UNIVERSIDADE ESTADUAL PAULISTA
CAMPUS DE PRESIDENTE PRUDENTE
FACULDADE DE CIÊNCIAS E TECNOLOGIA
Programa de Pós-Graduação em Ciências Cartográficas

FÁBIO FELICIANO DE OLIVEIRA

DESENVOLVIMENTO DE UMA PLATAFORMA DE SOFTWARE PARA MODELAGEM DIGITAL DE TERRENOS BASEADA EM TIN

Presidente Prudente
2010

Fábio Feliciano de Oliveira

Desenvolvimento de uma Plataforma de Software para a Modelagem Digital de Terrenos baseada em TIN

Dissertação apresentada ao Programa de Pós-Graduação em Ciências Cartográficas da FCT/UNESP - Campus de Presidente Prudente para a obtenção do título de Mestre em Ciências Cartográficas.

Orientador: Prof.Dr. Messias Meneguette Jr

Co-orientador: Prof. Dr. Marco Antonio Piteri

UNIVERSIDADE ESTADUAL PAULISTA
Faculdade de Ciências e Tecnologia
Pós-Graduação em Ciências Cartográficas

Presidente Prudente

2010

Oliveira, Fábio Feliciano.
O47d Desenvolvimento de uma Plataforma de Software para a Modelagem Digital de Terrenos baseada em TIN / Fábio Feliciano de Oliveira. - Presidente Prudente : [s.n], 2010
98 f.

Orientador: Messias Meneguette Júnior
Dissertação (mestrado) - Universidade Estadual Paulista, Faculdade de Ciências e Tecnologia
Inclui bibliografia

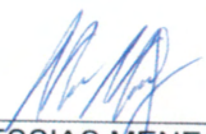
1. Modelagem digital de terreno. 2. Ferramentas opensource. 3. Estrutura de dados topológica. 4. Visualização. I. Meneguette Júnior, Messias. II. Universidade Estadual Paulista. Faculdade de Ciências e Tecnologia. III. Título.

CDD 623.71

Ficha catalográfica elaborada pela Seção Técnica de Aquisição e Tratamento da Informação – Serviço Técnico de Biblioteca e Documentação - UNESP, Câmpus de Presidente Prudente.

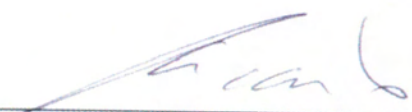
unesp
UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Presidente Prudente

BANCA EXAMINADORA

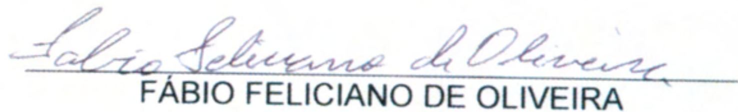


PROFA. DRA. MESSIAS MENEGUETTE JUNIOR
(ORIENTADORA)

PROF. DA. JOÃO FERNANDO CUSTÓDIO DA SILA
(FCT/UNESP)



PROF. DR. RICARDO LUÍS BARBOSA (USP/SP)



FÁBIO FELICIANO DE OLIVEIRA

PRESIDENTE PRUDENTE (SP), 27 DE SETEMBRO DE 2010

RESULTADO: APROVADO

Faculdade de Ciências e Tecnologia
Seção de Pós-Graduação
Rua Roberto Simonsen, 305 CEP 19060-900 Presidente Prudente SP
Tel 18 3229-5352 fax 18 3 223-4519 posgrad@fct.unesp.br

Resumo

A representação da superfície terrestre tem sido um desafio recorrente tanto para a comunidade acadêmica como para a indústria. Apesar de conseguirmos representar superfícies topográficas com um bom grau de precisão por meio de mapas, estes nem sempre são as melhores ferramentas na compreensão de relevos mais complexos. Mais recentemente, com a diminuição progressiva dos custos de aquisição e manutenção de computadores e o desenvolvimento de um amplo conjunto de técnicas computacionais, o uso do computador tornou-se imprescindível na representação da superfície de terrenos. Apesar de existirem sistemas de *software* comerciais que possuem funcionalidades associada à modelagem digital de terrenos, muitas vezes eles estão limitados a resolução de problemas específicos, não abordando de forma mais genérica várias questões associadas à representação e manipulação computacional da superfície de um terreno. Outro aspecto importante é a carência de *software* livre nessa área. Nesse sentido, a maior contribuição desse trabalho é a especificação e implementação da arquitetura de um sistema de *software opensource* voltado para a representação de modelos digitais de terrenos baseado em TIN (Triangular Irregular Network) e capaz de suportar um vasto conjunto de aplicações. A implementação do sistema obedece às filosofias de programação orientada a objetos e programação genérica, dois paradigmas atuais e que possibilitam a integração de várias tecnologias (CGAL, GDAL, OGR, OpenGL, OpenSceneGraph e Qt) que tem se tornado padrão no mercado de desenvolvimento de software, a maioria delas disponibilizadas em domínio público. Por outro lado, o núcleo de representação do sistema tem a capacidade de trabalhar com múltiplas estruturas de dados topológicas que permite extrair em tempo constante, todas as nove relações de conectividade entre as entidades vértices, arestas e faces, existentes numa subdivisão planar triangular, de forma independente da dimensão do problema. Esta característica é de fundamental importância na implementação de aplicações de tempo real, visto que mediante os avanços das tecnologias de aquisição de dados sobre a superfície de terrenos, podem ser gerados modelos de malhas triangulares da ordem de milhões de pontos.

Abstract

The representation of land surface has been a challenge both for the academic community and in industry. Although we can represent topographical surfaces with a good degree of accuracy by means of maps, these are not always the best tools in the understanding of the complex reliefs. More recently, with progressive decrease in the cost of acquisition and maintenance of computers and development of a wide range of computational techniques, the use of computer became crucial for the representation of land surface. Although there exist commercial software systems with modeling digital land features, they often are limited to solving specific problems, not addressing more generally a number of issues related to the representation and computational manipulation of the land surface. Another important aspect is the lack of free software in this area. In this sense, the greatest contribution of this work is to specify the architecture of a open source software system focused on the representation of digital terrain models based on TIN (Triangular Irregular Network) and capable to support a wide range of applications. The system implementation follows the philosophy of object oriented programming and generic programming, two paradigms current and enabling the integration of various technologies (CGAL, GDAL, OGR, OpenGL, OpenSceneGraph and Qt) that have become standard in the market development software, most of them available as public domain. Furthermore, the core representation of the system has the ability to work with multiple topological data structures from which can be extracted, in constant time, all the nine connectivity relations between the entities vertices, edges and faces existing in a planar triangulation, independently of the problem dimension. This feature is of fundamental importance in the implementation of real time applications, since through the advances in technology of acquiring data on the land surface, it can be generated triangular mesh models of the order of million points.

Lista de Figuras

1 Representação Bidimensional e Tridimensional de Superfícies. 15

2 Produção de Modelo Digital de Terreno 22

3 Superfície gerada usando o modelo Grid. 26

4 Superfície gerada usando o modelo TIN. 27

5 Representação de uma superfície usando os modelos Grid e um TIN 29

6 Possíveis triangulações para um mesmo conjunto de pontos. 36

7 Influência da triangulação na superfície gerada. 37

8 Duas triangulações e seus menores ângulos. 40

9 Diagrama de Voronoi. 40

10 Relação entre o Diagrama de Voronoi e a Triangulação de Delaunay. 41

11 Critério da Circunferência Circunscrita Vazia. 43

12 Princípio de *Flip-Edge*. 44

13 Entidades Topológicas de um TIN 44

14 Exemplo de representação de um TIN 45

15 Relações de conectividade de um TIN. 47

16 Estrutura de Dados *Winged-Edge Modificada* 49

17 Estrutura de Dados *Half-Edge* 51

18 Estrutura de Dados TDS 53

19 Representação de Arestas na TDS 54

20 Vértices e Faces infinitas na TDS 55

21 Interpolação Local e Global 58

22 Interpolação Exata e Aproximada 59

23 Interpolação Contínua e Abrupta 59
24 Representação de uma superfície utilizando Diagrama de Voronoi. 60
25 Estratégias de busca de pontos 63
26 Interpolação pela Vizinhança Natural 65
27 Interpolação Bilinear de triângulos. 67
28 Metodologia utilizada da geração de um TIN 69
29 Estrutura da CGAL. 71
30 Linha do Tempo de desenvolvimento do OpenGL 74
31 Desenvolvedores de Extensões para o OpenGL 75
32 Arquitetura Cliente-Servidor do OpenGL 76
33 Grafo de Cena 79
34 Arquitetura do Sistema Proposto 85
35 Arquitetura Modelo de Dados. 91
36 UML da Representação Computacional TIN 94
37 UML carregamento de arquivos 104
38 UML Padrão de Projeto *Factory*. 106
39 Aplicação do padrão de projeto *Factory*. 107
40 Renderização de um TIN 111
41 Modelo Esquemático de *Vertex Array* com Vetor de Índice. 113
42 Modelo Esquemático da Renderização utilizando *Vertex Array*. 114
43 Modelo Esquemático da Renderização utilizando *Buffers Objects*. 115
44 Modos de Renderização de um TIN 119
45 Teste de intersecção do mouse com o terreno. 122
46 Funcionamento de *trackball* virtual. 124
47 Diagrama de Estados de um telefone 127
48 Diagrama de classe do Controller 128

49 Diagrama de estados da classe Controller 130
50 Renderização de MDTs 131
51 Janela de Visualização de Progresso da Triangulação. 132

Lista de Siglas

API = Application Programming Interface.

CAD = Computer aided design.

CGAL = Computational Geometry Algorithms Library.

DEM = Digital Elevation Model.

DSM = Digital Surface Model.

DTM = Digital Terrain Model.

EDT = Estrutura de Dados Topológica.

GDAL = Geospatial Data Abstraction Library.

GPS = Global Positioning System.

GUI = Graphical User Interface.

LASER = Light Amplification by Stimulated Emission of Radiation.

LIDAR = Ligth Detection Ranging.

MDE = Modelo Digital de Elevação.

MDS = Modelo Digital de Superfície.

MDT = Modelo Digital de Terreno.

MEF = Métodos de Elementos Finitos.

OGR = Simple Feature Library.

OpenGL = Opensorce Graphics Library.

OSG = Open Scene Graph.

Qt = Q-Toolkit.

RADAR = Radio Detecting and Ranging.

SIG = Sistemas de Informação Geográfica.

STL = Standard Template Library.

TDS = Triangulation Data Structure.

TIN = Triangular Irregular Network.

TTL = Template Triangulation Libray.

Sumário

1 Introdução **14**

1.1 Objetivos 15

1.2 Justificativa 16

1.3 Organização do Trabalho 17

2 Modelagem Digital de Terrenos **19**

2.1 Terminologia Utilizada 19

2.2 Histórico 21

2.3 Produção de Modelos Digitais de Terreno 22

2.4 Aquisição de Dados para MDTs 23

2.4.1 Levantamentos Topográficos 23

2.4.2 Restituição Fotogramétrica 24

2.4.3 Bases Cartográficas 24

2.4.4 Laser 24

2.4.5 Radar 25

2.5 Manipulação Computacional de MDTs 25

2.5.1 Modelo de Malhas/Grades Regulares (Grid) 26

2.5.2 Modelo de Malha Triangular Irregular (TIN) 27

2.6 Aplicações de Modelos Digitais de Terreno 29

2.6.1 Aplicações Científicas 30

2.6.2 Aplicações Industriais 31

2.6.3 Aplicações Operacionais 32

2.6.4 Aplicações Militares 33

3 Malhas Triangulares Irregulares (TIN) 35

3.1 Triangulação de Delaunay 38

3.1.1 Critério MaxMin 39

3.1.2 Diagramas de Voronoi 40

3.1.3 Critério da Circunferência Circunscrita Vazia 42

3.1.4 Princípio de *Flip-Edge* 43

3.2 Estrutura de Dados para o Modelo TIN 44

3.2.1 Estrutura de Dados Topológica 45

3.2.2 A Estrutura *Winged-Edge* Modificada 48

3.2.3 A Estrutura *Half-Edge* 51

3.2.4 Estrutura TDS (Triangulation Data Struct) 52

4 Interpolação 56

4.1 Classificação dos Métodos de Interpolação 57

4.1.1 Interpolação por Ponto ou Área 57

4.1.2 Interpolação Local ou Global 57

4.1.3 Interpolação Exata ou Aproximada 59

4.1.4 Interpolação Contínua ou Abrupta 59

4.2 Vizinho mais Próximo 60

4.3 Média Simples 61

4.4 Inverso da Distância 61

4.5 Interpolação pela Vizinhança Natural 64

4.6 Interpolação Linear 65

4.7 Interpolação Bilinear 66

5 Sistema Proposto 68

5.1 Ferramentas Utilizadas 69
5.1.1 Biblioteca CGAL 70
5.1.1.1 Biblioteta *Core* 71
5.1.1.2 *Kernel* Geométrico 71
5.1.1.3 Biblioteca Básica 72
5.1.1.4 Bibliotecas de Suporte 72
5.1.1.5 Computação Exata 73
5.1.2 OpenGL 73
5.1.3 OpenSceneGraph 78
5.1.4 Biblioteca GDAL/OGR 81
5.1.5 Sistema Gerenciador de Janelas Qt 83
5.2 Arquitetura do Sistema 84
5.3 Integração entre os Componentes do Sistema 86
5.3.1 Programação Orientada a Objetos 86
5.3.2 Programação Genérica 87
5.3.3 Padrões de Projeto 87
5.4 Modelo Computacional TIN 90
5.4.1 Navegação no TIN usando a *Winged-Edge* 94
5.4.2 Estendendo Vértices Arestas e Faces 96
5.4.3 *Handlers* 98
5.4.4 Iteradores 98
5.4.5 Circuladores 100
5.5 Integração com Sistemas Existentes 104
5.5.1 Padrão de Projeto *Factory* 106
5.6 Visualização 110
5.6.1 Modo Imediato 110

5.6.2 *Vertex Arrays* . . . 112
5.6.3 *Buffers Objects* . . . 114
5.6.4 Implementação da Renderização . . . 117
5.6.5 Interação com a cena 3D do Terreno . . . 121
5.6.6 Navegação na cena 3D no Terreno . . . 123
5.7 Interface Gráfica com o Usuário . . . 126

6 Conclusão **133**
6.1 Trabalhos Futuros . . . 134
6.1.1 Geração do TIN . . . 135
6.1.2 Operações de Pós-Processamento do TIN . . . 135
6.1.3 Suporte a Diferentes Sistemas de Projeção Cartográfica . . . 135
6.1.4 Visualização . . . 135

Referências **136**

1 Introdução

Conseguir representar a superfície terrestre ou parte dela de uma maneira conveniente e com determinado grau de precisão, tem sido um desafio para a comunidade científica. Os mapas têm sido o meio mais utilizado para esse fim. Mapas modernos são projetados com alto nível de rigor matemático, possibilitando que informações de natureza espacial (ex: posição, orientação, medidas de distância, área e volume) possam ser extraídas dos mesmos com boa segurança. Estes ainda possuem um sistema simbólico bem definido e geralmente intuitivo, permitindo ao usuário, dependendo do tipo do mapa, obter rapidamente informações específicas, além de promover uma visão geral graças a sua generalização (LI; ZHU; GOLD, 2005).

A representação da superfície de um terreno, na Cartografia convencional, pode fazer uso de pontos cotados, cores hipsométricas, sombreados, curvas de nível, entre outros. Em particular, curvas de nível são os meios mais usuais de representação estática e bidimensional de uma superfície topográfica. Nesta representação, a superfície do terreno é descrita por linhas contínuas unindo pontos que possuem um mesmo valor de altitude, ou seja, linhas com valores de elevação constante (isolinhas). Um dos maiores inconvenientes dessa forma de representação, é que os valores de elevação da superfície do terreno são representados apenas ao longo de isolinhas (anomalias do terreno entre as linhas não são identificadas), sendo necessária a utilização de métodos de interpolação para se inferir valores de elevação entre as isolinhas (EL-SHEIMY; VALEO; HABIB, 2005).

Por outro lado, como é ilustrado pela Figura 1, existe uma perda de informação devido ao fato dos mapas usarem uma representação bidimensional para uma realidade tridimensional. Além disso, percebe-se pela Figura 1a, que a retirada de informações tridimensionais (contidas apenas de forma implícita nesse tipo de mapas) não é intuitiva, cabendo aos usuários deste a construção e visualização mental da realidade 3D representada neste (Figura 1b). Essa limitação, mostra a necessidade de se usar uma representação 3D da superfície terrestre.

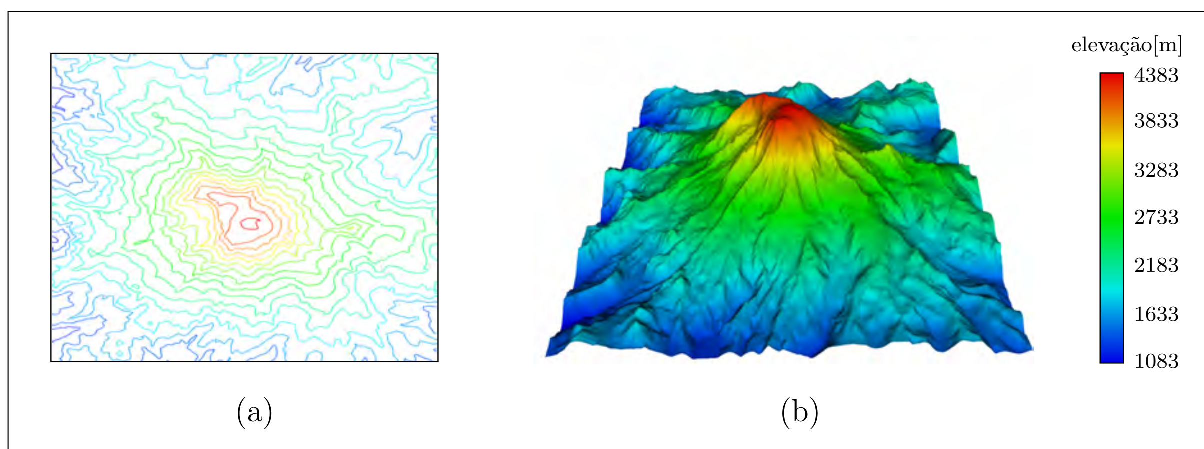


Figura 1: Representação bidimensional e tridimensional da superfície de um terreno: (a) representação 2D; (b) representação 3D.

Com o uso intensivo do computador na Cartografia e as consequentes facilidades de armazenamento, processamento e visualização de dados cartográficos, tornou-se mais evidente a necessidade de se elaborar e criar modelos digitais de terrenos apropriados para a representação e manipulação computacional de porções da superfície terrestre e que consigam transmitir mais facilmente ao usuário as suas características espaciais.

1.1 Objetivos

O objetivo central deste trabalho é fazer a especificação e implementação da arquitetura de uma plataforma de *software*, robusta e eficiente, voltada para a modelagem digital de terrenos que sirva como base para o desenvolvimento de aplicações em diferentes áreas do conhecimento. De modo a ilustrar a sua viabilidade, algumas funcionalidades foram desenvolvidas e já se encontram operacionais. No Capítulo 5 são dados maiores detalhes de todas funcionalidades já implementadas.

A arquitetura do sistema é fundamentada numa estrutura de dados topológica e utiliza várias tecnologias *opensource* que tem se tornado padrão na indústria. Essas tecnologias são altamente multidisciplinares e envolvem o conhecimento de várias áreas da computação como: Engenharia de *Software*, Algoritmos, Computação Gráfica e Geometria Computacional. A construção do sistema também seguiu às filosofias de orientação a objetos e programação genérica; dois paradigmas atuais que além de possibilitarem a integração entre as várias tecnologias utilizadas, também contribuirão para o aumento da manutenibilidade e extensibilidade do sistema proposto.

Mais especificamente, todos os aspectos relativos à criação e ao gerenciamento de janelas do sistema foram feitos com o auxílio do *framework* de desenvolvimento **Qt** (**Q**-Toolkit). As diferentes operações de visualização sobre o terreno representado foram implementadas por meio da biblioteca **OpenGL** (**O**pensource **G**raphics **L**ibrary). As tecnologias **Qt** e **OpenGL** são perfeitamente integradas, contribuindo para a eficiência e a robustez do sistema. Visando a eficiência e a pronta triangulação dos pontos, o núcleo de representação do sistema (arquitetura) será baseado numa estrutura de dados topológica, (mais especificamente na estrutura de dados topológica **TDS** (**T**riangulation **D**ata **S**tructure) fornecida pela **CGAL** (***C**omputational **G**eometry **A**lgorithms **L**ibrary*)), pois qualquer informação relativa à topologia pode ser obtida em tempo constante ou proporcional ao número de entidades envolvidas, independentemente da dimensão do problema.

Considerando a importância dessas tecnologias na indústria de *software* e principalmente no contexto desse trabalho, todas elas serão descritas em maiores detalhes na Seção 5.1.

1.2 Justificativa

Existe uma quantidade relativamente razoável de sistemas de *software* comerciais que fazem uso, ou contêm, ferramentas para a manipulação de **MDT** (**M**odelo **D**igital de **T**erreno). Entre eles podem ser citados: *Topograph, Idrisi, Envi, ArcGIS, LPS* e o *SOCCET SET*. Entretanto a maior parte desses sistemas são *softwares* voltados para SIG (Sistemas de Informação Geográfica), possuindo apenas módulos para a modelagem de terrenos, não sendo, portanto específicos para esse fim. Mesmo assim, os módulos fornecidos por esses *softwares*, muitas vezes são exclusivos para o uso em aplicações específicas como projetos de estradas ou modelagem hidrológica. Isso acaba levando a dependência do uso de mais de um sistema de *software* para a resolução de problemas relacionados à modelagem digital de terrenos. Por exemplo: um *software* é usado para gerar o **MDT**, outro para cálculo de visibilidade entre pontos no terreno gerado, outro para cálculo de declividades e assim por diante.

Outro fator importante é a ausência de *software* livre nessa área. Apesar da existência de programas consistentes como *Spring, Landserf* ou *Topocal*, esses sofrem dos mesmos problemas relacionados aos *softwares* de uso comercial, não existindo um sistema de *software* exclusivo e genérico o suficiente para o uso em um amplo conjunto de problemas relacionados à modelagem digital de terrenos.

Ainda os sistemas de modelagem digital terreno mais acessíveis são em geral baseados em estruturas de dados simples e por isso são relativamente lentos, não sendo próprios para a manipulação de terrenos representados por conjunto da ordem de milhares de pontos e nem permitem que a eles se agreguem, com facilidade, outras aplicações. Desta maneira, o objetivo desse trabalho é propor a criação de um sistema de *software* específico para a modelagem digital de terreno.

1.3 Organização do Trabalho

No Capítulo 2, é feita uma descrição da Modelagem Digital de Terrenos e da sua importância na representação de fenômenos relacionados à topografias ou superfícies similares. Também são dadas descrições dos principais termos relacionados a área assim como a terminologia adotada no contexto deste trabalho e um breve histórico do desenvolvimento da Modelagem Digital de Terreno ao longo do tempo.

Esse Capítulo, também contem uma visão geral das etapas relacionadas a Modelagem Digital de Terrenos. Mais precisamente são descritas as etapas de aquisição de dados, que basicamente consiste na obtenção e digitalização do conjunto de amostras da superfície do terremo, manipulação que está relacionada essencialmente com a construção e a representação computacional de sua superfície sendo descritos os modelos de representação por Grid e por **TIN** (**T**riangular **I**rregular **N**etwork) e finalmente a etapa de aplicações onde são dados exemplos de diversas áreas, profissionais e aplicações que podem fazer uso de **MDTs**

O Capítulo 3 focaliza a geração de modelos **TIN**s por meio de técnicas automáticas de triangulação. Entretanto, dado o fato de uma triangulação não ser única, é discutida a influência que diferentes tipos de triangulações, para uma mesma amostra de pontos de um terreno, podem ter na superfície final modelada, e quais características que uma triangulação deve ter para melhor representá-la. Neste caso é feita uma descrição detalhada dos conceitos, definições e propriedades da Triangulação de Delaunay e o porquê dessa técnica de triangulação ser tão indicada para a geração de malhas triangulares (neste caso **TIN**) utilizadas na representação de superfícies.

Além da questão da geração de triangulações, no Capítulo 3 também é discutido o conceito de **EDT** (**E**strutura de **D**ados **T**opológica) e as vantagens da representação computacional de **TIN**s por meio destas. O Capítulo se encerra com a descrição das **EDTs** *Winged-Edge Modificada*, *Half-Edge* e **TDS** e a suas vantagens e desvantagens na

representação de malhas triangulares. Lembrando que neste trabalho foi utilizada a **TDS** na representação de **TIN**s.

O Capítulo 4 contem a descrição da importância dos métodos de interpolação na Modelagem Digital de Terreno e as suas aplicações em **SIG** (**S**istemas de **I**nformação **G**eográfica). Também é dada uma rápida descrição de algumas formas de classificação existentes de métodos de interpolação. Ainda, nesse Capítulo, são discutidos os seguintes interpoladores: vizinho mais próximo, média simples, inverso da distância, interpolação pela vizinhança natural e interpolação linear e bilinear.

No Capítulo 5 é feita a apresentação do sistema proposto. Inicialmente é feita uma descrição da metodologia utilizada na geração e representação de **TIN**. Em seguida, são enumeradas e descritas todas as tecnologias e ferramentas utilizadas no desenvolvimento do sistema proposto, a saber, as bibliotecas **CGAL**, **OSG** (***O**pen **S**cene **G**raph*), **GDAL** (***G**eospatial **D**ata **A**bstraction **L**ibrary*), **OGR** (*Simple Feature Library*) e **Qt**. Uma vez descritas as tecnologias utilizadas, é feita uma discussão da arquitetura do sistema, com a elucidação de seus componentes e tecnologias utilizadas na sua implementação.

Dada a complexidade e quantidade de tecnologias utilizadas no desenvolvimento do sistema, também é descrito nesse Capítulo os paradigmas de orientação a objetos e programação genérica, assim como todos os padrões de projeto que foram utilizados. O Capítulo é encerrado com uma descrição das características e detalhes de implementação de todos os módulos constituintes do sistema.

Por fim, no Capítulo 6 são apresentadas algumas conclusões sobre o que foi implementado do sistema proposto, assim como uma discussão sobre a integração dos vários componentes de softwares utilizados na sua constituição. Nesse Capítulo também são feitas algumas propostas de trabalhos futuros que podem ser executados a partir do presente trabalho.

2 *Modelagem Digital de Terrenos*

Segundo Sakude (1992) a modelagem digital de terrenos representa a reconstrução da superfície de um terreno pelo uso de um modelo de aproximação ou interpolação des dados pontuais, com coordenadas (x, y, z), obtidos sobre o terreno.

A existência de um **MDT** associado a uma porção da superfície terrestre possibilita que uma série de operações possa ser realizada sobre ela, como por exemplo: extração de curvas de níveis e redes de drenagem; projetos de construção de estradas, túneis e dimensionamento dos deslocamentos de terras; e ainda, estudos de impacto ambiental, como o cálculo de áreas alagadas na construção de barragens hidroelétricas e do respectivo volume do reservatório. Outra aplicação muito interessante está associada a jogos e a simulação de vôos que é uma atividade fundamental no treinamento de pilotos, principalmente na área militar (SULEBAK, 2000).

Como pode ser observado, o uso de **MDTs** é decisivo na modelagem de fenômenos relacionados à topografias ou superfícies similares, e outras etapas como análise e visualização de superfícies (WEIBEL; HELLER, 1991). Desta forma, a produção e o uso de **MDT** são de fundamental importância em atividades de planejamento, auxílio à tomada de decisão e desenvolvimento de projetos ligados as áreas como SIG, Engenharia Cartográfica, Ambiental, Militar, Civil (EL-SHEIMY; VALEO; HABIB, 2005; LI; ZHU; GOLD, 2005).

2.1 Terminologia Utilizada

Entretanto, segundo Li, Zhu e Gold (2005), o conceito de criação de modelos digitais usados na representação de superfícies terrestre é relativamente recente existindo muitos significados para o termo "Modelo Digital de Terreno" (**MDT**). A seguir são dadas descrições dos principais termos relacionados a área de modelagem digital de terrenos, assim como o significado associados aos mesmos no contexto desse trabalho.

Modelo Digital de Elevação (MDE): termo (em inglês, **DEM** = **D**igital **E**levation **M**odel) que geralmente é utilizado quando apenas o relevo é representado pelo modelo (WEIBEL; HELLER, 1991), ou seja, o modelo deve conter apenas valores de elevação associadas ao relevo da superfície modelada (ZHOU; LEES; TANG, 2008), excluindo-se valores de elevação de vegetação (ex: árvores e arbustos) ou feições construídas pelo homem (ex: prédios e casas). Geralmente esse termo também está associado à criação de matrizes bidimensionais contendo tais valores de elevação utilizando espaçamento constante (EL-SHEIMY; VALEO; HABIB, 2005);

Modelo Digital de Superfície (MDS): termo (em inglês, **DSM** = **D**igital **S**urface **M**odel) usado quando são incluídos não apenas dados referentes ao relevo da superfície modelada, mas também outras feições como construções (prédios, casas, galpões) e vegetação. A remoção, com um bom grau de precisão, desses artefatos é de extrema importância na geração de **MDT** e **MDE** principalmente em áreas urbanas ou com muita vegetação;

Modelo Digital de Terreno (MDT): termo (em inglês, **DTM** = **D**igital **T**errain **M**odel) que assim como **MDE** é utilizado quando apenas o relevo é representado pelo modelo. Entretanto um **MDT** possui maior complexidade se comparado com **MDE**, pois este pode conter, em adição aos valores de elevação, outros elementos geográficos ou feições **SIG** como rios, estradas, linhas de quebra etc. Ainda, um **MDT** pode conter informações derivadas do próprio terreno como informações de declividade e visibilidade (EL-SHEIMY; VALEO; HABIB, 2005; LI; ZHU; GOLD, 2005);

No contexto deste trabalho será utilizado o termo **MDT** (**M**odelo **D**igital de **T**erreno), visto que no sistema proposto, é permitida a associação de outras informações ao modelo (não apenas valores de elevação). É importante destacar que o sistema utiliza um modelo 2.5D para a representação digital de terrenos. O termo 2.5D se deve ao fato dos valores de elevação (coordenada z) serem tratados apenas como um atributo associado a dados de posições (coordenadas (x, y)) sobre o terreno, não sendo permitida ainda, a associação de mais de um valor de elevação (coordenada z) a uma mesma posição (coordenadas (x, y)) sobre o terreno (WEIBEL; HELLER, 1991).

2.2 Histórico

Modelos de Terreno sempre tiveram um papel de destaque em áreas como planejamento e gerenciamento de recursos, engenharia civil e ciências da terra . Entretanto, originalmente Modelos de Terrenos se limitavam a modelos físicos feitos de materiais como borracha, plástico, argila, gesso etc. Em meados dos anos 50 com a introdução de técnicas matemáticas e o uso de computadores na modelagem de terrenos, tornou-se possível a modelagem digital de porções da superfície terrestre permitindo o surgimento dos primeiros **MDT** (EL-SHEIMY; VALEO; HABIB, 2005; LI; ZHU; GOLD, 2005).

Segundo Li, Zhu e Gold (2005), o desenvolvimento da Modelagem Digital de Terrenos se deu da seguinte maneira:

- **Meados do anos 50:** Miller e Laflamme (1958) introduzem **MDT** na engenharia civil e fazem uso deste no monitoramento de mudanças na superfície terrestre;
- **Anos 60:** A Modelagem Digital de Terrenos se torna uma importante área de pesquisa para a Sociedade Internacional de Fotogrametria e Sensoriamento Remoto (*International Society for Photogrammetry and Remote Sensing*). Técnicas provenientes da área de fotogrametria, constituem a principal fonte geradora de **MDTs**;
- **Final dos anos 60:** Maior parte das pesquisas voltadas para a modelagem de superfícies e estudos sobre a geração de contornos a partir de **MDE**. Neste período são feitas propostas de vários métodos de interpolação utilizando-se diferentes tipos de média móvel, assim como diversos métodos de triangulação foram propostos;
- **Anos 70:** Mudança de foco para o controle de qualidade e estratégias de amostragem com estudos experimentais e análise teórica tendo como objetivo a geração de modelos matemáticos de predição de acurácia de **MDT**;
- **Final anos 80:** Produção em larga escala de modelos digitais de terrenos com o uso de plotadores analíticos (*analitical plotters*) como ferramentas de aquisição de dados, e devido ao avanço de técnicas fotogramétricas de “correspondência de imagens”;
- **A partir dos Anos 90:** com o desenvolvimento da Ciência da Computação, barateamento de computadores e o advento da Cartografia Digital com o surgimento dos primeiros **SIG**s, os **MDTs** tornaram-se um componente importante de infraestrutura de dados espaciais, sendo atualmente utilizados em um grande conjunto de aplicações nas áreas de geociências e engenharia;

2.3 Produção de Modelos Digitais de Terreno

De uma perspectiva geral, a problemática da Modelagem Digital de Terreno pode ser dividida em três grandes áreas de pesquisa, a saber: *Aquisição*, *Manipulação* e *Aplicações* (Figura 2).

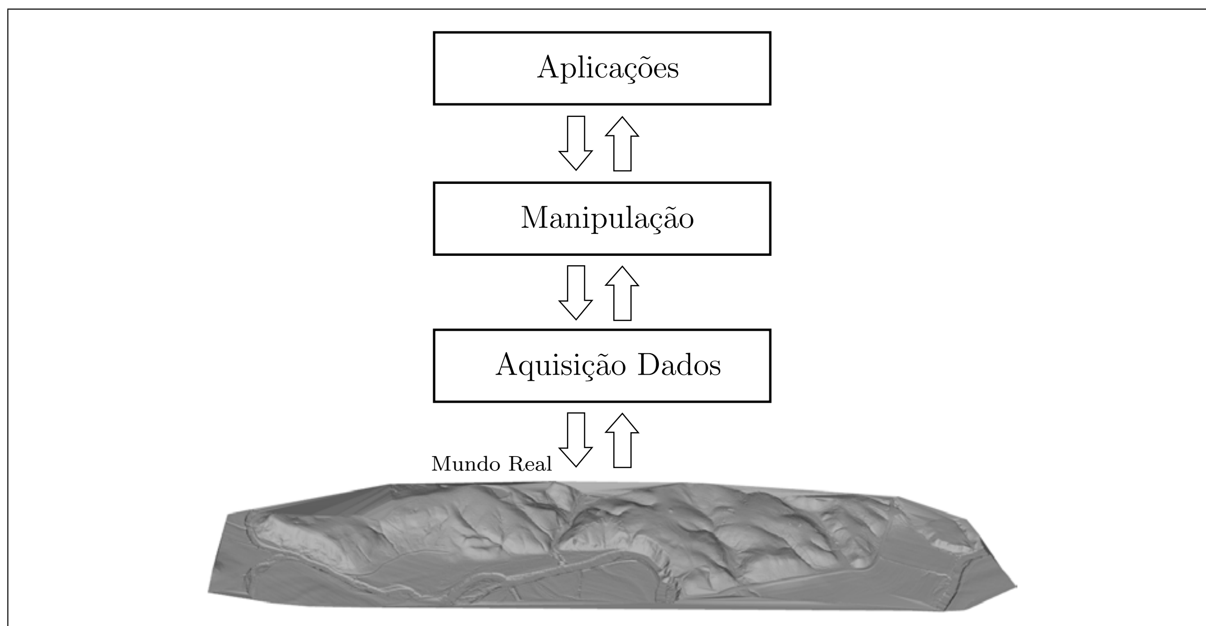


Figura 2: Produção de Modelo Digital de Terreno

Basicamente a etapa de aquisição compreende a obtenção e digitalização de um conjunto de amostras da superfície do terreno. Os dados para elaboração dos modelos podem ser obtidos a partir de mapas existentes, técnicas fotogramétricas, levantamentos terrestres, ou por meio de outros sistemas como **RADAR** (**R**adio **D**etecting and **R**anging) e **LASER** (**L**ight **A**mplification by **S**timulated **E**mission of **R**adiation). A etapa de aquisição dos dados forma a base de todas as operações que poderão ser efetuadas sobre o **MDT**, sendo que quanto maior for a precisão dos dados coletados, mais preciso será o modelo.

Independentemente do procedimento utilizado para se obter os dados, para que os mesmos possam ser utilizados de forma conveniente, estes devem ser organizados computacionalmente na forma de uma estrutura de dados, de modo que seja possível a recuperação e extração das relações entre os dados, de uma forma automática e eficiente, gerando as informações necessárias a uma dada aplicação.

A fase de manipulação está relacionada essencialmente com a construção e a representação computacional da superfície de um terreno, ou seja, com a estrutura combinatória e topológica que será construída a partir da amostragem realizada na fase anterior. O mo-

delo de dados usado por um **MDT** deve permitir operações básicas sobre o mesmo como: modificações, refinamentos e procedimentos de conversão entre diferentes estruturas de dados que podem ser utilizadas para representá-lo (EL-SHEIMY; VALEO; HABIB, 2005).

As aplicações são procedimentos de análise executados sobre o **MDT** como: testes de visibilidade entre pontos na superfície do terreno, cálculos de distância, área e volume, geração de mapas de declividades, visualização do modelo em perspectiva, perfis e seções transversais, entre outras.

O presente trabalho se limita a etapa de modelagem relativa a manipulação computacional de **MDTs**.

2.4 Aquisição de Dados para MDTs

A fase de amostragem dos dados referente à superfície de um terreno a ser modelada constitui a tarefa mais importante de todo o processo de geração de modelos digitais de terrenos, isto porque ela serve como base para todas posteriores operações que possam ser feitas sobre o modelo (EL-SHEIMY; VALEO; HABIB, 2005). Basicamente essa fase se resume a aquisição de dados, caracterizadas pelos levantamentos que irão permitir obter as coordenadas (x, y, z) que representem a superfície do terreno.

A fase de aquisição dos dados não pode gerar um número insuficiente de amostras (gerando uma perda de precisão do modelo) nem um número exagerado de amostras (dados redundantes podem sobrecarregar o sistema) (FELGUEIRAS, 2001). Além dos dados de elevação, quando possível, ainda devem ser obtidas informações adicionais que ajudem a descrever a superfície em questão como canais de drenagem e outras descontinuidades (WEIBEL; HELLER, 1991).

Ainda segundo Weibel e Heller (1991), a escolha das fontes de dados e das técnicas de amostragem é um fator crítico com relação à qualidade de **MDT**. Desta forma a escolha da fonte de dado mais adequada deve levar em conta fatores como: eficiência, custo, precisão e maturidade tecnológica das ferramentas. A seguir é dada uma breve descrição de possíveis fontes de dados para a modelagem digital de terrenos.

2.4.1 Levantamentos Topográficos

As amostras são coletadas diretamente do terreno por meio de equipamentos topográficos como teodolito, estações totais e níveis. Trata-se de uma técnica demorada de

aquisição de dados, entretanto a mesma produz como resultado dados com um alto grau de acurácia. Segundo Casaca, Matos e Baio (2007), geralmente essa técnica de aquisição é aplicada ao planejamento e desenvolvimento de projetos específicos de pequenas áreas.

2.4.2 Restituição Fotogramétrica

Nessa técnica, primeiramente é feita a captura de pares de fotografias de uma mesma região do terreno tomadas em estações terrenas (fotogrametria terrestre) ou em estações aéreas (aerofotogrametria). O modelo tridimensional da área fotografada é obtido a partir da observação estereoscópica deste par de fotos. Mais precisamente, as fotografias são orientadas de modo a reproduzir a posição de onde foram tomadas e a observação de ambas as fotografias em estereoscopia produz o modelo do terreno. Essa técnica tem a vantagem de produzir amostras com um alto grau de precisão (dependendo da resolução das imagens utilizadas), além de poder ser aplicada a projetos de grandes dimensões (CASACA; MATOS; BAIO, 2007).

2.4.3 Bases Cartográficas

Também é possível gerar dados para a produção de **MDT** a partir de bases cartográficas como, por exemplo, mapas altimétricos ou topográficos (onde o relevo é representado por meio de curvas de nível) e perfis. Essas bases cartográficas podem ser digitalizadas utilizando técnicas manuais, semi-automática ou automáticas, servindo como uma opção aos métodos diretos de captura de dados do terreno.

Ainda dada à existência de um grande volume de bases cartográficas, esse método indireto geralmente é aplicado a grandes extensões territoriais, cartograficamente representadas em escalas pequenas. As principais aplicações são ligadas à simulação de vôos, representação do relevo para fins militares e outros projetos em que sejam suficientes **MDTs** de média ou baixa escala (WEIBEL; HELLER, 1991).

2.4.4 Laser

Os dispositivos de mapeamento por **LASER** (**L**ight **A**mplification by **S**timulated **E**mission of **R**adiation) ou **LIDAR** (**Li**gth **D**etection **R**anging) são uma recente inovação no mapeamento topográfico. Basicamente esses sistemas são compostos por um *scanner laser*, um sistema de navegação inercial, um receptor **GPS** (**G**lobal **P**ositioning **S**ystem) e controladores e dispositivos de gravação de dados. Geralmente esse dispositivo é acoplado

em um avião, sendo capaz de obter medidas (coordenadas (x, y, z)) de um grande número de pontos da superfície terrestre.

O sistema opera da seguinte forma: enquanto o avião voa ao longo de sua trajetória, é emitido um pulso **LASER** na direção da superfície terrestre e o sinal retornante (refletido pela superfície) é detectado. A posição dos pontos no terreno é dada pela medida do tempo de transmissão e retorno dos pulsos **LASER**, a velocidade da luz e a distância do *scanner* em relação ao terreno. O dispositivo de navegação inercial captura os dados de orientação da aeronave, e o dispositivo de recepção **GPS** recupera a sua posição. Então os sinais dos três dispositivos são processados e combinados gerando um conjunto de pontos 3D representando as coordenadas do terreno.

Atualmente *scanners lasers* possuem uma acurácia da ordem de 10 a 15 centímetros, constituindo-se dessa forma como uma boa alternativa de ferramenta efetiva no mapeamento topográfico e uma excelente fonte de dados para a produção de **MDT** (WOLF; DEWITT, 2000).

2.4.5 Radar

Técnicas de interferometria utilizando **RADAR** e **LASER** podem ser consideradas atualmente como as mais avançadas e efetivas tecnologias de aquisição de dados topográficos (SULEBAK, 2000).

Ao contrário de sensores ópticos que dependem de uma fonte externa de iluminação, o **RADAR** é um sistema ativo, isto é, esses sensores emitem sinais eletromagnéticos sobre um alvo ou superfície e gravam e/ou analisam o sinal retornado, desta forma, imagens de **RADAR** não dependem por exemplo da iluminação solar. Ainda, dado o comprimento de onda utilizado, imagens de **RADAR** não são afetadas por condições meteorológicas (ex: cobertura de nuvens, chuva e névoa), como as imagens geradas por sensores ópticos (HENDERSON; LEWIS, 1998).

2.5 Manipulação Computacional de MDTs

A existência de modelos digitais de terrenos pode facilitar análises espaciais relacionadas ao relevo como: análises de visibilidade, cálculos de áreas e volumes, visualização em 3D ou em perspectiva do mesmo (ABDUL-RAHMAN; PILOUK, 2007).

Dentre os diversos métodos existentes que podem ser utilizados para se obter uma representação de superfícies em um formato digital (**MDT**) os mais utilizados são: grades regulares (também conhecido como matriz de elevação ou Grid) e malha triangular irregular (**TIN**) (EL-SHEIMY; VALEO; HABIB, 2005; LI; ZHU; GOLD, 2005).

O método mais apropriado depende de variáveis como: quantidade, resolução e escala dos dados disponíveis (amostrados sobre o terreno, ou retirados de bases cartográficas), natureza da superfície a ser modelada (relevo mais plano ou irregular), e complexidade das técnicas que serão utilizadas na análise e manipulação do **MDT**.

2.5.1 Modelo de Malhas/Grades Regulares (Grid)

No modelo de dados Grid, a superfície do terreno é representada por uma estrutura de dados matricial onde cada uma de suas entradas contêm um valor de elevação (z) do terreno amostrada em intervalos regulares no plano (x, y) (EL-SHEIMY; VALEO; HABIB, 2005). Na Figura 3 é mostrada a representação 3D de um terreno utilizando o modelo Grid.

