Um Visualizador para uma Extensão de CGAL ao Plano Projetivo Orientado

Fábio Pakk Selmi-Dei

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um Visualizador para uma Extensão de CGAL ao Plano Projetivo Orientado

Fábio Pakk Selmi-Dei

Maio de 2005

Banca Examinadora:

- Prof. Dr. Pedro Jussieu de Rezende (Orientador)
- Prof. Dr. Carlos Eduardo Ferreira
  IME-USP
- Prof. Dr. Alexandre Xavier Falcão
  IC-UNICAMP
- Prof. Dr. Siome Klein Goldenstein
  IC-UNICAMP

Substitua pela ficha catalográfica

Um Visualizador para uma Extensão de CGAL ao Plano Projetivo Orientado

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Fábio Pakk Selmi-Dei e aprovada pela Banca Examinadora.

Campinas, 11 de abril de 2005.

Prof. Dr. Pedro Jussieu de Rezende
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Substitua pela folha com a assinatura da banca

Prefácio

Visualizadores são softwares capazes de gerar, através de recursos gráficos computacionais, figuras geométricas a partir de estruturas de dados e seus estados. Suas imagens facilitam a compreensão e depuração de algoritmos, bem como aumentam a intuição do usuário sobre os objetos geométricos e o espaço que os abriga.

O presente trabalho descreve o projeto e a criação de um visualizador geométrico para uma extensão de CGAL ao plano projetivo orientado ($\mathbb{T}^2$).

CGAL é uma biblioteca de algoritmos geométricos e estruturas de dados desenvolvida por um consórcio de universidades com o objetivo de ser uma ferramenta de fácil acesso usada no desenvolvimento de aplicações que necessitem resolver problemas geométricos em $\mathbb{R}^2$. Através do trabalho [dO04], esta biblioteca foi estendida para incorporar $\mathbb{T}^2$, preservando sua robustez, corretude e confiabilidade.

O plano projetivo orientado é um espaço geométrico estritamente maior que o plano cartesiano $\mathbb{R}^2$, porém com geometria semelhante. Uma das principais características de $\mathbb{T}^2$ é o uso de coordenadas homogêneas sinaladas, o que permite lidar com o conceito de pontos no infinito de maneira homogênea ao tratamento dos pontos do plano, possibilitando o projeto de algoritmos geométricos que não mais precisam tratar separadamente muitos casos particulares, tornando-os mais simples e sucintos.

Neste contexto, o visualizador aqui descrito tem por finalidade a criação de um ambiente de visualização que permite a observação das características intrínsecas à geometria projetiva orientada, o que é de grande benefício para o usuário-programador da extensão de CGAL para $\mathbb{T}^2$.

Abstract

A graphical viewer is a software that enables the display of geometric figures from data structures and their varying states. The images it provides improve comprehension, make debugging easier and raise the users' intuition regarding geometric objects and their embedding space.

The present work describes the design and creation of a geometrical viewer for an oriented projective plane ($\mathbb{T}^2$) extension of CGAL.

CGAL is a library of geometric algorithms and data structures developed by a consortium of universities with the goal of producing an easy-to-use tool for building applications that require problem solving in $\mathbb{R}^2$. In [dO04], Oliveira describes an extension of this library that incorporates $\mathbb{T}^2$ into CGAL, while adhering to its robustness, correctness and reliability.

The oriented projective plane is a geometric space strictly larger than the Cartesian plane $\mathbb{R}^2$, though with similar geometry. One of the main features of $\mathbb{T}^2$ is the use of signed homogeneous coordinates, which enables one to work with points at infinity in a way similar to working with proper points on the plane, allowing for the design of algorithms that no longer need to handle many particular cases, making them simpler and shorter.

In this context, the viewer described here has the purpose of providing a visualization system that allows for the perception of the intrinsic characteristics of the oriented projective geometry, which is of great benefit to programmers of the extension of CGAL to $\mathbb{T}^2$.

Agradecimentos

Agradeço ao meu orientador, Pedro J. de Rezende, pela sua dedicação como professor e orientador. Sua contribuição para minha formação acadêmica inclui, também, a orientação em dois trabalhos de iniciação científica durante minha graduação. Sou grato pela sua confiança em minha capacidade.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo auxílio financeiro através de bolsa de mestrado e de iniciação científica, durante minha pós-graduação e graduação, respectivamente.