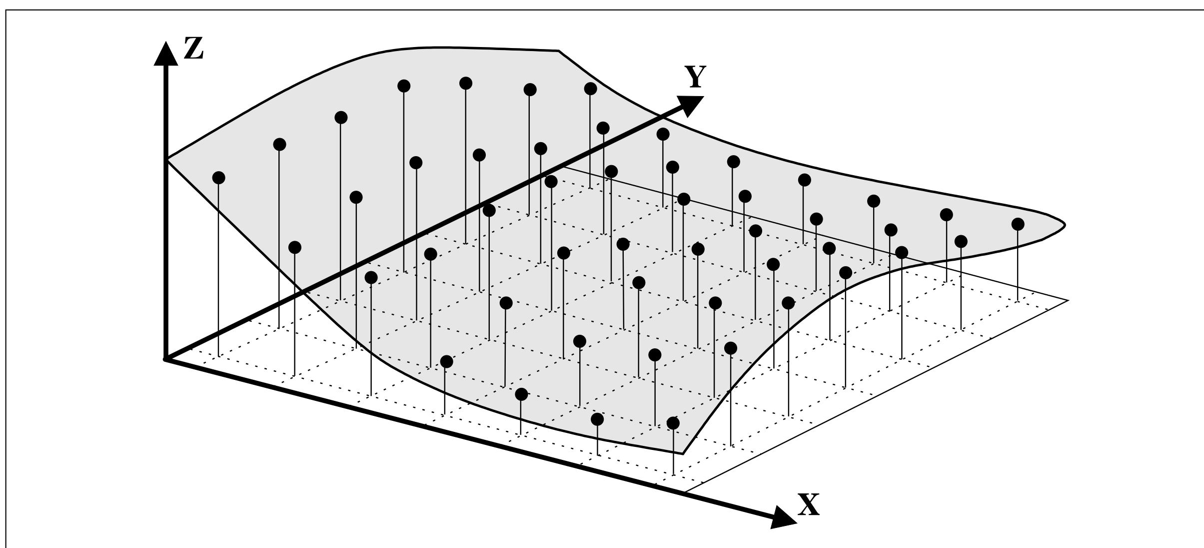


Figura 3: Superfície gerada usando o modelo Grid.
Fonte: adaptado de Ruhoff, Hendges e Pereira (2003)

Trata-se de um modelo de fácil armazenamento e manipulação computacional, uma vez que todas as relações topológicas entre os pontos amostrados sobre o terreno já se encontram guardados de forma implícita na estrutura de dados matricial, ou seja, relações de vizinhança entre pontos no Grid podem ser encontradas sem necessidade de computação adicional (ABDUL-RAHMAN, 1994).

Ainda de acordo com Abdul-Rahman (1994), esse modelo também possui fácil integração com bases de dados raster, sendo muito utilizado por técnicas de processamento de imagem, além de possuir um formato conveniente para algoritmos de processamento e análise de superfícies.

A grande desvantagem em se utilizar esse modelo se deve ao fato da distribuição espacial dos pontos utilizados, não estar relacionada com as características do terreno modelado (PETRIE; KENNIE, 1987), podendo gerar inconveniências como um grande número de dados redundantes em áreas mais planas do terreno, além de não conseguir representar de forma precisa áreas com relevos mais complexos não podendo, por exemplo, representar características do terreno que sejam menores que o espaçamento entre os pontos da malha.

Ainda esse modelo apenas representa pontos discretos no plano (x, y), não conseguindo dessa forma modelar uma superfície contínua. A continuidade da superfície modelada é obtida por meio de métodos de interpolação.

2.5.2 Modelo de Malha Triangular Irregular (TIN)

No modelo **TIN** a superfície do terreno é modelada por meio de uma malha de triângulos, adjacentes e não sobrepostos, de diferentes tamanhos e orientação (Figura 4). Sendo que esses triângulos são computados a partir de pontos (com coordenadas (x, y, z)) irregularmente espaçados amostrados sobre a superfície do terreno, desta forma, os pontos amostrados constituem os vértices da triangulação, e as arestas são geradas a partir dos segmentos formados pela conexão desses pontos (RASE, 2001).

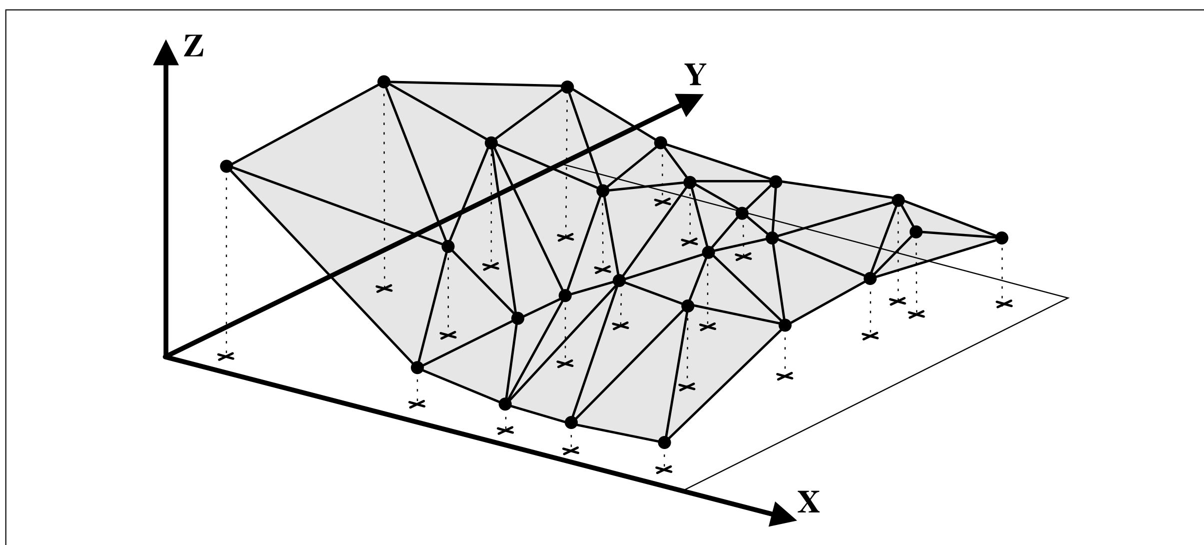


Figura 4: Superfície gerada usando o modelo TIN.
Fonte: adaptado de Ruhoff, Hendges e Pereira (2003)

O modelo **TIN**, geralmente, é considerado superior a outros modelos derivados de conjuntos de pontos amostrados sobre um terreno, devido ao fato de conseguir representar dados digitais de elevação em sua posição original, ou seja, esse modelo preserva a exata localização de cada ponto amostrado sobre terreno, melhorando assim a precisão do modelo (JONES; KIDNER; WARE, 1994).

Além de manter os pontos amostrados em sua localização original, esse modelo consegue representar de forma exata feições lineares (também chamadas de linhas de quebra) associadas a um terreno como, por exemplo, cadeias de montanhas, vales e estradas contribuindo ainda mais para a precisão do modelo.

A principal desvantagem do modelo **TIN** em relação ao Grid é devida a maior complexidade na criação e manuseio da estrutura de dados utilizada para sua representação, e na elaboração de procedimentos que atuem sobre a topologia do modelo, com o propósito de extrair as relações de conectividade e adjacências entre os elementos da malha triangular.

Entretanto dada a maior complexidade das estruturas de dados utilizadas, este modelo tem a vantagem de ser adaptativo, ou seja, o modelo pode ser adaptado para uma variação de resolução local em áreas mais acidentadas ou irregulares (a malha pode ser mais densa nessas regiões) conseguindo dessa forma representar melhor as características de terrenos mais complexos. Já em regiões com pouca ou nenhuma variação de altura (regiões mais planas), pode ser utilizado um conjunto menor de pontos (a malha irá possuir triângulos maiores), podendo ser eliminados do modelo triângulos e vértices redundantes (triângulos cujo os vértices possuem uma mesma elevação) sem perdas na acurácia ou precisão do modelo (DANOVARO et al., 2007; RASE, 2001). Ainda devido ao fato do modelo ser adaptativo, podem ser gerados modelos de superfície com diferentes níveis de resolução, para por exemplo, aumentar a performance de aplicações de visualização de superfícies em tempo real (RASE, 2001).

A Figura 5, mostra a representação de um terreno (Figura 5a) usando o modelo Grid (Figura 5b) e o modelo **TIN** Grid (Figura 5c). Para ambas as representações foram utilizadas a mesma quantidade de pontos. Entretanto observa-se que o modelo **TIN** concentra um número maior de pontos em regiões mais acidentadas (irregulares) e um número menor de pontos em regiões mais planas, conseguindo dessa forma, representar melhor a superfície do terreno.

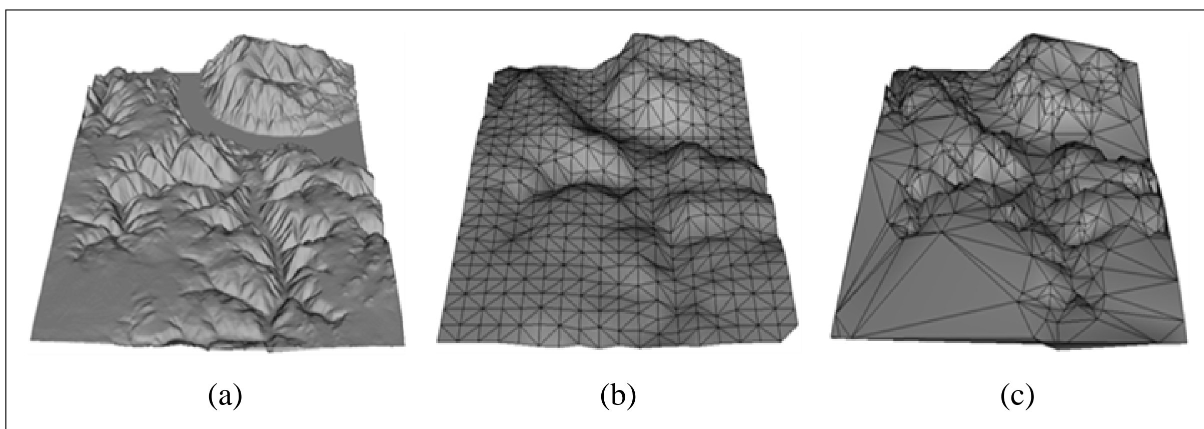


Figura 5: Representação de uma Superfície (a) usando um modelo Grid (b) e um modelo TIN (c).

Além de representar melhor a topografia de uma área da superfície terrestre, o modelo **TIN** consegue realizar um armazenamento mais eficiente dos dados, já que o seu tamanho é proporcional à quantidade de dados (pontos amostrados) usados para representar o terreno, o que não ocorre com o modelo Grid, cujo tamanho é proporcional à área representada.

Como o foco deste trabalho é a representação de **MDTs** utilizando o modelo **TIN**, na Seção 3.2 é feita a descrição de algumas estruturas de dados topológicas que podem ser utilizadas na sua representação.

2.6 Aplicações de Modelos Digitais de Terreno

Uma das vantagens da geração de **MDTs** é devida ao fato de que estes possibilitam a extração de diversas informações de um dado fenômeno que se deseja estudar, sem a necessidade de se ter que trabalhar, ou ir até o local da área modelada (SIMOES, 1993). Fora essa comodidade inicial, um grande número de aplicações necessita de **MDTs** sendo que estes podem ser utilizados por diversas áreas de estudo.

Segundo Sulebak (2000) o conhecimento sobre a topografia de um terreno é de grande importância para a área de Ciências da Terra, sendo essencial na análise e resolução de problemas relacionados à modelagem de processos nas áreas de hidrologia, climatologia, geomorfologia e ecologia. Em particular, de acordo com Barbosa (1999), **MDTs** podem ser utilizados na Cartografia, por exemplo, para a geração de ortofotos, mapas topográficos e temáticos, em **SIG** dentre outros.

Entretanto o uso de **MDTs** não é restrito área de Ciências da Terra, sendo que o mesmo pode ser necessário, e até mesmo um pré-requisito, para muitas aplicações de engenharia civil e militar, e também em áreas industriais como telecomunicação, navegação, transporte e planejamento de infra-estrutura (SULEBAK, 2000).

Desta forma, diversos tipos de profissionais como geólogos, hidrólogos, ecólogos, geomorfologistas, engenheiros civil e de minas, geógrafos, geofísicos, cientistas de solo e clima, dentre outros, podem necessitar em algum momento fazer uso de **MDTs**. Sendo que geralmente os usuários mais comuns de **MDTs** são: empresas de construção civil; administração pública; universidades; gabinetes de engenharia; empresas exploradoras de estradas; mineradoras; indústria petrolífera; agências de meio ambiente e de oceanografia; empresas de consultoria e órgãos de planejamento ambiental.

2.6.1 Aplicações Científicas

O uso de **MDTs** em combinação com diferentes dados espaciais pode resultar em uma importante base de dados para análises topográficas em diferentes áreas como:

SIG: a integração de **MDTs** com os dados raster ou vetoriais de um **SIG** pode permitir a execução de análises topográficas mais avançadas (SULEBAK, 2000);

Sensoriamento Remoto: pode fazer uso de **MDTs** em conjunto com **SIG** na correção ou retirada de informações temáticas de imagens de satélite ou radar, levando em consideração o relevo local e a geometria do sensor, produzindo desta forma produtos geocodificados como por exemplo imagens de radar com correção do relevo ou imagens de satélite geocodificadas (SULEBAK, 2000);

Ecologia: pode fazer uso de **MDTs** no estudo do impacto do relevo sobre as formas de vida que habitam uma dada região (SULEBAK, 2000);

Hidrologia: pode fazer uso de **MDTs**, por exemplo, modelagem hidrológica de bacias hidrográficas, na construção de modelos de rotas de escoamento ou de movimentação de geleiras (SULEBAK, 2000; EL-SHEIMY; VALEO; HABIB, 2005);

Geomorfologia: a existência de **MDTs** associado a uma área pode facilitar a compreensão do seu relevo e dos seus processos construtivos (SULEBAK, 2000). Ainda, **MDTs** podem ser utilizados na geração de modelos da camadas inferiores da superfície terrestre utilizados em mapeamentos geológico e geofísico. Esses modelos podem ser

utilizados, por exemplo, na definição de estratos geológicos específicos e em estudos de simulação de reservatórios de petróleo (BARBOSA, 1999);

Climatologia: o uso de **MDTs** é uma ferramenta importante na investigação da influência do relevo em fenômenos climatológicos como fluxo de temperatura, umidade de ar, regime de chuvas, formação de neblina dentre outros. Por exemplo, os modelos de processo de conversão entre o solo e a atmosfera e de movimentação das camadas inferiores da atmosfera usados na previsão do tempo e modelagem climática necessariamente devem fazer uso de **MDTs** (SULEBAK, 2000);

2.6.2 Aplicações Industriais

Modelos Digitais de Terreno podem ser utilizados pela indústria tanto na pesquisa e desenvolvimento de novos produtos e tecnologias, como na melhoria de serviços e produtos já existentes.

Indústria de Telecomunicação: faz uso de ferramentas de modelagem de redes de multi-transmissão, e de apoio no posicionamento ideal de torres de rádio e antenas de transmissão, de tal forma a se obter a melhor propagação terrestre possível de seus sinais. Geralmente essas ferramentas necessitam da existência de **MDTs** recentes e informações sobre o uso da terra (SULEBAK, 2000);

Indústria Aérea: informações de elevação e uso de terrenos combinadas com bases de dados de aeroportos podem ser usadas, por exemplo, na criação de sistemas de *software* de gestão de vôo, sistemas anti-colisão e sistemas de aviso de proximidade do solo. Essa indústria ainda pode fazer uso de **MDTs** na criação de simuladores de vôo, com cenários realista, para o treino de pilotos e tripulação e na simulação de corredores aéreos, com dados reais, para aeroportos (SULEBAK, 2000);

Indústria Telemática: a confiabilidade de sistemas de navegação por **GPS** usados, por exemplo, em automóveis, depende da utilização de dados precisos e atualizados. Apesar dos usuários dessas aplicações estarem mais interessados em informações de coordenadas planas, dados de elevação (**MDTs**) também podem ser adicionadas nas bases de dados utilizadas por esses sistemas. Essa informação pode ser usada, por exemplo, por transportadoras no planejamento de sua rede de transporte (SULEBAK, 2000);

Indústria de Mineração e Petróleo: dados combinados de **MDTs** e mapas de gravidade, podem ser utilizados por empresas de prospecção, por exemplo, na identificação derrames de petróleo e outros fenômenos em imagens de satélite, e em conjunto com outras informações na identificação de áreas de prospecção. Além de servir como um dado de apoio em atividades de exploração, **MDTs** também podem ser utilizados no monitoramento do impacto dessas atividades, como por exemplo, problemas de afundamento em regiões de mineração (SULEBAK, 2000);

Indústria do Turismo: sistemas de dados de informações turísticas podem fazer uso de informações turística juntamente com mapas, imagens de satélite e ortofotos com modelos de elevação digital (**MDTs**). Essa integração de informações pode permitir aos usuários desses sistemas não somente a extração de informações usuais como infra-estrutura, acomodações, atividades de lazer, horários de abertura de restaurantes etc, mas também opções como visualização, navegação e animações 3D de pontos turísticos de maior interesse (SULEBAK, 2000);

2.6.3 Aplicações Operacionais

Aplicações operacionais estão mais relacionadas à administração pública e serviços governamentais. **MDTs** podem ser usados, por exemplo, no aprimoramento do planejamento e/ou gestão de recursos naturais, proteção ambiental, redução de riscos, agricultura, conservação do solo, assistência em áreas atingidas por calamidade, avaliação de áreas de risco, dentre outros (SULEBAK, 2000).

Plano de manejo florestal: **MDTs** podem ser introduzidos em ferramentas de manejo florestal, de forma que diferentes critérios relacionados à topografia da floresta possam ser considerados no processo de planejamento como: cálculos da variável de declividade e a sua relação com processos de erosão devido a desmatamentos de encostas, cálculo da variável aspecto e os efeitos da iluminação solar sobre o crescimento de uma floresta ou derivação da curvatura do terreno e a sua relação com a umidade do solo (SULEBAK, 2000);

Planejamento de Quebra-Mar: esse tipo de construção é de fundamental importância para a proteção de portos. A topografia do fundo do mar e a morfologia de regiões costeiras, têm efeitos significativos sobre a direção e o tamanho de ondas em águas rasas, desta forma, **MDTs** dessas regiões podem ser utilizados na produção de

simulações computacionais realistas para a localização da melhor região, no sentido de máxima proteção, para a construção de quebra-mares (SULEBAK, 2000);

Predição de Movimento de Massa: deslocamentos de terra, como deslizamentos ou avalanches, possuem um alto custo financeiro na reparação de danos causados em infra-estrutura e construções além, é claro, da possibilidade de perda de vidas. Deslocamentos de terra podem ser causados por uma combinação de várias condições ambientais, de tal forma que o uso de **MDTs** em conjunto com informações geológicas, meteorológicas, de vegetação e solo, pode permitir a construção de modelos de predição de áreas de risco. Informações sobre áreas de risco podem ser usadas, por exemplo, para que seja evitada a construção edificações nessas áreas ou então que sejam tomadas medidas de proteção específica (SULEBAK, 2000; EL-SHEIMY; VALEO; HABIB, 2005);

Predição de áreas com risco de inundação: a modelagem de bacias hidrográficas está atrelada a fatores como tipo do solo, vegetação e dados topográficos. Desta maneira, torna-se necessária a análise de diferentes fontes de dados, como **MDTs** de alta precisão, dados sobre precipitação, retenção de água e capacidade de armazenamento de rios durante sua modelagem. Uma vez criado o modelo hidrológico, podem ser executadas sobre o mesmo simulações como por exemplo, duração e extensão de inundações provocadas por rios, mediante a precipitação de chuva em uma dada região (SULEBAK, 2000; LI; ZHU; GOLD, 2005; EL-SHEIMY; VALEO; HABIB, 2005);

2.6.4 Aplicações Militares

Segundo El-Sheimy, Valeo e Habib (2005) a área militar é uma grande produtora e consumidora de **MDTs**. Muitos os aspectos de ações militares dependem de informações topográficas precisas e confiáveis sendo que compreensão do terreno é de vital importância para ações como a determinação de posição ótima para radares, lançadores de mísseis ou equipamentos de comunicação (BARBOSA, 1999).

Simuladores de Vôo: o treinamento de pilotos é uma tarefa cara, difícil e em alguns casos extremamente perigosa. Dados acurados de **MDTs** podem ser utilizados em simulares de vôo tanto para propósitos de treinamento como em sistemas embarcados nas próprias aeronaves servindo, por exemplo, como guia de aproximação de pistas sem suporte em ações militares (LI; ZHU; GOLD, 2005; SULEBAK, 2000).

Campos Virtuais de Batalha: fornecem uma simulação dinâmica em ambiente estéreo, que pode ser usada para recapitular batalhas, avaliar resultados, ou em treinamento e ganho de experiência. Esses simulares podem utilizar **MDTs** na extração de informações topográficas como intervisibilidade, formações defensivas do relevo e acessibilidade do campo de batalha (LI; ZHU; GOLD, 2005)

3 Malhas Triangulares Irregulares (TIN)

Triangulações possuem uma grande importância na resolução de problemas da área de Geometria Computacional, por exemplo, algumas técnicas de busca assumem divisões planares na forma de triangulações; algoritmos de intersecção de poliedros geralmente exigem que a superfície poliedral seja pré-triangularizada. Ainda, triangulações planares podem ser utilizadas em aplicações que envolvam interpolação de superfície tanto para propósito de visualização gráfica, como para cálculos na análise numérica (PREPARATA; SHAMOS, 1998).

Formalmente uma triangulação $T(S)$ de um conjunto finito de pontos S num plano qualquer ($S \subset R^2$), pode ser definida como um conjunto de triângulos cujos vértices são pontos de S e cujas arestas são formadas por pares de pontos em S, sendo que cada ponto pertencente a S deve ocorrer em pelo menos um triângulo, e a intersecção das arestas é permitida apenas nos vértices da triangulação (SCHNEIDER; EBERLY, 2003).

Ainda segundo Hjelle e Daehlen (2006), além da sua importância na Geometria Computacional, triangulações possuem aplicações em diversas outras áreas como:

- **SIG:** modelagem digital de terrenos e relações entre objetos geográficos;
- **Indústria de exploração de Gás e Petróleo:** usada para representar superfícies de separação entre diferentes estruturas geológicas e também são usadas para representar propriedades dessas estruturas;
- **Sistemas CAD (Computer Aided Design):** triangulações são muito utilizadas pela indústria de manufatura, principalmente a industria automotiva;
- **Engenharia:** utilizada por campos da engenharia focados na simulação de fenômenos físico que fazem uso de uso de **MEF** (**M**étodos de **E**lementos **F**initos);

- **Sistemas de Visualização e Computação Gráfica:** utilizada na interpolação e visualização de superfícies;

Em particular no contexto desse trabalho é dado foco no uso de triangulações na produção de **MDT**, entretanto, como é ilustrado pela Figura 6, uma triangulação não é única, existindo diferentes triangulações para um mesmo conjunto de pontos amostrados sobre a superfície de um terreno.

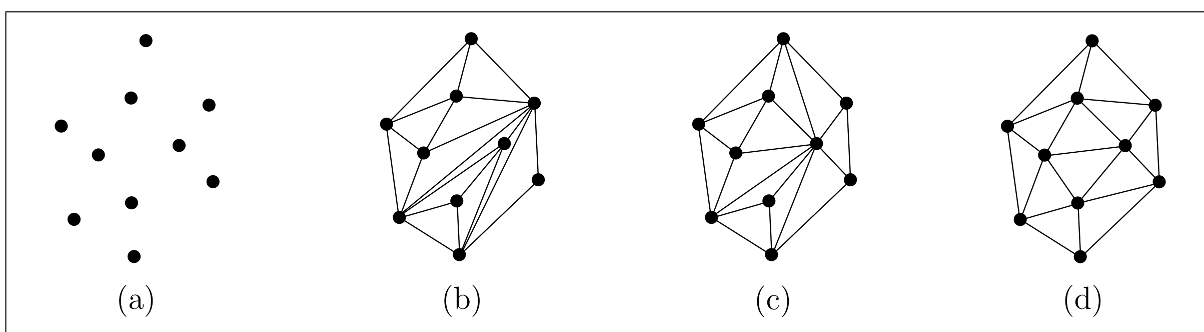


Figura 6: Possíveis triangulações para um mesmo conjunto de pontos.

Surge então a questão de gerar uma triangulação, a partir de um conjunto de pontos amostrados sobre um terreno, de maneira que ela mais se aproxime da superfície do terreno. Desta maneira, torna-se necessário a adoção de algum critério para a criação da rede de triângulos.

Berg et al. (2008) fornece um exemplo interessante de como diferentes triangulações podem influenciar no resultado final da modelagem digital de uma superfície topográfica. A Figura 7 mostra duas triangulações formadas a partir do mesmo conjunto de pontos. Analisando o espaçamento e a altura associadas a cada ponto pode-se supor que esses pontos representam uma cadeia de montanhas. A triangulação mostrada na Figura 7a reflete bem essa suposição, entretanto, observando a triangulação mostrada na Figura 7b vê-se algo como se fosse um vale cortando essa cadeia montanha, indo contra a suposição inicial. Percebe-se que toda essa mudança na superfície modelada foi causada por uma única troca de aresta na triangulação da Figura 7b.

O problema com a triangulação mostrada na Figura 7b, é devido ao fato da altura no ponto p ser determinada por dois pontos distantes do mesmo, isso por que p está sobre uma aresta compartilhada por dois triângulos muito alongados e finos, ou seja: o formato desses dois triângulos é a causa do problema. Desta forma, a triangulação que contêm ângulos muitos pequenos, não é a triangulação ideal. Neste caso a triangulação ideal é aquela que contêm triângulos o mais equilátero possível, ou seja, a triangulação

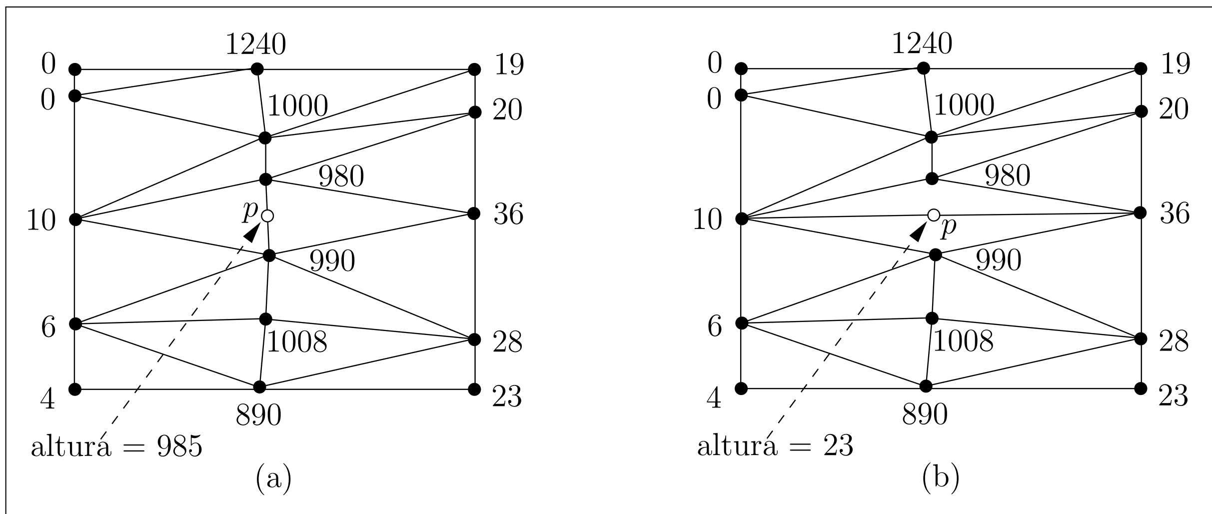


Figura 7: Influência da triangulação na superfície gerada: (a) altura do ponto p sobre arestas conectando vértices próximos reflete realidade da superfície; (b) altura do ponto p sobre arestas conectando vértices distantes não reflete realidade da superfície.
Fonte: adaptado de Berg et al. (2008)

mostrada na Figura 7a. Sendo assim dependendo da aplicação pode-se otimizar uma triangulação, como por exemplo no caso acima, para evitar triângulos muito alongados ou quase degenerados (triângulos cujos vértices são quase colineares).

Muitos critérios podem ser utilizados na definição de uma triangulação ideal sendo que particularmente uma triangulação ótima para propósitos de representação digital de superfícies deve possuir as seguintes características:

- Para um dado conjunto de pontos, a triangulação resultante deve ser única (se o mesmo algoritmo for usado), não importando a ordem com que os pontos são usados na construção da rede de triângulos (HJELLE; DAEHLEN, 2006);
- Deve ser gerada uma rede com triângulos ótimos, ou seja, os triângulos gerados devem ser o mais próximo possível do triângulo equilátero (BERG et al., 2008), isto porque triângulos não equiláteros, acarretam problemas de interpolação em regiões que mudam rapidamente de inclinação (BARBOSA, 1999). Ainda, triângulos muito alongados (quase degenerados) podem causar erros de arredondamento em implementações de funções de interpolação (PREPARATA; SHAMOS, 1998);
- Cada triângulo deve ser formado usando pontos próximos entre si, de tal forma que a soma das três arestas que formam o triângulo dê um valor baixo (LI; ZHU; GOLD, 2005);

3.1 Triangulação de Delaunay

Uma solução para o problema das várias triangulações para um mesmo conjunto de pontos, assim como a geração de uma triangulação com triângulos ótimos, é a utilização da técnica de Triangulação de Delaunay. Segundo Berg et al. (2008), a Triangulação de Delaunay é ideal para a representação da superfície de um terreno a partir de pontos amostrados sobre o mesmo. De fato, dentre todas as possíveis formas de se gerar uma rede de triângulos a partir de pontos irregularmente espaçados, o método de Delaunay é o mais utilizado (LI; ZHU; GOLD, 2005).

Esse tipo particular de triangulação tem sido extensivamente estudada, possuindo muitas características interessantes e uma base matemática teórica sólida. Além disso, essa triangulação é fácil de se computar ao contrário de outras triangulações que apesar de terem uma base teórica sólida são de difícil computação (HJELLE; DAEHLEN, 2006).

A Triangulação de Delaunay é ideal para propósito de representação de superfícies topográficas por possuir as seguintes características:

- os triângulos gerados são os mais equiláteros possíveis (BARBOSA, 1999; PITERI et al., 2007);
- os lados dos triângulos gerados são os menores possíveis (lados grandes ou desproporcionais implicam em triângulos pouco equiláteros), ou seja, as arestas da triangulação são construídas usando pontos o mais próximos entre sí (BARBOSA, 1999);
- a triangulação é única, caso não exitam mais que três pontos co-circulares (PITERI et al., 2007);

Segundo Hjelle e Daehlen (2006) existem três definições diferentes para uma Triangulação de Delaunay:

Definição 1 *Uma triangulação é tida como Triangulação de Delaunay se considerada ótima no Critério MaxMin e se for definida sobre o fecho convexo do conjunto de pontos.*

Definição 2 *A Triangulação de Delaunay é o dual do Diagrama de Voronoi.*

Definição 3 *Uma Triangulação de Delaunay $TD(S)$ de um conjunto de pontos S sobre um plano, é uma triangulação em que o interior do circuncírculo formado por qualquer*

triângulo de $TD(S)$ não contêm nenhum ponto de S (Critério da Circunferência Circunscrita Vazia).

Sendo que, segundo Piteri et al. (2007), no plano, as Definições 1 e 3 são equivalentes. Nas Seções a seguir, é dada uma breve descrição dos critérios utilizados por cada uma dessas definições.

3.1.1 Critério MaxMin

Pelo Critério MaxMin, para cada possível triangulação $T^k(S)$ de um conjunto de pontos S é associado um vetor $I\left(T^k\right) = (\alpha_1, \alpha_2, \cdots, \alpha_n)$, onde n é o número de triângulos em T^k e cada entrada α_i representa o menor ângulo interno de cada um dos triângulos pertencentes a T^k, sendo ainda que cada vetor $I\left(T^k\right)$ tem as suas entradas ordenadas de forma crescente,

$$I\left(T^k\right) = (\alpha_1, \alpha_2, \cdots, \alpha_n), \ \alpha_i \leq \alpha_j, \ 1 \leq i, \ j \leq n.$$

A ordenação de todos os vetores $I\left(T^k\right)$ impõe uma ordenação linear sobre o conjunto de todas as possíveis triangulações de S, desta maneira se um vetor $I\left(T^i\right)$ for lexicograficamente maior que um vetor $I\left(T^j\right)$, ou seja, se $I\left(T^i\right) > I\left(T^j\right)$ então a triangulação T^i é "melhor" que a triangulação T^j. Hjelle e Daehlen (2006) fornece como exemplo os vetores:

$$I\left(T^a\right) = (14.04^\circ, 26.57^\circ, 36.87^\circ, 40.60^\circ, 40.60^\circ, 49.40^\circ, 50.91^\circ)$$
$$I\left(T^b\right) = (14.04^\circ, 15.26^\circ, 19.44^\circ, 26.57^\circ, 29.74^\circ, 36.87^\circ, 45.00^\circ)$$

das triangulações mostradas na Figura 8a e 8b. Percebe-se que a primeira entrada de ambos vetores são iguais, entretanto a segunda entrada em $I\left(T^a\right)$ é lexicograficamente maior que a segunda entrada em $I\left(T^b\right)$, sendo assim a triangulação mostrada na Figura 8a é "mais ótima" que a triangulação mostrada na Figura 8b.

Desta forma, uma triangulação T^k de um conjunto de pontos S é tida como ótima pelo critério MaxMin se o seu vetor $I\left(T^k\right)$ for lexicograficamente o maior vetor dentre todos os vetores de todas possíveis triangulações de S, ou seja, ao encontrar-se o menor ângulo em cada triangulação dentre todas as possíveis triangulações, a triangulação ótima será aquela em que esse ângulo for máximo.

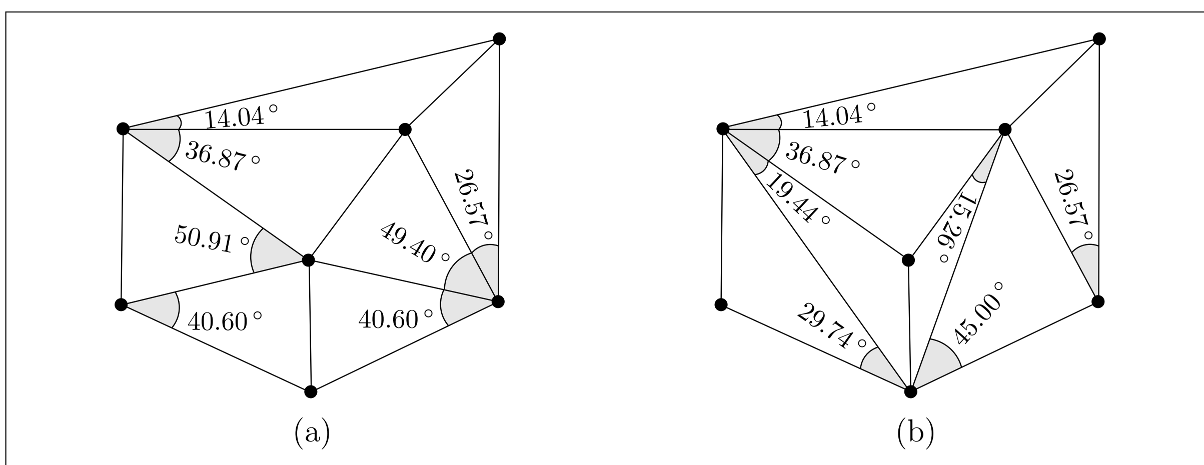


Figura 8: Duas triangulações e seus menores ângulos. Triangulação (a) possui o seu menor ângulo maximizado em relação a triangulação (b).

Fonte: adaptado de Hjelle e Daehlen (2006)

3.1.2 Diagramas de Voronoi

O Diagrama de Voronoi no plano é uma estrutura geométrica, topológica e combinatória que representa informações de proximidade, por exemplo, de um conjunto de pontos. Ou seja, dado um conjunto de pontos, o Diagrama de Voronoi representa as regiões que estão mais próximas de um ponto em particular, que de quaisquer outros pontos (EL-SHEIMY; VALEO; HABIB, 2005).

A Figura 9 apresenta um exemplo de um conjunto de pontos no plano e do seu Diagrama de Voronoi associado.

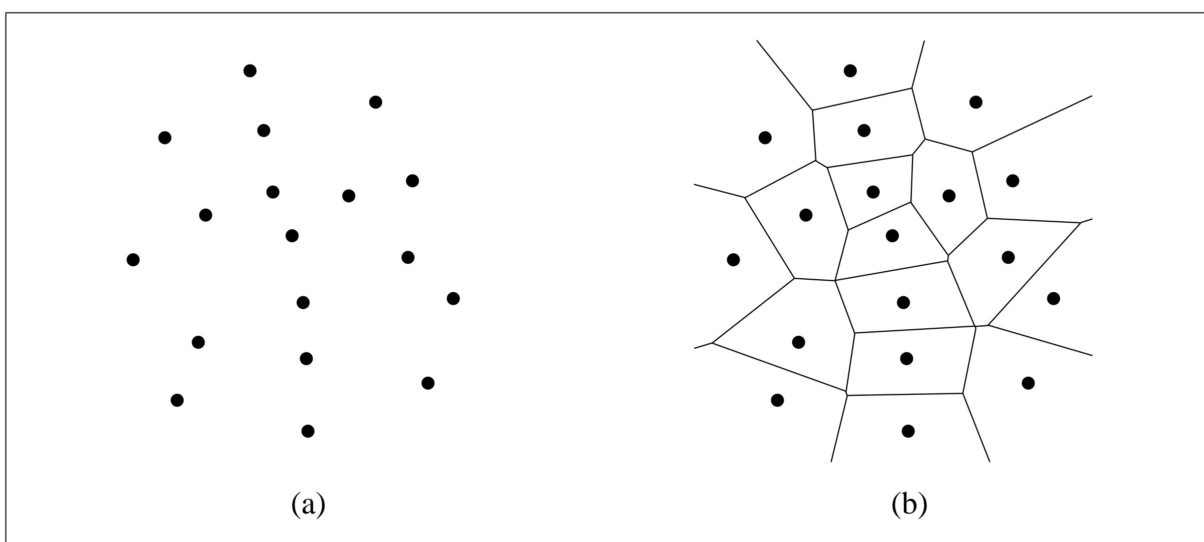


Figura 9: Diagrama de Voronoi. (a) Conjunto de pontos no plano; (b) Diagrama de Voronoi associado.

Segundo (PITERI et al., 2007) a Triangulação de Delaunay está estreitamente relacionada com o Diagrama de Voronoi na medida que utilizando um Diagrama de Voronoi como base pode-se gerar uma Triangulação de Delaunay, ou seja, a Triangulação de Delaunay é o dual do Diagrama de Voronoi de tal forma que:

- um ponto do Diagrama de Voronoi corresponde a um triângulo da Triangulação de Delaunay;
- um segmento do Diagrama de Voronoi corresponde a uma aresta da Triangulação de Delaunay;
- uma região do Diagrama de Voronoi corresponde a um vértice da Triangulação de Delaunay;

Na Figura 10 é mostrado o relacionamento entre um Diagrama de Voronoi e uma Triangulação de Delaunay. As linhas sólidas representam as arestas da Triangulação de Delaunay enquanto as linhas hachuradas são as segmentos do Diagrama de Voronoi. Os pontos preenchidos são os vértices da Triangulação de Delaunay enquanto os vazios são pontos do Diagrama de Voronoi. Por fim, a área hachurada mais clara representa uma região do Diagrama de Voronoi enquanto a área hachurada mais escura é um triângulo da Triangulação de Delaunay.

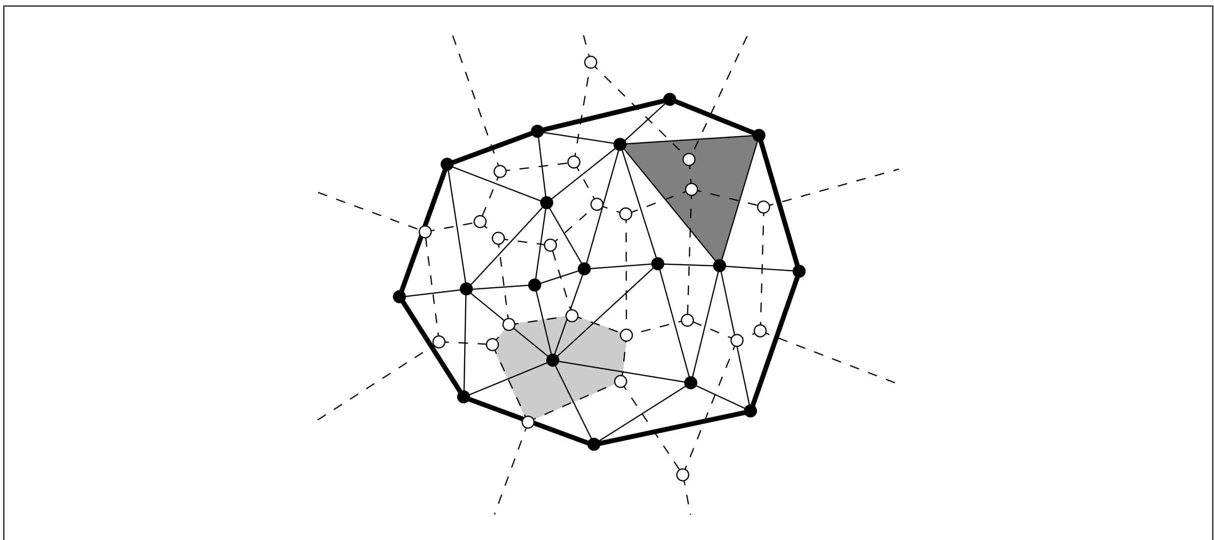


Figura 10: Relacionamento entre a Triangulação de Delaunay e o Diagrama de Voronoi.

Segundo El-Sheimy, Valeo e Habib (2005) a Triangulação de Delaunay possui algumas propriedades interessantes como consequência da estrutura do Diagrama de Voronoi:

Fecho Convexo: a face externa da Triangulação de Delaunay é o fecho convexo do conjunto de pontos.

Propriedade Circuncírculo: o circuncírculo de qualquer triângulo da Triangulação de Delaunay é vazio, ou seja, não contêm nenhum outro ponto da triangulação (veja Seção 3.1.3).

Propriedade Círculo Vazio: dois pontos p_i e p_j são conectados por uma aresta da Triangulação de Delaunay, se e somente se, existir um círculo vazio (não contêm nenhum outro ponto da triangulação) passando por p_i e p_j.

Propriedade Par mais Próximo: numa Triangulação de Delaunay $TD(S)$ de um conjunto de pontos S, o par de pontos mais próximos (p_i e p_j) forma uma aresta em $TD(S)$.

3.1.3 Critério da Circunferência Circunscrita Vazia

O Critério da Circunferência Circunscrita define que para uma Triangulação de Delaunay (caso planar) nenhum dos vértices da triangulação pode ser interior às circunferências circunscritas a qualquer um dos seus triângulos (PREPARATA; SHAMOS, 1998).

Na Figura 11, observa-se três triangulações para um mesmo conjunto de pontos. Percebe-se na triangulação mostrada na Figura 11a que nenhuma das circunferências circunscrita aos triângulos contêm algum ponto da malha, ou seja, trata-se de uma Triangulação de Delaunay. Já nas Figura 11b e Figura 11c isso não acontece, ou seja, essas triangulações não satisfazem o critério da circunferência circunscrita vazia, sendo assim essas triangulações não são de Delaunay.

Geralmente esse critério é utilizado na implementação computacional da Triangulação de Delaunay, isto porque, ele é mais simples do ponto de vista de implementação computacional do que o critério MaxMin (não é preciso gerar todas possíveis triangulações para um conjunto de pontos), e também por ser um método direto (não é preciso primeiramente gerar o Diagrama de Voronoi para só então derivar a Triangulação de Delaunay deste) (HJELLE; DAEHLEN, 2006).

Na Seção a seguir, é mostrado como esse critério pode ser utilizado na geração de uma Triangulação de Delaunay.

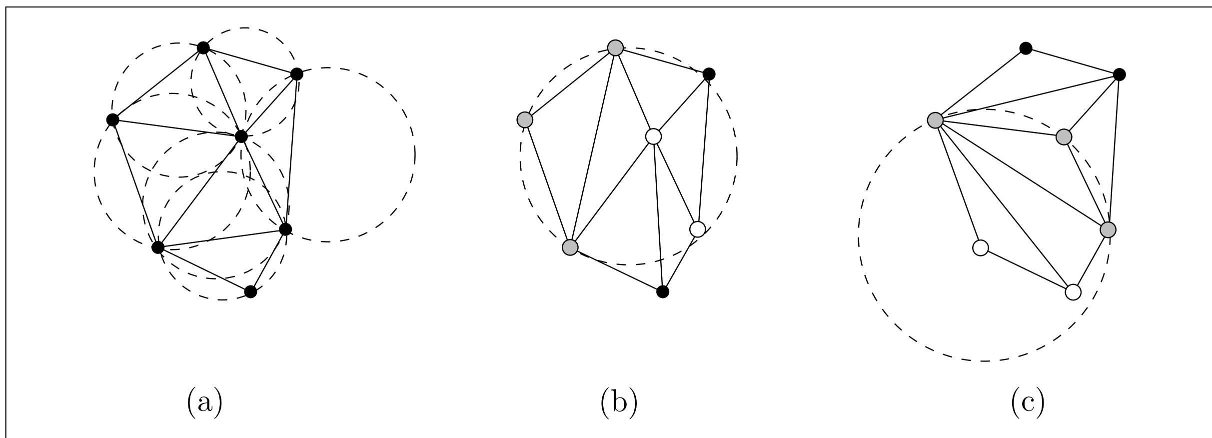


Figura 11: Critério da Circunferência Circunscrita Vazia. (a) Triangulação que satisfaz o critério da circunferência circunscrita vazia; (b) e (c) Triangulações que não satisfazem o critério da circunferência circunscrita vazia.
Fonte: adaptado de Piteri et al. (2007)

3.1.4 Princípio de ***Flip-Edge***

Dada uma triangulação $T(S)$ de um conjunto de pontos $S \subseteq R^2$, se toda aresta $e \in T(S)$ for localmente de Delaunay, então $T(S)$ é uma Triangulação de Delaunay. Conforme Piteri et al. (2007), uma aresta é localmente de Delaunay se:

- ela pertencer ao fecho convexo da triangulação, ou,
- se ela for adjacente aos triângulos $\Delta\,(p_i p_v p_k)$ e $\Delta\,(p_i p_v p_t)$, e o ponto p_t se localiza no exterior da circunferência que circunscreve o $\Delta\,(p_i p_v p_k)$(Figura 12c).

Segundo o princípio de *flip-edge* e conforme é mostrado na Figura 12, se a aresta $p_k p_t$ adjacente aos triângulos $\Delta\,(p_k p_t pv)$ e $\Delta\,(p_k p_i p_t)$ não for localmente de Delanay (Figura 12a), e sua remoção resultar num quadrilátero estritamente convexo (Figura 12b), então, é possível trocá-la pela aresta, $p_i p_v$ localmente de Delaunay (Figura 12c).

O princípio de *flip-edge* pode ser utilizado tanto na geração direta da Triangulação de Delaunay como na conversão de uma triangulação arbitrária em uma Triangulação de Delaunay. Sendo que, além da sua simplicidade (pode ser realizado usando o critério da circunferência circunscrita vazia) este também tem por característica ser local, isto é, ao aplicá-lo em torno do ponto mais recentemente inserido, as trocas de arestas não se propagam por toda a triangulação gerando poucas mudanças, bem localizadas, na topologia da triangulação (PITERI et al., 2007).

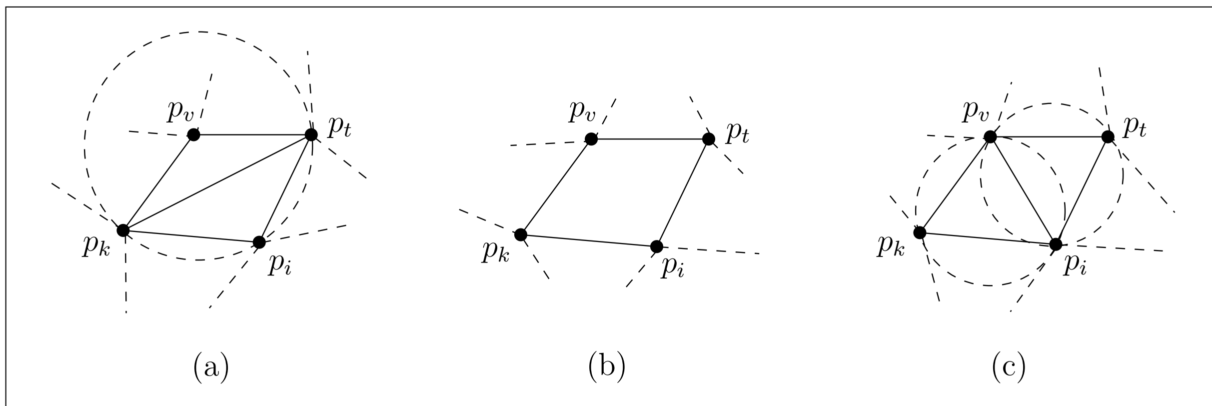


Figura 12: Princípio de *Flip-Edge*. (a) O triângulo $\Delta(p_k p_i p_t)$ não satisfaz o critério da circunferência circunscrita vazia; (b) A remoção da aresta $p_k p_t$ gera um quadrilátero estritamente convexo; (c) Troca da aresta $p_k p_t$ pela aresta $p_i p_v$ localmente de Delaunay . **Fonte**: adaptado de Piteri et al. (2007)

3.2 Estrutura de Dados para o Modelo TIN

Conforme pode ser observado na Figura 13, um **TIN** nada mais é do que uma subdivisão espacial simples composta pelas entidades vértices (v), arestas (e) e faces (f).

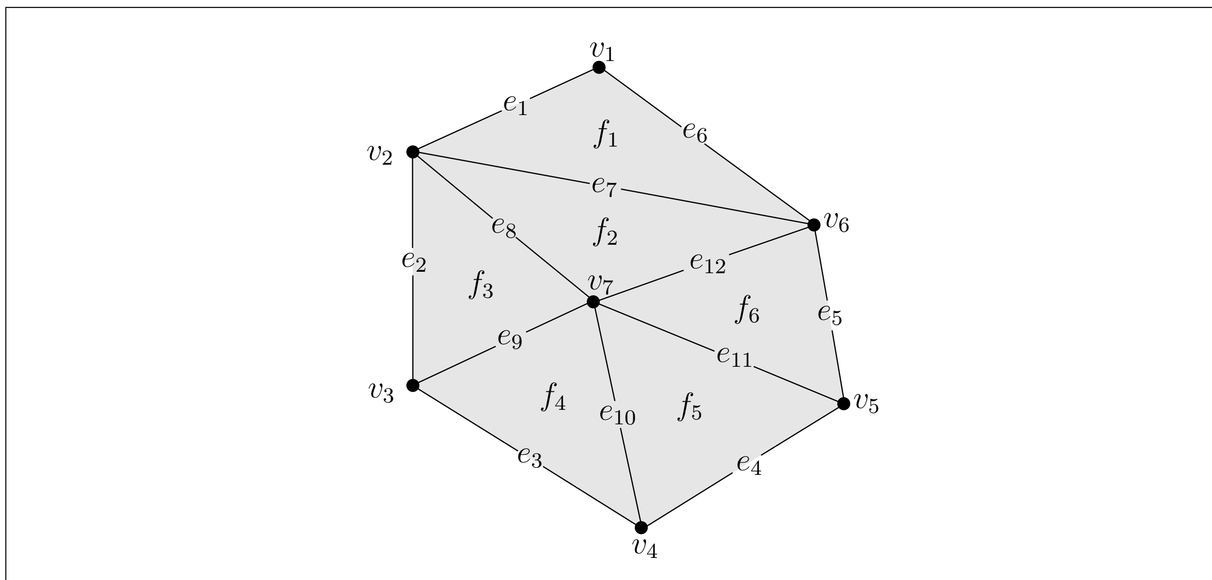


Figura 13: Entidades Topológicas de um TIN

Segundo Varela (2004), um **TIN** pode ser facilmente representado por uma lista de faces e uma lista de vértices, onde cada face é descrita como um conjunto de três ponteiros para os elementos da lista de vértices. A Figura 14 apresenta um exemplo dessa forma de representação para a triangulação mostrada na Figura 13.

Embora esse modelo de representação possua uma fácil compreensão e implementação computacional, ele é ineficiente para operações sobre o **TIN** que requeiram consultas sobre

Lista de Vértices	
Vértices	**Coordenadas**
v_1	x_1, y_1, z_1
v_2	x_2, y_2, z_2
v_3	x_3, y_3, z_3
v_4	x_4, y_4, z_4
v_5	x_5, y_5, z_5
v_6	x_6, y_6, z_6
v_7	x_7, y_7, z_7

Lista de Faces	
Face	**Vértices**
f_1	v_1, v_2, v_6
f_2	v_2, v_7, v_6
f_3	v_2, v_3, v_7
f_4	v_3, v_4, v_7
f_5	v_4, v_5, v_7
f_6	v_5, v_6, v_7

Figura 14: Exemplo de representação de um TIN

relações de conectividade (adjacência e incidência) entre os seus componentes (vértices, aresta e faces), como por exemplo, encontrar todas as faces ou arestas incidentes a um dado vértice. Isso porque, apesar da possibilidade de se fazer buscas sobre um **TIN** representado por esse modelo, esta é muito custosa, requerendo uma completa varredura sobre toda a lista de faces ou vértices (VARELA, 2004).

Desta forma, a estrutura de dados utilizada para representar o modelo **TIN** idealmente deve se preocupar com a manutenção tanto da geometria como dos relacionamentos entre as suas entidades vértices, aresta e faces, isto é, com a topologia da triangulação.

3.2.1 Estrutura de Dados Topológica

Cada entidade vértice, aresta e face do modelo **TIN** tem associado informações de natureza geométrica e topológica. As informações de natureza geométrica podem conter dados como:

- coordenadas (x, y) e valores de elevação z associados aos seus vértices;
- equações de curvas associadas às suas arestas;
- equações do plano, polinômios ou vetores normais associados às suas faces;

Já as informações de natureza topológicas representam dados do modelo que se mantém invariantes com o uso de transformações geométricas (ex: rotações, translações e escalas), ou seja, dados sobre as relações de conectividade, como relações de adjacência e incidência, entre as entidades topológicas vértices arestas e faces do modelo. Particularmente, podem ser retiradas do modelo **TIN** nove relações de conectividade entre os seus componentes:

- **Adjacência Vértice-Vértice (VV):** dado um vértice retorna todos os vértices adjacentes a ele, sendo que, dois vértices são adjacentes se estiverem conectados por uma aresta;

- **Incidência Vértice-Aresta (VE):** dado um vértice retorna todas as arestas incidentes a ele, sendo que, uma aresta é incidente a um vértice se o mesmo for um de seus pontos extremos;

- **Incidência Vértice-Face (VF):** dado um vértice retorna todas as faces incidentes a ele, sendo que, uma face é considerada incidente a um vértice se este estiver contido nela;

- **Incidência Aresta-Vértice (EV):** dada uma aresta retorna os dois vértices incidentes a ela, sendo que, os dois vértices incidentes a ela são o seus pontos extremo;

- **Adjacência Aresta-Aresta (EE):** dada uma aresta retorna as arestas adjacentes a ela, sendo que duas arestas são adjacentes se estas possuem uma face e um vértice incidente em comum;

- **Incidência Aresta-Face (EF):** dada uma aresta retorna as suas duas faces incidentes, sendo que, uma face é incidente a uma aresta se a aresta em questão for uma das arestas que a define;

- **Incidência Face-Vértice (FV):** dada uma face retorna todos os vértices incidentes a ela, ou seja, retorna todos os vértices que compõe a face;

- **Incidência Face-Aresta (FE):** dada uma face retorna todas as arestas incidentes a ela, ou seja, retorna todas as aresta que compõe a face;

- **Adjacência Face-Face (FF):** dada uma face retorna todas as faces adjacentes a ela, sendo que duas faces são adjacentes se possuírem uma aresta em comum;

A Figura 15 apresenta graficamente essas nove relações de conectividade que podem ser obtidas entre as entidades vértices, arestas e faces do modelo **TIN**.

Desta maneira, uma estrutura de dados é dita "topológica" se esta for capaz de conter de forma conveniente, tanto informações de natureza geométrica como topológica do modelo que ela representa. Ainda, uma **EDT** é dita "completa" se ela for capaz de prover as relações de adjacências entre todas as entidades topológicas definidas (no caso do modelo **TIN** as nove relações de adjacência entre as entidades vértice, aresta e face) em tempo

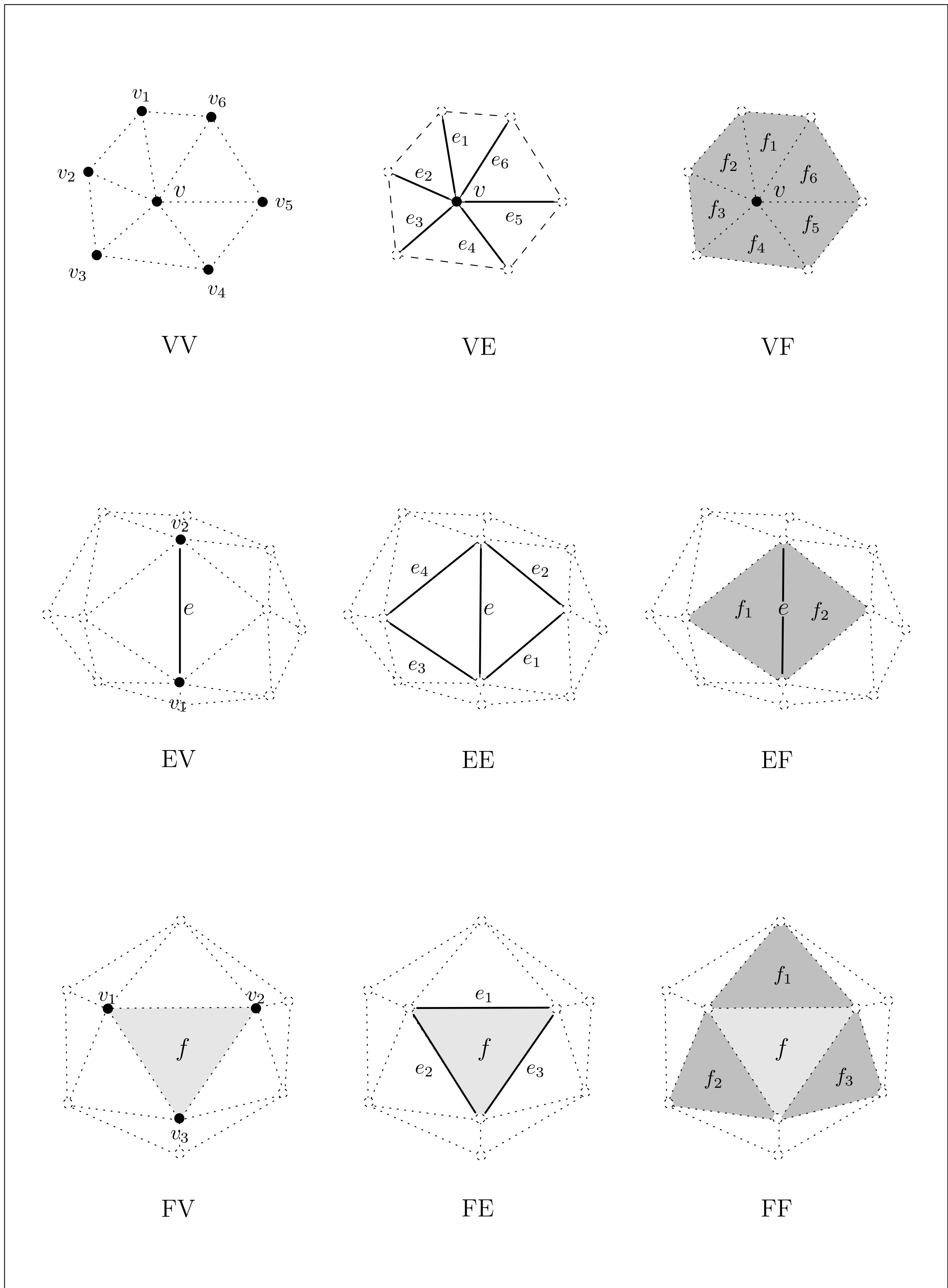


Figura 15: Relações de conectividade entre as entidades topológicas vértice, aresta e faces de um TIN.

ótimo, isto é, em tempo constante ou linearmente proporcional ao número de entidades retornadas.

Segundo Hjelle e Daehlen (2006) e Al-Salami (2009) várias **EDTs** podem ser utilizadas na representação computacional de um **TIN** sendo que a sua escolha deve levar em conta fatores como:

- **Para quais propósitos o TIN será utilizado:** por exemplo, em aplicações de visualização deve ser utilizada uma **EDT** de rápido acesso a dados, ou seja, uma **EDT** que possua informação topológica o suficiente para a rápida extração de sequências de triângulos para visualização.
- **Quais operações serão suportadas pelo TIN:** se serão feitas operações de busca, união, remoção de vértices etc.
- **Em qual ambiente ou plataforma o TIN será utilizado:** por exemplo, se o **TIN** será implementado para ser executado na memória principal (memória RAM) de um computador ou se será implementado em um sistema de banco dados.

A seguir é apresentada uma discussão sobre algumas **EDTs** que podem ser utilizadas na representação computacional de **TIN**.

3.2.2 A Estrutura *Winged-Edge* Modificada

Segundo Piteri (1999) a estrutura de dados *Winged-Edge* (BAUMGART, 1975) é considerada a gênese das **EDT**. Como é mostrado na Figura 16, nessa estrutura a maior parte das informações topológicas são organizadas em torno da entidade aresta, a qual tem conhecimento dos seus vértices, arestas e faces adjacentes, dai o fato dela ser referenciada como uma **EDT** baseada em arestas.

A escolha da aresta como representante das informações topológicas é devida ao fato de que em uma subdivisão planar arbitrária, o número de aresta e faces incidentes a um vértice, assim como o número de vértices e arestas associadas uma face ser variável, enquanto que o número de vértices e faces incidentes em uma aresta é constante, sendo essa uma característica de modelos poliedrais *manifold* (PITERI, 1999; GOIS; PITERI, 2001).

Entretanto a *Winged-Edge* não possui uma representação direta para a entidade face com mais de uma componente conexa (faces com buracos em seu interior). Essa representação só foi possível com a introdução da entidade ciclo, a partir dai a *Winged-Edge* passou a ser referenciada como *Winged-Edge Modificada*.

Na Figura 16a pode ser vista uma representação da estrutura *Winged-Edge Modificada*, onde cada aresta *we* contêm os ponteiros *vt1* e *vt2* para os seus dois vértices delimitadores, os ponteiros *fc1* e *fc2* para as duas faces que incidem sobre ela e finalmente os ponteiros *pcw*, *ncw*, *pccw* e *nccw* para as quatro arestas que incidem sobre os seus vértices delimitadores, quando ela é percorrida no sentido anti-horário e horário. Essa forma de estruturação permite que se encontre as faces, arestas ou vértices ligados a uma aresta em tempo constante, isto é, todas as relações de conectividade de uma aresta são obtidas em tempo constante, sendo que as relações de conectividade entre as entidades vértices e faces são obtidas em tempo linearmente proporcional ao número de entidades envolvidas (VARELA, 2004).

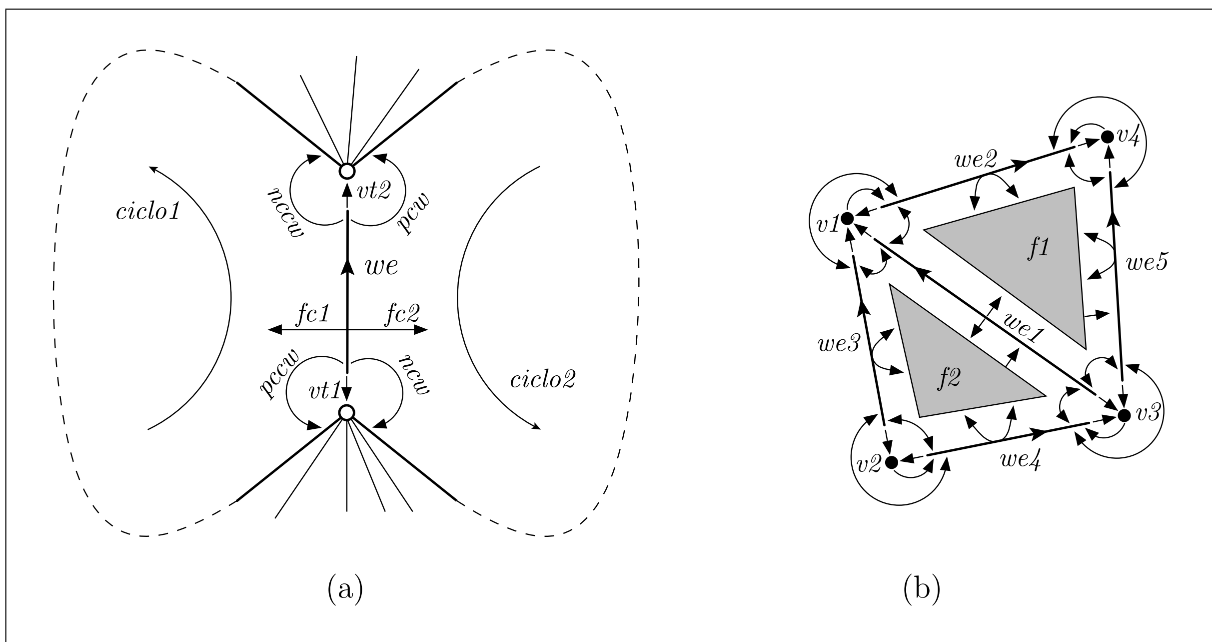


Figura 16: (a) Estrutura de Dados *Winged-Edge Modificada*; (b) Representação interna de uma triangulação utilizando a Estrutura de Dados *Winged-Edge Modificada*

Na Figura 16b é mostrada a representação interna de uma triangulação utilizando-se a *Winged-Edge Modificada*. Como pode ser observado nesta figura, além da informação contida em cada aresta, cada vértice (v) e face (f) também fornece acesso (por meio de ponteiros) para uma de suas arestas (we) incidentes, isto porque, é a partir desta aresta que será retirada toda a informação topológica necessárias para a obtenção das suas relações de adjacência. De fato, de acordo com Piteri (1999), ao se utilizar a *Winged-Edge Modificada* na representação de subdivisões planares, as nove relações de adjacência mostradas na Figura 15 podem ser obtidas com o auxílio de apenas duas primitivas topológicas:

Primitiva *ccw_nev(v, we, nwe)*: dado um vértice v e uma de suas arestas incidentes *we*, essa primitiva retorna a aresta *nwe*, que é a próxima aresta incidente a v depois de *we* no sentido anti-horário.

Primitiva *ccw_nel(l, we, nwe)*: dado um ciclo l (representa uma face propriamente dita no caso de faces com apenas uma componente conexa) e uma de suas arestas incidentes *we*, retorna a aresta *nwe* que é a aresta seguinte a *we* quando se percorre o ciclo(face) l no sentido anti-horário.

É interessante notar que, uma vez encontrada a próxima aresta incidente a um vértice ou a uma face , por meio das primitivas *ccw_nev(.)* e *ccw_nel(.)*, pode-se facilmente encontrar as demais relações de adjacência dessas entidades. Por exemplo, pode-se encontrar todos os vértices adjacentes a um dado vértice v seguindo os seguintes passos:

- **Passo1:** pegar a aresta *we* incidente a v (lembrando que como é mostrado na Figura 16b, todo vértice dá acesso a uma de suas arestas incidentes);
- **Passo2:** se o vértice apontado pelo ponteiro *vt1* de *we* for igual a v então o vértice apontado pelo ponteiro *vt2* de *we* será o vértice adjacente a v; já se o vértice apontado por *vt2* de *we* for igual a v então o vértice apontado por *vt1* de *we* será o vértice adjacente a v;
- **Passo3:** encontrar a próxima aresta *we* adjacente a v usando a primitiva *ccw_nev(...)* e repetir o passo 2 até terminar de percorrer todo o ciclo de vértices incidentes;

Também é possível notar pela Figura 16b que os ponteiros *fc1* e *fc2* de todas as arestas pertencentes à fronteira da triangulação (*e2*, *e3*, *e4* e *e5*) compartilham uma mesma face. Essa característica pode ser utilizada para descobrir implicitamente se uma aresta da triangulação pertence à sua fronteira ou não.

3.2.3 A Estrutura *Half-Edge*

A estrutura *Half-Edge* (WEILER, 1985) pode ser considerada como uma variação da *Winged-Edge* (VARELA, 2004), também tendo a aresta como entidade responsável pela manutenção das informações topológicas (BOISSONNAT et al., 2002). Entretanto, como pode ser observado na Figura 17, na *Half-Edge* cada aresta é representada por duas semi-arestas de sentidos opostos, sendo que, cada semi-aresta *he* possui uma orientação definida sobre os seus vértices delimitadores, sendo um vértice de origem e um vértice de destino.

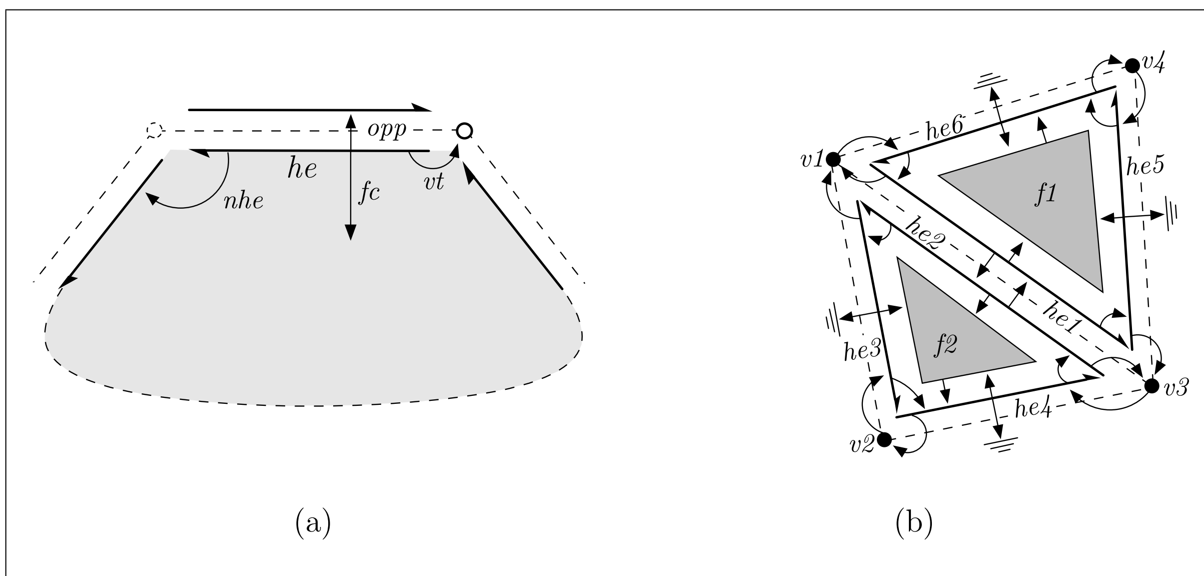


Figura 17: (a) Estrutura de Dados *Half-Edge*; (b) Representação interna de uma triangulação utilizando a Estrutura de Dados *Half-Edge*

Segundo Boissonnat et al. (2002), e como é ilustrado na Figura 17a, cada semi-aresta *he* pode ser implementada minimamente, no sentido de demandar pouco espaço de memória computacional conseguindo ainda manter informações suficientes para operações topológicas, usando-se apenas quatro ponteiros:

- um ponteiro *vt* para o seu vértice origem;
- um ponteiro *nhe* para a próxima semi-aresta no sentido anti-horário, pertencente à mesma face;
- um ponteiro *opp* para a semi-aresta de sentido oposto, que compartilha mesma aresta (*twin-edge*);
- um ponteiro *fc* para a face à sua esquerda (face a qual a semi-aresta pertence).

Ainda, conforme é ilustrado na Figura 17b, geralmente em implementações de **EDTs** que utilizam a *Half-Edge*, as entidades faces e vértices possuem um ponteiro para uma de suas semi-aresta incidente e as semi-arestas que fazem parte da fronteira da triangulação (*he3*, *he4*, *he5* e *he6*) não tem associadas a sí uma semi-aresta oposta (é associado um valor nulo ao ponteiro *opp*). Lembrando ainda que, toda informação adicionada à entidade aresta, como por exemplo equações de curva associadas, é duplicada em cada uma das duas semi-arestas que a compõe.

Existem implementações da *Half-Edge* em algumas bibliotecas como por exemplo a **TTL** (**T**emplate **T**riangulation **L**ibray) e a **CGAL**.

3.2.4 Estrutura TDS (Triangulation Data Struct)

A **TDS** é uma **EDT** que foi desenvolvida para dar suporte ao pacote de *software* de triangulação bidimensional da biblioteca **CGAL**.

Ao contrário da *Winged-Edge*, *Winged-Edge Modificada* e *Half-Edge* que são **EDTs** baseadas em arestas e que podem ser utilizadas para representar subdivisões planares arbitrárias, a **TDS** é uma estrutura baseada em faces e vértices, concebida especificamente para a representação de triangulações, ou seja, subdivisões planares onde necessariamente cada face é um triângulo.

Segundo Boissonnat et al. (2002) a decisão da representação da **TDS** ser baseada em vértices e faces e a restrição da estrutura poder representar apenas triangulações, tem como vantagens uma menor exigência com relação ao espaço de memória computacional utilizado e também de poder ser generalizada para triangulações de dimensão 3 ou maior.

Como pode ser observado na Figura 18a cada face triangular da **TDS** contêm três ponteiros para os seus três vértices, e três ponteiros para as suas faces adjacentes, ou seja, cada face fornece acesso direto aos seus três vértices e faces adjacentes. Cada vértice também fornece acesso direto a uma de suas faces incidentes por meio de um ponteiro, sendo que por meio dessa face qualquer outra face incidente ao vértice pode ser obtida (BOISSONNAT et al., 2002; PION; YVINEC, 2010).

Ainda segundo a Figura 18a os três ponteiros para os vértices e faces adjacentes são indexados como o valores 0, 1 e 2 no sentido anti-horário, sendo que vértices e faces adjacentes com um mesmo índice são posicionados de maneira oposta.

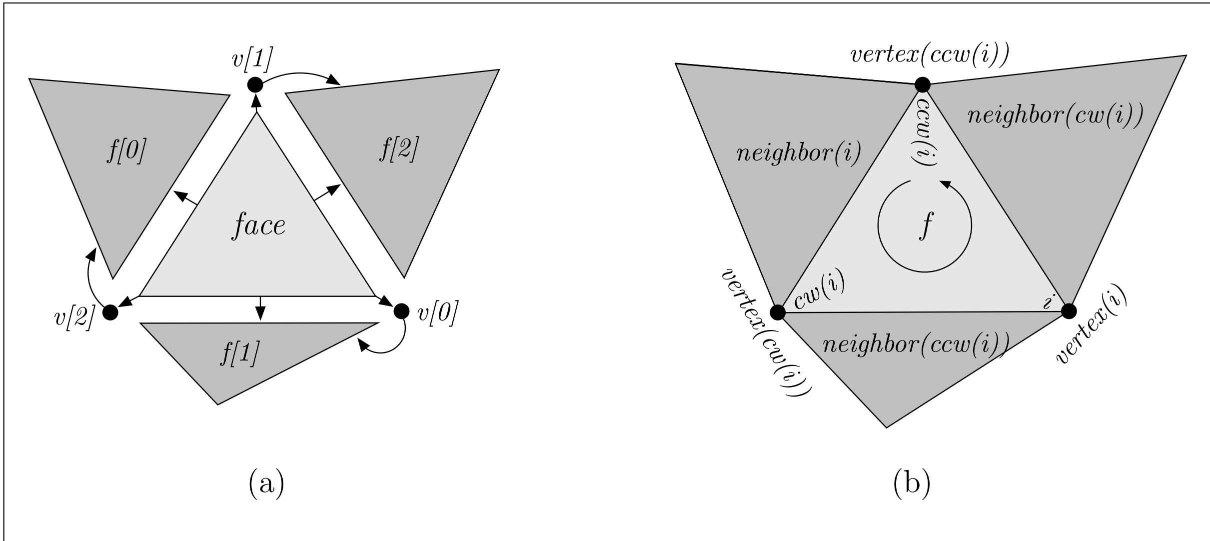


Figura 18: (a) Estrutura de Dados *Triangulation Data Struct (TDS)*; (b) Primitivas de acesso às entidades topológicas vértice e face da TDS.

Conforme é mostrado na Figura 18b, a entidade face da estrutura **TDS** fornece as primitivas *vertex(i)* e *neighbor(i)* para acessar seus vértices e faces adjacentes respectivamente, sendo *i* o valor do índice associado ao vértice ou face adjacente acessado (veja os índices associados a vértices e faces adjacentes na Figura 18a). Lembrando que cada vértice indexado por *i* (*vertex(i)*), é oposto à face adjacente com o mesmo índice (*neighbor(i)*) (Figura 18b).

Também são fornecidas as primitivas *ccw(i)* e *cw(i)* que são usadas para computar *i+1* e *i-1* com módulo 3 (PION; YVINEC, 2010), isto é, essas primitivas recebem como parâmetro um índice e devolvem como resultado um novo índice no sentido anti-horário e horário, respectivamente.

Por exemplo, pode ser observado na Figura 18b que dado um vértice *vertex(i)* de uma face *f*, a primitiva *vertex(ccw(i))* retorna o próximo vértice de *f* partindo do vértice *vertex(i)* no sentido anti-horário, e a primitiva *vertex(ccw(i))* retorna o próximo vértice de *f* partindo do vértice *vertex(i)* no sentido anti-horário. Ou então, dada uma face *f* e uma de suas faces adjacentes *neighgor(i)*, a primitiva *neighbor(cw(i))* retorna a próxima face adjacente de *f* partindo de *neighbor(i)* no sentido horário, e a a primitiva *neighbor(ccw(i))* retorna a próxima face adjacente de *f* partindo de *neighbor(i)* no sentido anti-horário.

Nota-se ainda na Figura 18a a existência de ponteiros apenas para os vértices(*p[0]*, *p[1]*, *p[2]*) e faces(*f[0]*, *f[1]*, *f[2]*) adjacentes à face *f* não existindo nenhum ponteiro para as arestas de *f*. Isto ocorre devido ao fato da entidade aresta ser representada apenas implicitamente na **TDS**.

Essa representação pode ser feita por meio das relações de adjacências entre as entidades face e vértice, por exemplo, como é mostrado na Figura 19, uma aresta e de uma f pode ser facilmente representada pela tupla (f,i), sendo i o valor do índice do vértice f oposto à aresta e. Os vértices delimitares de e nesse caso serão os dois vértices de f com índice diferente i, nesse caso os vértices *vertex(cw(i))* e *vertex(ccw(i))*.

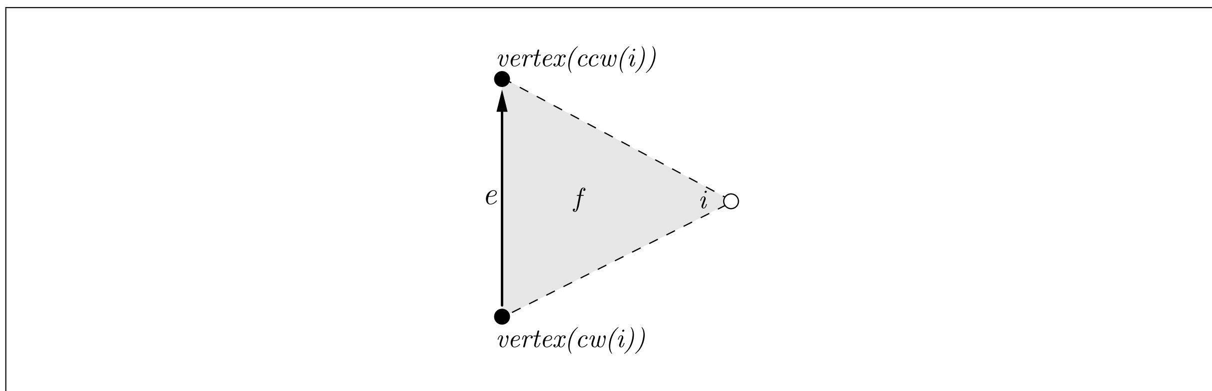


Figura 19: Representação de Arestas na TDS

Embora exista alguma economia de espaço de memória computacional usando uma representação implícita para arestas, em contrapartida é obtida uma estrutura menos poderosa, lembrando ainda que qualquer informação que precise ser adicionada a uma aresta, deve ser adicionada a cada uma de suas faces incidentes, gerando uma duplicação de informações (BOISSONNAT et al., 2002).

Ainda, na **CGAL** uma triangulação é tratada como um conjunto de faces triangulares mais uma face aberta complementar ao fecho convexo formado pelos vértices da triangulação (BOISSONNAT et al., 2002; PION; YVINEC, 2010). Tendo em vista que a **TDS** pode ser utilizada apenas na representação de faces triangulares, e a dificuldade em se lidar com faces abertas, toda triangulação representada pela **TDS** possui um vértice fictício, chamado de *vértice infinito* , sendo que toda aresta que pertencer ao fecho convexo da triangulação deve formar uma *face infinita* com esse vértice. Desta forma cada aresta da triangulação é incidente a exatamente duas faces e o conjunto de faces, aresta e vértices da triangulação é topologicamente equivalente à triangulação bidimensional de uma esfera (BOISSONNAT et al., 2002).

Na Figura 20a na página seguinte, é mostrado o relacionamento entre uma triangulação e os seus vértices, arestas e faces infinitas, onde: cada linha sólida representa uma aresta da triangulação, sendo que as linhas mais escuras representam arestas que fazem parte do fecho convexo, e as linhas tracejadas as *arestas infinitas* (arestas que estão ligadas ao vértice infinito). Cada círculo representa um vértice da triangulação, sendo que

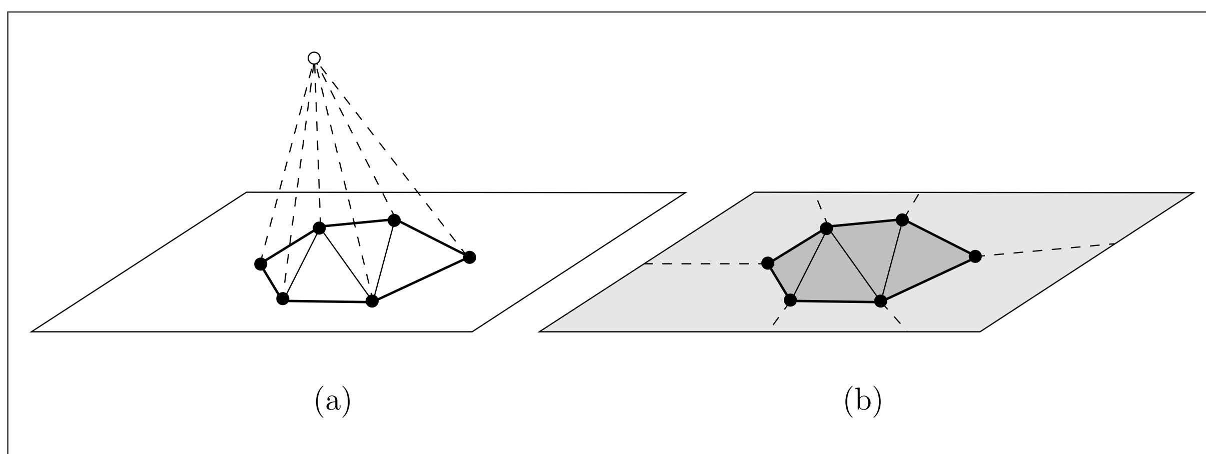


Figura 20: (a) Vértice Infinito e Arestas Infinitas da TDS; (b)Faces Finitas e Faces Infinitas da TDS

o círculo não preenchido representa o *vértice infinito*. Ainda, como pode ser visto na Figura 20b, as faces da triangulação são representadas pela área hachurada, sendo que a área mais clara representa as faces infinitas da triangulação.

Todos os aspectos relativos à construção, e manipulação da estrutura de dados topológica que representa uma triangulação de pontos irregularmente distribuídos no espaço (**TIN**), assim como a geração da triangulação propriamente dita, foram feitos utilizando o pacote de triangulação bidimensional, e a **EDT TDS** da biblioteca **CGAL**. Na Seção 5.4, é mostrado como esses componentes são utilizando na implementação do modulo de representação de **TIN** do sistema proposto, sendo que no trecho de código na página 92 desta mesma Seção, é dado um exemplo de como criar e representar uma triangulação utilizando esses componentes.

4 *Interpolação*

Métodos de interpolação são funções matemáticas que têm como parâmetros pontos de controle (dados pontuais) estimados ou medidos sobre uma superfície (LI; ZHU; GOLD, 2005). Os métodos de interpolação são usados para calcular atributos (elevação, temperatura, densidade demográfica) de dados pontuais com valores desconhecidos usando as informações (atributos) dos pontos na sua vizinhança (YANG et al., 2004).

Segundo Mitas e Mitasova (2005), métodos de interpolação são muito utilizados por **SIG**s, dando suporte para a execução de diversas tarefas como: transformações entre diferentes representações discretas e contínuas (ex: transformações de dados vetoriais como pontos irregularmente espaçados ou linhas de contorno para a representação raster), ou reamostragem entre entre diferentes resoluções raster (ex: conversão de um Grid com resolução de 1 metro em um Grid com resolução de 2 metros).

Na modelagem de terrenos interpoladores geralmente são utilizados na determinação de um valor desconhecido de elevação de um ponto sobre o terreno utilizando como referência pontos conhecidos (previamente amostrados sobre o terreno) na sua vizinhança, de tal forma que seja obtida uma continuidade na superfície modelada. Entretanto o uso de técnicas de interpolação não está restrita apenas ao processo de reconstrução de superfícies, podendo ser utilizadas em vários estágios da modelagem como controle de qualidade e medição de acurácia (LI; ZHU; GOLD, 2005)

Particularmente, no caso de **MDTs** representados por meio de Grid ou **TIN**, métodos de interpolação são utilizados na estimação dos valores de elevação desconhecidos, por exemplo, no espaço entre pontos de um Grid, ou ao longo das arestas e faces de um **TIN** (KUMLER, 1994).

Para Landim (2000) um método de interpolação ideal deve levar em conta alguns fatores como: seu ajuste aos dados (reais) respeitando um nível de precisão pré-definido pelo usuário, ser contínuo e suave em todos os locais e sua aplicabilidade a diferentes configurações e padrões de densidade dos dados.

4.1 Classificação dos Métodos de Interpolação

Os métodos de interpolação espacial podem receber diferentes classificações de acordo com suas características (ex: quantidade, correlação e distribuição espacial dos pontos utilizados pela função interpoladora) e tipo de superfície gerada (ex: superfícies abruptas, suaves e contínuas etc). Nas seções a seguir é dada uma rápida descrição de algumas formas de classificação existentes.

4.1.1 Interpolação por Ponto ou Área

Métodos de interpolação por ponto fazem uso de um conjunto de pontos com localização e atributos conhecidos (ex: valor de elevação e temperatura) para determinar os atributos de outro pontos cuja apenas a localização é conhecida.

Esse tipo de interpolador é usado com dados que podem ser coletados em localizações pontuais específicas como: valores de elevação, de porosidade e tipo de solo em pontos geográficos de uma dada região, ou, leituras (temperatura, umidade, etc) de estações meteorológicas.

Métodos de interpolação por área, utilizam dados de uma variável de um conjunto de áreas para determinar os valores desconhecidos desta variável para outras áreas.

Basicamente esses interpolares são utilizados na resolução de problemas de transferência de dados de um conjunto de áreas (áreas fonte) para outro conjunto de áreas (áreas destino). Um exemplo de uso desse tipo de interpolação é a estimação de dados de população por distritos político a partir de dados de população por áreas de censo.

4.1.2 Interpolação Local ou Global

Os métodos de interpolação local são baseados na premissa que os pontos de amostra influenciam o resultado de um valor de interpolação apenas até uma certa distância. Por exemplo, o valor de elevação desconhecido de um ponto no terreno provavelmente será mais parecido com os valores de elevação medidos para pontos mais próximo do que para pontos mais distantes (MITAS; MITASOVA, 2005).

Como pode ser observado na Figura 21a, na interpolação local um novo valor é interpolado usando como referência apenas um conjunto pontos da amostra na sua vizinhança. Segundo Petrie e Kennie (1987), esse tipo de interpolação possui a característica de que

alterações em dados da amostra usados pela função interpoladora afetam apenas os pontos mais próximos dos locais de alteração.

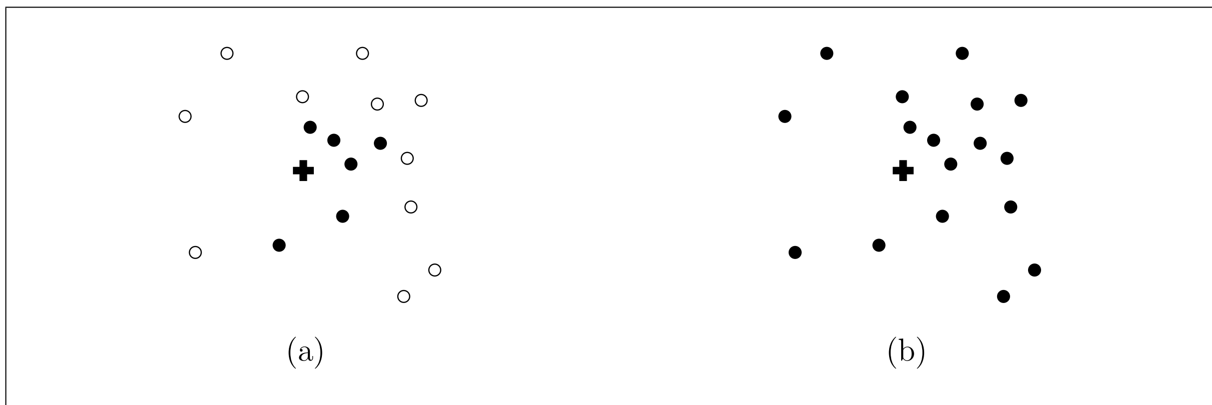


Figura 21: (a) Interpolação Local; (b) Interpolação Global.

Técnicas de interpolação global utilizam todos os pontos amostrados na estimação de valores desconhecidos (Figura 21b). Interpolares desse tipo tentam estabelecer uma superfície tridimensional única, também conhecida como superfície de tendência, usando como base um polinômio de alto grau, cujo os coeficientes são calculados a partir de todo conjunto de dados amostrado. Uma vez definidos todos os coeficientes desse polinômio, valores desconhecido de atributos (elevação, temperatura, precipitação de chuva ...) da superfície modelada podem ser estimados.