Agradeço, por fim, aos meus pais, meu irmão e pessoas muito próximas, pelo estímulo e relevância dado aos estudos e suporte afetivo. Aos meus pais, agradeço também pelo suporte financeiro.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Desde os primeiros estudos sobre geometria, milhares de anos atrás, a figura geométrica sempre esteve presente. Com linhas, circunferências e outras primitivas geométricas, as ilustrações foram um instrumento de grande valor para elucidar demonstrações, desenvolver a intuição e elaborar contra-exemplos.

A partir do surgimento da geometria computacional, na década de 1970, grandes esforços vêm sendo realizados a fim de representar primitivas geométricas através de estruturas de dados eficientes, permitindo que algoritmos geométricos resolvam problemas clássicos. Estes esforços, sem dúvida, têm grandes méritos e deram resultados. No entanto, manter a intuição geométrica nos estudos focados puramente no projeto e na implementação de algoritmos e estruturas de dados é uma tarefa difícil e, freqüentemente, pesquisadores desta área recorrem a figuras geométricas ainda feitas manualmente.

Visualizadores são ferramentas com a capacidade de gerar figuras geométricas automaticamente a partir de estruturas de dados e seus estados. Utilizando-se, para tanto, de recursos gráficos computacionais, os visualizadores produzem imagens pelo menos tão relevantes quanto as figuras geométricas. Imagens tridimensionais e a capacidade de animação são alguns dos recursos adicionais que este tipo de software proporciona. Através de animações, os visualizadores são capazes de exibir os passos de um algoritmo, demonstrando sua lógica, e de produzir movimentos contínuos que simulam um passeio virtual do observador em construções geométricas tridimensionais. Assim, estes softwares proporcionam o aumento da intuição do pesquisador, a compreensão dos alunos, a capacidade de detectar erros na implementação de algoritmos e estruturas de dados, bem como inspiram a criação de novos algoritmos e linhas de pesquisa.

Muitos visualizadores geométricos surgiram e grande parte deles são restritos ao plano cartesiano. Mais informações sobre estes trabalhos serão apresentadas no capítulo 2.

O presente trabalho propõe a criação de um visualizador para o plano projetivo orientado $\mathbb{T}^2$, um espaço geométrico semelhante ao plano cartesiano $\mathbb{R}^2$, porém estritamente

maior. Uma das principais propriedades de $\mathbb{T}^2$ é o uso de coordenadas homogêneas sinaladas, o que permite lidar com o conceito de pontos no infinito de maneira homogênea ao tratamento dos pontos do plano. Para a geometria computacional, isso possibilita a criação de algoritmos geométricos que não mais precisam tratar separadamente muitos casos particulares, tornando-os mais simples e sucintos. Suas principais propriedades e conceitos, assim como os modelos de visualização desse plano, são apresentados no capítulo 3.

O capítulo 4 descreve a biblioteca geométrica CGAL e sua extensão para incorporar o plano projetivo orientado. CGAL é uma biblioteca de algoritmos geométricos e estruturas de dados desenvolvida com o objetivo de ser uma ferramenta de fácil acesso usada no desenvolvimento de aplicações que necessitem resolver problemas geométricos. CGAL possui uma variedade de primitivas geométricas implementadas de maneira eficiente e rigoroso cuidado com a precisão numérica, corretude e robustez dos algoritmos. Através do trabalho [dO04], esta biblioteca foi estendida para incorporar $\mathbb{T}^2$, preservando sua robustez, corretude e confiabilidade.

O proposto visualizador para a extensão de CGAL ao plano projetivo orientado, denominado "$\mathbb{T}^2$ Viewer", está detalhado no capítulo 5. A especificação deste software abrange desde os principais objetivos e requisitos até aspectos de implementação e plataforma de desenvolvimento. Diagramas de diversos níveis de abstração são usados para complementar as descrições de projeto, e figuras produzidas por este visualizador são usadas para ilustrar algumas de suas funcionalidades.

O capítulo 6 descreve trabalhos futuros que podem ser realizados neste software e o capítulo 7 apresenta as principais conclusões e benefícios provenientes deste trabalho.