Esse tipo interpolação não deve ser utilizada com um grande número de pontos pois nesse caso, deve ser empregado um polinômio de alta ordem, sendo necessária a solução de um sistema de equações de grau igual ao número de pontos amostrados exigindo um alto custo computacional. Ainda, polinômios de alto grau produzem muitas oscilações (o polinômio se torna instável), o que pode gerar valores ruins de interpolação (PETRIE; KENNIE, 1987). Também recomenda-se o uso da interpolação global apenas quando existir uma hipótese ou tendência nos dados amostrados como, por exemplo, dados de temperatura que diminuem com a altitude e a latitude, ou dados amostrados sobre um terreno possuem uma tendência de aumento ou diminuição nos seus valores de elevação no sentido Este-Oeste.

Interpoladores globais ainda tem como características a produção de superfícies atenuadas (sem alterações abruptas) e, dado ao fato de serem utilizados todos os pontos amostrados na construção da superfície, quaisquer alterações nestes dados irão afetar toda a superfície gerada (PETRIE; KENNIE, 1987).

4.1.3 Interpolação Exata ou Aproximada

Como é mostrado na Figura 22a, interpoladores exatos geram superfícies que necessariamente passam por todos os pontos da amostra (LEDOUX; GOLD, 2005). Já no caso dos interpoladores aproximados, a superfície gerada não passa por todos os pontos de referência utilizados na interpolação (Figura 22b).

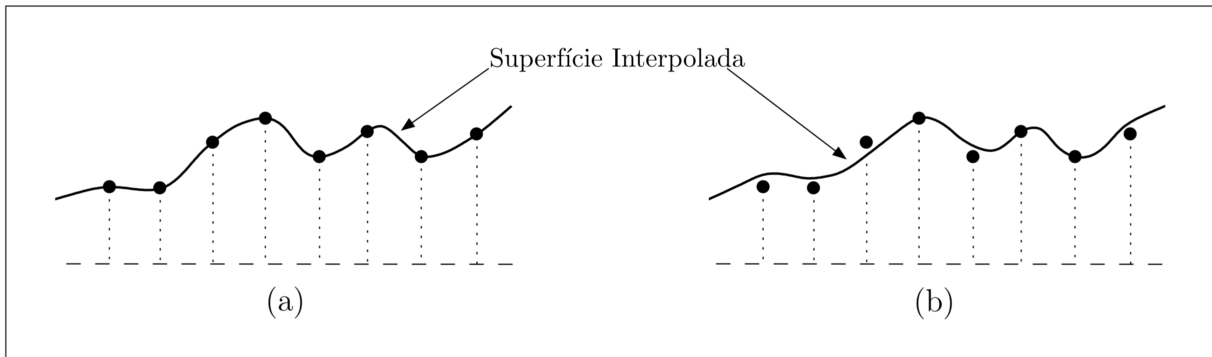


Figura 22: (a) Interpolação Exata; (b) Interpolação Aproximada.

4.1.4 Interpolação Contínua ou Abrupta

Segundo El-Sheimy, Valeo e Habib (2005), interpoladores contínuos geram superfícies com a propriedade de que, ao se aproximar de qualquer ponto dessa superfície por diferentes direções, sempre será encontrado um mesmo valor z para esse ponto (Figura 23a). Geralmente interpoladores desse tipo produzem superfícies com variações graduais.

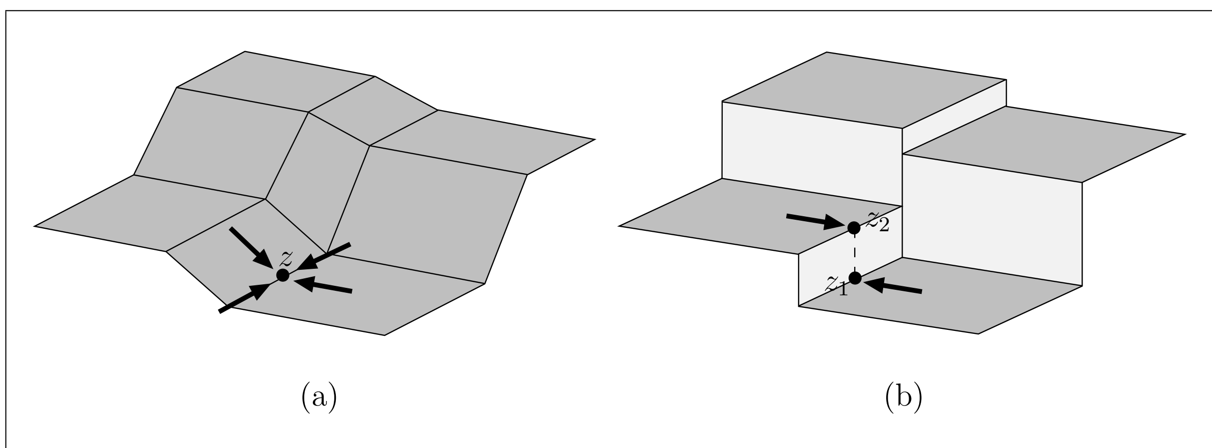


Figura 23: (a) Interpolação Contínua; (b) Interpolação Abrupta.

Como pode ser visto na Figura 23b, interpoladores abruptos, por sua vez, geram superfícies onde são encontrados diferentes valores z para um ponto da superfície, depen-

dendo da direção em que for feita a aproximação desse ponto. Nesse caso a superfície gerada possui muitas variações abruptas.

4.2 Vizinho mais Próximo

O método de interpolação pelo vizinho mais próximo assume que o valor desconhecido de um atributo numa dada posição (x, y) de uma superfície, é igual ao atributo do seu ponto vizinho mais próximo (LEDOUX; GOLD, 2005), ou seja, cada posição (x, y) interpolada recebe o valor do dado amostrado mais próximo.

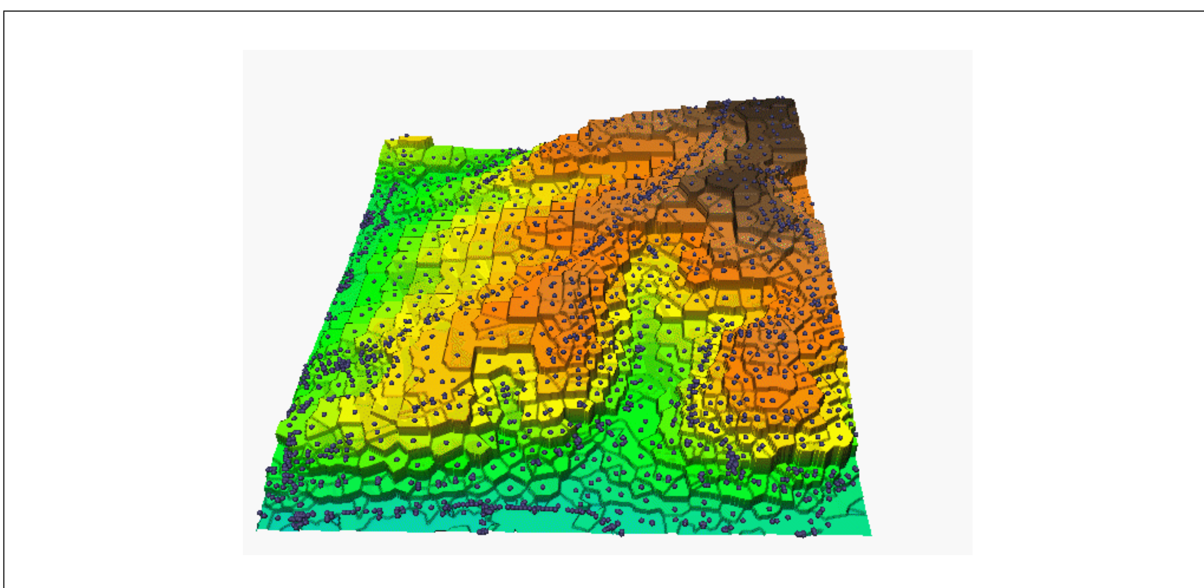


Figura 24: Representação de uma superfície utilizando Diagrama de Voronoi.
Fonte: Mitas e Mitasova (2005)

Esse método tem por característica produzir superfícies não contínuas, sendo que, se ele for usado com dados muito próximos entre sí, acaba gerando superfícies com o comportamento de um Diagrama de Voronoi, como a superfície representada por polígonos de Voronoi mostrada na Figura 24.

Segundo Ledoux e Gold (2005), o método de interpolação pelo vizinho mais próximo pode ser utilizado no sensoriamento remoto na diminuição de avarias ou borramentos em imagens de satélite, ou então com dados temáticos como: fatiamento de imagens, tipos de rochas e cobertura do solo. A maior dificuldade na implementação computacional desse método está relacionada à busca eficiente das relações de proximidade entre os dados.

4.3 Média Simples

Essa técnica de interpolação faz a estimação de um atributo desconhecido numa dada localização (x, y), por meio da média simples dos seus n vizinhos mais próximos. Trata-se de um método de interpolação rápido porém sem muita precisão. Segundo Namikawa et al. (2003), esse interpolador geralmente é utilizado quando se requer maior rapidez na geração de Grids, para avaliar erros grosseiros ocorridos durante a digitalização de dados da amostra.

O método de interpolação pela média simples possui a seguinte formulação:

$$f(x, y) = \frac{1}{n}\left(\sum_{i=1}^{n} a_i\right) \tag{4.1}$$

onde:

- $f(x, y)$ é a função interpolante;
- n é o número de vizinhos;
- a_i é o valor do atributo associado a um vizinho;

4.4 Inverso da Distância

Segundo Mitas e Mitasova (2005), esse método de interpolação é baseado na idéia de que pontos mais próximos são mais parecidos, enquanto pontos mais distantes não possuem muitas semelhanças (são mais independentes).

O método de interpolação inverso da distância estima o valor do atributo de um dado pontual, como a média ponderada dos atributos dos pontos da amostra. Como se trata de um método que usa o cálculo de uma média ponderada, cada ponto da amostra tem um peso associado a si; onde esses pesos são inversamente proporcionais à distância entre o ponto a ser interpolado e um ponto da amostra em questão, ou seja, pontos da amostra mais próximos do ponto a ser estimado têm um maior peso/influência sobre este, sendo que a soma dos pesos dos pontos utilizados na interpolação deve ser igual a 1.0 (EL-SHEIMY; VALEO; HABIB, 2005). Quando um ponto a ser estimado coincide com um ponto da amostra, este recebe peso 1.0, enquanto todos os outros pontos vizinhos recebem peso 0.0, de tal forma que o ponto que esta sendo estimado receba o valor exato do ponto da amostra ali situado (LANDIM, 2000).

De acordo com Ledoux e Gold (2005) interpolação pelo inverso da distância possui a seguinte formulação:

$$f(x,y) = \frac{\sum_{i=1}^{k} w_i a_i}{\sum_{j=1}^{n} w_i} \tag{4.2}$$

onde:

- $f(x, y)$ é a função interpolante;
- n é o número de vizinhos;
- a_i é o valor do atributo associado a um vizinho;
- w_i é o peso associado a cada vizinho.

Segundo Li, Zhu e Gold (2005), as seguintes equações de função da distância podem ser utilizadas na associação de pesos (w_i) aos pontos de controle (pontos da amostra):

$$w_i = \frac{1}{d_i^2} \tag{4.3}$$

$$w_i = \left(\frac{R - d_i}{d_i}\right)^2 \tag{4.4}$$

$$w_i = e^{-d_i^2/K^2} \tag{4.5}$$

onde:

- w_i é o peso associado ao ponto de controle i;
- R representa o raio do círculo de busca (Figura 25a na página seguinte);
- d_i é a distância euclidiana entre o ponto de controle e o ponto a ser interpolado;
- K é uma constante.

Geralmente a equação 4.3 é utilizada de forma a cancelar a raiz quadrada usada no cálculo da distância euclidiana melhorando o desempenho computacional do método (LEDOUX; GOLD, 2005; SLOCUM, 1999).

O uso desse método ainda envolve a definição de estratégias de busca dos pontos vizinhos a posição (x, y) que será interpolada. Segundo Petrie e Kennie (1987), dentre as possíveis estratégias de busca de pontos na vizinhança pode-se citar: uso de círculos, elipses e quadriláteros como área de busca (ex: Figura 25a e 25b); pegar os n vizinhos mais próximos (ex: Figura 25c); ou dividir a área ao redor da posição a ser interpolada em setores (quadrantes ou octantes) e pegar n vizinhos mais próximo em cada setor (ex: Figura 25d e 25e).

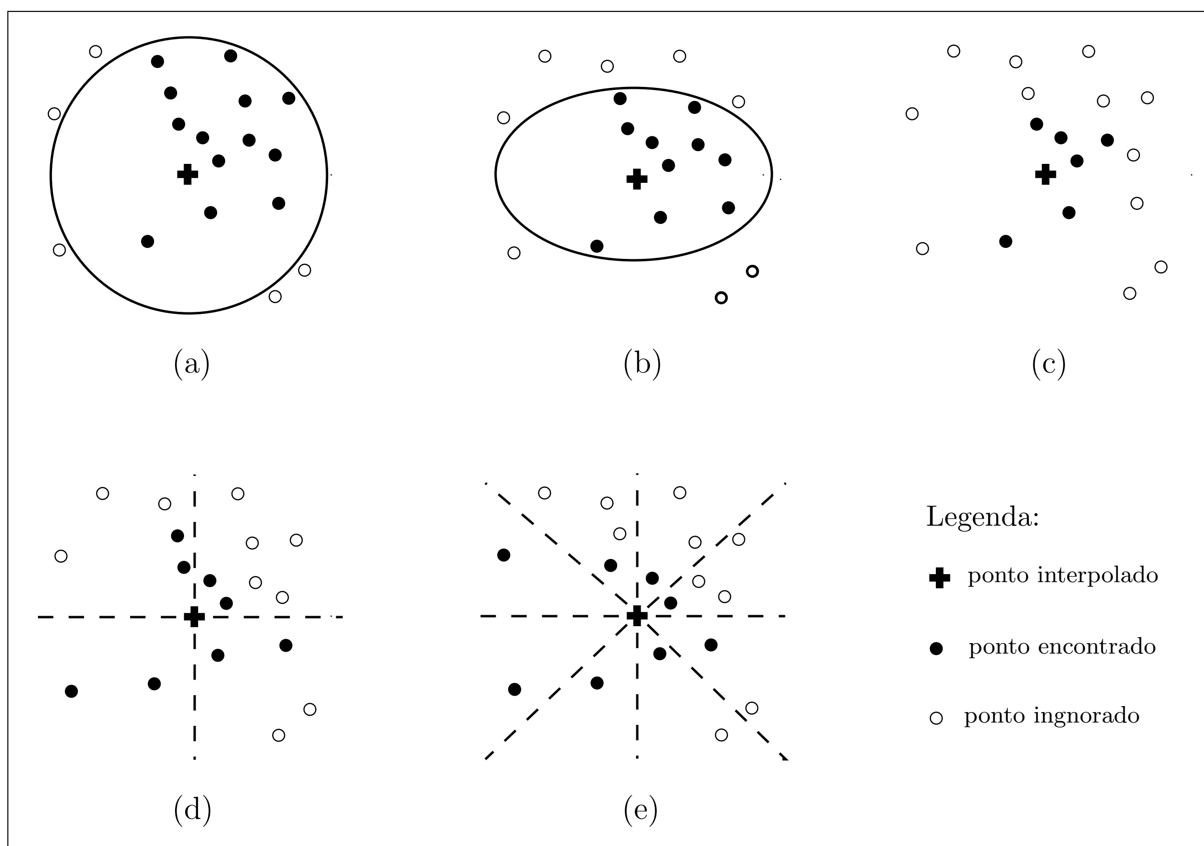


Figura 25: Estratégias de busca de pontos: (a) Círculo de busca; (b) Elipse de busca; (c) n vizinhos mais próximos; (d) n vizinhos mais próximos em cada quadrante; (d) n vizinhos mais próximos em cada octante.
Fonte: Li, Zhu e Gold (2005)

Apesar desse método ser de fácil implementação computacional, e estar presente em quase todos sistemas **SIG**, devido aos critérios de busca e associação de peso não levarem em conta a distribuição espacial dos dados, esse método não produz bons resultados (não garante continuidade) no caso de dados agrupados ou com tendências (MITAS; MITASOVA, 2005; LEDOUX; GOLD, 2005).

4.5 Interpolação pela Vizinhança Natural

O método de interpolação pela vizinhança natural é baseado no Diagrama de Voronoi (vide Seção 3.1.2). Ao contrário dos métodos de interpolação descritos anteriormente que utilizam a distância como fator de ponderação no cálculo da influência dos pontos na vizinhança de um ponto a ser estimado, esse método usa medidas de área como fator de ponderação na estimação de um novo valor.

Como pode ser observado na Figura 26, basicamente são utilizados dois Diagrama de Voronoi:

- O primeiro Diagrama de Voronoi (Figura 26a) é gerado a partir de todos os dados da amostra, onde cada polígono de Voronoi V_{pi} representa a área de influência (também chamada de região de vizinhança natural) de um ponto p_i da amostra.
- O segundo Diagrama de Voronoi (Figura 26b) é gerado a partir de todos dos dados da amostra mais o ponto p_k que será interpolado, nesse caso é criada uma nova região de Voronoi V_{pk} (área hachurada) associada ao ponto a ser interpolado.

Conforme pode ser visto na Figura 26c, ao se fazer a sobreposição desses dois Diagramas de Voronoi, percebe-se que a região de Voronoi V_{pk} associada ao ponto a ser interpolado (área hachurada na Figura 26b) sobrepõe algumas regiões V_{pi} associadas aos dados da amostra (área hachurada na Figura 26a).

Cada área V_k (área hachurada na Figura 26c) gerada a partir da intersecção das regiões de Voronoi dos pontos da amostra (V_{pi}) com a região do ponto a ser estimado (V_{pk}) tem um peso w_i associado a sí. Segundo Ledoux e Gold (2005) e como é mostrado na equação a seguir, esse peso representa a porcentagem dessas áreas de intersecção em relação à região de Voronoi do ponto a ser interpolado:

$$w_i = \frac{Area(V_p) \bigcap Area(V_{pi})}{Area(V_{pi})} \tag{4.6}$$

sendo que uma vez calculado o peso w_i associado a cada área, p_k pode ser estimado como a média dos pontos p_i ao seu redor ponderados por w_i.

Esse método de interpolação tem como características produzir bons resultados com dados cuja a distribuição é altamente irregular (LEDOUX; GOLD, 2005), e ao contrário de outros interpoladores onde o usuário deve pré-definir o número de pontos vizinhos

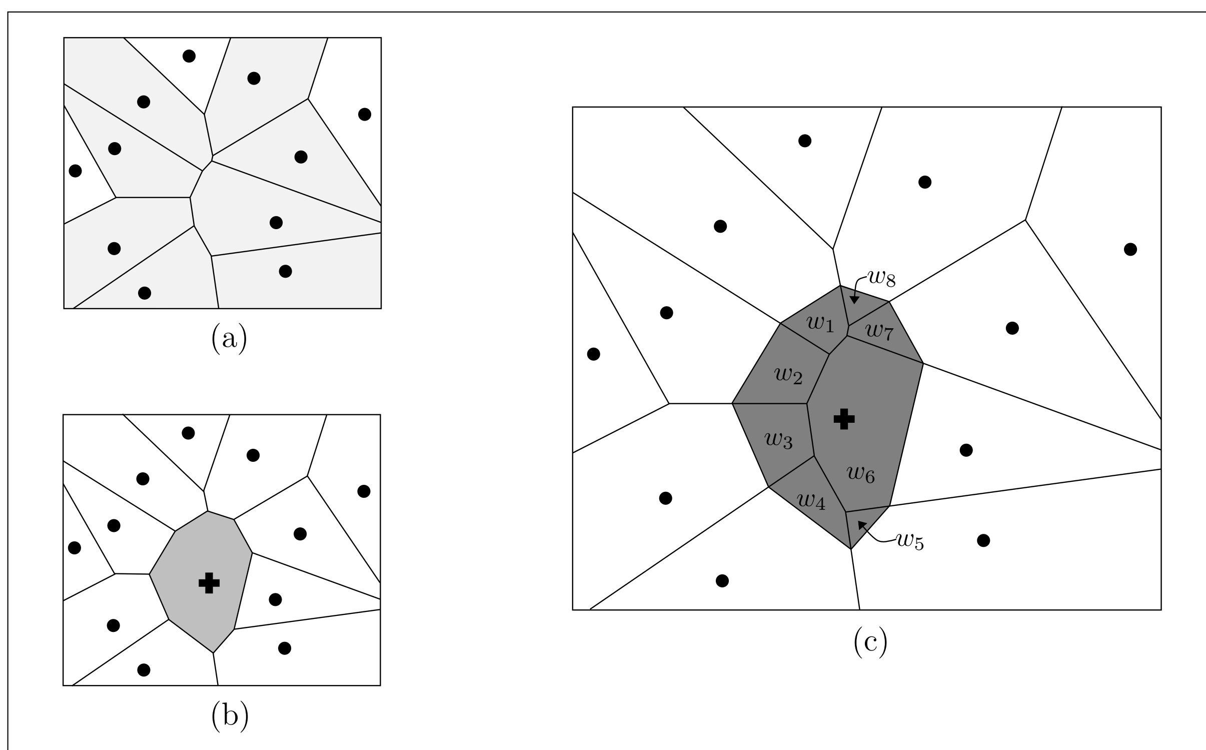


Figura 26: Interpolação pela Vizinhança Natural: (a) Diagrama de Voronoi gerado a partir de uma amostra de pontos; (b) Diagrama de Voronoi gerado a partir dos pontos uma amostra mais um ponto com atributo desconhecido a ser interpolado; (c) Sobreposição dos Diagramas de Voronoi (a) e (b) com valores de peso (w_i) associados à áreas de interseção.

usados na estimação de um valor desconhecido, o número de pontos da amostra usados na interpolação pela vizinhança natural é variável, dependendo da configuração espacial dos dados da amostra (MITAS; MITASOVA, 2005).

4.6 Interpolação Linear

Esse tipo de interpolação é muito utilizado com o modelo **TIN**. Segundo Li, Zhu e Gold (2005), nesse método, os vértices de cada face triangular do **TIN** definem o plano:

$$z = a_0 + a_1 x + a_2 y \tag{4.7}$$

onde a_0, a_1 e a_2 são os coeficientes e (x, y, z) são as coordenadas de um ponto sobre esse plano. Como no modelo **TIN** cada vértice da triangulação representa um ponto de controle (ponto da amostra), os três coeficientes da equação 4.7 são encontrados utilizando os três pontos $p_1\,(x_1, y_1, z_1)$, $p_2\,(x_2, y_2, z_2)$ e $p_3\,(x_3, y_3, z_3)$ pertencentes a face triangular

(que contêm o ponto p_k a ser interpolado) gerando o sistema de equações lineares a seguir:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}^{-1} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} \tag{4.8}$$

Uma vez definidos os coeficientes a_0, a_1 e a_2, o valor de elevação z_k de qualquer ponto p_k com coordenadas (x_k, y_k) pertencente ao triângulo formado por p_1, p_2 e p_3 pode ser obtido substituindo (x_k, y_k) na equação 4.7, ou seja, para qualquer ponto p_k a ser estimado, deve-se buscar o triângulo que o contem e através da solução do sistema linear 4.8, obter o valor z_k desse ponto.

Para Namikawa et al. (2003), apesar desse método de interpolação possuir um esforço computacional mínimo e garantir continuidade entre as superfícies de triângulos vizinhos, não é garantida uma suavidade na transição entre as superfícies.

4.7 Interpolação Bilinear

Esse método de interpolação pode ser utilizado tanto para **MDTs** representados por meio de Grids como **TIN**. Segundo (LI; ZHU; GOLD, 2005) a interpolação bilinear pode ser obtida a partir de quatro pontos que não sejam colineares, sendo definida pelo polinômio:

$$z = a_0 + a_1x + a_2y + a_3xy \tag{4.9}$$

onde a_0, a_1 e a_2 são os coeficientes; (x, y) são as coordenadas de um ponto a ser interpolado e z o valor interpolado. Os coeficientes da equação 4.9 podem ser determinados a partir de quatro equações que façam uso de quatro pontos de controle $p_1\,(x_1, y_1, z_1)$, $p_2\,(x_2, y_2, z_2)$, $p_3\,(x_3, y_3, z_3)$ e $p_4\,(x_4, y_4, z_4)$ gerando o sistema de equações lineares a seguir:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 & x_1y_1 \\ 1 & x_2 & y_2 & x_2y_2 \\ 1 & x_3 & y_3 & x_3y_3 \\ 1 & x_4 & y_4 & x_4y_4 \end{bmatrix}^{-1} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} \tag{4.10}$$

Uma vez definidos os coeficientes a_0, a_1, a_2 e a_2, o valor de elevação z_k de qualquer ponto p_k, com coordenadas (x_k, y_k) conhecidas, pode ser obtido substituindo (x_k, y_k) na

equação 4.9 na página anterior.

Se os dados estiverem organizados na forma de um Grid, para cada nova posição a ser interpolada, basta encontrar uma região formada por quatro pontos do Grid que contenha essa nova posição, e usar esses pontos na resolução do sistema 4.10.

A interpolação bilinear, conforme mostrado na Figura 27, também pode ser utilizada com **TIN**s seguindo os seguintes passos:

- Dado um ponto $p_k(x_k, y_k, z_k)$ com a coordenada z_k a ser interpolada, encontrar os pontos $p_a(x_a, y_a, z_a)$, $p_b(x_b, y_b, z_b)$ e $p_c(x_c, y_c, z_c)$ da face que o contêm;
- Assumindo que os pontos $p_{ab}(x_{ab}, y_{ab}, z_{ab})$, p_k e $p_{ac}(x_{ac}, y_{ac}, z_{ac})$ possuem um mesmo valor para y, isto é $(y_{ab} = y_k = y_{ac})$, fazer uma interpolação linear no plano das coordenadas x_{ab} e x_{ac} dos pontos p_{ab} e p_{ac} usando os pontos p_a,p_b e p_a,p_c respectivamente;
- Uma vez conhecidas as coordenadas x e y de p_{ab} e p_{ac} encontrar as coordenadas z_{ab} e z_{ac} por meio de uma interpolação linear no espaço usando os pontos p_a,p_b e p_a,p_c respectivamente;
- Fazer uma interpolação linear no espaço de z_p usando os pontos p_{ab} e p_{ac} .

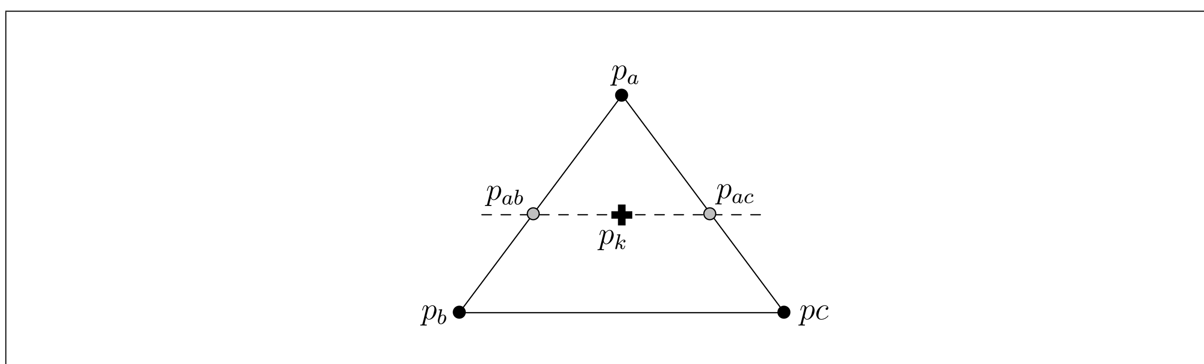


Figura 27: Interpolação Bilinear de triângulos.

Apesar desse método gerar um melhor ajuste da superfície ao longo de um Grid ou faces triangulares de um **TIN** do que a interpolação linear, esse método ainda tem como característica gerar transições bruscas.

5 *Sistema Proposto*

No sistema proposto, a reconstrução da superfície do terreno é feita a partir de uma triangulação sobre uma amostra de pontos sobre o terreno (**TIN**).

Apesar da superfície do terreno poder ser representada primariamente por um conjunto de pontos irregularmente espaçados, esta forma de representação não oferece informações adicionais de natureza topológica, como relações de incidência, adjacência, conectividade, e proximidade, nem permitem explorar propriedades de natureza geométrica como, por exemplo, declividade e orientação do terreno.

Isto torna a existência de uma triangulação fundamental, na medida em que é ela que estabelece a conexão entre os pontos (dimensão 0), criando mais duas entidades que até então não existiam, as arestas (dimensão 1) e as faces triangulares (dimensão 2), dando origem a uma estrutura topológica e combinatória, que consegue capturar informações relevantes sobre o terreno e dá a abstração necessária para a criação de um modelo computacional que constitui a essência do sistema proposto.

A Figura 28 ilustra diagramaticamente a metodologia utilizada pelo sistema proposto, na reconstrução da superfície de um terreno. Inicialmente, o sistema receberá como entrada um conjunto de pontos amostrados sobre a superfície do terreno (Figura 28a), em seguida, esse conjunto de pontos será projetado ortogonalmente sobre um plano (Figura 28b), onde então será aplicada a técnica de Triangulação de Delaunay sobre esses pontos (Figura 28c) assim como será criada a estrutura de dados topológica que irá representar a topologia dessa triangulação. Uma vez terminada a triangulação dos pontos, estes são levados de volta à sua posição original (Figura 28d), sendo importante ressaltar que toda informação de natureza topológica gerada durante a triangulação dos pontos no plano é preservada quando estes são trazidos de volta à sua posição no espaço.

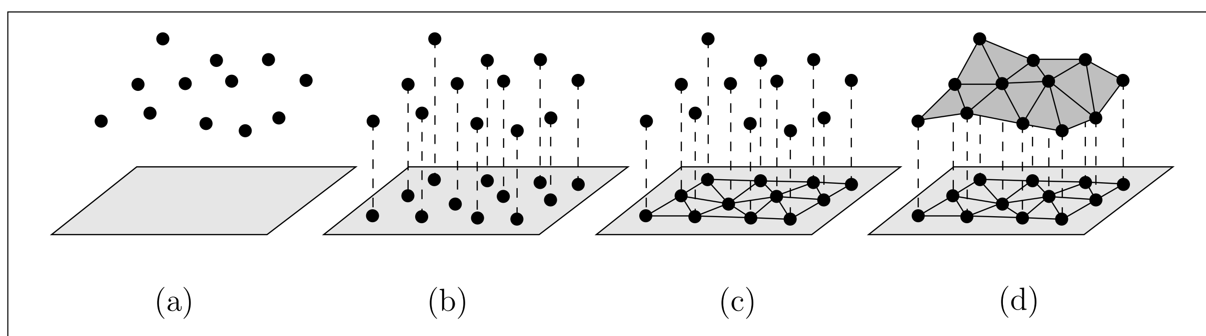


Figura 28: Triangulação de um conjunto de pontos amostrados sobre a superfície de um terreno: (a) Pontos no espaço; (b) Projeção ortogonal; (c) Triangulação dos pontos no plano; (d) Associação da cota e ilustração da malha de elementos triangulares sobre o terreno.

Fonte: adaptado de Piteri et al. (2007)

5.1 Ferramentas Utilizadas

O sistema foi construído utilizando a linguagem de programação **C/C++**. Essa linguagem de programação contem todos os conceitos e ferramentas necessárias para a utilização dos paradigmas de programação orientada a objetos e programação genérica, além de fornecer a biblioteca padrão **STL** (**S**tandard **T**emplate **L**ibrary), que contém várias estruturas e componentes que facilitaram enormemente o desenvolvimento do projeto além de servir como modelo de boas práticas de programação. Nesse primeiro momento, o sistema foi desenvolvido sobre a plataforma *Windows* com o auxílio do compilador *Microsoft VisualStudio*.

Considerando a grandeza e multidisciplinaridade do projeto foram utilizadas as seguintes bibliotecas *software* livre e multiplataforma: **CGAL**, **GDAL**, **OGR**, **Qt**, **OpenGL** e **OSG**.

Cada uma dessas ferramentas foi escolhida levando-se em consideração: desempenho, estabilidade, robustez e inserção em sua área de atuação. Vale realçar que essas bibliotecas também são implementadas em **C/C++** e tem se tornado padrão no mercado de desenvolvimento de *software*.

Nas seções a seguir é dada uma descrição mais detalhada das funcionalidades de cada uma dessas bibliotecas assim como a sua utilização no sistema proposto.

5.1.1 Biblioteca CGAL

A área da Geometria Computacional oferece inúmeras soluções na forma de algoritmos e estruturas de dados para manipulação e análise eficiente de dados espaciais (VERBREE; OOSTEROM; QUAK, 2000).

Entretanto a implementação computacional de algoritmos geométricos se torna uma tarefa complicada dada a divergência entre a aritmética em ponto-flutuante (rápida e inexata associada ao *hardware* de sistemas computacionais) e a aritmética exata de números reais assumida na concepção desses algoritmos em seus respectivos documentos, tornando esses tipos de algoritmos muito sensíveis a erros de arredondamento (VELTKAMP, 2001).

Por exemplo, ao considerar-se o problema de encontrar o centro de uma circunferência a partir de três pontos pertencentes à mesma, normalmente a precisão dos tipos numéricos reais oferecidos pelo *hardware* computacional será suficiente para a correta resolução do problema. Porém, se os três pontos utilizados forem "quase colineares", a implementação do algoritmo em questão poderá: gerar um resultado com valores absurdos; parar devido a erros de divisão por zero; ou até mesmo entrar em *loop-infinito*. Sendo obrigatório, nesse caso, o uso de aritmética exata (SHEWCHUK, 1996).

Ainda existem outros problemas como a falta de tratamento explícito dos casos degenerados e a complexidade inerente às implementações eficientes desses algoritmos tornam o desenvolvimento desse tipo de *software* ainda mais difícil (VELTKAMP, 2001).

Considerando esse e outros problemas relacionados à geração de *software* ligado à solução de problemas de natureza geométrica, no presente trabalho foi usada a biblioteca de estruturas de dados e algoritmos geométricos **CGAL**. Trata-se de uma biblioteca de *software opensource*, escrita utilizando a linguagem de programação C++, que tem por propósito disponibilizar para a indústria e meios acadêmicos, implementações de estruturas de dados e algoritmos geométricos confiáveis e eficazes para a resolução de problemas básicos na área de Geometria Computacional, com isso facilitando o desenvolvimento de aplicações geométricas mais complexas (KETTNER; NäHER, 2004).

Basicamente a **CGAL** é composta por uma coleção de estruturas de dados e algoritmos que operam sobre as mesmas, sendo estruturada em três camadas e uma biblioteca de suporte a parte. Como pode ser observado na Figura 29, a **CGAL** é constituída pela biblioteca *core*, o *kernel* geométrico, biblioteca básica e bibliotecas de suporte (FABRI et al., 2000).

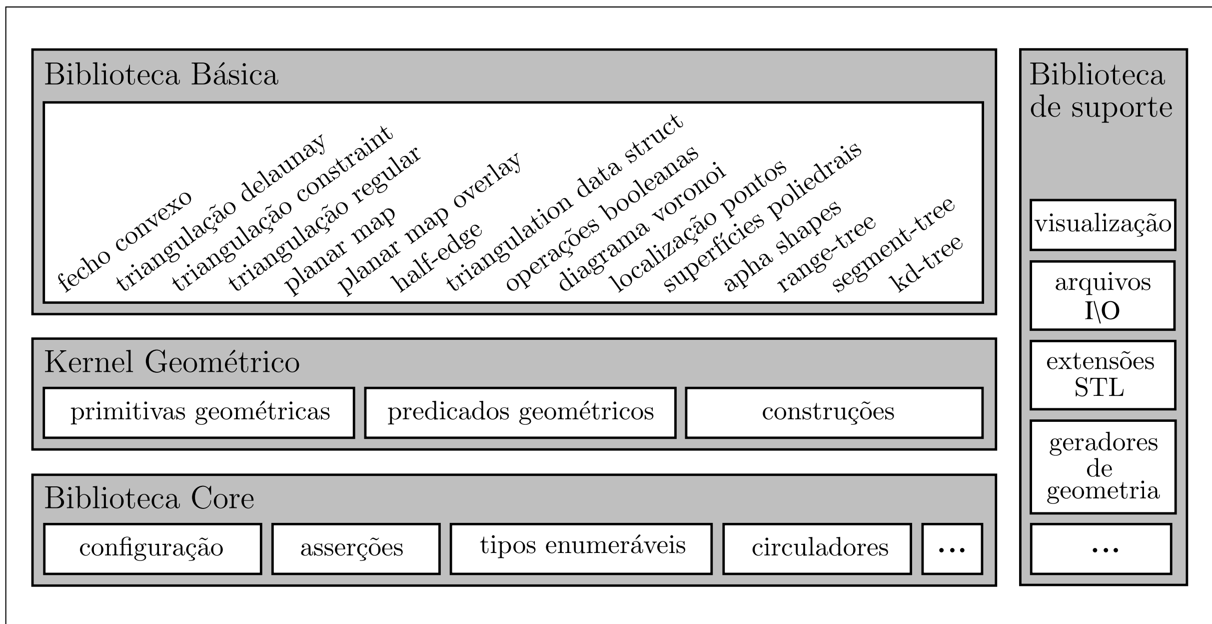


Figura 29: Estrutura da CGAL.

5.1.1.1 Biblioteta *Core*

O *core* possui funcionalidades não geométricas da **CGAL** como funções de configuração, asserções (*asserts*) para verificar se o comportamento do código está correto ou se o usuário está chamando as funções fornecidas pela biblioteca de maneira correta (*preconditions*) e se estas estão fazendo o que é prometido (*postconditions*), tipos enumeráveis (*enum*), assim como funcionalidades que servem como base para o correto funcionamento do *kernel* geométrico e a biblioteca básica.

5.1.1.2 *Kernel* Geométrico

O *kernel* é composto por implementações de primitivas geométricas básicas de tamanho constante (pontos, segmentos, linhas, círculos, triângulos, retângulos, esferas, tetraedros etc) e operações que podem ser aplicadas sobre as mesmas (testes e cálculo de intersecção, testes de orientação entre pontos, cálculos de distâncias e área, comparação entre coordenadas, cálculo do ponto médio de um segmento etc). Essas operações são classificadas em dois tipos: predicados e construções.

São considerados predicados todas as operações sobre um objeto geométrico que retornem um tipo booleano (verdadeiro ou falso), como, por exemplo, testes de intersecção entre dois segmentos de reta ou testes que retornem tipo enumeráveis, como por exemplo, uma função que verifique o posicionamento de um ponto em relação a uma reta orien-

tada (se ponto está à direita, sobre, ou a esquerda da reta). Desta forma os predicados encapsulam testes ou decisões e são utilizados como blocos de construção de algoritmos geométricos.

A **CGAL** classifica como construções todas as operações que não retornam tipos booleanos ou enumerações, ou seja, são consideradas construções funções que computam valores numéricos como distâncias, tamanho ou área de primitivas geométricas ou ainda funções que geram outras primitivas como, por exemplo, o cálculo do ponto médio de um segmento.

5.1.1.3 Biblioteca Básica

A biblioteca básica é construída sobre o *kernel*. Ela contem implementações de estruturas de dados de uso comum na área de Geometria Computacional como as estruturas de dados topológicas *Half-Edge* e **TDS** usadas para a representação de arranjos, polígonos, poliedros ou malhas triangulares e estruturas espaciais de busca como a *Kd-Tree*, *Range-Tree* ou *Segment-Tree* usadas, por exemplo, para verificar se um dado objeto pertence a um conjunto, encontrar todos os elementos de um conjunto que pertencem a uma área (como um retângulo) ou encontrar os objetos mais próximos a um ponto. Ainda, a biblioteca básica da **CGAL** inclui um conjunto de algoritmos que operam sobre essas estruturas de dados como: Fecho Convexo, Diagrama de Voronoi, Triangulação de Delaunay, Localização de Pontos, *Alpha Shapes* dentre outros.

5.1.1.4 Bibliotecas de Suporte

Finalmente, as bibliotecas de suporte, possuem funcionalidades de natureza não geométrica mas ao contrário do *core* as suas funcionalidade não são necessárias para o *kernel* ou para a biblioteca básica. Dessa forma, elas oferecem ferramentas de apoio ao desenvolvimento de *software* que faça uso da **CGAL** como tipos numéricos exatos (por meio da integração com diversos pacotes de *software* que fornecem tipos com de aritmética exata), suporte à operações de entrada e saída em arquivos, geradores randômicos de entidades geométricas (ex: polígonos, conjuntos convexos de pontos e segmentos), extensões para a **STL** (ex: circuladores e algorítimos de ordenação), e interfaces de integração com várias ferramentas como a *GeomView*, *LEDA Windows*, *GeoWin* e **Qt**.

Na Seção a seguir, é feita uma descrição de como é tratada a computação exata nas estruturas e algorítimos geométricos fornecidos pela **CGAL**.

5.1.1.5 Computação Exata

Todos os algoritmos fornecidos pela **CGAL** assumem o uso de computação exata. Entretanto a computação de tipos exatos consome muito mais tempo de execução computacional que os tipos ponto-flutuante presentes em *hardware* (SHEWCHUK, 1996). Para resolver esse problema, a **CGAL** disponibiliza, em complemento aos tipos exatos, tipos numéricos que fazem uso de técnicas de filtragem no computo de predicados. Neste caso, a comparação de predicados é feita primeiramente usando o tipo flutuante rápido, mas inexato, fornecido pelo *hardware*. Ao mesmo tempo é calculado um erro de arredondamento associado a essa operação, que é usado para estimar a qualidade da solução obtida. Se a solução não for considerada correta, é feita uma nova avaliação do predicado, só que agora utilizando aritmética exata, ou seja, a aritmética exata é usada apenas quando necessário, mantendo a robustez do sistema sem gerar grandes prejuízos na sua eficiência (velocidade de execução).

Desta maneira, a **CGAL** oferece uma forma inteligente de uso da aritmética exata, permitindo ao desenvolvedor escolher entre usar aritmética exata (com filtragem), tipos inexatos (*hardware*) ou ambos. Por exemplo: geralmente os algoritmos de triangulação necessitam do uso de aritmética exata apenas nos seus predicados, sendo suficiente o uso de tipos ponto-flutuante em suas construções garantindo uma melhor eficiência dos mesmos.

A **CGAL** foi utilizada na implementação das funcionalidades de geração e representação computacional de malhas triangulares (**TIN**) no sistema proposto. Na Seção 5.2 na página 84, é dada uma visão geral da arquitetura do sistema e de como a **CGAL** está inserida dentro dele, enquanto na Seção 5.4 na página 90, é feita uma descrição detalhada (inclusive com exemplo de código) de quais funcionalidade da **CGAL** foram utilizadas e de quais partes do sistema foram desenvolvidas utilizando ela.

5.1.2 OpenGL

O **OpenGL** é um sistema gráfico simples e interativo para modelagem e exibição tridimensional, bastante rápido e portável, que permite ao desenvolvedor de *software* escrever programas que acessem o *hardware* gráfico desde plataformas *desktop* a grandes *workstations* (SHREINER et al., 2004; GRAPHICS, 1998).

Desde sua introdução em 1992 o **OpenGL** tem se tornado a **API** (**A**pplication **P**rogramming **I**nterface) de desenvolvimento de aplicações gráficas 2D e 3D mais utili-

zada pela indústria de *software*, permitido o desenvolvimento dessas aplicações em uma grande variedades de plataformas computacionais (KHRONOS GROUP, 2010). Na Figura 30 é mostrado um histórico do desenvolvimento dessa ferramenta.

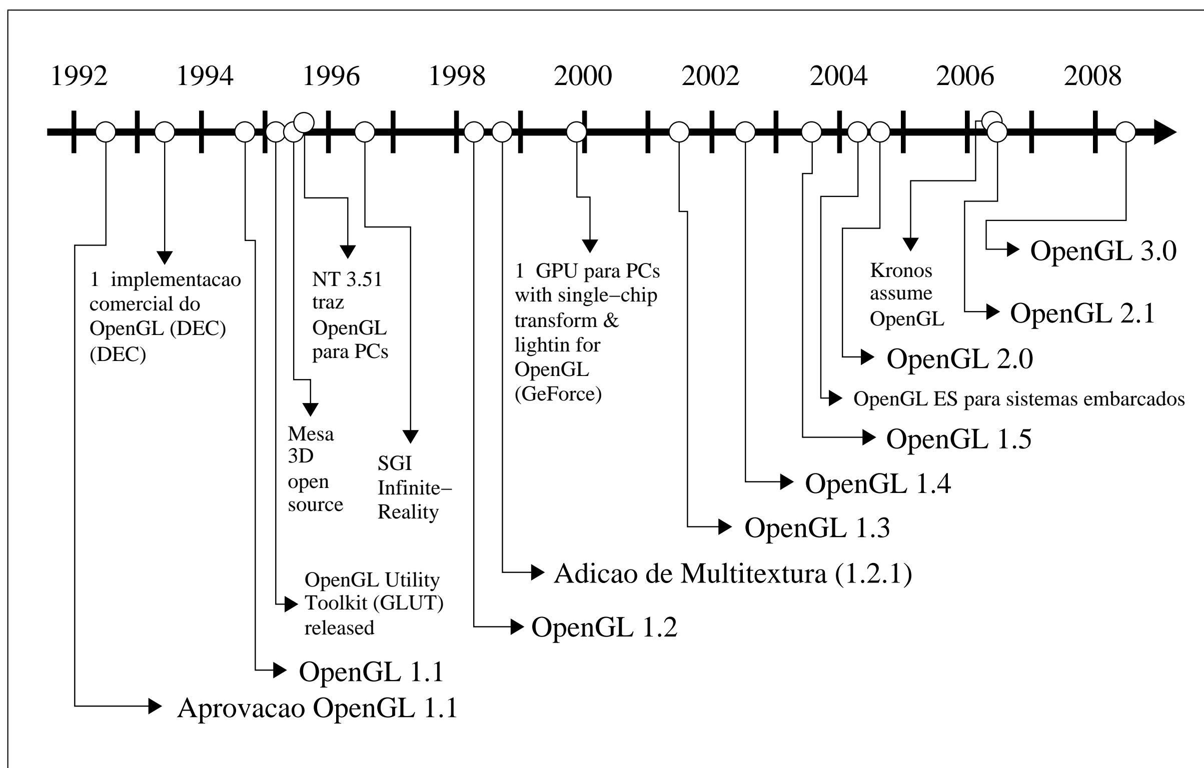


Figura 30: Linha do Tempo de desenvolvimento do OpenGL
Fonte: adaptado de Kilgard et al. (2008)

Segundo Kilgard et al. (2008), Khronos Group (2010), o **OpenGL** foi planejado seguindo as seguintes filosofias:

- **Definido por uma Especificação:** o **OpenGL** é uma especificação, controlada pelo grupo Khronos Group (2010). Trata-se de um consórcio aberto, formado por empresas de desenvolvimento de *software* como *3Dlabs*, *ATI*, *Discreet*, *Evans & Sutherland*, *Intel*, *NVIDIA*, *Activion*, *Blizard*, *Motorola*, *Nokia*, *SGI* e *Sun Microsystems*, dedicadas a criação de padrões de APIs que possibilitem a produção de aplicações de mídia em uma grande variedade de plataformas. Dentre as principais implementações do **OpenGL** recebem destaque as fornecidas pelas empresas *AMD*, *Intel* e *NVIDIA*;

- **Alta-Performance:** todas as funcionalidades oferecidas pelo **OpenGL**, inicialmente assumem, aceleração por *hardware* (sendo esta dependente da implementação);

- **Renderização usando Máquina de Estado:** o **OpenGL** trata da especificação de uma **API** procedural, com independência de sistemas de janelas e com estrutura simples, por exemplo não fornece grafos de cena ou estruturas mais complexas;

- **Extensibilidade:** o padrão **OpenGL** está em constante evolução, permitindo que extensões fornecidas por fabricantes de placas gráficas possam ser utilizadas por desenvolvedores de aplicações, de tal forma que estes tenham acesso aos mais recentes avanços de *hardware*. Quando uma extensão é amplamente testada e aceita, esta é adicionada sua **API**. Na Figura 31 são mostrados os principais desenvolvedores de extensões do **OpenGL**;

- **Multi-Plataforma:** o **OpenGL** não possui nenhuma função de gerenciamento de janela, ou tratamento de eventos do mouse e teclado, entretanto ele pode interagir com um sistema operacional e seu sistema de janelas, como por exemplo: **Windows, Linux** e *Macintosh*;

- **Utilizado por diferentes linguagem de programação:** pode ser utilizado em conjunto com diversas linguagens existentes no mercado, dentre elas: **C/C++**, Delphi, Python e Java.

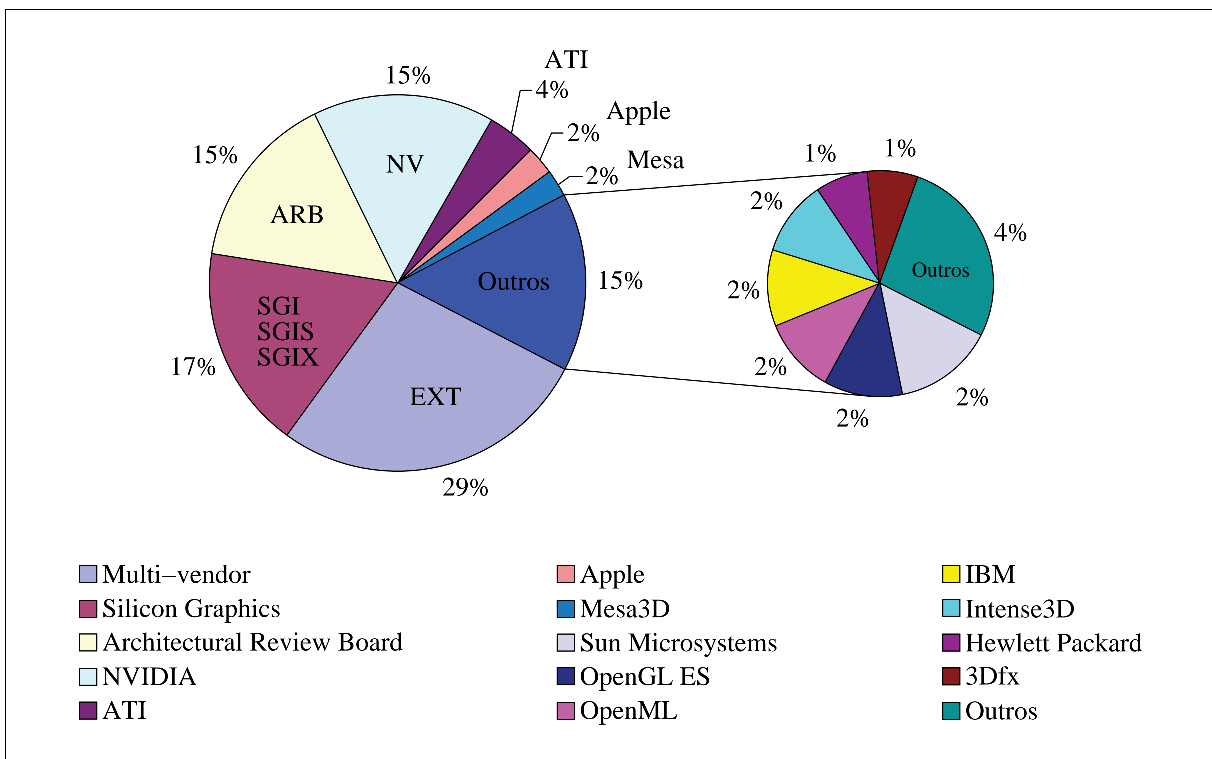


Figura 31: Desenvolvedores de Extensões para o OpenGL
Fonte: Kilgard et al. (2008)

Ainda segundo Wright, Lipchak e Haemel (2007), **OpenGL** é projetado utilizando a arquitetura Cliente-Servidor. A natureza do cliente e do servidor, assim como o meio de comunicação existente entre eles é específica para cada implementação do **OpenGL**, por exemplo, o servidor e o cliente podem estar em máquinas distintas, ou então serem processos distintos na mesma máquina. No caso da plataforma PC, o cliente é a **CPU** e a memória **RAM**; o servidor é a placa gráfica (GPU e memória de vídeo) sendo que ambos estão locados na mesma máquina.

Na Figura 32 é apresentado graficamente, como **OpenGL** é "visto" por uma aplicação do usuário. Neste caso, observa-se que o **OpenGL** funciona como uma camada intermediária entre a aplicação do usúario e o *hardware* gráfico, livrando o desenvolvedor de ter de lidar com toda a complexidade associada ao mesmo. Percebe-se ainda que o *hardware* gráfico oferece suporte ao **OpenGL** por meio de *drivers*.

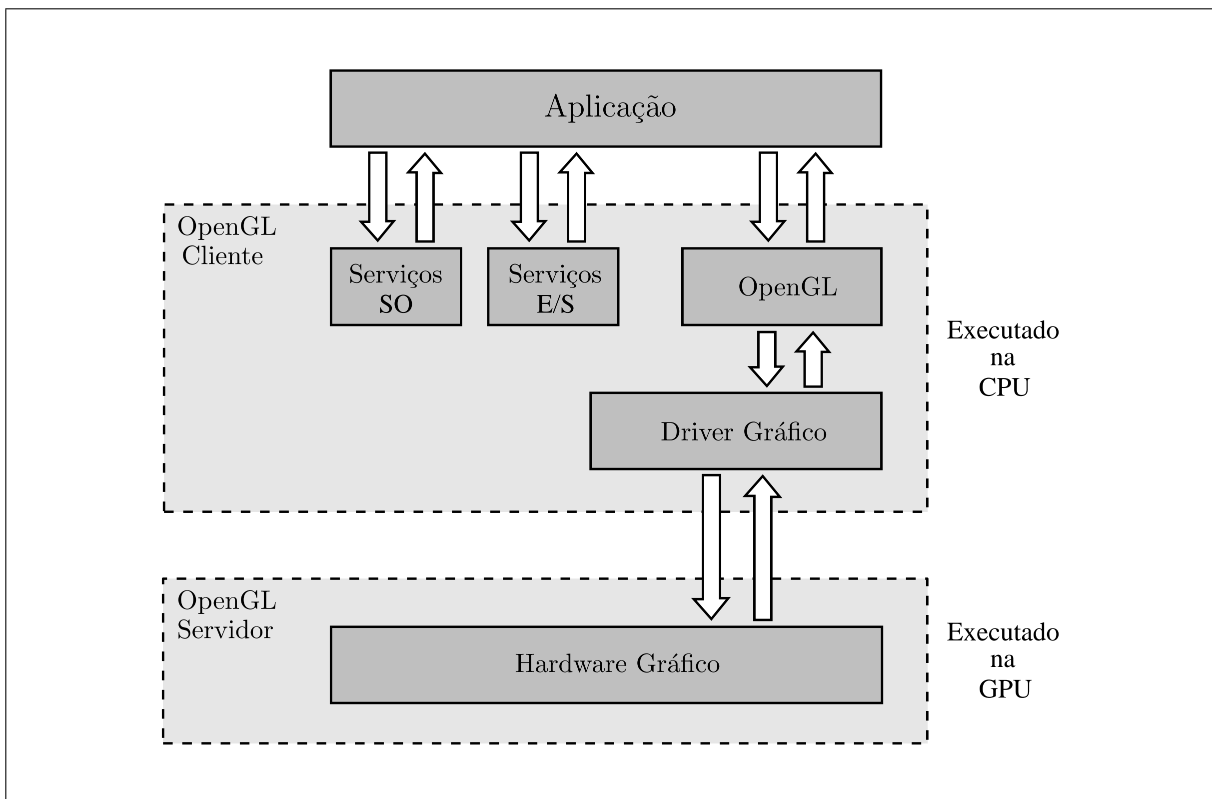


Figura 32: Arquitetura Cliente-Servidor do OpenGL

Também é mostrada na Figura 32 a arquitetura Cliente-Servidor do **OpenGL**. Neste caso toda chamada da aplicação do usuário para o **OpenGL** é feita para no lado cliente (CPU), que faz um processamento dos comandos passados (ex: cópia e reformatação de dados) e os repassa para o servidor (GPU).

O objetivo da arquitetura Cliente-Servidor do **OpenGL**, no caso da plataforma PC, é dividir o trabalho entre a CPU (cliente) e a GPU (servidor) de tal forma que os seus comandos possam ser executados de maneira assíncrona. Desta maneira o lado cliente do **OpenGL** pode retornar o controle de execução para a aplicação antes que um dado comando termine de ser executado. Essa execução assíncrona é necessária uma vez que se todos os comandos do **OpenGL** tiverem que ser executados antes de retornar o controle para a aplicação, ambos cliente ou servidor podem ficar ociosos esperando dados um do outro.

O modo de programação do **OpenGL** prioriza a renderização, sendo que geralmente são seguidos os seguintes passos:

- especificação de objetos (pontos, linhas, polígonos etc);
- especificação das propriedades desses objetos (cor, textura, materiais etc...);
- formação de uma imagem desses objetos usando uma câmera virtual.

De acordo com Graphics (1998), o **OpenGL** oferece as seguintes vantagens como ferramenta de desenvolvimento de aplicações gráficas:

- **Padrão da Indústria:** como o **OpenGL** é especificado e mantido por um consórcio de empresas e implementado pela própria indústria de *hardware*, ele acaba se tornado um padrão de **API** gráfica multiplataforma;
- **Estabilidade:** o **OpenGL** já vem sendo implementado há mais de 17 anos em diferentes plataformas; existindo um controle rigoroso na adição de novas funcionalidades sempre mantendo a compatibilidade com versões anteriores pelo uso da extensão *GL_ARB_compatibility*;
- **Confiável e Portável:** toda a aplicação que faça uso do **OpenGL**, irá produzir o mesmo resultado visual em qualquer *hardware* que dê suporte ao **OpenGL**, independentemente do sistema operacional ou sistema de janelas utilizado;
- **Escalável:** aplicações que utilizam o **OpenGL** podem ser executadas desde a plataforma PC até grandes estações de trabalho (*workstations*) e super-computadores, ou seja, a aplicação pode ser escalada para qualquer classe de máquina que dê suporte a esse padrão;

- **Boa Documentação:** além da boa quantidade de livros publicados sobre o **OpenGL**, também existe uma boa quantidade de exemplos de código e tutoriais disponíveis na internet, facilitando a obtenção de diferentes tipos de informação.

Dentre os recursos gráficos disponíveis pelo **OpenGL**, podem ser destacados a renderização 3D das primitivas pontos, linhas e polígonos, mapeamento de superfícies com textura, modelos de iluminação, transformações geométricas (rotação, translação e escala) e de sistemas de coordenadas, gerenciamento de câmeras virtuais (aplicação de transformações de perspectiva) e efeitos de transparência, neblina (*fog*), mistura de cores (*blending*) e anti-serrilhamento (*antialiase*).

O **OpenGL**, em conjunto com a biblioteca **OSG**, foi utilizado na implementação das funcionalidades de visualização 3D de malhas triangulares (**TIN**) no sistema desenvolvido. Na Seção 5.2 na página 84, é dada uma visão geral da arquitetura do sistema e de como o **OpenGL** está inserido dentro dela, enquanto na Seção 5.6 na página 110, é feita uma descrição detalhada de como os dados de um **TIN** são gerenciados pelo **OpenGL** de tal forma a melhorar o desempenho da sua renderização.

5.1.3 OpenSceneGraph

Um grafo de cena é uma estrutura de dados na forma de árvore que permite a organização hierárquica e espacial de objetos que constituem uma cena 3D de tal forma a melhorar a eficiência da sua renderização (MARTZ, 2007).

Basicamente um grafo de cena contem informações que definem um mundo virtual, possuindo desde descrições de baixo nível sobre a geometria e aparência de objetos na cena, a informações espaciais de alto nível como posicionamento, orientação, animações e transformações dos objetos que constituem a cena, assim como dados de aplicações do usuário (ECKEL; JONES, 2004).

Toda essa informação é encapsulada dentro dos diferentes tipos de nodos que compõem o grafo de cena. Cada tipo de nodo pode conter informações específicas sobre a descrição da cena 3D modelada como:

- **Descrição Geométrica:** define a maneira como o objeto é representado; por exemplo, o tipo de geometria utilizada (pontos, linhas, conjunto de polígonos), raio, tamanho, etc;
- **Câmera:** define a visão que o usuário tem da cena virtual;

- **Transformações:** posiciona objetos no mundo virtual por meio de transformações geométricas como: translação, rotação e escala;
- **Aparência:** define os atributos de aparência dos objetos como: cor, material, textura, sombra, parâmetros de reflexão e transparência;
- **Iluminação:** define os modelos de iluminação utilizados (ex: *Flat Shading*, *Phong* e *Gouraud*);
- **Comportamento:** definem o comportamento de objetos dinâmicos (objetos que mudam de posição ou de forma entre um quadro e outro);

Na Figura 33 é mostrado como uma cena 3D pode ser modelada por meio de um grafo de cena:

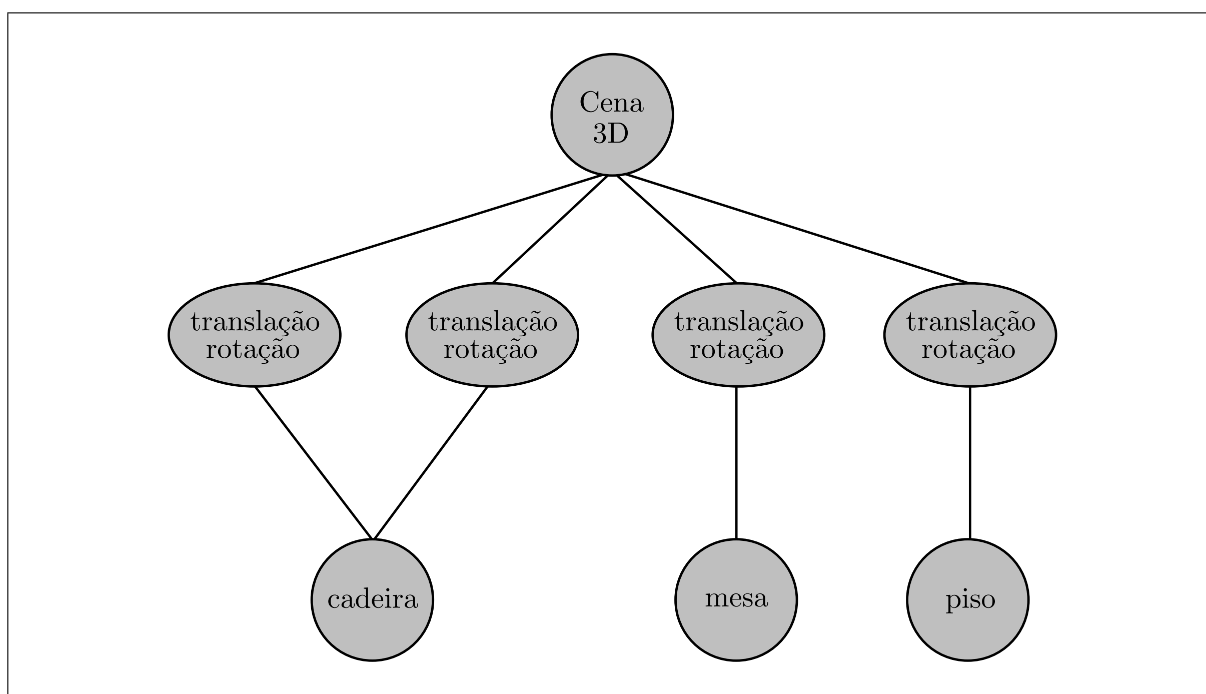


Figura 33: Representação de cenas virtuais por meio de grafos de cena.

Esse grafo representa uma cena composta por um piso, uma mesa e duas cadeiras. O nodo raiz representa a cena como um todo, desta forma ele possui quatro nodos filhos representando o piso, a mesa e as duas cadeiras que compõem a cena. Esses tipo de nodos em particular, aplicam transformações geométricas (rotação e translação) posicionando e orientando os seus nodos filhos, nesse caso os nodos folha do grafo que contêm a descrição geométrica de cada objeto da cena. Nota-se ainda na Figura 33, que existe apenas um nodo folha representando os dois objetos cadeira, isto porque na cena representada, as

duas cadeiras são idênticas (possuem os mesmos dados de geometria); nesse caso os seus dois nodos pai posicionam a cadeira em dois locais diferentes, produzindo dessa forma a ilusão de duas cadeiras na cena.

Além da questão do ganho de eficiência na renderização, segundo Silva, Raposo e Gattass (2004) e Martz (2007) o uso de grafos de cena ainda possui as seguintes vantagens:

- **Produtividade:** como o grafo de cena é responsável pelo gerenciamento de toda a parte gráfica, é necessário um número menor de linhas de código para criar uma cena do que utilizando uma interface de programação baixo nível como o **OpenGL**. Ainda, estes fornecem funcionalidades adicionais de alto-nível, não oferecidas por APIs de baixo nível, como: renderização de texto, efeitos de partículas, sombras e suporte à abertura de diferentes formatos de arquivos de modelos 3D;
- **Portabilidade:** grafos de cena encapsulam funcionalidades multi-plataforma de baixo nível, como acesso à superfícies de renderização e dispositivos de entrada e saída, podendo reduzir ou até mesmo extinguir a necessidade da produção de código específico para determinadas plataformas.
- **Escalabilidade:** grafos de cena geralmente são projetados para funcionar tanto em configurações simples, como computadores de mesa e placas gráficas aceleradoras convencionais, como em *hardware* complexo, como *clusters* de máquinas gráficas ou ou sistemas multiprocessados.

Como o **OpenGL** não oferece nenhum tipo de ferramenta global em termos de gerenciamento de cenas 3D e dadas as vantagens associadas ao uso de grafo de cena, optou-se pela utilização o **OSG**, na implementação da visualização dos **TIN**s gerados pelo sistemas.

O **OSG** é uma biblioteca gráfica 3D livre de alta performance, que constrói um grafo de cena sobre o **OpenGL**, permitindo a montagem e gerenciamento eficiente de cenas 3D. Ele é inteiramente escrito em **C/C++**, utilizando a **STL**, conceitos de orientação a objetos além de implementar diferentes padrões de projetos como por exemplo, cadeia de responsabilidade (*Chain of responsability Pattern*), composição(*Composite Pattern*), visitador(*Visitor Pattern*), observador (*Observer Pattern*), adaptador (*Adaptor Pattern*) e decorador (*Decorator Pattern*) (MARTZ, 2007; SILVA et al., 2003).

O uso dessa biblioteca irá facilitar o desenvolvimento da parte de visualização do sistema visto além das funcionades de renderização fornecidas por meio do **OpenGL**, ela também fornece métodos para a execução de testes de intersecção do mouse com a cena

(interação do usuário com a cena 3D), renderização de texto em cenas 3D, *bilboarding* e rotinas para o carregamento de imagens/texturas. Além disso, o **OSG** ainda possui ferramentas para a criação de primitivas particularmente eficientes para a renderização de triangulações como *triangle strips* e *triangles fan* (HJELLE; DAEHLEN, 2006), assim como rotinas de visibilidade e eliminação de vértices (*Frustum Culling*, *Occlusion Culling*, *Level of Detail*) diminuindo dessa forma o número total de vértices passado da aplicação para a placa de vídeo melhorando de forma significativa o desempenho e a interatividade do usuário com o sistema, que são pontos críticos em qualquer sistema de *software*.

A biblioteca **OSG** foi utilizada na implementação das funcionalidades de visualização 3D de malhas triangulares (**TIN**) no sistema desenvolvido. Na Seção 5.2 na página 84, é dada uma visão geral da arquitetura do sistema e de como o **OSG** está inserido dentro dele, enquanto na Seção 5.6 na página 110, é feita uma descrição detalhada de como ele é utilizado na implementação das rotinas de visualização e interação do usuário com **TIN**s gerados pelo sistema.

5.1.4 Biblioteca GDAL/OGR

A **GDAL** e **OGR** são bibliotecas de *software opensource* escritas em **C/C++**, que oferecem suporte à leitura e escrita de uma grande variedade de formatos de arquivos de dados espaciais raster e vetoriais respectivamente, assim como ferramentas de linha de comando (aplicações console) para o processamento e conversão entre esses formatos (GDAL, 2010).

Apesar de em teoria essas bibliotecas estarem separadas (possuem modelos de dados e **API** diferentes), ambas residem e são distribuídas na forma de uma única biblioteca de *software* (WARMERDAM, 2008), sendo que futuramente pode ser efetuado uma fusão da **OGR** com a **GDAL** de tal forma que a **GDAL** se torne uma biblioteca de leitura e escrita de dados raster e vetorial (GDAL, 2010).

Apesar dessas bibliotecas serem desenvolvidas em **C/C++**, elas também podem ser utilizadas com outras linguagens de programação como: Ruby, Perl, Python, Java e C#. Essas bibliotecas também são muito utilizadas tanto no desenvolvimento de *software* livre como comercial, fazendo parte de aplicações como *ERDAS ER Viewer*, *ESRI ArcGIS 9.2*, *Google Earth*, *GRASS GIS*, *IDRISI* e *MapServer*.

De uma maneira geral, as bibliotecas **GDAL** e **OGR** possuem as seguintes características (GDAL, 2010; WARMERDAM, 2008):

- **Modelo de dados unificado:** ambas bibliotecas fornecem um modelo de dados único, para a leitura e escrita de diferentes formatos de arquivos raster e vetoriais;
- **Conformidade com Open Geospatial Consortium (OGC):** dado o fato da OCG não fornecer padronização específicas para **API** C++, essas bibliotecas tentam seguir todas as especificações OGC para a representação de dados raster e vetoriais em banco de dados espaciais;
- **Não necessidade de configuração:** não é esperado do usuário nenhum conhecimento sobre o formato do arquivo que será aberto ou algum tipo de configuração para determinados formatos de arquivo;
- **Portabilidade:** essas bibliotecas podem ser utilizadas nas plataformas: *Linux, Solaris, Mac OS X, Solaris* e *Windows*(NT/2000/XP/Vista/CE) sendo suportadas as arquiteturas de 32 e 64 bits;
- **Minimização de perdas de desempenho:** ambas bibliotecas são desenvolvidas levando em consideração não somente a questão da quantidade de formatos de arquivos suportados, mas também a questão do desempenho, de tal forma a evitar que o usuário sinta a necessidade de criar seus próprios métodos de leitura para um formato de arquivo raster ou vetorial para obter uma performance melhor;
- **Suporte a banco de dados:** além do suporte a leitura e escrita de dados na forma de arquivos, essas bibliotecas também podem ser utilizadas para o acesso a dados guardados em Sistemas Gerenciadores de Banco de Dados ou em servidores remotos.

As bliotecas **GDAL** e **OGR** foram utilizadas na implementação do modulo responsável pela leitura de arquivos do sistema desenvolvido. Na Seção 5.2, é dada uma visão geral da arquitetura do sistema e de como essas bibliotecas estão inseridas dentro dela, enquanto na Seção 5.5, é feita uma descrição dessas bibliotecas e quais partes do sistema foram desenvolvidas utilizando elas.

5.1.5 Sistema Gerenciador de Janelas Qt

O **Qt** pode ser considerado como um padrão da indústria de alto desempenho, possuindo um grande conjunto de *widgets* (controles na terminologia *Windows*) que são amplamente utilizados em aplicações atuais como menus, menus de contexto, botões e *dockable toolbars* que são os componentes mais usados no contexto desse projeto.

Apesar de ser completamente escrito em C++, ele pode ser utilizado com outras linguagens como, por exemplo: C, Python, C# e Java (Qt Jambi) permitindo ainda ao programador usar o mesmo código fonte para criar aplicações executáveis em código nativo nas seguintes plataformas: *Windows*, *Mac OS X, Linux*, Sistemas Embarcados com *Linux (QTopia), Solaris, HP-UX* e versões do *Unix* com *X11*.

Ele tem sido utilizado intensamente desde 1995 por diversas grandes companhias e organizações como: *Adobe, Boeing, IBM, Motorola, NASA, Skype* e outras. Dentre as aplicações comerciais criadas usando o **Qt** estão incluídas, ferramentas de animação 3D, processamento digital de filmes, design e automação eletrônica (para design de chip), exploração de petróleo e gás, serviços financeiros e processamento de imagens médicas. E mais, também é amplamente usado pela comunidade *opensource* já que ele é a base do *KDE Linux Desktop Environment*.

Também é importante realçar que as tecnologias **Qt**, **OpenGL** e **OSG** estão perfeitamente integradas, não sendo necessário fazer, por exemplo, nenhum tipo de conversão para mostrar uma figura (nesse caso um terreno) renderizada pelo **OpenGL/OSG** numa janela do **Qt**, aumentando ainda mais a eficiência e a robustez do sistema.

O **Qt** foi utilizado na implementação da interface gráfica do sistema desenvolvido. Na Seção 5.2, é dada uma visão geral da arquitetura do sistema e de como o **Qt** está inserido nela, enquanto na Seção 5.7, é feita uma descrição de alguns detalhes de implementação e funcionades da interface gráfica do sistema.

5.2 Arquitetura do Sistema

Na Figura 34 é dada visão geral da arquitetura do sistema e da integração de seus componentes. Como pode ser observado, foi feita uma divisão conceitual do sistema proposto em cinco módulos:

Representação Computacional de TIN: esse módulo é responsável pela criação de um **TIN** por meio da técnica de Triangulação de Delaunay e pela sua representação computacional por meio de uma **EDT**. É interessante notar todos os outros componentes do sistema têm acesso a esse módulo sem ter que se preocupar como **TIN** é construído ou representado. Como pode ser visto na Figura 34, foram utilizadas a técnica de Triangulação de Delaunay com *constraints* e a **EDT TDS**, fornecidas pela biblioteca **CGAL**, na geração e manutenção de **TIN**s no sistema desenvolvido. Entretanto futuramente esse módulo poderá ser utilizado como uma camada de acesso unificada para implementações de técnicas de triangulação e **EDTs** fornecidas por outras bibliotecas de *software* como por exemplo a **TTL**.

Interface Gráfica com o Usuário: é por meio desse módulo que o usuário tem acesso à representação computacional de um **TIN** gerado pelo sistema assim como à todas aplicações que poderão ser construídas sobre o mesmo. Esse módulo foi desenvolvido utilizando a biblioteca **Qt**.

Serviços de Leitura e Escrita de Arquivos: esse módulo engloba todas funcionalidades de leitura e escrita de diferentes formatos de arquivos raster e vetoriais no sistema desenvolvido, garantindo assim a sua integração com outras plataformas de *software*. Como é mostrado na Figura 34, esse módulo foi desenvolvido utilizando as bibliotecas de *software* **GDAL** e **OGR**.

Visualização 3D de Terrenos: esse módulo é responsável pela visualização 3D dos **MDTs** representado a partir de **TIN**s gerados pelo sistema. O módulo de visualização foi implementado utilizando as bibliotecas **OpenGL** e **OSG**.

Aplicações: o modulo de aplicações representa todas possíveis aplicações que poderão ser implementadas a partir das funcionalidades fornecidas pelos módulos anteriormente descritos.

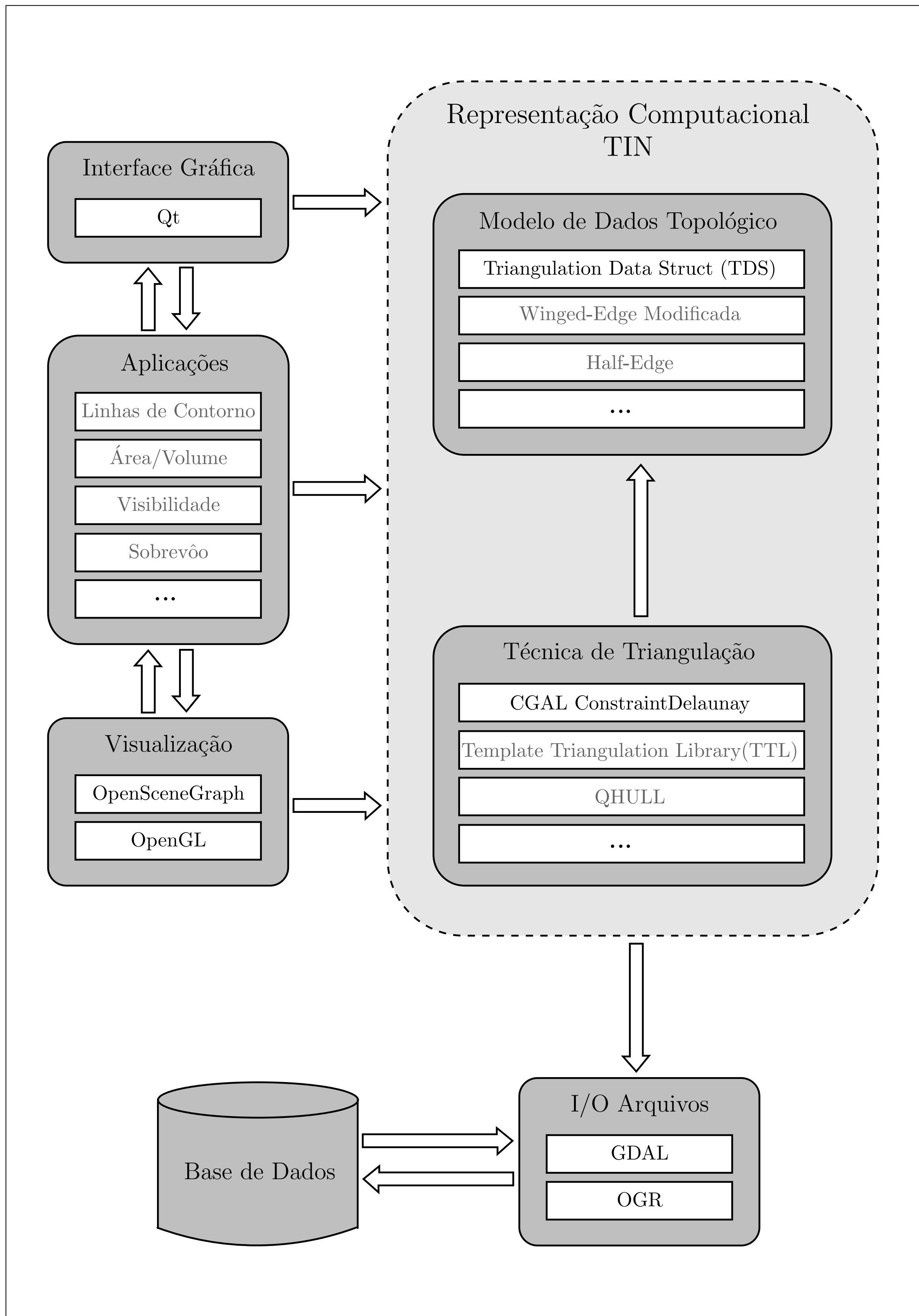


Figura 34: Modelo Esquemático da Arquitetura do Sistema Proposto

5.3 Integração entre os Componentes do Sistema

Separadamente, todos os módulos e tecnologias discutidas anteriormente não fazem nenhum sentido para o sistema, sendo assim é preciso criar uma estrutura lógica para fazer com que essas tecnologias e módulos "conversem entre sí". Considerando a dimensão do projeto e a diversidade de componentes agregados para a sua constituição, o sistema foi desenvolvido obedecendo aos paradigmas de orientação a objetos e programação genérica, sendo que quando possível também foram utilizados padrões de projeto.

5.3.1 Programação Orientada a Objetos

A programação orientada a objetos já pode ser considerada uma tecnologia madura e bem estabelecida, cujos conceitos têm sido estudados desde o final da década de 60. Em particular nos últimos dez anos têm surgido muitas metodologias e estudos visando potencializar o seu uso, além de um considerável crescimento na utilização e aceitação dessa tecnologia em toda a indústria de *software*, de tal forma que a mesma pode ser encontrada em grandes áreas, como: banco de dados, redes, internet e interfaces gráficas com o usuário (O'DOCHERTY, 2005; SINTES, 2002).

A programação orientada a objetos oferece um mecanismo poderoso para se lidar com a complexidade de um projeto computacional, fornecendo ferramentas para tornar os programas mais claros, seguros e favorecendo a manutenção destes. Ainda, esta é particularmente conveniente para o desenvolvimento de aplicações gráficas de uma maneira geral. Isto porque, componentes desses programas podem ser implementados na forma de objetos intuitivos (O'DOCHERTY, 2005). Por exemplo, um ponto ou segmento de reta podem ser realizados de forma direta, como componentes gráficos que possuem dados como posição, tamanho e cores, assim como, métodos para mudar os seus valores e para exibir o componente em si, não existindo uma separação entre os dados e os métodos ou funções que fazem algum tipo de processamento sobre os mesmos.

Como pode ser observado nas Seções 5.4, 5.5, 5.6 e 5.7, foram utilizados extensivamente os conceitos de herança e polimorfismo da programação orientada a objetos no desenvolvimento do sistema proposto.

5.3.2 Programação Genérica

Apesar da orientação a objetos fornecer ferramentas para se lidar com toda a complexidade relacionada ao desenvolvimento de *software*, em alguns casos não é vantajoso o acoplamento entre os algoritmos e estruturas de dados fornecido pelo desenvolvimento de *software* orientado a objetos. Por exemplo, segundo Vinhas et al. (2002), no desenvolvimento de *software* **SIG**, um grande número de algoritmos não dependem da implementação de uma estrutura de dados em particular, mas apenas de algumas propriedades semânticas fundamentais destas como a habilidade de ir de um elemento da estrutura ao próximo ou de comparar dois elementos. Por isso o desenvolvimento de **SIG** pode ser melhorado através da combinação da idéia das técnicas de orientação a objeto com o paradigma de programação genérica.

De acordo com Pirkelbauer et al. (2008), a programação genérica tem se consolidado como uma importante ferramenta no desenvolvimento de bibliotecas de *software* com alto grau de eficiência e reusabilidade isto porque, a programação genérica é voltada para a construção de algoritmos que independem de um determinado tipo ou estrutura de dados e que sejam tão eficientes (com relação a velocidade de execução e utilização de recursos) quanto, a implementação desses mesmos algoritmos acoplados a determinados tipos ou estruturas de dados (VINHAS et al., 2002).

No desenvolvimento do sistema proposto, foi efetuado o uso da programação genérica no desenvolvimento da funcionalidade de agregação de dados do usuário no componente de *software* responsável pela criação e manutenção de **TIN**s (vide Seção 5.4.2 na página 96).

5.3.3 Padrões de Projeto

O desenvolvimento de *software* orientado a objetos é uma tarefa difícil, o *software* orientado a objetos reutilizável é mais difícil ainda. Sendo assim, tendo por objetivo conseguir um maior aproveitamento desse paradigma de programação também foram utilizados alguns padrões de projeto no desenvolvimento do sistema proposto.

Padrões de projeto provêem soluções elegantes, de fácil reuso e manutenção, para problemas comuns da área de programação. A seguir são dadas algumas definições para o termo padrão de projetos (BUCHMANN et al., 1996; GAMMA et al., 1998; LASATER, 2007):

- "São soluções recorrentes para problemas que ocorrem com maior frequência na área da programação";

- "Conjunto de regras descrevendo como completar determinadas tarefas na área de desenvolvimento de *software*";
- "Identificam e especificam abstrações que estão acima do nível de classes e instância, ou componentes".

Dentro do contexto da programação orientada a objetos, o uso de padrões de projeto não apenas auxilia na identificação e especificação de objetos úteis na solução de determinados problemas como também oferece soluções para o gerenciamento da comunicação entre objetos, e metodologias para o desenvolvimento de *software* visando a sua extensibilidade (futuras mudanças no código não devem afetar partes já resolvidas). Mais especificamente, os padrões de projetos podem ser divididos em três categorias: criacionais, estruturais e comportamentais.

Padrões Criacionais: contêm indicações sobre a melhor maneira (melhores lugares e qual a ordem) de se criar instâncias de um objeto em sistemas de *software* complexos.

Padrões estruturais: contêm descrições de como ferramentas da orientação a objetos (como herança e polimorfismo) podem ser utilizadas na criação de estruturas maiores a partir da combinação de classes e objetos. Por exemplo, como combinar as estruturas de dados e algoritmos fornecidos pela **CGAL** e pela **STL** na criação do modelo computacional que consiga representar a superfície de um terreno usando **TIN** (como os dados serão guardados ou compartilhados por essas estruturas).

Padrões Comportamentais: se preocupam e modelar a troca de informações entre objetos. Por exemplo, se o usuário clicar com ou mouse sobre uma tela do sistema, qual componente de *software* será responsável por capturar esse evento, quem será responsável por processar esse evento, ou seja, como os diferentes componentes do módulo de visualização do sistema (**Qt**, o **OSG** e o **OpenGL**) irão se comportar (interagir entre si) mediante os eventos gerados pelo usuário.

Resumindo, padrões de projeto resolvem muitos problemas que programadores orientados a objeto podem se deparar por representar uma experiência coletiva de boas práticas de programação e resolução de determinados problemas recorrentes da área de desenvolvimento e projeto de *software*, permitido a estes aprenderem com o sucesso de outros programadores ao invés de com seus próprios fracassos.

No presente trabalho foram utilizados os seguintes padrões de projeto no desenvolvimento do sistema proposto:

- **Padrão de Projeto Iterador (*Iterator*)**: esse padrão de projeto foi utilizado no acesso ao conjunto de vértices, arestas e faces contidos na estrutura de dados responsável pela representação computacional do **TIN**, e também no acesso aos dados (nesse caso pontos com coordenadas (x, y, z)) de superfícies contidos em arquivos em disco. A Seção 5.4.4 contem uma descrição mais detalhada desse padrão de projeto e de como ele está inserido no sistema desenvolvido.

- **Padrão de Projeto *Factory***: esse padrão de projeto foi utilizado no desenvolvimento do módulo de *software* responsável pela leitura de diferentes formatos de arquivos raster e vetorial pelo sistema. Uma melhor descrição desse padrão e o seu uso é dado na Seção 5.5.1.

Além dos padrões de projeto citados acima, também foram utilizados os seguintes conceitos durante o desenvolvimento do sistema:

- **Handler:** o conceito de *handler* foi utilizado no acesso individual aos elementos vértices, aresta e faces armazenados na estrutura de dados utilizada na representação computacional do **TIN**. São dados mais detalhes sobre esse conceito e a sua utilização na Seção 5.4.3.

- **Circuladores (*Circulators*)**: esse conceito foi utilizado na implementação das classes responsáveis pelo acesso às relações de conectividade dos vértices e faces do **TIN** representado pelo sistema. Maiores detalhes sobre esse conceito e a sua inserção no sistema são dados na Seção 5.4.5.

- **Máquina de Estado Hierárquica (*Statechart*)**: todo o sistema de controle da interface gráfica do sistema desenvolvido foi criado utilizando máquinas de estado hierárquicas. Na Seção 5.7 é dada uma melhor descrição desse conceito e a sua inserção no sistema.

5.4 Modelo Computacional TIN

No presente trabalho a representação de um modelo **TIN** é feita pela classe *Triangulation*. Essa classe classe é responsável pela criação, representação e manipulação computacional de um **TIN**.

A classe *Triangulation* foi desenvolvida tendo como objetivo, independentemente da biblioteca de triangulação ou **EDT** utilizada, fornecer as seguintes funcionalidades:

- Dado um conjunto de pontos gerar um **TIN** por meio de uma Triangulação de Delaunay;
- Fornecer uma representação explícita para as entidades vértices, arestas e faces do modelo **TIN**, mesmo que as estruturas que de fato representem esse modelo não possuam uma representação explícita para essas entidades (como a **TDS** que representa arestas apenas implicitamente);
- Permitir consultas eficientes em relação a topologia da triangulação gerada;
- Possuir um alto grau de customização permitindo ao usuário adicionar informações à entidades vértice, aresta e face do modelo **TIN** por meio de programação genérica.

Como é mostrado na Figura 35, a classe *Triangulation* foi criada tendo como objetivo fornecer uma representação de modelo **TIN** que sirva como base para a implementação de aplicações e algoritmos que atuem sobre este.

Basicamente essa classe funciona da seguinte forma: dado um conjunto de pontos, regular ou irregularmente espaçados, no plano, a criação do **TIN** é feita utilizando o algoritmo de Triangulação de Delaunay. Toda representação computacional das informações geométricas e topológicas é feita por meio de uma **EDT**.

Entretanto no presente trabalho não foi implementado nenhum software para geração de triangulações ou **EDT** para a representação computacional destas. De fato, conforme é mostrado na Figura 35, observa-se que a classe *Triangulation* pode ser construída sobre diferentes bibliotecas de *software* existentes, de geração e representação computacional de malhas triangulares. Isto porque, a classe *Triangulation* foi projetada para funcionar como uma camada de acesso unificada às funcionalidades existentes nessas bibliotecas, escondendo desta maneira toda a sua complexidade associada, fornecendo uma interface mais amigável ao usuário.

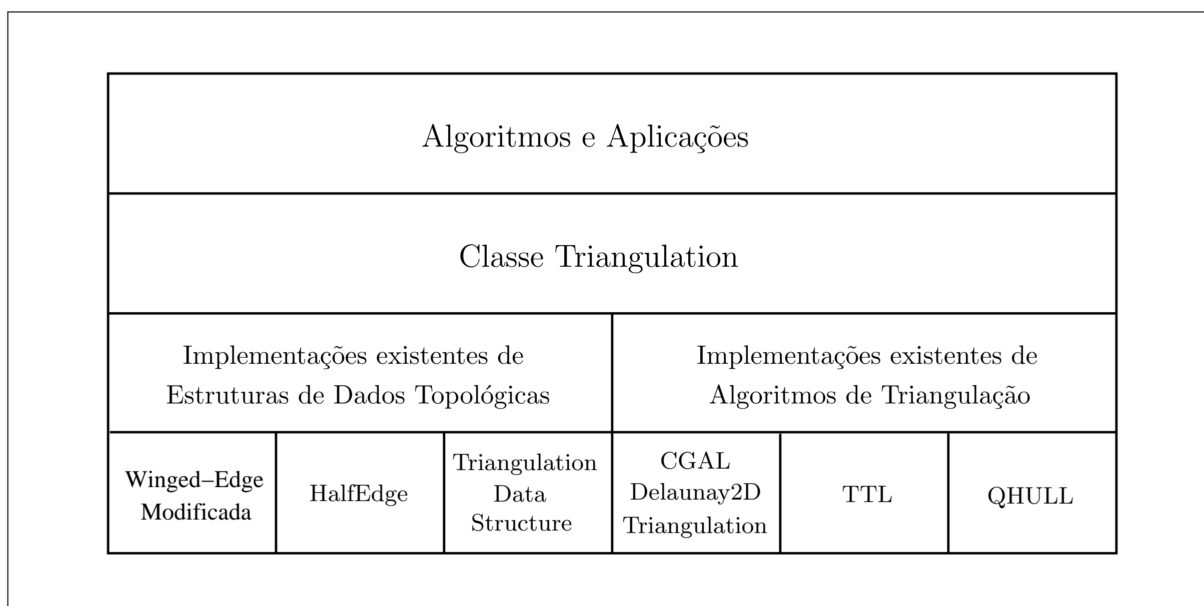


Figura 35: Arquitetura Modelo de Dados.