Capítulo 2

Trabalhos Anteriores

Neste capítulo estão descritos, em ordem cronológica, vários visualizadores geométricos que antecederam a presente dissertação.

Computational Geometry Workbench [Sac90]: Ambiente para criação, edição e manipulação de objetos geométricos e animação de algoritmos geométricos. Suas funcionalidades auxiliam na implementação e depuração de novos algoritmos. É implementado em Smalltalk V e funciona exclusivamente na plataforma Macintosh.

XTango [XTa90]: Sistema de animação de algoritmos onde um usuário sem conhecimentos específicos sobre animação é capaz de criá-las através de um paradigma abstrato de projeto de animações, presente no XTango, facilitando seu uso. Os algoritmos podem pertencer a qualquer domínio para serem animados. XTango é implementado em C, com código-fonte aberto e funciona na plataforma Unix/X11.

XYZ GeoBench [XYZ90]: Ambiente de programação geométrica, com ferramentas para a criação, edição e manipulação de objetos geométricos. Ele é implementado em Object Pascal, para a plataforma Macintosh, e tem código-fonte aberto.

Zeus [Bro91]: Sistema de animação de algoritmos com múltiplas janelas de visualização, cada uma das quais pode exibir um aspecto distinto sobre a funcionalidade do algoritmo bem como servir de entrada para a manipulação direta pelo usuário. Algoritmos de geometria computacional, de sistemas operacionais, de projeto de hardware e de protocolos de comunicação já foram animados através deste sistema. Zeus é implementado em Modula-3 e está disponível somente para algumas plataformas Unix.

Geomview [Geo92]: Interface para a visualização e manipulação de objetos geométricos, podendo ser utilizada como um visualizador para objetos estáticos ou objetos

dinamicamente gerados por outros programas. Geomview também pode servir como entrada de dados, recebendo os eventos gerados pelo usuário e os transmitindo a outros programas. Ele foi feito em C, com código-fonte aberto e funciona na plataforma Linux/Unix.

GeoLab [dRJ93]: Ambiente de programação e animação de algoritmos geométricos. GeoLab utiliza bibliotecas compartilhadas de algoritmos e uma abordagem incremental para a composição de novos objetos geométricos. GeoLab foi feito em C++, com código-fonte aberto e funciona sobre a plataforma SunOS/XView.

GeoSheet [Lee95]: Ferramenta para a visualização de algoritmos geométricos em ambientes distribuídos, permitindo o acesso remoto. A entrada e a saída do GeoSheet é baseada no programa Xfig. GeoSheet foi implementado em C++.

GeoPrO [Gon96]: Ambiente de programação e visualização de algoritmos geométricos na geometria projetiva orientada (descrita no capítulo 3). É possível incluir novos algoritmos e estruturas de dados neste ambiente que, pela sua natureza distribuída, permite a interação simultânea de mais de um terminal de computador conectado a um servidor central que executa o núcleo da aplicação. A visualização e a execução do algoritmo ficam, portanto, desvinculadas. Este ambiente foi feito em C++ e é livre, com código-fonte aberto. Ele funciona exclusivamente na plataforma IRIX que executa em estações gráficas Silicon Graphics.

A tabela 2.1 resume as principais características de cada visualizador geométrico descrito acima. Outras ferramentas mais simples de visualização existem e são mencionadas na seção 4.1 em vista de sua relevância para o presente trabalho.

Visualizador	**Linguagem de Programação**	**Plataforma**	**Interação com Usuário**	**Suporte a Animação**
C. G. Workbench	Smalltalk V	Macintosh	Entrada e Saída	Sim
XTango	C	Unix/X11	Saída apenas	Sim
XYZ GeoBench	Object Pascal	Macintosh	Entrada e Saída	Sim
Zeus	Modula-3	Unix	Entrada e Saída	Sim
Geomview	C	Linux/Unix	Entrada e Saída	Sim
GeoLab	C++	SunOS/XView	Entrada e Saída	Sim
GeoSheet	C++	Unix	Entrada e Saída	Não
GeoPrO	C++	IRIX	Entrada e Saída	Não

Tabela 2.1: Resumo dos Visualizadores Geométricos Apresentados

Capítulo 3

Plano Projetivo Orientado $\mathbb{T}^2$

Neste capítulo descrevemos sucintamente o plano projetivo orientado $\mathbb{T}^2$ e como este se relaciona com o plano cartesiano $\mathbb{R}^2$. Através de algumas definições, e do uso de coordenadas homogêneas, mostramos os benefícios mais evidentes proporcionados pela geometria projetiva orientada ([Sto91, dRS94]). Em seguida, mostramos como esta geometria (no caso de duas dimensões) pode ser representada através de dois modelos de $\mathbb{T}^2$.