Nesse primeiro momento, todos os aspectos relativos à construção, e manipulação da estrutura de dados topológica que irá representar uma triangulação de pontos irregularmente distribuídos no espaço (**TIN**), assim como a geração da triangulação propriamente dita, foram feitos com o auxílio da biblioteca **CGAL**. O pacote de triangulação da **CGAL** dá suporte tanto para triangulação de conjuntos de pontos no plano (bidimensional), como para triangulação de pontos no espaço (tridimensional).

No plano, a **CGAL** oferece algoritmos incrementais para triangulação básica (não existe controle sobre a qualidade das faces triangulares geradas), de Delaunay (com ou sem linhas de quebra) e Triangulações Regulares. Sendo a sua representação feita através da estrutura de dados topológica **TDS**.

O pacote de triangulação bidimensional também oferece funcionalidades como localização de pontos, inserção e remoção de vértices na triangulação assim como rotinas ligadas a topologia da triangulação como, por exemplo, encontrar todas as faces adjacentes a um vértice. Ainda, dada a importância da triangulação no desenvolvimento de *software* **SIG**, em particular na geração de modelos digitais de terrenos, esse pacote ainda permite o uso de pontos tridimensionais na triangulação 2D de tal forma que desenvolvedores de aplicações **SIG** possam computar malhas triangulares de pontos pertencentes ao terreno de forma direta, sem ter que primeiro projetar os pontos no plano, executar a triangulação e finalmente voltar com os pontos em suas coordenadas originais, ou seja, todas as operações (ex: testes de orientação, localização de pontos e comparação entre coordenadas)

são computadas usando apenas as coordenadas (x, y) do conjunto de pontos fornecidos (BOISSONNAT et al., 2002).

No trecho de código a seguir é dado um exemplo de como gerar uma triangulação utilizando a **CGAL**:

```
typedef CGAL::Simple_cartesian<double>                                   CK;
typedef CGAL::Filtered_kernel<CK>                                        K;
typedef CGAL::Triangulation_euclidean_traits_xy_3<K>                     Gt;
typedef CGAL::Triangulation_vertex_base_2<Gt>                            Vb;
typedef CGAL::Constrained_triangulation_face_base_2<Gt>                  Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>                      Tds;
typedef CGAL::No_intersection_tag                                        Itag;
typedef CGAL::Constrained_Delaunay_triangulation_2<Gt,Tds,Itag>         CgalTriangulation;

typedef CgalTriangulation::Point_3<K>                                    Point3D;

...

//cria a triangulação
CgalTriangulation trg;

//insere pontos na triangulação
trg->insert(Point3D(x1,y1,z1));
trg->insert(Point3D(x1,y1,z1));
...
trg->insert(Point3D(xn,yn,zn));


//insere constraints (cada constraint é definido por dois pontos)
trg->insertConstraint(Point3D(cx1,cy1,cz1), Point3D(cx2,cy2,cz2));
trg->insertConstraint(Point3D(cx3,cy3,cz3), Point3D(cx4,cy4,cz4));
...
trg->insertConstraint(Point3D(cxn,cyn,czn), ...);
```

Nas linhas 1 e 2 é criado um *kernel* que suporta a construção de objetos geométricos (no caso da presente triangulação pontos) utilizando coordenadas cartesianas do tipo *double*, e que faz o cômputo de predicados geométricos usando tipos exatos com filtragem (vide Seção 5.1.1.5 na página 73 sobre tipos exatos com filtragem).

Na linha 3, é dito que apesar de estar sendo utilizada uma técnica de Triangulação de Delaunay bidimensional, os pontos usados como entrada são tridimensionais. Neste caso, a coordenada z desses pontos deve ser ignorada, sendo utilizadas apenas as coordenadas (x, y), o que equivale a projeção dos pontos de entrada no plano (x, y), para a efetuação da triangulação (vide Figura 28 na página 69).

Nas linhas 4, 5 e 6 são definidas respectivamente: as classes responsáveis pela representação das entidades vértice e face da triangulação, e a **EDT** responsável pela representação da triangulação propriamente dita, isto é, a **TDS**. Ainda é interessante notar na linha 6 que a **TDS** é definida utilizando apenas os tipos dados criados para a representação as entidades vértice e face (linhas 4 e 5); como a **TDS** não representa arestas de forma explícita (vide Seção 3.2.4 na página 52), esta não recebe como parâmetro nenhum tipo de dado que represente a entidade aresta.

Finalmente na linha 8 é definido o tipo de dado utilizado para geração do **TIN** utilizando a técnica de Triangulação de Delaunay com *constraint* (linhas de quebra). Os três parâmetros utilizados na sua definição indicam que esse tipo respectivamente, receberá como entrada pontos tridimensionais, será utilizada a **EDT TDS** para a representação da triangulação gerada e que não será permitida a intersecção entre linhas de quebra.

Na linha 10, é recuperado o tipo geométrico ponto (utilizado na representação das coordenadas dos vértices da triangulação) para que usuário possa entrar com as coordenadas dos os pontos e linhas de quebra que serão utilizados na geração do **TIN**.

Por fim, nas linhas 15 a 28, é dado um exemplo de como criar e inserir pontos (linhas 18 a 21) e linhas de quebra (linhas 25 a 28) numa triangulação representada pelo tipo de dado definido na linha 8.

Como um dos objetivos desse trabalho é o reaproveitamento de código a classe *Triangulation* foi desenvolvida na forma de uma biblioteca de *software* de tal modo que esta também possa ser utilizada em outros projetos. Na Figura 36 na próxima página é mostrado um modelo UML da classe *Triangulation* e suas principais classes associadas.

Como já foi dito anteriormente, a classe *Triangulation* concentra toda informação sobre a construção e representação de um **TIN**. A seguir é dada uma breve descrição de alguns de seus principais métodos:

- **insert():** insere um ponto na triangulação;
- **insertContour():** insere um conjunto de pontos que definem um contorno na triangulação;
- **getNumberVertices():** retorna o total de vértices da triangulação;
- **getNumberEdges():** retorna o total de arestas da triangulação;
- **getNumberFaces():** retorna o total de faces da triangulação;

- **getMinHeight():** retorna o menor valor da coordenada z dos vértices da triangulação;

- **getMaxHeight():** retorna o maior valor da coordenada z dos vértices da triangulação;

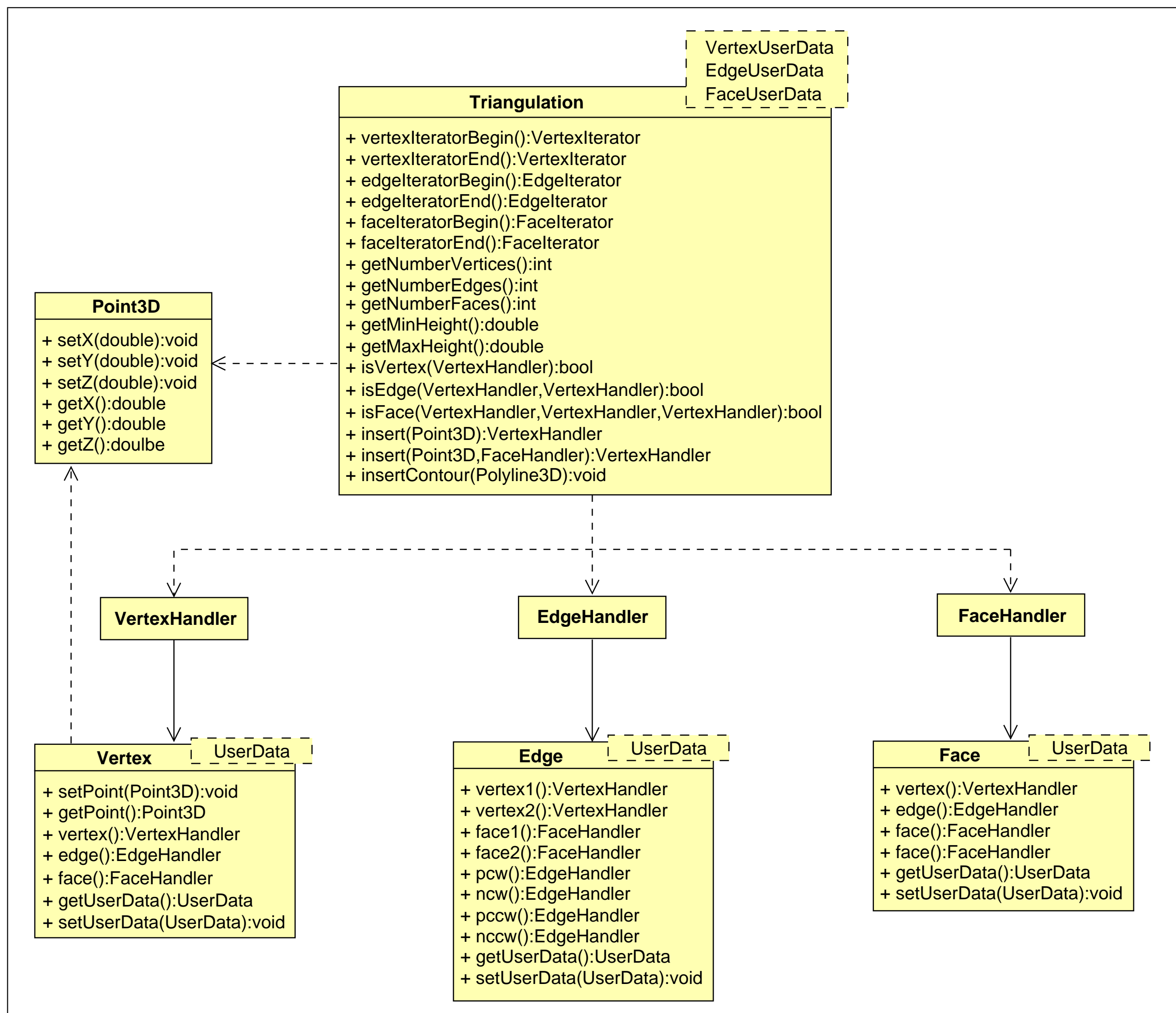


Figura 36: UML da Classe *Triangulation* e suas principais classes associadas.

5.4.1 Navegação no TIN usando a *Winged-Edge*

Verifica-se na Figura 36, que a classe *Triangulation* tem o apoio de diferentes classes para conseguir efetivamente representar um **TIN**.

As coordenadas (x, y, z) dos vértices de uma triangulação são representadas por meio da classe `Point3D`.

A entidade aresta da triangulação é representada por meio da classe `Edge`. Dado o fato do número de vértices e faces incidentes em uma aresta serem invariantes optou-se por construir a interface de acesso da classe `Edge` nos moldes da **EDT** *Winged-Edge* (vide Seção 3.2.2). Desta maneira a classe `Edge` define os seguintes métodos para o acesso à toda informação topológica de uma aresta:

- **vertex1() e vertex2():** fornece acesso aos dois vértices delimitadores da aresta;
- **face1() e face2():** fornece acesso às duas faces adjacentes;
- **pccw(), nccw(), pcw() e ncw():** fornece acesso às quatro arestas adjacentes;

Na Figura 16a pode ser vista uma representação gráfica da informação topológica retornada por esses métodos. Os vértices que compõem o modelo **TIN** são representados por meio da classe `Vertex`. Como é ilustrado no modelo UML da Figura 36 na página precedente, essa classe define os seguintes métodos para o acesso à informação topológica e geométria de um vértice:

- **getPoint():** retorna um ponto contendo as coordenadas (x, y, z) do vértice;
- **vetex():** retorna um dos seus vértices adjacentes;
- **edge():** retorna uma de suas arestas adjacentes;
- **face():** retorna uma de suas faces adjacentes;

Finalmente as faces que compõem o modelo **TIN** são representados por meio da classe `Face`. Como ilustrado na Figura 35, essa classe define os seguintes métodos para o acesso à informação topológica de uma face:

- **vertex():** retorna um dos seus vértices adjacentes;
- **edge():** retorna uma de suas arestas adjacentes;
- **face():** retorna uma de suas faces adjacentes;

Nota-se que somente a classe responsável pela representação de arestas (`Edge`), fornece acesso a todas as relações de adjacência. Na Seção 5.4.4, é mostrado como pode ser feita a recuperação das relações de adjacência das entidades vértice e face.

5.4.2 Estendendo Vértices Arestas e Faces

Como pode ser observado na Figura 36, as classes Vertex, Edge e Face contêm informações suficiente apenas para representar o **TIN**; mais especificamente informações sobre as coordenadas dos vértices e de conectividade entre as entidades vértices arestas e faces.

Dependendo da aplicação ou algoritmo que irá fazer uso de um **TIN**, pode ser necessário adicionar outras informações nessas entidades, como: vetor normal associado a faces ou vértices; se uma aresta representa uma linha de quebra; ou um ponteiro para uma *string* contendo uma descrição do vértice.

Dessa forma, a classe *Triangulation* permite ao usuário adicionar outras informações nas classes Vertex, Edge e Face. Conforme é mostrado na Figura 36, classe *Triangulation* é implementada na forma de uma classe *template* com os parâmetros:

- **VertexUserData:** permite ao usuário adicionar outros atributos à classe Vertex;
- **EdgeUserData:** permite ao usuário adicionar outros atributos à classe Edge;
- **FaceUserData:** permite ao usuário adicionar novos atributos à classe Face;

Uma vez definidos o novos atributos para as classes Vertex, Edge e Face, esses podem ser acessados por meio dos métodos getUserData() e setUserData() definidos nessas classes (Figura 36).

Os métodos getUserData() e setUserData() são criados apenas quando o usuário adiciona novas informações nas classes Vertex, Edge e Face, por exemplo, se o usuário customizar apenas a classe Face adicionando um vetor normal a ela, os métodos getUserData() e setUserData() serão disponíveis apenas nessa classe, ou seja não serão gerados para as classes Vertex, Edge.

Como é visto na Figura 36, a classe *Triangulation* tem os seus parâmetros *template* inicializados por padrão com os valores DefaultVertexUserData, DefaultEdgeUserData e DefaultFaceUserData, indicando que nesse caso não será feita nenhum tipo de customização nas classes Vertex, Edge e Face.

Na página seguinte é apresentado um trecho de código ilustrando como pode ser feita a extensão das classes Vertex, Edge e Face.

```
1  //define dados do usuário que serão adcionados à classe Vertex
2  class VertexUserData
3  {
4          ...;//definição de atributos e métodos
5  };
6
7  //define dados do usuário que serão adcionados à classe Edge
8  class EdgeUserData
9  {
10         ...;//definição de atributos e métodos
11 };
12
13 //define dados do usuário que serão adcionados à classe Face
14 class FaceUserData
15 {
16         ...;//definição de atributos e métodos
17 };
18
19
20 //define o novo tipo "TriangulationUser" com os dados do usuario
21 typedef Triangulation<VertexUserData,EdgeUserData,FaceUserData> TriangulationUser;
22 //cria uma triangulação com vértices, aresta e faces customizadas
23 TriangulationUser trg;
24
25
26 /*****************Adiciona e Recupera os dados do usuário nos vértices************/
27 //dado um vértice qualquer de trg, e os dados do usuario para este vértice
28 VertexHander vh = ...;
29 VertexUserData vd = ...;
30
31 vh->setUserData(vd); //adiciona os dados do usuário ao vértice
32 vd = vh->getUserData();   //recupera os dados do usuário adicionados ao vértice
33
34
35 /*****************Adiciona e Recupera os dados do usuário nas Arestas*************/
36 //dado uma aresta qualquer de trg, e os dados do usuario para esta aresta
37 EdgeHander eh = ...;
38 EdgeUserData ed = ...;
39
40 eh->setUserData(ed); //adiciona os dados do usuário à aresta
41 ed = eh->getUserData();   //recupera os dados do usuário adicionados na aresta
42
43
44 /*****************Adiciona e Recupera os dados do usuário nas Faces***************/
45 //dado uma face qualquer de trg, e os dados do usuario para esta face
46 FaceHander fh = ...;
47 FaceUserData fd = ...;
48
49 fh->setUserData(fd); //adiciona os dados do usuário à aresta
50 fd = fh->getUserData();   //recupera os dados do usuário adicionados na aresta
```

5.4.3 *Handlers*

Um *handler* é um tipo que representa uma referência para outro tipo. *Handlers* funcionam como ponteiros da linguagem **C/C++**, possuindo os operadores de desreferenciamento “ `*` ” e de acesso “ `->` ”. Entretanto, ao contrário de ponteiros, *handlers* não possuem operadores de incremento “ `++` ” e decremento“ `--` ”.

No sistema desenvolvido, *handlers* são utilizados como interface de acesso individual aos elementos vértices, aresta e faces armazenados na estrutura de dados utilizada na representação computacional do **TIN** (neste caso a **TDS**), utilizando a interface fornecida pelas classes `Vertex`, `Edge` e `Face`.

Basicamente foram desenvolvidos os seguintes *handlers*:

- **VertexHandler:** fornece acesso a um vértice do **TIN**;
- **EdgeHandler:** fornece acesso a uma aresta do **TIN**;
- **FaceHandler:** fornece acesso a uma face do **TIN**;

5.4.4 Iteradores

Uma vez gerado o **TIN**, os seus vértices, as arestas e faces podem ser acessados por meio de iteradores.

O uso de iteradores permite ao usuário acessar esses elementos de forma linear, ou seja, independentemente da forma como esses elementos estão arranjados na **EDT** que mantém o **TIN**, os mesmos são acessados como se estivem arranjados sequencialmente em um vetor ou lista, ou seja, o iterador trata uma estrutura de dados como uma sequência (VINHAS et al., 2002).

A classe *Triangulation* fornece os seguintes iteradores:

- **VertexIterator:** fornece acesso a todos os vértices que compõem a triangulação, onde o método `vertexIteratorBegin()` que retorna um `VertexIterator` para o primeiro vértice da triangulação e o método `vertexIteratorEnd()` que retorna um `VertexIterator` representando o final do conjunto de vértices.
- **EdgeIterator:** fornece acesso a todas as arestas da triangulação, onde o método `edgeIteratorBegin()` que retorna um `EdgeIterator` para a primeira aresta da

triangulação e o método `edgeIteratorEnd()` que retorna um `EdgeIterator` representando o final do conjunto de arestas.

- **FaceIterator:** fornece acesso a todas as faces que compõem a triangulação. A classe *Triangulation* fornece o método `faceIteratorBegin()` que retorna um `FaceIterator` para a primeira face da triangulação e o método `faceIteratorEnd()` que retorna um `FaceIterator` representando o final do conjunto de faces.

Para todos os iteradores desenvolvidos, foi adotado o conceito de iterador utilizado pela **STL**. De fato, todos os iteradores utilizados no sistema são especializações do tipo de dado *Bidirectional Iterator* fornecido pela **STL**. Dessa maneira, a exemplo da **STL**, todos iteradores fornecidos pela classe *Triangulation* possuem em comum os seguintes operadores:

- **Operador ***
  Retorna o elemento (vértice, aresta ou face) na posição atual do iterador;
- **Operador ->**
  Fornece acesso às funções membro do elemento (vértice, aresta ou face) para o qual o iterador aponta;
- **Operador ++**
  Move o iterador para o próximo elemento (vértice, aresta ou face) da triangulação;
- **Operador --**
  Move o iterador para o elemento (vértice, aresta ou face) anterior da triangulação;
- **Operadores == e !=**
  Usados para verificar se dois iteradores representam a mesma posição;
- **Operador ++**
  Faz a atribuição de um iterador;

No trecho de código apresentado na página seguinte, é mostrado um exemplo de como percorrer todos os vértices, arestas e faces de uma triangulação utilizando os iteradores fornecidos pela classe *Triangulation*.

```
Triangulation trg;


// Percorre todos os vértices da triangulação
for(Triangulation::VertexIterator vit = trg.vertexIteratorBegin();
    vit != mesh.vertexIteratorEnd();
    ++vit)
{
    ...; //faz algum processamento com *vit, vit-> ou VertexHandler vh = vit
}


// Percorre todas as arestas da triangulação
for(Triangulation::EdgeIterator eit = trg.edgeIteratorBegin();
    eit != trg.edgeIteratorEnd();
    ++eit)
{
    ...; //faz algum processamento com *eit, eit-> ou EdgeHandler eh = eit
}


// Percorre todas as faces da triangulação
for(Triangulation::FaceIterator fit = trg.FaceIteratorBegin();
    fit != trg.FaceIteratorEnd();
    ++fit)
{
    ...; //faz algum processamento com *fit, fit-> ou FaceHandler fh = fit
}
```

5.4.5 Circuladores

O conceito de iterador, é útil para acessar elementos de uma dada estrutura como sequências lineares como por exemplo, todos os vértices, arestas ou faces que compõem um **TIN**. Entretanto segundo Devillers et al. (2010), sequências circulares ocorrem com muita frequência em estruturas de dados combinatórias e geométricas. Por exemplo, no caso de um **TIN** todas as arestas ou faces adjacentes a um vértice formam uma sequência circular.

Desta forma, no sistema desenvolvido foi utilizado o conceito de circulador (*circulator*), fornecido pela biblioteca **CGAL**, no acesso ao conjuntos de elementos adjacentes ou incidentes a um dado elemento de um **TIN** (ex: faces incidentes a um vértice). A seguir é dada uma breve descrição dos circuladores desenvolvidos:

- **AdjVV:** dado um vértice, fornece acesso a todos os seus vértices adjacentes;
- **AdjVE:** dado um vértice, fornece acesso a todas as suas arestas incidentes;

- **AdjVF:** dado um vértice, fornece acesso a todas as suas faces incidentes;
- **AdjFV:** dada uma face, fornece acesso a todos os seus vértices incidentes;
- **AdjFE:** dada uma face, fornece acesso a todas as suas arestas incidentes;
- **AdjFV:** dada uma face, fornece acesso a todas as suas faces adjacentes;

Todos circuladores descritos anteriormente possuem em comum os seguintes operadores:

- **Operador ***
  Retorna o elemento (vértice, aresta ou face) adjacente ou incidente corrente;
- **Operador ->**
  Fornece acesso às funções membro do elemento (vértice, aresta ou face) adjacente ou incidente corrente;
- **Operador ++**
  Move o circulador para o próximo elemento adjacente ou incidente no sentido anti-horário;
- **Operador --**
  Move o circulador para o próximo elemento adjacente ou incidente no sentido horário;
- **Operadores == e !=**
  Usados para verificar se dois circuladores representam um mesmo elemento adjacente ou incidente;
- **Operador =**
  Operador de atribuição entre o circuladores;

Não foi desenvolvido nenhum circulador para a obtenção dos vértices, arestas ou faces adjacentes a uma aresta. Isto porque os elementos adjacentes a uma aresta não formam uma sequência circular podendo ser obtidos de forma direta na classe `Edge`.

No trecho de código mostrado na próxima página, são dados exemplos de como os circuladores fornecidos pela classe *Triangulation* podem ser utilizados para encontrar todas as relações de adjacência da entidade vértice:

```
1  //dado um vertice qualquer apontado por vh
2  VertexHandler vh = ...;
3
4
5  //Adjacência Vértice-Vértice
6  AdjVV vv, v_end;
7  vv = v_end = vh;
8  //percorre todos os vértices adjacentes ao vértice apontado por vh
9  do
10 {
11     ...; //faz algum processamento com *vv, vv-> ou VertexHandler vh = vv;
12
13 }while(++vvc != vvend)
14
15
16 //Adjacência Vértice-Aresta
17 AdjVE ve, e_end;
18 ve = e_end = vh;
19 //percorre todas as arestas adjacentes ao vértice apontado por vh
20 do
21 {
22     ...; //faz algum processamento com *ve, ve-> ou EdgeHandler eh = ve;
23
24 }while(++ve != e_end);
25
26
27 //Adjacência Vértice-Face
28 AdjVF vf, f_end;
29 vf = f_end = vh;
30 //percorre todas as faces adjacentes ao vértice apontado por vh
31 do
32 {
33     ...; //faz algum processamento com *vf, vf-> ou FaceHandler fh = vf;
34
35 }while(++vf != f_end);
```

No trecho de código a seguir são dados exemplos de como os circuladores fornecidos pela classe *Triangulation* podem ser utilizados para encontrar todas as relações de adjacência da entidade face:

```
1  //dado uma face qualquer apontada por fh
2  FaceHandler fh = ...;
3
4
5  //Adjacência Face-Vértice
6  AdjFV fv, v_end;
7  fv = v_end = fh;
8  //percorre todos os vertices adjacentes a face apontada por fh
9  do
10 {
11     ...; //faz algum processamento com *fv, fv-> ou VertexHandler vh = fv;
```

```
12
13  }while(++fv != v_end)
14
15
16  //Adjacência Face−Aresta
17  AdjFE fe, e_end;
18  fe = e_end = fh;
19  //percorre todas as arestas adjacentes a face apontada por fh
20  do
21  {
22      ...; //faz algum processamento com *fe, fe−> ou EdgeHandler eh = fe;
23
24  }while(++fe != e_end);
25
26
27  //Adjacência Face−Face
28  AdjFF vf, f_end;
29  ff = f_end = fh;
30  //percorre todas as faces adjacentes a face apontada por fh
31  do
32  {
33      ...; //faz algum processamento com *ff, ff−> ou FaceHandler fh = ff;
34
35  }while(++vf != e_end);
```

5.5 Integração com Sistemas Existentes

Uma questão fundamental na proposta de elaboração de um novo sistema de *software* de modelagem digital de terrenos é a sua compatibilidade com sistemas já existentes. Logo, o sistema deve ser capaz de suportar a entrada de dados, tanto no formato vetorial (arquivos de pontos irregularmente espaçados ou contornos), como no formato raster (matrizes de elevação), garantindo assim a sua integração com outras plataformas de *software*.

Para ser mais preciso, o suporte aos vários formatos de dados raster e vetoriais foi desenvolvido respectivamente, com o auxílio das bibliotecas **GDAL** e **OGR** , que oferecem inúmeras facilidades de leitura, escrita e conversão entre esses formatos (WARMERDAM, 2008).

Entretanto, ao contrário das bibliotecas **GDAL** e **OGR** que contêm interfaces específicas para os formatos de dados raster e vetorial, optou-se no presente trabalho, pela criação de um camada de acesso unificada para esses dois formatos de dados. Na Figura 37, é mostrado o modelo UML das classes desenvolvidas para o acesso a arquivos do tipo raster e vetorial.

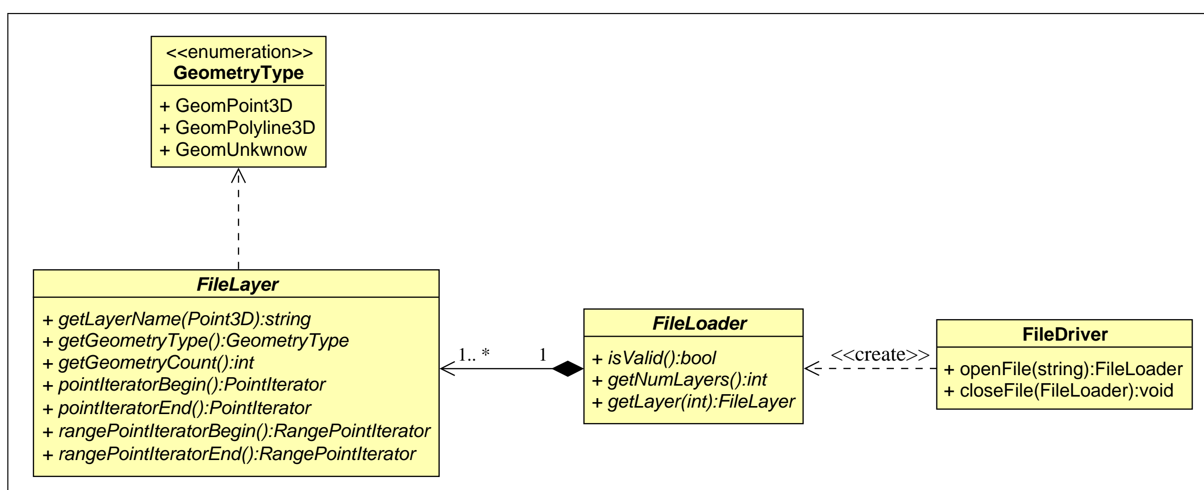


Figura 37: UML das classes responsáveis pelo carregamento de arquivos.

A classe **FileDriver** é responsável pela abertura de um arquivo de dados sendo que dependendo da sua extensão, ela pode utilizar a **GDAL** ou **OGR** para esse fim. Basicamente essa classe fornece os seguintes métodos:

- **openFile():** esse método é responsável pela abertura de um arquivo de dados raster ou vetorial, recebendo como parâmetro uma *string* contendo o nome do arquivo a ser

aberto e a sua extensão e devolvendo um objeto do tipo **FileLoader** representando um arquivo aberto;

- **closeFile():** esse método é responsável pelo fechamento de um arquivo previamente aberto por meio do método `openFile()`.

A classe abstrata **FileLoader** representa uma arquivo de dados aberto pela classe **FileDriver**. Basicamente essa classe funciona como uma camada de acesso aos formatos de arquivos raster ou vetorial suportados pelas bibliotecas **GDAL/OGR**. Essa classe tem associada um conjunto de objetos do tipo **FileLayer** que representam os dados do arquivo propriamente dito. A classe **FileLoader** basicamente fornece os seguintes métodos:

- **isValid():** esse método retorna um valor do tipo booleano que é utilizado para verificar se o arquivo foi aberto como sucesso.
- **getNumLayers():** retorna o total de *layers* que o arquivo contêm;
- **getLayer(int):** retorna uma *layer* associada ao arquivo;

A classe abstrata **FileLayer** representa um conjunto de dados do mesmo tipo contido no arquivo, neste caso conjunto de pontos 3D regularmente (raster) / irregularmente (vetorial) espaçados ou polilinhas 3D representando contornos. O acesso a esses dados é feito por meio dos iteradores *PointIterator* e *RangePointIterator*. A seguir é dada uma descrição dos principais métodos fornecidos por essa classe:

- **getGeometryType():** retorna uma enumeração do tipo **GeometryType** (Figura 37) com a informação sobre o tipo de geometria contida na layer. Como o sistema basicamente utiliza pontos 3D (**GeomPoint3D**) ou Polilinha 3D representando contornos (**GeomPolyline3D**), somente esse tipo de geometria pode ser retornada por uma **FileLayer**. Para quaisquer outros tipos de geometria contida no arquivo é retornado o valor **GeomUnknow**;
- **getGemetryCount():** retorna o total de geometrias (pontos ou contornos) da *layer*;
- **pointIteratorBegin() e pointIteratorEnd():** fornece acesso aos pontos 3D contidos na layer;
- **rangePointIteratorBegin() e rangePointIteratorEnd():** fornece acesso às linhas de contorno contidas na layer;

5.5.1 Padrão de Projeto *Factory*

As classes mostradas na Figura 37, foram desenvolvidas utilizando o padrão de projeto *factory*. Esse padrão de projeto define uma interface para a criação de objetos delegando às suas subclasses a decisão de qual classe instanciar (GAMMA et al., 1998).

Basicamente esse padrão pode ser utilizado na definição de classes responsáveis pela criação, inicialização, configuração ou qualquer outra operação relacionada a instanciação de objetos (LASATER, 2007).

Ainda, segundo CAMARA et al. (2001), o padrão de projeto *factory* possui muitas aplicações no desenvolvimento de *software* **SIG**, podendo ser utilizado em situações onde a instanciação de um objeto é dependente da especificação de um campo tipo como:

- seleção de *drivers* de banco de dados baseado em seu nome;
- escolha da função de conversão de dados baseada na extensão de um arquivo;
- carregamento de um sistema de projeção cartográfica baseado no seu nome.

De acordo com Lasater (2007) e conforme ilustrado na UML mostrada na Figura 38, o padrão de projeto *factory* é baseado em dois componentes principais: uma fábrica e um produto.

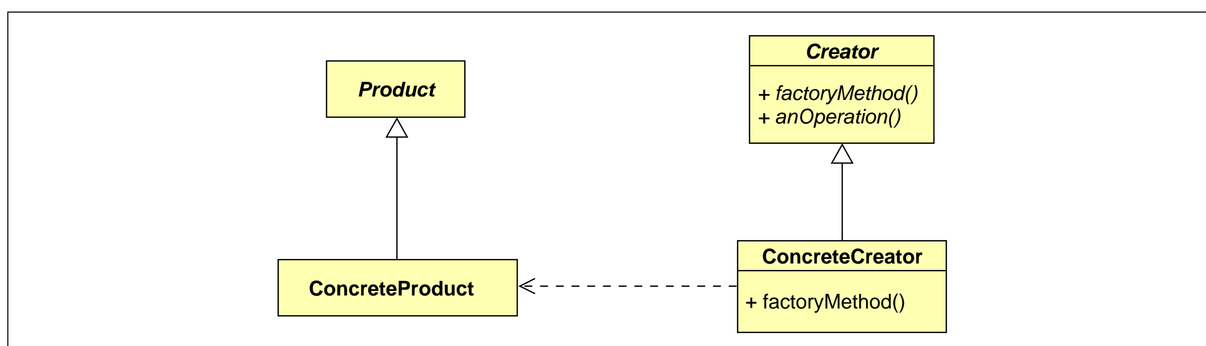


Figura 38: UML Padrão de Projeto *Factory*.

A seguir é dada uma rápida descrição de cada um dos componentes desse padrão de projetos.

- **Creator :** faz a definição do método responsável pela instanciação de objetos derivados de **Product** (no caso da Figura 38 o método `factoryMethod()`). Esse método pode receber como parâmetro um valor chave indicando qual subclasse de **Product** deverá ser instanciada;

- **Product :** define a interface utilizada pelos objetos instanciados por **Creator**;
- **ConcreteCreator :** faz a sobrecarga do método `factoryMethod()` retornando uma instância de **ConcretProduct**;
- **ConcretProduct :** classe contendo dados ou funcionalidades que implemente a interface especificada por **Product**;

A Figura 39, contem um digrama de classe ilustrando como o padrão de projeto *factory* foi utilizado na criação das classes responsáveis pelo acesso a arquivos de dados raster e vetoriais:

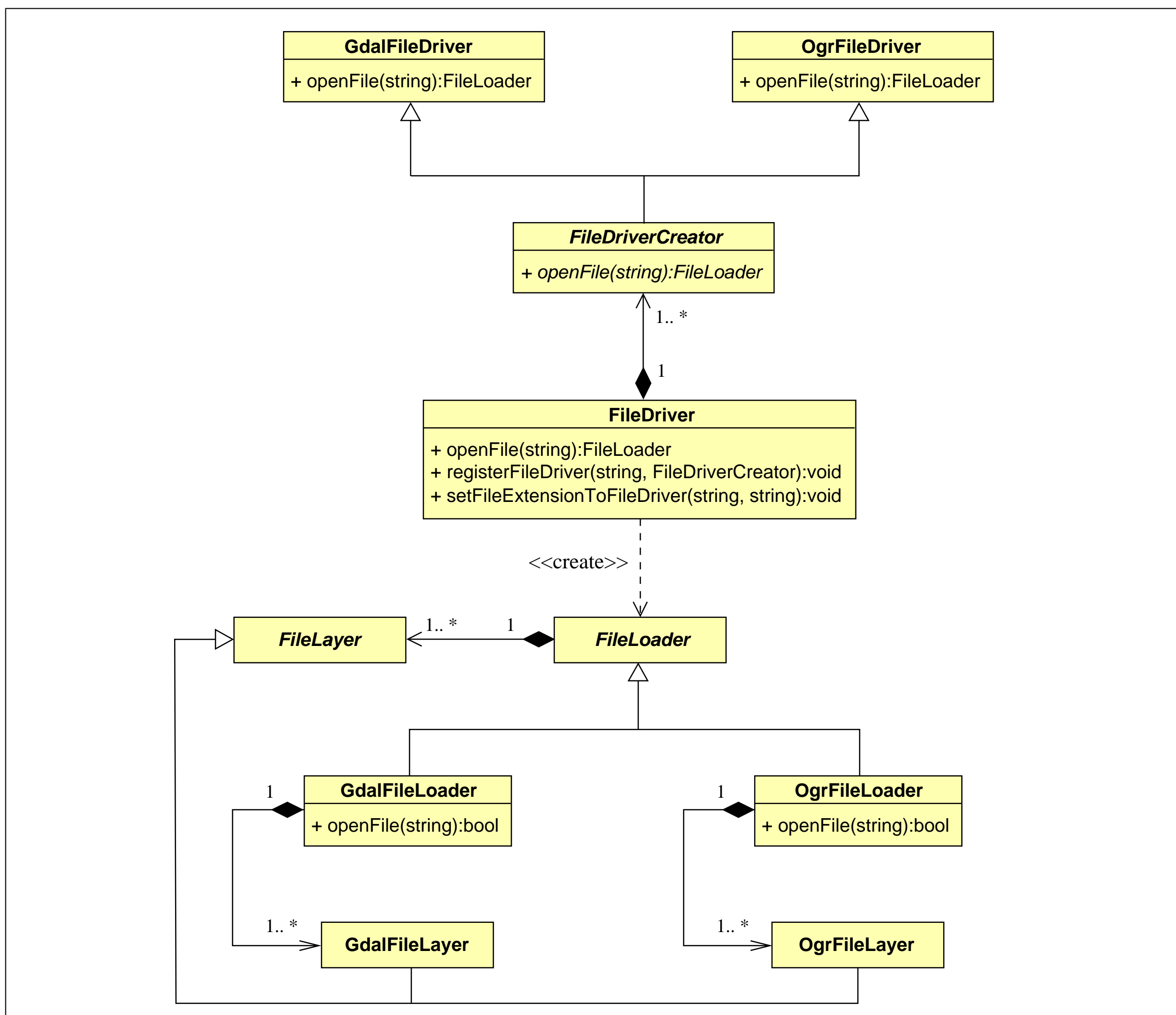


Figura 39: Aplicação do padrão de projeto *Factory*.

A classe abstrata **FileDriverCreator** está para a classe **Creator** na Figura 38. Nessa classe o método `openFile()` é responsável pela correta instanciação de objetos do tipo **FileCreator**, ou seja, o método `openFile()` representa o método `factoryMethod()` de

Creator na Figura 38. As classes derivadas **GdalFileDriver** e **OgrFileDriver** implementam o método `openFile()` definido na classe **FileDriverCreator** (utilizando as bibliotecas **GDAL** e **OGR** respectivamente), desta forma essas classes representam a classe **ConcreteCreator** na Figura 38.

Já a classe abstrata **FileLoader** é uma representação da classe **Product** na Figura 38. As classes derivadas **GdalFileLoader** e **OgrFileLoader** implementam os métodos definidos na classe **FileLoader** para a manipulação de arquivos raster e vetorial, dessa forma essas classes representam a classe **ConcreteProduct** na Figura 38.

A classe **FileDriver** funciona como um repositório de classes "*fábrica*" (nesse caso as classes **GdalFileDriver** e **OgrFileDriver**) sendo responsável pela correta instanciação de um objeto do tipo **FileLoader** (nesse caso **GdalFileLoader** ou **OgrFileLoader**) de acordo com a extensão do arquivo que será aberto.

Por exemplo se o usuário entrar com um arquivo vetorial (ex: "*.shp") a classe **FileDriver** irá instanciar um objeto do tipo **OgrFileLoader** para a abertura e manipulação desse arquivo. Já se o usuário entrar com um arquivo raster (ex: "*.tif" ou "*.jpg") a classe **FileDriver** irá instanciar um objeto do tipo **GdalFileLoader**.

De fato, todo código responsável pela abertura de arquivos se encontram nas classes concretas derivadas de **FileLoader**. Conforme mostrado na Figura 39, as classes **GdalFileLoader** e **OgrFileLoader** também possuem um método `open()`, isto porque é nesses métodos que é implementado todo do código de abertura de arquivo utilizando as bibliotecas **GDAL** e **OGR**.

Desta forma basicamente são executados os seguintes passos durante a abertura de um arquivo:

1. O método `openFile()` de uma instância de **FileDriver** recebe o nome de um arquivo e recupera o "*driver*" (classe concreta derivada de **FileDriver**; neste caso **GdalFileDriver** ou **OgrFileDriver**) que lhe dá suporte baseado na sua extensão.

2. O nome do arquivo então é passado para o método `openFile()` do "*driver*" recuperado, para que este crie uma instância da classe concreta derivada **FileLoader** (**GdalFileLoader** ou **OgrFileLoader**) que detém todo o conhecimento e funcionalidades necessárias para a abertura e manipulação do arquivo.

3. Finalmente o nome do arquivo é repassado para o método `openFile()` da instância da classe concreta derivada de **FileLoader** (**GdalFileLoader** ou **OgrFileLoader**)

que de fato irá efetuar a abertura do arquivo.

Uma vantagem do uso do padrão de projeto "*factory*" reside no fato de que a abertura de arquivos é feita de forma transparente para o usuário, que deve se preocupar apenas em saber quais métodos são fornecidos pela classe abstrata **FilerLoader** (que contem os métodos de acesso aos dados do arquivo), não interessando se essa classe usa uma implementação fornecida pela classe **GdalFileLoader** ou **OgrFileLoader** ou quem é de fato responsável pela abertura do arquivo em disco.

Uma outra vantagem do uso desse padrão de projeto está relacionada à questão de extensibilidade. Conforme pode ser visto na Figura 39 na página 107, além do método `openFile()`, a classe **FileDriver** também possui os métodos `registerFileDriver()` e `setFileExtensionToFileDriver()`.

Esses dois métodos, permitem ao usuário, respectivamente, adicionar novos "*drivers*" (classes concretas derivadas de **FileDriverCreator**) para a leitura de formatos de arquivos não suportados e associar as extensões de arquivos suportadas por um determinado driver.

Basicamente, devem ser executados os seguintes passos para a criação e adição de novos *"drivers"* de carregamento de arquivos:

1. Criar uma classe derivada de **FileLoader** com todo o código necessário para a abertura e acesso aos dados do arquivo.

2. Criar uma classe derivada de **FileDriverCretor**, que consiga fazer uma instanciação da classe de derivada de **FileLoader** criada anteriormente.

3. Adicionar o novo driver e extensões suportadas por este, por meio dos métodos `registerFileDriver()` e `setFileExtensionToFileDriver()` de **FileDriver**.

5.6 Visualização

Um dos componentes fundamentais de sistemas de *software* **SIG** é o modulo de visualização. Particularmente, a visualização de **MDT** possui um papel de destaque na sua compreensão e avaliação. A visualização tem por propósito mostrar graficamente o **MDT** e qualquer informação derivada do mesmo, servindo como um meio de comunicação entre o modelo digital da superfície do terreno e o usuário do modelo, podendo ser usado, por exemplo, como uma ferramenta de tomada de decisão por meio de inspeção visual, ou simplesmente exibir resultados de processamentos sobre o mesmo.

Ainda, muitas vezes a natureza da superfície de um terreno pode ser melhor entendida apenas por meio de sua representação gráfica. Desta forma a visualização se torna uma ferramenta importante na exploração (gráfica) de dados e informações acerca da superfície que está sendo modelada. Ainda, o uso de representações gráficas geralmente é focada na representação intuitiva dos dados, contribuindo ainda mais para o entendimento do fenômeno contido neles.

Com o desenvolvimento da computação gráfica, a visualização 3D tem se tornado um dos principais meios de exibição de **MDTs** (LI; ZHU; GOLD, 2005). Entretanto, um **TIN** gerado para representar um **MDT** pode ser bastante complexo, podendo chegar a ordem de um milhão de faces triangulares, sendo necessário o uso de técnicas eficientes de renderização de triangulação.

No sistema desenvolvido, as diferentes operações de visualização sobre o terreno representado foram implementadas por meio das bibliotecas **OpenGL** e **OSG**. Nas seções a seguir é dada uma descrição de como a renderização de triangulações, representando a superfícies de terrenos, foi realizada.

5.6.1 Modo Imediato

Uma maneira de se gerar imagens de terreno no **OpenGL** é pelo uso do modo imediato de renderização. Nesse modo o **OpenGL** faz uma chamada ao *hardware* gráfico para cada elemento da triangulação (pontos, linhas, triângulos) que será renderizada.

A seguir é dado um exemplo de código de uma possível forma de renderização dos triângulos que compõem um **TIN** utilizando esse modo.

```
1  glBegin(GL_TRIANGLES);
2      glNormal3f(x0,y0,z0);  //vetor normal do primeiro vértice do triângulo
3      glColor3f(r0,g0,b0);        //cor do primeiro vértice do triângulo
4      glVertex3f(nx0,ny0,nz0);    //coordenadas do primeiro vértice do triângulo
5
6      glNormal3f(x1,y1,z1);  //vetor normal do segundo vértice do triângulo
7      glColor3f(r1,g1,b1);        //cor do segundo vértice do triângulo
8      glVertex3f(nx1,ny1,nz1);    //coordenadas do segundo vértice do triângulo
9
10     glNormal3f(x2,y2,z2);  //vetor normal do terceiro vértice do triângulo
11     glColor3f(r2,g2,b2);        //cor do terceiro vértice do triângulo
12     glVertex3f(nx2,ny2,nz2);    //coordenadas do terceiro vértice do triângulo
13 glEnd();
```

No trecho de código acima, verifica-se que além das coordenadas de cada triângulo que compõem o modelo, também é associado um vetor normal e uma cor para cada vértice da triangulação. No caso do presente trabalho, o vetor normal é utilizado na aplicação de modelos de iluminação dando uma melhor noção de profundidade na imagem gerada do terreno. Já a cor associada a cada vértice é utilizada para dar uma melhor compreesão da variação de altitude ao longo do terreno, por exemplo, podem ser utilizadas cores mais fracas para vértices com valores de elevação pequenos e cores mais fortes para vértices com valores de elevação mais altos. Na Figura 40 é mostrada a imagem gerada de um **TIN**, com um total de 3.135.364 faces, usando esse esquema de renderização.

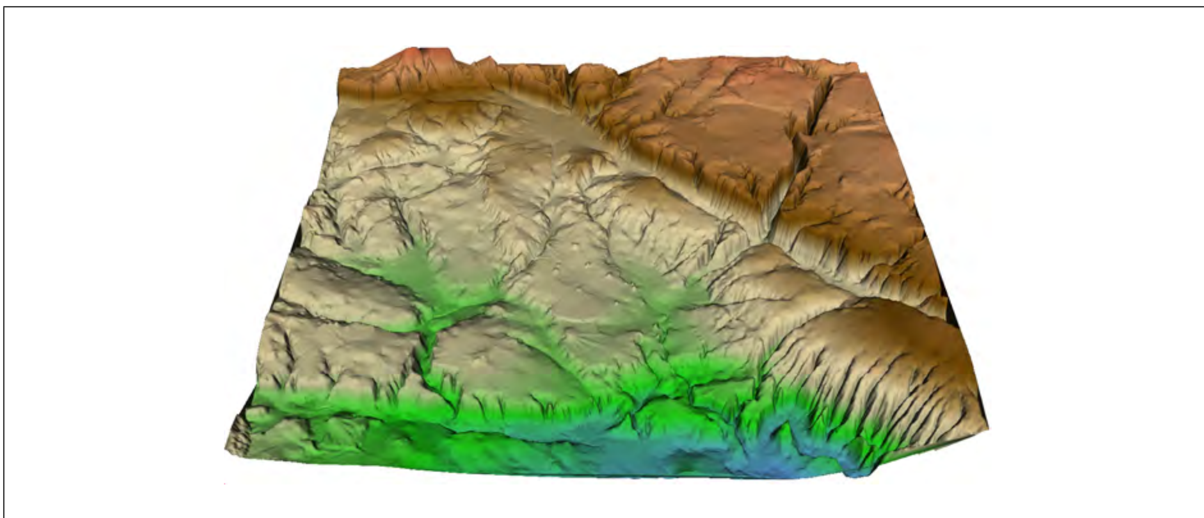


Figura 40: Renderização de um TIN

Voltando no trecho de código anterior, observa-se que para cada triângulo renderizado é feito um total de onze chamadas a função sendo, duas chamadas para dizer o tipo de primitiva que será renderizada (linhas 1 e 13) e mais 9 chamadas para passar o vetor normal (linhas 2, 6 e 10), a cor (linhas 3, 7 e 11) e as coordenadas (linhas 4, 8 e 12)

de cada triângulo. Esse número excessivo de chamadas a função pode causar uma perda de desempenho na renderização de triangulações com um número elevado de triângulos, por exemplo, para renderizar as 3.135.364 faces do **TIN** mostrado na Figura 40, seriam necessárias 34.489.004 chamadas a função.

Além do número elevado de chamadas a função, cada atributo (coordenadas, vetores normais e cores) dos triângulos renderizados, necessariamente deve passar pela CPU, que acaba sendo sobrecarregada, e como geralmente o *hardware* gráfico possui um poder de processamento maior que a CPU (processa os dados muito mais rapidamente do que a CPU tem poder de gerar) ele ficará a maior parte do tempo ocioso esperando por dados dela (é muito difícil saturar o *hardware* gráfico com dados nesse modo). Desta forma, o maior problema em se utilizar o modo imediato na renderização de triangulações é a transferência de dados ineficiente da aplicação (lado cliente) para o *hardware* gráfico (lado servidor).

5.6.2 *Vertex Arrays*

Uma forma de amenizar o problema do número excessivo de chamadas a função e a transferência ineficiente de dados da aplicação (modelo **TIN**) para o *hardware* gráfico é por meio da funcionalidade de *Vertex Arrays* do **OpenGL**.

O uso de *Vertex Arrays* permite a renderização de primitivas a partir de dados guardados em blocos de memória (MARTZ, 2007), mais precisamente, no tipo de dado vetor da linguagem de programação **C/C++**. A seguir é dado um exemplo de código de renderização de um **TIN** utilizando *Vertex Arrays*, onde cada triângulo possui os atributos coordenadas, vetor normal e cor associados aos seus vértices.

```
1  GLfloat vertices[] = {x0,y0,z0, x1,y1,z1, ... ,xn,yn,zn}
2  GLfloat cores[]    = {r0,g0,b0, r1,g1,b1, ... ,rn,gn,bn}
3  GLfloat normais[]  = {nx0,ny0,nz0, nx1,ny1,nz1, ... ,nxn,nyn,nzn}
4  GLint   indices[]  = {0,1,3,1,2,3,...}
5
6  //Habilita o uso de vertex arrays de coordenadas, normais e cores
7  glEnableClientState(GL_NORMAL_ARRAY);
8  glEnableClientState(GL_COLOR_ARRAY);
9  glEnableClientState(GL_VERTEX_ARRAY);
10
11 //diz para o Opengl onde os dados usados na renderização estão armazendos
12 glVertexPointer(3, GL_FLOAT, 0, normais);
13 glColorPointer(3, GL_FLOAT, 0, cores);
14 glVertexPointer(3, GL_FLOAT, 0, vertices);
15
```

```
16 //...
17 //faz a renderização da triangulação
18 glDrawElements(GL_TRIANGLES, 3, GL_INT, indices);
19 //...
```

Pode-se observar nas linhas 1, 2 e 3, que os atributos coordenada, cor e normal dos vértices da triangulação, são agrupados na forma dos vetores `vertices[]`, `cores[]` e `normais[]`. Na linha 4 também é criado o vetor `indices[]`. Cada entrada desse vetor representa uma coordenada, uma normal e uma cor contida nos vetores `vertices[]`, `cores[]` e `normais[]`. Como nesse caso estão sendo renderizados triângulos, cada três entradas do vetor de índices definem um triângulo. A Figura 41, contem uma ilustração de como dos dados de uma triangulação podem ser guardados em blocos de memória (vetores), e como cada triângulo desta é representado por meio de um vetor de índices.

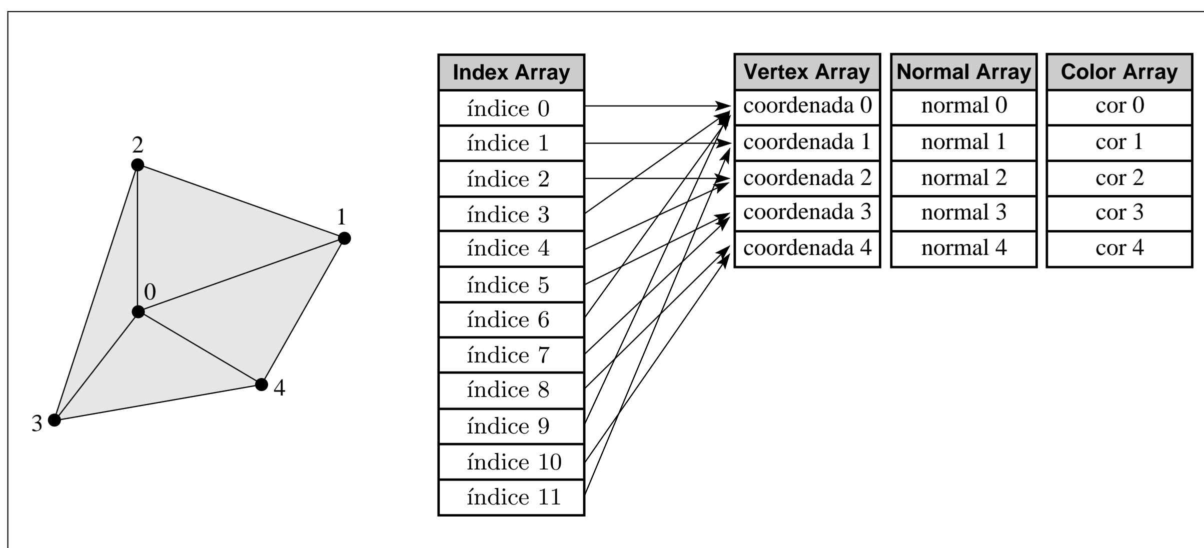


Figura 41: Modelo Esquemático de *Vertex Array* com Vetor de Índice.

Nas linhas 7 a 9 é habilitado o uso de *Vertex Arrays* para dados de coordenadas, normais e cor. Nas linhas 12 a 14 é dito para o **OpenGL** onde os dados usados na renderização estão armazenados. Finalmente na linha 18 é chamado o método `glDrawElements(...)` para renderizar toda a triangulação.

Tomando como base o exemplo do modo imediato onde eram necessárias 11 chamadas a função para renderizar cada triângulo de um **TIN**; usando *Vertex Arrays* é necessário fazer apenas uma chamada para a função `glDrawElements()` para renderizar todos os triângulos deste mesmo **TIN**, isto porque agora a quantidade de chamadas a função não está mais atrelada à quantidade de geometria que será renderizada. Os demais comandos do exemplo de código anterior (7 a 9 e 12 a 14) são utilizados apenas para a inicialização dos *Vertex Arrays*, ou seja, não precisam ser executados toda vez que a triangulação

for renderizada. Sendo assim as 34.489.004 chamadas a função para para renderizar as 3.135.364 faces do **TIN** mostrado na Figura 40 usando o modo imediato de renderização do **OpenGL**, caem para 1 chamada a função usando *Vertex Arrays*, ou seja, o uso de *Vertex Arrays* reduz drasticamente o *overhead* de chamadas a funções.

Outra vantagem em se utilizar *Vertex Arrays* está relacionada ao agrupamento das informações da geometria (nesse caso coordenadas, normal e cor dos vértices de um **TIN**) que será renderizada em blocos de dados (vetores `vertices[]`, `cores[]` e `normais[]`). Isto porque, a transferência desses blocos de memória da aplicação (lado cliente) para a o *hardware* gráfico (lado servidor) é bem mais eficiente do que a transferência individual de dados usado pelo modo imediato (WRIGHT; LIPCHAK; HAEMEL, 2007).

5.6.3 *Buffers Objects*

Apesar do uso de *Vertex Array* diminuir consideravelmente o número de chamadas da aplicação para o *hardware* gráfico e aumentar a eficiência da transferência de dados, considerando a arquitetura Cliente-Servidor do **OpenGL**, toda vez que for necessário renderizar a triangulação, os dados deverão ser transferidos da memória cliente (nesse caso o modelo **TIN** carregado na memória RAM) para o servidor (nesse caso a placa gráfica). A Figura 42 ilustra melhor como é feita essa transferência de dados da memória cliente para o servidor.

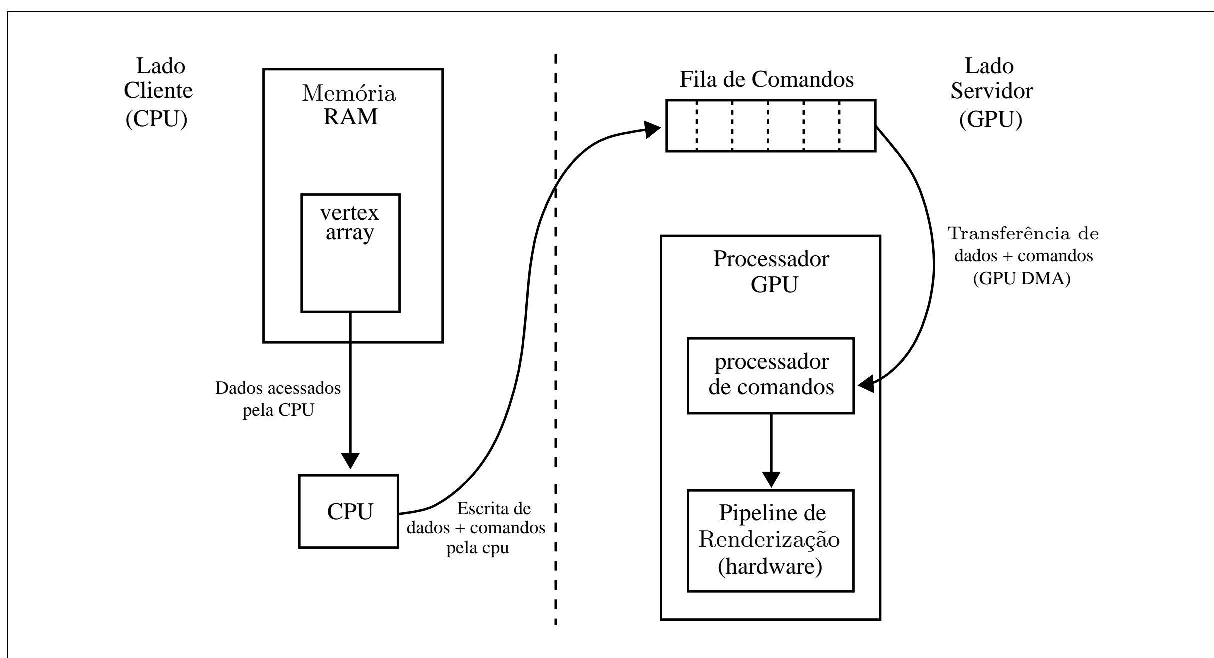


Figura 42: Modelo Esquemático da Renderização utilizando *Vertex Array*.
Fonte: adaptado de Kilgard et al. (2008)

Como pode ser observado na Figura 42, dados de renderização (ex: *Vertex Arrays*) se encontram na memória RAM do cliente. Todos os dados ou comandos (rotação, translação, habilitação de texturas, iluminação) passados da aplicação para o **OpenGL** necessariamente devem ser processados pela **CPU** que faz a escrita desses na fila de comandos do **OpenGL**, que só então faz o processamento desses dados e comandos e executa o *pipeline* de renderização (cria uma imagem a partir dos dados e comandos passados).

Segundo Shreiner et al. (2004), essa transferência ainda pode ser lenta ou desnecessária para dados que não sofram mudanças frequentemente (ex: não tem as coordenadas ou cores dos seus vértices alterados) ou então para o caso em que o cliente e o servidor se encontram em máquinas distintas.

Dadas estas questões, o **OpenGL** permite que porções da memória do servidor (placa gráfica) sejam reservadas pelo cliente (aplicação) por meio de *Buffer Objects*. De acordo com Martz (2007), *Buffer Objects* permitem que dados da aplicação (ex: vértices, normais e cores) sejam guardados em regiões de memória de alta-performance do *hardware* gráfico evitando o *overhad* da transferência de dados durante a renderização.

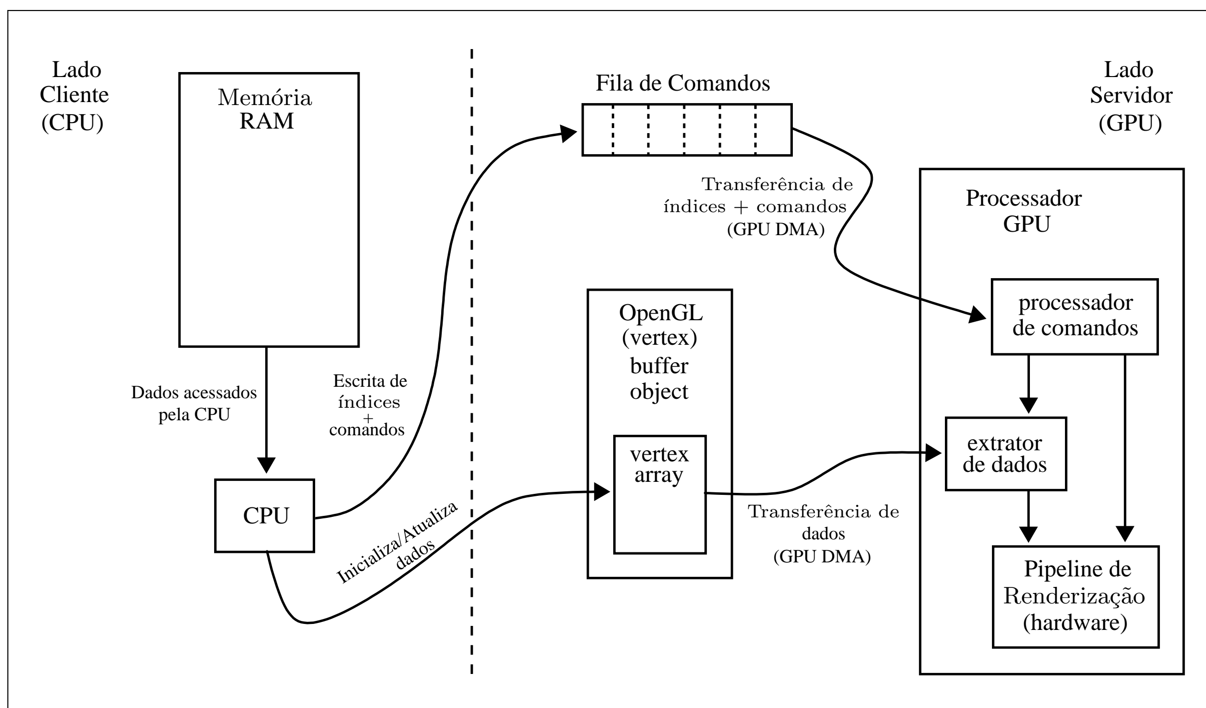


Figura 43: Modelo Esquemático da Renderização utilizando *Buffers Objects*.
Fonte: adaptado de Kilgard et al. (2008)

Como é ilustrado na Figura acima, quando *Buffer Objects* são utilizados, os dados da aplicação ficam armazenados na memória da placa gráfica, passando pela **CPU** apenas dados de índices e comandos. Desta maneira, toda vez que o **OpenGL** necessitar de

dados para a renderização ele fará acesso direto a eles na memória do *hardware* gráfico ou seja, os dados não precisarão ser transferidos da aplicação para a placa gráfica passando pela CPU.

O exemplo de código a seguir, mostra uma possível forma de renderização do **TIN** mostrado na Figura 40 usando *Buffers Objects*. Como nos exemplos de código anteriores, considera-se que serão renderizadas apenas as faces do **TIN**, sendo que cada vértice tem associado a sí as suas coordenadas, um vetor normal e uma cor.

```
#define VERTICES              0
#define NORMAIS               1
#define CORES                 2
#define INDICES               3
#define NUM_VERTICES      ...
#define NUM_BUFFERS           4

GLuint  buffers[NUM_BUFFERS];
GLfloat vertices[] = {x1,y1,z1,x2,y2,z2,...,xn,yn,zn}
GLfloat cores[]    = {r0,g0,b0, r1,g1,b1, ... ,rn,gn,bn}
GLfloat normais[]  = {nx0,ny0,nz0, nx1,ny1,nz1, ... ,nxn,nyn,nzn}
GLint   indices[]  = {0,1,3,1,2,3,...}

//cria os buffer objects
glGenBuffers(NUM_BUFFERS, buffers);

//faz a alocação e inicialização do Buffer Object com as coordenadas dos vertices
glBindBuffer(GL_ARRAY_BUFFER, buffers[VERTICES])
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*NUM_VERTICES*3, vertices, GL_STATIC_DRAW);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glEnableClientState(GL_VERTEX_ARRAY);

//faz a alocação e inicialização do Buffer Object com as normais dos vertices
glBindBuffer(GL_ARRAY_BUFFER, buffers[NORMAIS])
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*NUM_VERTICES*3, normais, GL_STATIC_DRAW);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glEnableClientState(GL_NORMAL_ARRAY);

//faz a alocação e inicialização do Buffer Object com as cores dos vertices
glBindBuffer(GL_ARRAY_BUFFER, buffers[CORES])
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*NUM_VERTICES*3, cores, GL_STATIC_DRAW);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
glEnableClientState(GL_COLOR_ARRAY);

//faz a alocação e inicialização do Buffer Object com os indices dos vertices
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[INDICES]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices GL_STATIC_DRAW);

//Faz a renderização da triangulação
glDrawElements(GL_TRIANGLES, NUM_VERTICES, GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));
```

No exemplo de código mostrada na página anterior, percebe-se que os dados que serão guardados na memória da placa gráfica também são agrupados na forma de vetores (linhas 9 a 12). Nas linhas compreendidas no intervalo 15 a 37 são feitas a criação, alocação e inicialização de porções da memória do *hardware* gráfico com esses vetores. Por fim, a renderização de toda a triangulação é feita com apenas uma chamada função `glDrawElements` na linha 40.

5.6.4 Implementação da Renderização

Segundo (WRIGHT; LIPCHAK; HAEMEL, 2007), a melhor maneira de se renderizar geometrias estáticas é guardando os seus dados no *hardware* gráfico e fazer uso de vetores de índices. No sistema proposto basicamente são utilizados duas formas de renderização: se a placa gráfica utilizada suportar o uso de *Buffers Objects*, é utilizado esse método de renderização; se a placa não der suporte ao uso de *Buffers Objects*, então a triangulação é renderizada utilizando *Vertex Arrays*. Considerando as limitações do modo imediato de renderização, este não será executado em nenhum momento.

Dada a complexidade do sistema desenvolvido, como já foi dito anteriormente, toda a renderização será feita por meio da biblioteca **OSG**. Particularmente para o caso de renderização da triangulação, essa biblioteca fornece mecanismos para lidar com toda a sua complexidade utilizando *Vertex Arrays* ou *Buffers Objects*.

Além disso, no sistema desenvolvido, foram utilizadas funcionalidades do **OSG** como: rotinas para teste de interseção do mouse com a geometria renderizada e manipuladores de câmeras.

Também foram utilizadas na renderização, rotinas de criação de vetores normais para os vértices do **TIN** e de iluminação fornecidas pelo **OSG**. Para Li, Zhu e Gold (2005), o uso de modelos de iluminação tem um papel fundamental na adição de realismo na renderização de **MDTs**. Ainda segundo Shreiner et al. (2004), grande parte dos objetos tridimensionais renderizados, não aparentam ter profundidade enquanto não iluminados (vide Figura 44(e) e 44(f)).

No trecho de código apresentado na próxima página, é exemplificado como a renderização do **TIN** mostrado na Figura 40 pode ser feita utilizando o **OSG**

```
//cria o vetor de indices
osg::ref_ptr<osg::DrawElementsUInt> indices = new osg::DrawElementsUInt(GL_TRIANGLES);

//cria os vetores que irão conter os dados da triangulação
osg::ref_ptr<osg::Vec3Array> vertices = new osg::Vec3Array();
osg::ref_ptr<osg::Vec3Array> normais  = new osg::Vec3Array();
osg::ref_ptr<osg::Vec4Array> cores    = new osg::Vec4Array();

//faz a inicialização do vetores de índices, vertices, normais e cores
indices.push_back(id)...
vertices->push_back(osg::Vec3(x,y,z))...
normais->push_back(osg::Vec3(nx,ny,nz))...
cores->push_back(osg::Vec4(r,g,b,1.0))...

//cria e inicializa o objeto que ira representar a geometria do TIN
osg::Geometry* geometry = new osg::Geometry;
geometry->addPrimitiveSet(indiceTriang.get());
geometry->setVertexArray(vertices.get());
geometry->setNormalArray(normais.get());
geometry->setColorArray(cores.get());
geometry->setColorBinding(osg::Geometry::BIND_PER_VERTEX);

//os dados de geometria devem ser guardados na memoria da placa gráfica
geometry->setDataVariance(osg::Object::STATIC);
geometry->setUseDisplayList(false);
geometry->setUseVertexBufferObjects(true);

//cria e inicializa o grafo de cena representando a triagulação
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable(geometry);

//cria um visualizador para o grafo de cena criado
osgViewer::Viewer viewer;
viewer.setSceneData(geode.get());
viewer.run();
```

No trecho de código acima, todos os dados de renderização são colocados nos vetores (ao estilo **STL**) `vertices`, `normais` e `cores`. Também é utilizado um vetor de índice (`indices`) para evitar dados duplicados durante a renderização. O objeto `geometry` é responsável por guardar dados da geometria e como esta será renderizada, nesse caso nas linhas 24 a 26 é dito para usar *Buffers Objects* na renderização da geometria, sendo que, se a placa gráfica não der suporte a *Buffers Objects* então, automaticamente serão utilizados *Vertex Arrays*, conferindo desta forma uma maior consistência ao módulo de visualização do sistema.

Nas linhas 29 e 30, é criado e inicializado o grafo de cena representando a triangulação, e nas linhas 33 a 35 é criado um visualizador responsável pela renderização da triangulação.

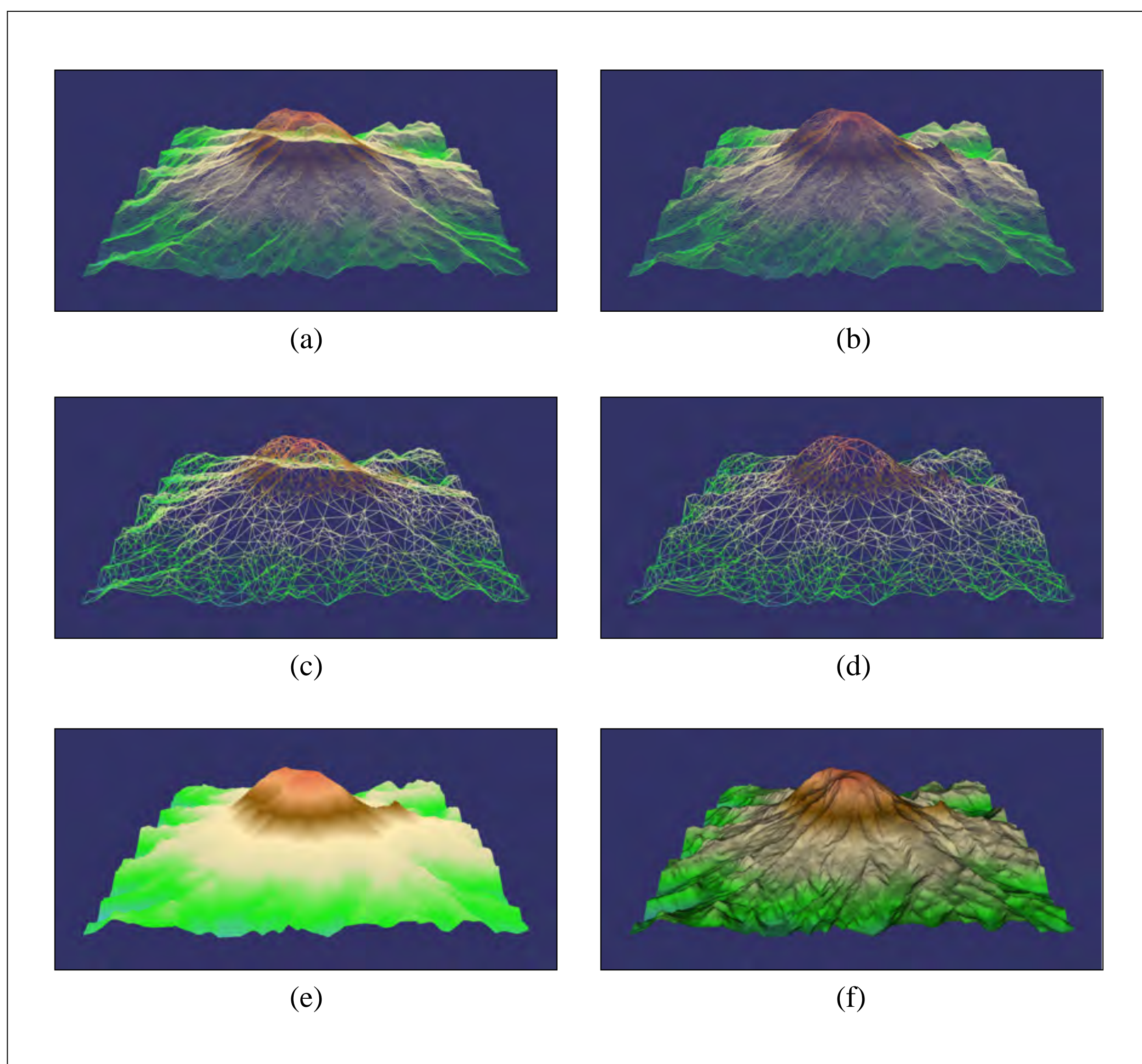


Figura 44: Modos de Renderização de um TIN: (a) Renderização por pontos; (b) Renderização por pontos com a eliminação de pontos escondidos; (c) Renderização *wireframe*; (d) Renderização *wireframe* com a eliminação de linhas escondidas; (e) Renderização sólida sem iluminação; (f) Renderização sólida com iluminação.

Como é mostrado na Figura acima, foram implementados os seguintes modos de renderização de terrenos no presente sistema:

- **Renderização de Pontos:** nesse caso são renderizados apenas os vértices do modelo **TIN** (Figura 44a), sendo permitido a renderização com eliminação de pontos escondidos (Figura 44b).

- **Renderização WireFrame:** nesse caso são renderizados apenas a aresta do modelo **TIN** (Figura 44c), também sendo permitido ao usuário habilitar a eliminação de arestas escondidas (Figura 44d).

- **Renderização Sólida:** nesse caso são renderizadas as faces triangulares do **TIN**. Essas podem ser renderizadas sem a aplicação de modelos de iluminação (Figura 44f) ou com modelo de iluminação (Figura 44e).

No caso dos modos de renderização com a eliminação de pontos ou linhas escondidas (Figuras 44b e 44d) foi utilizada uma técnica de renderização em dois passos descrita por McReynolds e Blythe (2005).

Esse método assume que o objeto a ser renderizado é compostos por polígonos. Primeiramente o objeto é renderizado como polígonos, sendo que apenas o *buffer* de profundidade é atualizado (não atualiza o *framebuffer*). Como nessa primeira etapa só o *buffer* de profundidade é atualizado, a renderização pode ser feita sem modelos de iluminação ou aplicação de textura, só interessando o valor de profundidade dos pixeis gerados.

Na segunda etapa de renderização, são desenhados apenas os pontos ou linhas dos polígonos, sendo que o buffer de profundidade criado na primeira etapa permite apenas o desenho de pontos ou linhas que não são obscurecidos pelos polígonos renderizados anteriormente. Por exemplo, no caso da renderização de um terreno, nenhum ponto ou linha obscurecido, por exemplo, por uma colina é renderizado, nesse caso para casa pixel renderizado é feito um teste com buffer de profundidade criado anteriormente para saber se ele é escurecido ou não.

Desta forma, foram executados os seguintes passos na renderização do terreno com a eliminação de pontos ou linhas escondidas:

1. Desabilitar a escrita de dados no FrameBuffer;
2. Habilitar a escrita no Buffer de Profundidade;
3. Renderizar apenas os triângulos do **TIN**;
4. Habilitar a escrita de dados no FrameBuffer;
5. Renderizar apenas as arestas ou vértices do **TIN**;

Neste caso a eliminação de pontos ou linhas escondidas tem influência apenas no aspecto visual, tendo por objetivo apenas clarificar e melhorar a aparência da renderização do terreno quando utilizados os modos de renderização por pontos ou linhas, não contribuindo desta forma para uma ganho de performance do sistema. Na realidade como nesses modo o terreno deve ser renderizado duas vezes, o uso dos modos de renderização

de pontos e linhas sem a eliminação de geometria escondida tem uma melhor performance do que os modos de renderização de pontos e linhas com eliminação.

5.6.5 Interação com a cena 3D do Terreno

Segundo Martz (2007), aplicações 3D devem permitir à interação do usuário com a cena 3D mostrada, como por exemplo a seleção de porções da imagem exibida. Desta maneira, no presente trabalho, além da questão de renderização do terreno, uma outra preocupação no desenvolvimento do sistema foi dar suporte à interação do usuário com a cena 3D utilizando o mouse.

A biblioteca de *software* **OSG** oferece uma série de funcionalidades que permitem uma iteração eficiente do usuário com o grafo de cena (ex: seleção de parte ou regiões da cena e a retirada de informações dessas áreas selecionadas), por meio das rotinas:

- intersecção entre um segmento de reta e a cena 3D;
- intersecção entre politopos definidos por uma série de planos e a cena 3D;
- intersecção entre um plano e a cena 3D;

As rotinas de teste de intersecção fornecidas pelo **OSG** permitem que ao clicar ou passar com o mouse sobre alguma região da janela de visualização determinar quais objetos ou quais porções desses estão embaixo dele. Por exemplo, dada uma cena 3D de um terreno mostrada em uma janela, o **OSG** permite verificar se o mouse está sobre o terreno e a retirada de informações como sobre qual face do terreno o mouse se encontra, ou os vértices e arestas que formam essa face, não interessando se a cena está transladada, rotacionada, escalada etc. O **OSG** se encarrega de fazer todo o mapeamento das coordenadas (x, y) de tela do mouse, para as coordenadas de mundo (x, y, z) da cena 3D.

Como é mostrado na Figura 45, as rotinas de teste de intersecção entre o mouse e o terreno renderizado foram utilizadas para que o usuário ao passar o mouse sobre o terreno receba informações sobre as coordenadas (x, y)e elevação (z) daquele ponto no terreno.

A funcionalidade de verificação das coordenadas do terreno utilizando o mouse foi implementada em três passos:

1. Captura da posição do mouse sobre a janela gráfica onde o terreno é renderizado e transformação das coordenadas de tela do mouse para coordenadas de mundo (neste caso coordenadas do terreno);

2. Teste de intersecção, utilizando um segmento de reta, entre o mouse e o terreno renderizado;

3. Caso tenha ocorrido intersecção, retirada da informação sobre o ponto onde a intersecção ocorreu, e os vértices do triângulo que contém o ponto de intersecção.

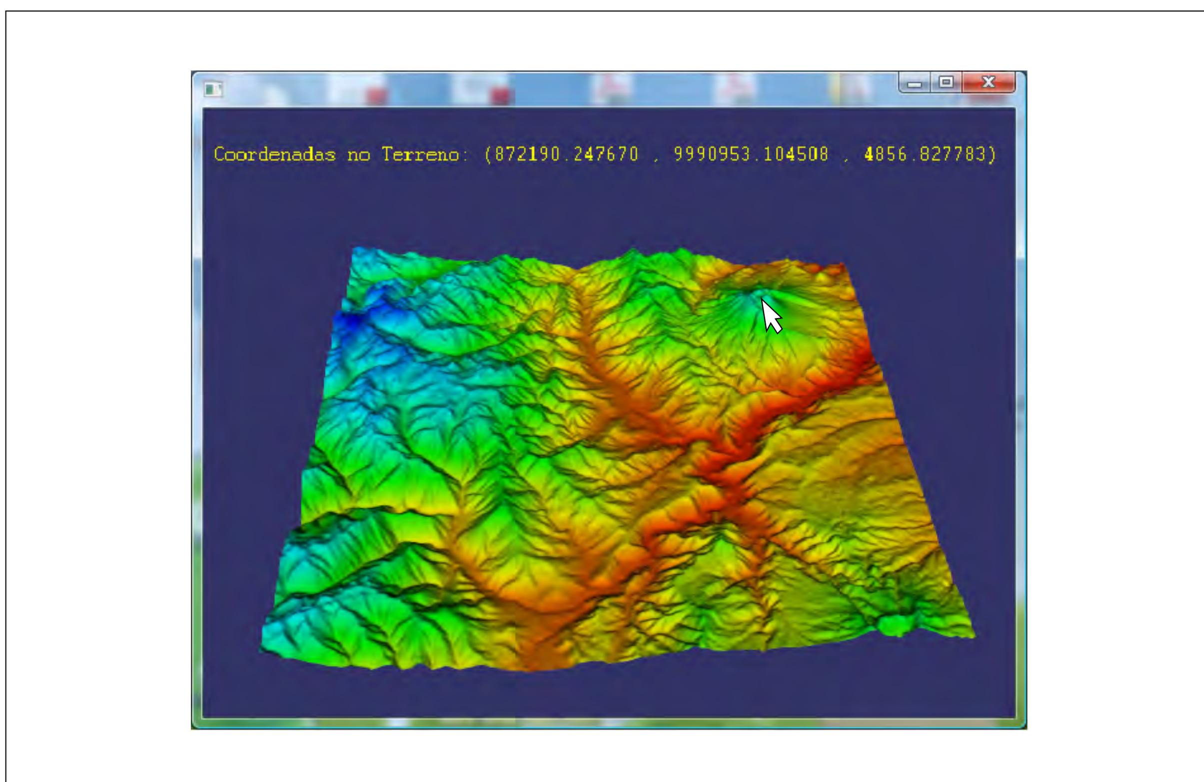


Figura 45: Teste de intersecção do mouse com o terreno.

A idéia do algoritmo é de "disparar um raio" a partir da posição do mouse na cena 3D e recuperar informações sobre as porções da cena atingidas por esse "raio". Esse modelo de seleção assume que a cena deve ser composta por polígonos, isto porque, não é suportada a intersecção entre segmentos e linhas ou segmentos e pontos. Para ser efetuado seleções com esses tipos de primitivas, pode ser utilizado, por exemplo, intersecção entre um politopo (definindo um volume ao redor do mouse), e a cena 3D (MARTZ, 2007).

Juntamente com as rotinas de intersecção, o **OSG** também permite a criação de uma *Kd-Tree* da cena modelada, aumentando dessa maneira a performance dos testes de intersecção, permitindo assim que esses possam ser executados em tempo real.

5.6.6 Navegação na cena 3D no Terreno

Para Metello et al. (2007), uma das maiores contribuições da tecnologia de desenvolvimento de jogos foi o uso de técnicas de computação gráfica no aperfeiçoamento de procedimentos de navegação em ambientes virtuais 3D. Ainda segundo Metello et al. (2007), o uso de técnicas de navegação 3D tem sido adotada pela comunidade **SIG** em sistemas como "Google Earth", para dar ao usuário uma experiência de navegação mais realista.

Desta forma, o sistema desenvolvido permite ao usuário executar uma navegação 3D no terreno modelado em tempo real utilizando o manipulador de câmera "*Trackball*" oferecido pela biblioteca **OSG**.

Como pode ser visto na Figura 46, o *trackball* virtual é um método de interface com o usuário que permite rotações arbitrárias de modelos tridimensionais utilizando cliques e movimentação do mouse em uma janela de visualização bidimensional.

De acordo com as Figuras 46a e 46b, esse método consiste em criar uma esfera imaginária inscrita na janela de visualização e com o centro projetado no centro da janela de visualização. Quando o usuário pressiona, por exemplo, o botão esquerdo do mouse na janela de visualização, a posição do mouse é projetada em um ponto dessa esfera imaginária. Quando o mouse é movimentado (ainda pressionado), essa esfera é rotacionada de forma a manter o mesmo ponto projetado na esfera sobre o ponteiro do mouse.

Por exemplo , movimentos horizontais do mouse (Figura 46(c)) podem rotacionar a esfera, e consequentemente toda a cena 3D, ao redor do eixo Z, movimentos verticais do mouse (Figura 46c) podem rotacionar a esfera (cena 3D) em torno do eixo X.

No caso da **OSG**, uma câmera virtual é rotacionada ao invés da cena, aplicando nesta câmera o inverso da matriz de rotação obtida a partir da movimentação do seu *trackball* virtual. Além dar suporte à rotações, por meio de *trackball* virtual, o **OSG** também permite fazer uma aproximação e afastamento da câmera em relação a cena 3D utilizando o mouse.

Basicamente a navegação 3D, utilizando o mouse sobre uma janela de visualização 2D, sobre um **TIN** renderizado pelo sistema é feita da seguinte maneira:

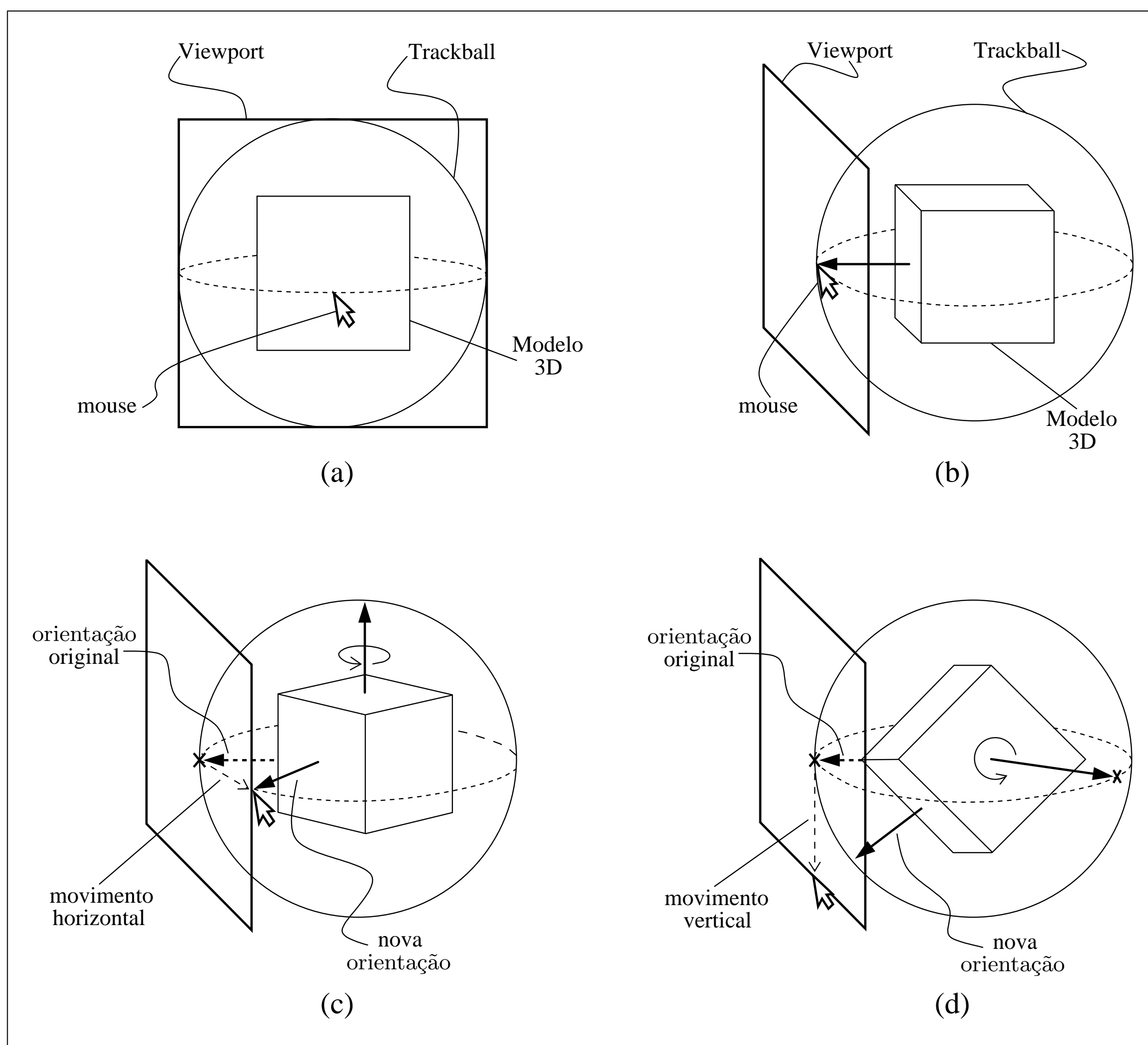


Figura 46: Funcionamento de *trackball* virtual.
Fonte: http://viewport3d.com/trackball.htm

Botão esquerdo pressionado com movimento vertical para esquerda: o terreno é rotacionado em torno do eixo Z no sentido anti-horário;

Botão esquerdo pressionado com movimento vertical para direita: o terreno é rotacionado em torno do eixo Z no sentido horário;

Botão esquerdo pressionado com movimento horizontal para cima: o terreno é rotacionado em torno do eixo X no sentido horário;

Botão esquerdo pressionado com movimento horizontal para cima: o terreno é rotacionado em torno do eixo X no sentido anti-horário;

Botão direito pressionado com movimento horizontal para cima: *zoom in* do terreno;

Botão direito pressionado com movimento horizontal para cima: *zoom out* do terreno;

Botão do meio pressionado com movimento horizontal para cima: translação positiva do terreno sobre o eixo Z;

Botão do meio pressionado com movimento horizontal para cima: translação negativa do terreno sobre o eixo Z;

5.7 Interface Gráfica com o Usuário

O uso de interfaces gráficas tem se tornado cada vez mais presente no desenvolvimento de sistemas de softwares, muitas vezes constituindo uma parte significante de sua implementação (JIMENEZ; IRIBARNE, 2005).

Em sistemas de modelagem digital de terrenos, a interação do usuário com o MDT, assim como a representação gráfica é fundamental para a compreensão do mesmo (por meio da analise visual). Em particular, operações como cliques, clique e araste, *zoom* e *pan* tem um papel de destaque na exploração visual deste (LI; ZHU; GOLD, 2005).

Segundo Jimenez e Iribarne (2005), todo o comportamento da interface gráfica (assim como do sistema de forma geral), pode ser melhor modelado e desenvolvido pelo uso de Máquinas de Estado Finitas Hierárquicas, também conhecidas como *StateCharts*.

Statecharts fornecem um mecanismo para se modelar graficamente o "ciclo de vida" do sistema, isto é, os possíveis estados em que o sistema pode estar (ex: esperando dados do usuário, executando uma triangulação e visualizando um **TIN**), a quais eventos o sistema pode responder (ex: carregar dados, executar triangulação e encerrar sua execução) e como ele deve reagir a esses eventos (ex: uma vez gerado o **TIN**, o evento executar triangulação não faz mais sentido).

A seguir é dada uma rápida descrição dos principais conceitos e componentes básicos de *Statecharts* (ALHIR, 2003; DRUSINSKY, 2006; LARGMAN, 2004):

Estados (*States*): condição ou situação específica em que o sistema pode se encontrar durante ciclo de vida.

Transição (*Transition*): representa um relacionamento entre dois estados indicando que quando um dado evento ocorrer, acontecerá uma mudança do estado corrente do sistema, nesse caso, do estado "origem" da transição para o seu estado destino.

Evento (*Event*): ocorrência significativa ou notável em um pico ou intervalo de tempo muito curto, por exemplo, uma requisição.

Na Figura 47 é mostrado um exemplo dado por Largman (2004) de como o comportamento de um telefone pode ser modelado utilizando *Statechart*. Percebe-se nessa figura que o telefone possui os estados "ocioso", enquanto estiver no gacho, e "ativo", quando estiver fora do gancho. Também pode ser observada a existência de duas transições que

fazem a troca do estado do telefone de “ocioso” para “ativo” e vice-versa. Essas transições são executadas conforme são gerados os eventos “tirar do gancho” e “colocar no gancho”, por exemplo: quando um evento “fora do gancho” ocorrer, é feita uma transição do telefone do estado “ocioso” para o estado “ativo”.

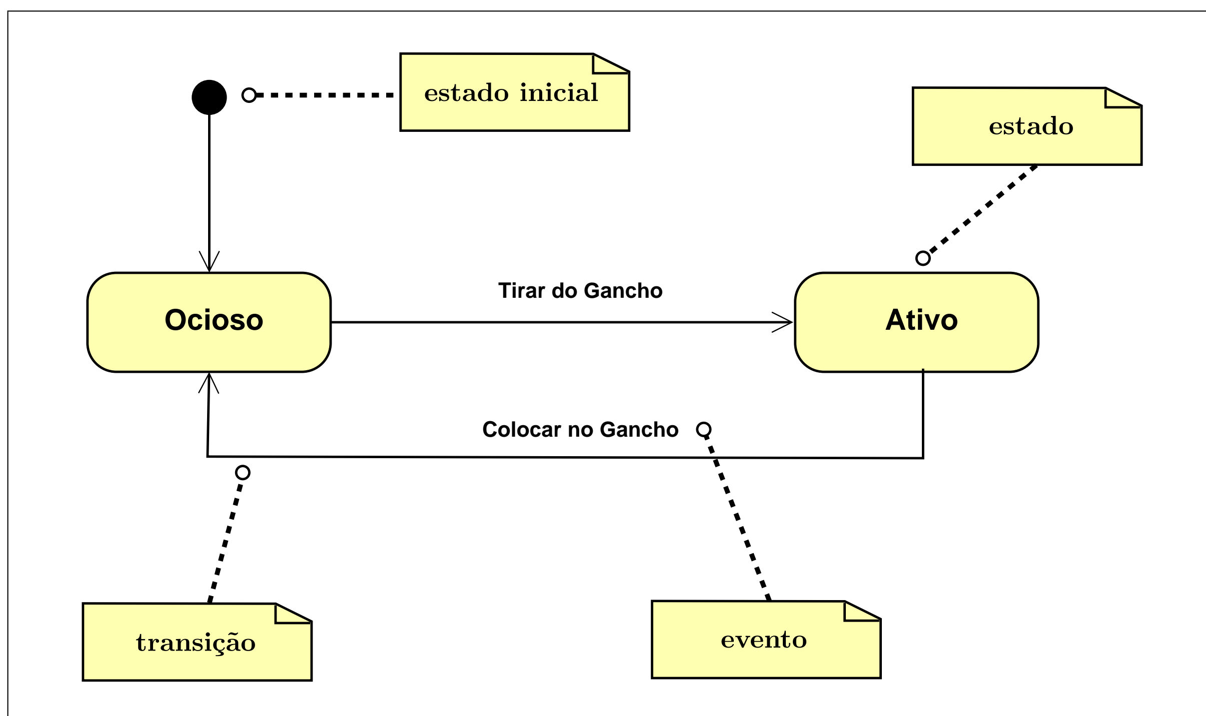


Figura 47: Diagrama de Estados modelando o comportamento de um telefone.
Fonte: adaptado de Largman (2004)

Basicamente *Statecharts* estendem o conceito de Máquinas de Estado Finitas, pela adição de novos componentes como estados paralelos, condições de guarda e tipos de eventos especiais (por exemplo toda vez que um estado se torna corrente é gerado o evento *onEnter* ou toda vez que um estado deixa de ser o estado corrente é gerado o evento *onExit*), reduzindo o número total de estados necessários para definir o comportamento do sistema, se mostrando dessa forma uma solução mais “escalável” do que Máquinas de Estado Finitas tradicionais (DRUSINSKY, 2006).

Entretanto, para Samek (2009), a contribuição mais importante de *Statechart*, em relação às Máquinas de Estado Finitas, foi a criação dos estados aninhados hierarquicamente (por esse motivo *Statecharts* também são chamados de Máquinas de Estados Hierárquicas). Esse novo tipo de estado permite que sub-estados reutilizem comportamento comum a partir do seus super-estados ou simplesmente ignorarem os eventos já tratados por estes, gerando assim uma “Herança Comportamental” de estados, isto é, comportamento ou eventos já tratados nos estados no nível superior da hierarquia não precisam ser tratados ou então podem ser reimplementados nos estados nos níveis inferiores.

No sistema desenvolvido, todo e qualquer tipo de iteração do usuário com o sistema foi feito por meio de componentes **GUI** (**G**raphical **U**ser **I**nterface). Mais precisamente, todos os aspectos relativos à criação e ao gerenciamento do janelas do sistema foram feitos com o auxilio da biblioteca **Qt**. Entretanto, com o intuito de lidar com toda a complexidade inerente ao desenvolvimento de sistemas de software que possuam interface gráfica e que tenham que lidar com eventos externos gerados pelo usuário, foi criada a camada intermediaria "Controle" responsável por gerenciar todos os eventos gerados pelo usuário por meio da interface gráfica do sistema. Desta forma, qualquer evento gerado pelo usuário por meio da **GUI**, é delegado ao "Controle" para que este faça o seu processamento e execute todas as ações necessárias de tal forma que a interface do sistema não tenha que lidar com a lógica da aplicação. Essa camada intermediária de controle foi implementada por meio de uma máquina de estado hierárquica.

O "Controle" foi implementado por meio da classe **Controller**. Na Figura 48 é mostrado um modelo UML da classe **Controller** e suas principais classes associadas.

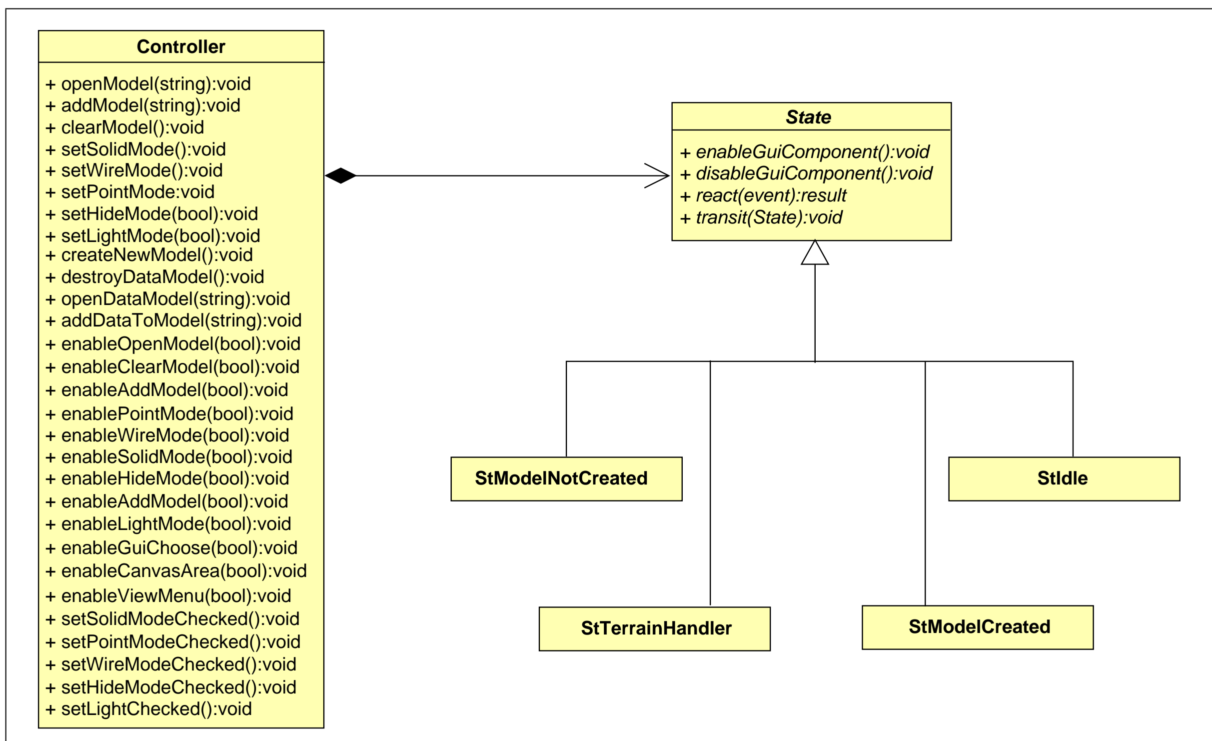


Figura 48: Diagrama de classe do Controller

Como já foi dito anteriormente, a classe **Controller** representa a máquina de estados responsável por processar todos os eventos gerados pela interface gráfica (janelas) do sistema, desta maneira não é de se admirar que essa classe possua uma grande quantidade de métodos. Entretanto toda essa funcionalidade não é manipulada de forma direta nessa classe, sendo que toda a sua lógica está distribuída entre as classes que representam os

possíveis estados em que sistema pode estar durante a sua execução. A seguir é dada uma descrição das classes responsáveis por modelar os estados do sistema:

- **StModelNotCreated:** essa classe representa o estado inicial do sistema quando nenhum modelo de terreno foi gerado. Esse estado é responsável por controlar toda a lógica de abertura de arquivos e geração do modelo que irá representar computacionalmente o terreno;
- **StModelCreated:** essa classe representa o estado do sistema quando um modelo de terreno já foi carregado. Toda a visualização e manipulação do terreno é feita nesse estado. Com o intuito de lidar com a complexidade dos processamentos, esse estado possui parte da sua funcionalidade dividida em dois estados internos **StIdle** e **StTerrainHandler** discutidos a seguir;
- **StIdle:** essa classe representa o estado do sistema quando um modelo digital de terreno já foi carregado mas ainda não é exibido no canvas. Esse estado contém toda funcionalidade que permite escolher qual modelo será visualizado no canvas. No presente momento apenas um modelo pode ser carregado por vez;
- **StTerrainHandler:** essa classe representa o estado do sistema quando um modelo digital de terreno já foi carregado e o mesmo está sendo visualizado e manipulado em um canvas. Nesse estado é feito todo o gerenciamento da visualização e navegação 3D;

Ainda, como pode ser observado na Figura 48, todos os estados possuem os métodos `enableGuiComponents()` e `disableGuiComponents()`. Esses métodos são executados, respectivamente, toda vez que o sistema entra e sai de um estado. Inicialmente o sistema possui todas as funcionalidades (botões, menus, canvas etc) da interface gráfica desabilitadas. O método `enableGuiComponents()` é responsável por habilitar apenas os componentes da interface utilizados pelo estado. Por exemplo, quando o sistema é executado, ele se encontra inicialmente no estado **ModelNotCreated**. Nesse estado a única funcionalidade possível no sistema é abertura de um arquivo com os dados para a construção do **MDT**. Sendo assim, os únicos componentes que devem estar habilitados na interface gráfica do sistema são os responsáveis pela abertura de arquivos com dados de elevação. Quando o sistema sai de um estado, o método `disableGuiComponents()` desabilita todos os componentes que foram habilitados na entrada do estado. Resumindo: inicialmente todos os botões e componentes da interface do sistema estão desabilitados, sendo que os

mesmos são habilitados/desabilitados durante a execução do sistema de acordo com o estado corrente do mesmo.

Também é possível observar na 48, que todos os estados possuem os métodos `react()` e `transit()`. O método `react()` permite que os componentes da interface gráfica do sistema emitam eventos para o controle de forma que o mesmo possa efetuar algum tipo de processamento (ex: abrir um arquivo e gerar o modelo **TIN**) ou mudar o seu estado corrente utilizando o método `transit()`.

Na Figura 49 é mostrado um modelo UML com o digrama da máquina de estados hierárquica implementada pela classe **Controller**, onde é possível ver os eventos e transições que são representados pelos métodos `react()` e `transit()`.

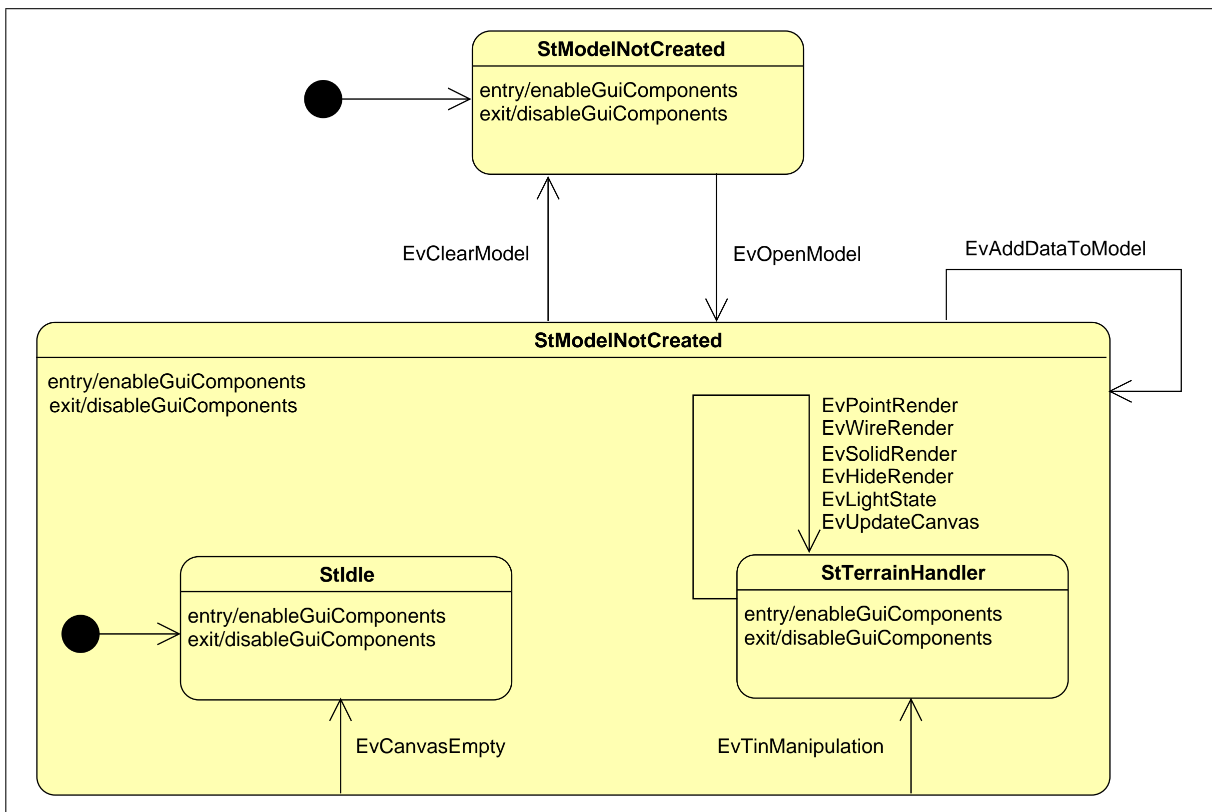


Figura 49: Diagrama de estados da classe Controller

O uso de uma máquina de estados, ajudou a lidar com a complexidade do sistema uma vez que muitas operações efetuadas são dependentes do seu estado de execução, não sendo convenientes, ou mesmo possíveis, durante alguns momentos da vida do programa.

Na Figura 50 é mostrada a janela principal do sistema desenvolvido.

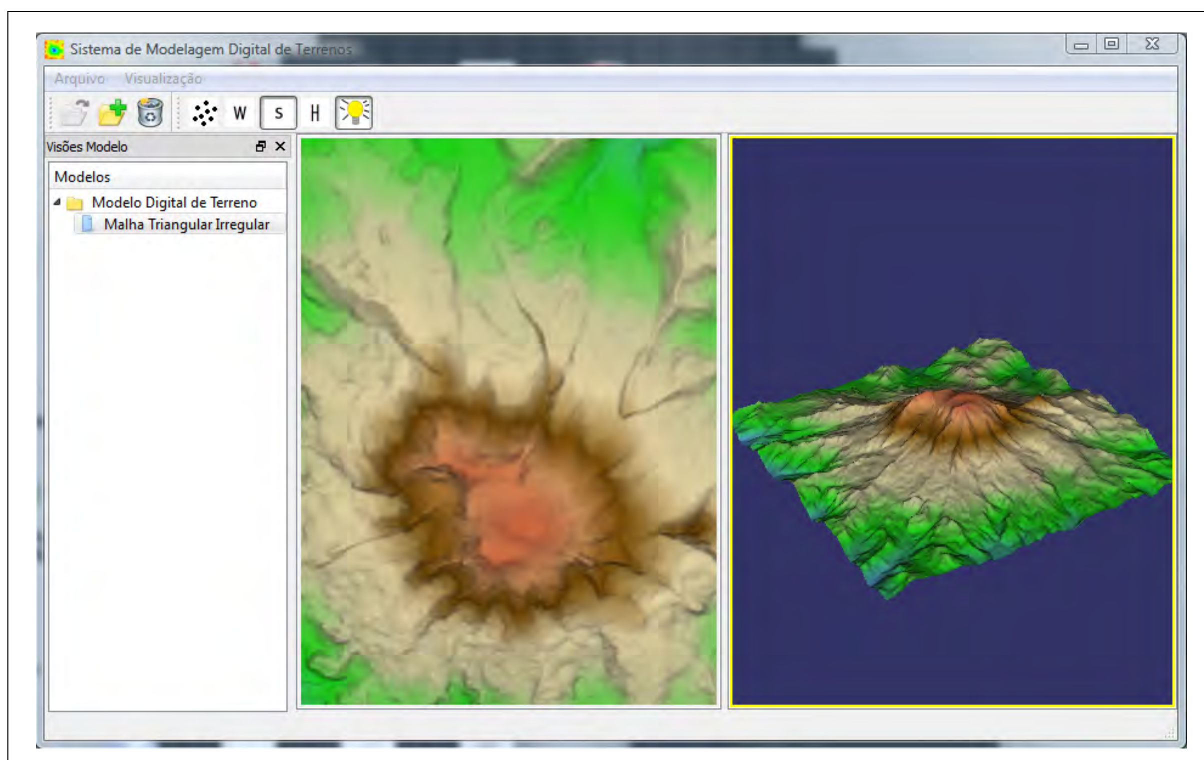


Figura 50: Renderização de MDTs

Apesar de ainda não integrado no sistema desenvolvido, outra preocupação foi de manter o sistema reativo (respondendo a eventos do usuário) durante processamentos intensos, neste caso, durante a geração do **TIN** (execução da triangulação).

Segundo Blanchette e Summerfield (2008), o **Qt** trata os eventos gerados pelos seus componentes gráficos da seguinte forma: enquanto um evento estiver sendo processado, eventos adicionais podem ser gerados e anexados à uma fila de execução de eventos. Entretanto, se um evento em particular (ex: geração da triangulação) consumir muito tempo de processamento, o sistema de janela para de responder, isto porque nenhum outro evento é tratado enquanto ele não terminar a sua execução.

Esse problema pode ser solucionado pelo uso de duas *thread*, isto é, uma *thread* para a execução da interface gráfica, e outra *thread* para a execução do evento ou processamento demorado. Usando essa abordagem, a interface gráfica se mantém responsiva (não trava ou pára de responder a eventos do usuário) independentemente de qualquer processamento que esteja sendo executado pelo sistema.

Desta maneira, a fim de verificar a validade dessa solução, foi desenvolvido um protótipo onde a interface gráfica e o método responsável por gerar a triangulação são executados em *threads* diferentes. Para isso foi utilizada a biblioteca de *QThread* para criação

e gerenciamento dessas *threads*.

Na Figura 51 é mostrada a interface gráfica que é executada concorrentemente com o processo de triangulação.

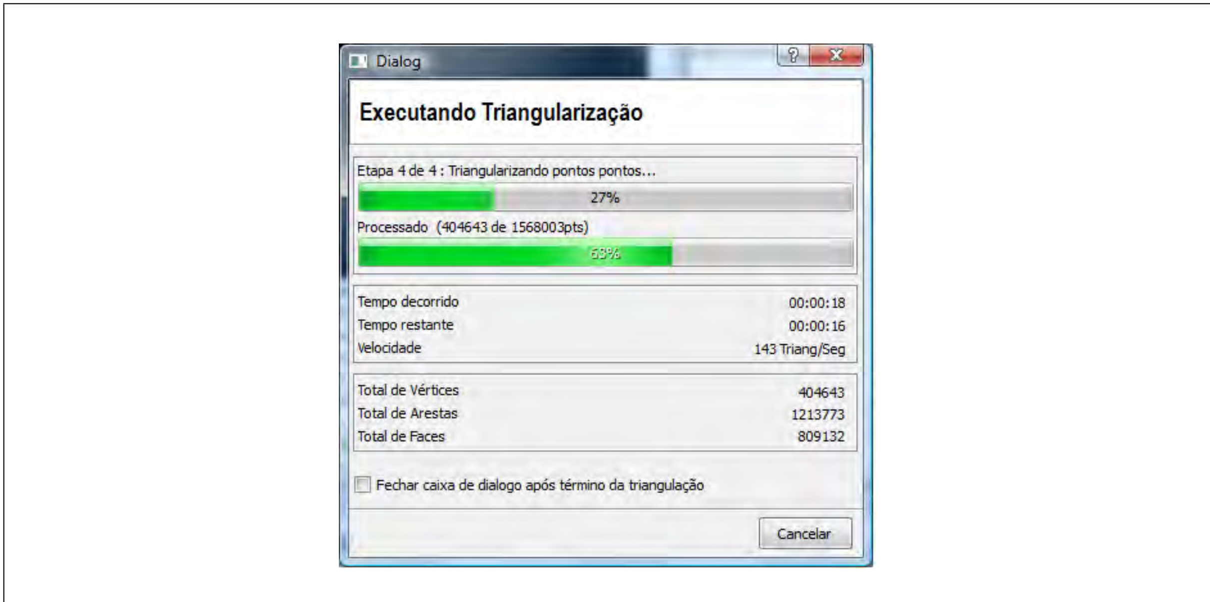


Figura 51: Janela de Visualização de Progresso da Triangulação.

Nesse protótipo, além da execução da **GUI** e da triangulação em *threads* separadas, também foi explorada a questão da intercomunicação entre essas duas *threads*. Como é mostrado na Figura 51, a interface gráfica exibe o estado corrente de execução da triangulação utilizando um temporizador que entre intervalos de um segundo, pega os dados sobre a situação corrente da triangulação (faz uma requisição para a sua *thread* de execução), como número de vértices, faces e arestas criados, e os exibe. A interface gráfica também faz o cálculo da velocidade de execução da triangulação (quantos triângulos são gerados por segundo), assim como uma projeção do tempo estimado para a sua conclusão.

6 *Conclusão*

Considerando a carência de ferramentas de software voltadas exclusivamente para a modelagem digital de terreno, torna-se necessário o desenvolvimento de novos sistemas computacionais que abordem essa problemática.

Desta forma, este trabalho teve como objetivo fazer a especificação da arquitetura de uma plataforma de *software* voltada para a Modelagem Digital de Terrenos visando não somente contribuir com uma nova opção de ferramenta *opensource*, mas com uma plataforma de desenvolvimento de software capaz de suportar um amplo conjunto de aplicações que façam uso de modelos digitais de terrenos. Apesar do projeto estar em sua fase inicial, de modo a ilustrar a sua viabilidade as seguintes funcionalidades foram desenvolvidas:

- Leitura de diferentes formatos de arquivos raster e vetorial, utilizando as bibliotecas **GDAL** e **OGR**, visando uma maior integração do sistema desenvolvido com sistemas de modelagem digital de terrenos já existentes;
- Geração de **TIN** a partir de pontos e linhas de contorno utilizando a técnica de Triangulação de Delaunay fornecida pela biblioteca **CGAL**;
- Visualização 3D de um **TIN** previamente gerado (utilizando as bibliotecas **OpenGL** e **OSG**), no modo ponto (apenas os vértices são renderizados), *wireframe* (apenas as arestas são renderizadas) e sólido (renderiza as faces triangulares), sendo ainda implementadas as funcionades de eliminação de pontos ou linhas escondidas na renderição por pontos e wireframe, assim como a possibilidade de habilitar ou desabilitar o uso de iluminaçao no caso do modo de renderização sólida;
- Criação uma interface gráfica para o sistema, utilizando as biblioteca **Qt**, de tal forma a facilitar o seu uso.

Dentre toda funcionalidade desenvolvida, merece destaque o fato de que o núcleo de representação de um **TIN** estar fundamentado sobre uma estrutura de dados topológica,

neste caso **EDT TDS** fornecida pela **CGAL**, que suporta de modo eficiente uma série de operações típicas realizadas sobre subdivisões planares como, por exemplo, consultas sobre relações de conectividade (adjacência e incidência) entre as entidades topológicas vértice, aresta e face de um **TIN** em tempo constante ou proporcinal ao número de entidades envolvidas.

Dada a amplitude do sistema e o fato da implementação do mesmo envolver o conhecimento de várias áreas da computação (ex: Engenharia de *Software*, Algoritmos, Computação Gráfica e Geometria Computacional), todas as funcionalidades descritas anteriormente foram desenvolvidas utilizando bibliotecas *opensource* já existentes e que têm se tornado padrões no mercado.

Contudo, a integração entre as várias bibliotecas utilizadas, se apresentou como um desafio adicional em relação à implementação do sistema. Desta maneira, todo o seu desenvolvimento foi executado seguindo a filosofia da programação orientada a objetos e programação genérica, também sendo feito, quando possível, o uso de padrões de projetos, garantindo dessa maneira uma melhor integração entre as tecnologias utilizadas. Ainda o uso dessas ferramentas e metodologias de programação também tiveram por objetivos facilitar a manutenção e a extensibilidade do sistema.

Outra questão abordada na implementação do sistema, foi o desenvolvimento do componente de software responsável por gerar e manter um **TIN** na forma de uma biblioteca de software que possa ser utilizada em outros projetos e que também sirva como uma camada de acesso unificada para outras implementações de **EDTs** e técnicas de triangulação além das fornecidas pela **CGAL**.

Em síntese, o projeto está em sua fase inicial e se constitui no desenvolvimento de uma plataforma de software voltada para a Modelagem Digital de Terreno, que permita avançar no desenvolvimento de outras aplicações em diferentes áreas do conhecimento.

6.1 Trabalhos Futuros

No sentido de mostrar a abrangência e o quanto o sistema proposto neste trabalho poderá evoluir, essa seção apresenta algumas funcionalidades que poderiam estender ou melhorar os recursos já existentes.

6.1.1 Geração do TIN

Apesar de o sistema desenvolvido também permitir a geração de **TIN**s a partir de pontos e linhas de contorno, a implementação do suporte e utilização de outras fontes de informações como por exemplo linhas de quebra, linhas estruturais e linhas de falésia podem aprimorar reconstrução da superfície do terreno.

6.1.2 Operações de Pós-Processamento do TIN

Uma vez gerado o **TIN** alguns processamentos podem ser executados sobre o mesmo como refinamentos pela inserção de pontos adicionais de maneira a eliminar triângulos muito longos ou quase degenerados ou então, a remoção de triângulos "ruins" (finos, longos ou muito longos) das bordas do **TIN**, melhorando dessa forma as características gerais da superfície gerada. Também podem ser executadas operações de simplificação do **TIN** gerando vários níveis de resolução do mesmo.

6.1.3 Suporte a Diferentes Sistemas de Projeção Cartográfica

No presente sistema, todos os pontos ou linhas de contorno utilizadas devem possuir coordenadas UTM. Futuramente outras projeções poderão ser suportadas utilizando ferramentas existentes como a biblioteca de funções para projeção de dados cartográficos *Proj.4* (NETTO; RIBEIRO, 2007).

6.1.4 Visualização

A renderização de **TIN**s pelo sistema implementado, pode ser melhorada pela execução de uma etapa de pré-processamento dos dados (ex: agrupar espacialmente os vértices e índices dos triângulos que serão renderizados e calcular *triangles strips*) antes de mandá-los para a GPU. Também podem ser utilizados níveis de detalhes (LOD).

Referências

ABDUL-RAHMAN, A. Digital terrain model data structures. *Buletin Ukur*, v. 5, p. 61–72, 1994.

ABDUL-RAHMAN, A.; PILOUK, M. *Spatial data modelling for 3D GIS*. New York: Springer-Verlag, 2007. 289 p.

AL-SALAMI, A. M. *TIN support in an open source spatial database*. 2009. Thesi in GeoInformatics — International Institute fo Geo-Information Science and Earth Observation, 2009.

ALHIR, S. S. *Learning UML*. Gravenstein Highway North, Sebastopol, CA, USA: O'Reilly, 2003. 252 p.

BARBOSA, R. L. *Geração de modelo digital do terreno por aproximações sucessivas utilizando câmaras digitais de pequeno formato*. Dissertação de Mestrado em Ciências — Faculdade de Ciências e Tecnologia/UNESP - Campus de Presidente Prudente, Presidente Prudente, dez. 1999.

BAUMGART, B. G. Winged-Edge polyhedron representation for computer vision. *National Computer Conference*, p. 589–596, 1975.

BERG, M. et al. *Computational geometry*: algorithms and applications. 3. ed. New York: Springer-Verlag, 2008. 386 p.

BLANCHETTE, J.; SUMMERFIELD, M. *C++ GUI programming with Qt 4*. 2. ed. Westford, Massachusetts USA: Prentice Hall, 2008. 752 p. ISBN 0-13-235416-0.

BOISSONNAT, J. D. et al. Triangulations in CGAL. *Computational Geometry Theory and Applications*, v. 22, p. 5–19, 2002.

BUCHMANN, F. et al. *A system of patterns*: pattern oriented software architecture. New York: JOHN WILEY & SONS, 1996. 157 p.

CAMARA, G. et al. Design patterns in GIS development: the TerraLib experience. In: III BRAZILIAN SYMPOSIUM ON GEOINFORMATICS, 3., 2001, Rio de Janeiro. Rio de Janeiro, 2001. p. 7.

CASACA, J.; MATOS, J.; BAIO, M. *Topografia geral*. 4. ed. Rio de Janeiro - RJ: LTC - Livros Técnicos e Científicos Editora S.A., 2007. 208 p.

DANOVARO, E. et al. Out-of-core multiresolution terrain modeling. In: BELUSSI, A. et al. (Ed.). *Spatial Data on the Web*. Berlin: Springer-Verlag, 2007. cap. 3, p. 43–63.

DEVILLERS, O. et al. Handles and circulators. In: *CGAL User and Reference Manual*. 3.6. CGAL Editorial Board, 2010. Disponível em: <http://www.cgal.org/Manual>. Acesso em: 18 mar. 2010.

DRUSINSKY, D. *Modeling and verification using UML statecharts*: a working guide to reactive system design, runtime monitoring and execution-based model checking. BOSTON: Elsevier Inc, 2006.

ECKEL, G.; JONES, K. *OpenGL Performer*: programmer guide. Mountain View, California: SGI techpubs library, 2004. 967 p.

EL-SHEIMY, N.; VALEO, C.; HABIB, A. *Digital terrain modeling*: acquisition, manipulation, and applications. Boston: Artech House, 2005. 257 p.

FABRI, A. et al. On the design of CGAL: a computational geometry algorithms library. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 30, n. 11, p. 1167–1202, 2000. ISSN 0038-0644.

FELGUEIRAS, C. A. Modelagem numérica de terrenos. In: CAMARA, G.; DAVIS, C.; MONTEIRO, A. M. V. (Ed.). *Introdução à ciência da geoinformação*. São José dos Campos: INPE, 2001, (Ciência e Engenharia da Geoinformação). cap. 7, p. 174–211. Disponível em: <http://www.dpi.inpe.br/gilberto/livro/introd>. Acesso em: 16 mar. 2010.

GAMMA, E. et al. *Design patterns*: elements of reusable object-oriented software. Reading, MA: Addison Wesley, 1998.

GDAL, D. T. *GDAL*: geospatial data abstraction library, version 1.7.2. 2010. Disponível em: <http://www.gdal.org>. Acesso em: 13/08/2010.

GOIS, J. P.; PITERI, M. A. Geração automática de malhas de elementos finitos e a estrutura de dados Winged-Edge modificada. In: XXIV CONGRESSO NACIONAL DE MATEMÁTICA APLICADA E COMPUTACIONAL - CNMAC 2001. *Seleta do XXIV CNMAC*. Belo Horizonte, 2001. v. 3, p. 121–130.

GRAPHICS, I. S. OpenGL: the industry's foundation for high-performance graphics. 1998.

HENDERSON, F. M.; LEWIS, A. J. (Ed.). *Principles & applications of imaging RADAR*: manual of remote sensing. 3. ed. New York: John Wiley & Sons Ltd, 1998. 866 p.

HJELLE, O.; DAEHLEN, M. *Triangulations and applications*. New York: Springer-Verlag, 2006. 234 p.

JIMENEZ, J.; IRIBARNE, L. Designing GUI components from UML use cases. *Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, Greenbelt, Maryland, p. 210 – 217, abr. 2005.

JONES, C. B.; KIDNER, D. B.; WARE, J. M. The implicit triangulated irregular network and multiscale spatial databases. *The Computer Journal*, v. 37, p. 43–57, 1994.

KETTNER, L.; NäHER, S. Two computational geometry libraries: LEDA and CGAL. In: GOODMAN, J. E.; O'ROURKE, J. (Ed.). *Handbook of Discrete and Computational Geometry*. Second. Boca Raton, FL: CRC Press LLC, 2004. cap. 64, p. 1435–1463.

KHRONOS GROUP. O*pen*GL: The industry's foundation for high performance graphics. 2010. Disponível em: <http://www.opengl.org/about/overview/>. Acesso em: 11 mai. 2010.

KILGARD, M. J. et al. Modern OpenGL: its design and evolution. *SIGGRAPHASIA2008*, 2008.

KUMLER, M. P. An intensive comparison of triangulated irregular networks (TINs) and digital elevation models (DEMs). *Cartographica: The International Journal for Geographic Information and Geovisualization*, v. 31, n. 2, p. 1 – 99, 1994. ISSN 1911-9925. Disponível em: <http://utpjournals.metapress.com/content/tm5674k7qh1t8575/>. Acesso em: 29 abr. 2010.

LANDIM, P. M. B. *Introdução aos métodos de estimação espacial para confecção de mapas*. 2000. Disponível em: <http://www.rlc.fao.org/es/prioridades/transfron/sig/pdf-/interpo.pdf>. Acesso em: 16 mar. 2010.

LARGMAN, C. *Applying UML and patterns*: an introduction to object oriented analysis and design and the unified process. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004. ISBN 0131489062.

LASATER, C. G. *Design pattern*: wordware applications library. Texas: Wordware Publishing, 2007. 386 p.

LEDOUX, H.; GOLD, C. Interpolation as a tool for the modelling of three-dimensional geoscientific datasets. In: PROCEEDINGS 4TH ISPRS WORKSHOP ON DYNAMIC AND MULTI-DIMENSIONAL GIS, Pontypridd, Wales, UK. *Anais...* Pontypridd, Wales, UK, 2005. p. 79–84.

LI, Z.; ZHU, Q.; GOLD, C. *Digital terrain modeling*: principles and methodology. London: CRC PRESS, 2005. 323 p.

MARTZ, P. *OpenSceneGraph quick start guide*: a quick introduction to the cross-platform open source scene graph API. California: Skew Matrix Software LLC, 2007. 136 p.

MCREYNOLDS, T.; BLYTHE, D. *Advanced graphics programming using OpenGL*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2005. ISBN 1558606599.

METELLO, M. et al. Continuous interaction with TDK: improving the user experience in TerraLib. In: IX BRAZILIAN SYMPOSIUM ON GEOINFORMATICS, 9., 2007, Campos do Jordão, SP. Campos do Jordão, SP, 2007. p. 13–22.

MILLER, C. L.; LAFLAMME, R. A. The digital terrain model: theory and application. *Photogrammetric Engineering*, v. 24, n. 3, p. 433–442, 1958.

MITAS, L.; MITASOVA, H. Spatial interpolation. In: LONGLEY, P. et al. (Ed.). *Geographical Information Systems: Principles, Techniques, Management and Applications*. 2. ed. [S.l.]: Wiley, 2005, (GeoInformation International). cap. 3, p. 404.

NAMIKAWA, L. M. et al. *Modelagem numérica de terrenos e aplicações*. São José dos Campos: INPE, 2003. 158 p. Disponível em: <http://mtc-m12.sid.inpe.br/col/sid.inpe-.br/marciana/2003/03.10.11.36/doc/publicacao.pdf>. Acesso em: 16 mar. 2010.

NETTO, S. O. A.; RIBEIRO, J. A. Emprego da biblioteca PROJ.4 nos sistemas de informação geográfica. In: EPIPHANIO, J. C. N.; GALVÃO, L. S.; FONSECA, L. M. G. (Ed.). *Anais...* São José dos Campos: Instituto Nacional de Pesquisas Espaciais (INPE), 2007. p. 2915–2921. ISBN 978-85-17-00031-7.

O'DOCHERTY, M. *Object oriented analysis and design*: understanding system development with UML 2.0. England: John Wiley & Sons Ltd, 2005. 559 p.

PETRIE, G.; KENNIE, T. J. M. Terrain modelling in surveying and civil engineering. *Computer-Aided Design*, Newton, MA, USA, v. 19, n. 4, p. 171–187, 1987. ISSN 0010-4485.

PION, S.; YVINEC, M. 2D Triangulation data structure. In: BOARD, C. E. (Ed.). *CGAL-3.6 User and Reference Manual*. CGAL Editorial Board, 2010. Disponível em: <http://www.cgal.org/Manual>. Acesso em: 18 mar. 2010.

PIRKELBAUER, P. et al. Runtime concepts for the C++ standard template library. In: *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2008. p. 171–177. ISBN 978-1-59593-753-7.

PITERI, M. A. *Geração automática de malhas hierárquico-adaptivas em domínios bidimensionais e tridimensionais*. Tese de Doutorado — Universidade Técnica de Lisboa - Instituto Superior Técnico, Lisboa, fev. 1999.

PITERI, M. A. et al. Triangulação de Delaunay e o princípio de inserção randomizado. *II Simpósio Brasileiro de Geomática - V Colóquio Brasileiro de Ciências Geodésicas*, Presidente Prudente - SP, p. 9, jul 2007.

PREPARATA, F. P.; SHAMOS, M. I. *Computational geometry*: an introduction. 2. ed. New York: Springer-Verlag, 1998. 398 p.

RASE, W.-D. Volume-preserving interpolation of a smooth surface from polygon-related data. *Journal of Geographical Systems*, Berlin, v. 3, n. 2, p. 199–213, ago 2001. ISSN 1435-5949.

RUHOFF, A. L.; HENDGES, E. R.; PEREIRA, R. S. *Curso de geoprocessamento*: processamento digital de imagens, modelagem numérica do terreno e análise espacial, geoestatística e programação em LEGAL. Rio Grande do Sul: Universidade Federal de Santa Maria, 2003. 41 p.

SAKUDE, M. T. S. Modelagem de terrenos por superfície triangulares de bezier. In: SIMPÓSIO BRASILEIRO DE COMPUTAÇÃO GRÁFICA E PROCESSAMENTO DE IMAGENS, 5., 1992, Aguas de Lindóia. *Anais...* Aguas de Lindóia, 1992. p. 213–222.

SAMEK, M. *Practical statecharts in C/C++*: event-driven programming for embedded systems. 2. ed. Burlington, MA, USA: Elsevier Inc., 2009. 712 p.

SCHNEIDER, P. J.; EBERLY, D. H. *Geometric tools for computer graphics*. San Francisco: Elsevier Science, 2003. 1007 p. (Computer Graphics and GeometricModeling). ISBN 1-55860-594-0.

SHEWCHUK, J. R. Adaptive precision floating point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, v. 18, p. 305–363, 1996.

SHREINER, D. et al. *OpenGL programing guide*: the official guide to learning OpenGL, version 1.4. 4. ed. Boston: Addison Wesley, 2004. 759 p.

SILVA, R. J. M. et al. Experiência de portais em ambientes arquitetônicos virtuais. *SVR 2003 - VI Symposium on Virtual Reality*, Ribeirão Preto, SP, Brasil, p. 117–128, Out 2003.

SILVA, R. J. M. da; RAPOSO, A. B.; GATTASS, M. Grafo de cena e realidade virtual. In: *Monografias em Ciência da Computação*. Rio de Janeiro - Brasil: Departamento de Informática, PUC-Rio, 2004. p. 52. Disponível em: <http://www.tecgraf.puc-rio.br/>. Acesso em: 26 mai. 2010.

SIMOES, M. G. *Modeladores digitais de terreno em sistemas de informação geográfica*. Mestrado em Ciência (M.Sc.) em Engenharia de Sistemas e computação. — Universidade Federal do Rio de Janeiro, Rio de Janeiro, Abr. 1993.

SINTES, T. *Teach yourself object oriented programming in 21 days*. Indianapolis: Sams Publishing, 2002. 698 p.

SLOCUM, T. A. *Thematic cartography and visualization*. New Jersey: Prentice Hall, 1999. 293 p.

SULEBAK, J. R. Applications of digital elevation models. *DYNAMAP Project*, p. 11, 2000.

VARELA, M. C. *Uma estrutura de dados para malhas progressivas*. Mestrado em Ciências em Engenharia de Sistemas e Computação — Universidade Federal do Rio de Janeiro, Rio de Janeiro, mar. 2004.

VELTKAMP, R. C. Generic geometric programming in the computational geometry algorithms library. *Computer Graphics Forum*, v. 18, n. 2, p. 131–137, 2001.

VERBREE, E.; OOSTEROM, P. van; QUAK, W. Computational geometry algorithms library in geographic information systems. In: *Proceedings of the First International Conference on Geographic Information Science*. Savannah: GIScience, 2000.

VINHAS, L. et al. Programação genérica aplicada a algoritmos geográficos. In: IV SIMPÓSIO BRASILEIRO DE GEOINFORMÁTICA, 4., GeoInfo2002, Caxambu, MG, Brazil,. *Anais...* Caxambu, MG, Brazil,, 2002. v. 1, p. 117–122.

WARMERDAM, F. The geospatial data abstraction library. In: HALL, G. B.; LEAHY, M. G. (Ed.). *Open Source Approaches in Spatial Data Handling*. Berlin: Springer-Verlag, 2008. cap. 5, p. 87–104.

WEIBEL, R.; HELLER, M. Digital terrain modelling. In: MAGUIRE, D.; GOODCHILD, M.; RHIND, D. (Ed.). *Geographical Information Systems*: Principles and applications. New York: John Wiley and Sons, 1991. cap. 19, p. 269–297.

WEILER, K. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Comput. Graph. Appl.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 5, n. 1, p. 21–40, 1985.

WOLF, P. R.; DEWITT, B. A. *Elements of photogrammetry*: with applications in GIS. 3. ed. Boston: McGraw-Hill Higher Education, 2000. 608 p. ISBN 0072924543.

WRIGHT, R. S.; LIPCHAK, B.; HAEMEL, N. *OpenGL SuperBible*: comprehesive tutorial and reference. 4. ed. Indianapolis, IN, USA: Addison Wesley, 2007. 1262 p.

YANG, C. et al. Twelve different interpolation metohds: a case study of Surffer 8.0. *INTERNATIONAL ARCHIVES OF PHOTOGRAMMETRY REMOTE SENSING AND SPATIAL INFORMATION SCIENCES*, v. 35, p. 778–785, 2004.

ZHOU, Q.; LEES, B.; TANG, G. *Advances in digital terrain analysis.* Berlin: Springer-Verlag, 2008. 463 p.