3.1 Definições

O plano projetivo orientado $\mathbb{T}^2$ é um espaço geométrico semelhante ao plano cartesiano $\mathbb{R}^2$, porém estritamente maior. Enquanto um ponto de $\mathbb{R}^2$ é representado pelas coordenadas cartesianas (x, y), onde $x, y \in \mathbb{R}$, um ponto de $\mathbb{T}^2$ é representado por coordenadas homogêneas $[w, x, y]$, onde $w, x, y \in \mathbb{R}$, com exceção da tupla $[0, 0, 0]$.

A relação entre os dois espaços geométricos $\mathbb{T}^2$ e $\mathbb{R}^2$ é consolidada através da seguinte relação entre suas respectivas coordenadas:

$$[w, x, y] = \left(\frac{x}{w}, \frac{y}{w}\right)$$

se $w \neq 0$.

Seja $p = [w, x, y] \in \mathbb{T}^2$, onde $w, x, y \in \mathbb{R}$ e $p \neq [0, 0, 0]$. Se $w > 0$, dizemos que p está no *aquém*, se $w < 0$ dizemos que p está no *além*. Se $w = 0$ (com $x \neq 0$ ou $y \neq 0$), dizemos que p está no infinito. Portanto, $[0, x, y]$, com $x, y \in \mathbb{R}$, é uma representação conveniente do ponto no infinito, na direção $\overrightarrow{(x, y)}$, em coordenadas homogêneas, desde que $[0, x, y] \neq [0, 0, 0]$.

Essa característica é uma das grandes vantagens do plano projetivo orientado sobre o plano cartesiano, pois aquele consegue lidar com o conceito de pontos no infinito de maneira homogênea ao tratamento dos demais pontos do plano. Para geometria compu-

tacional, isso possibilita a criação de algoritmos geométricos que não mais precisam tratar separadamente muitos casos particulares, tornando-os mais simples e sucintos.

Denominamos *antípoda* de p o ponto $\neg p = [-w, -x, -y]$. Embora p e $\neg p$ sejam representados pelo mesmo ponto de $\mathbb{R}^2$ (quando $w \neq 0$), $p \neq \neg p$. Dizemos que dois pontos $p, q \in \mathbb{T}^2$ são equivalentes se e somente se um for múltiplo positivo do outro. Em outras palavras, $q = [\alpha w, \alpha x, \alpha y]$, $\alpha > 0$ e $\alpha \in \mathbb{R}$, se e somente se $q = p$.

Uma reta de $\mathbb{T}^2$ é representada pela tripla $\langle W, X, Y \rangle$, com $W, X, Y \in \mathbb{R}$, exceto a tripla $\langle 0, 0, 0 \rangle$. O ponto $p = [w, x, y]$ pertence à reta $\langle W, X, Y \rangle$ se e somente se $Ww + Xx + Yy = 0$. O leitor pode verificar que se $p \in \langle W, X, Y \rangle$ então $\neg p \in \langle W, X, Y \rangle$. Obviamente, a afirmação no sentido inverso também é verdadeira, portanto $p \in \langle W, X, Y \rangle$ se e somente se o antípoda de p também pertence à reta, isto é, $(p \in \langle W, X, Y \rangle) \Leftrightarrow (\neg p \in \langle W, X, Y \rangle)$.

A geometria projetiva orientada admite todos os conceitos e operações da geometria euclidiana. No entanto, aquela possui vantagens adicionais: o uso de coordenadas homogêneas freqüentemente resulta em fórmulas mais simples, envolvendo apenas o uso de operações básicas da álgebra linear. Além disso, as coordenadas homogêneas permitem um tratamento consistente de pontos no infinito.

Intencionalmente apresentamos um resumo muito sucinto sobre este assunto, já que há literatura que trata sobre a geometria projetiva orientada em profundidade. O leitor sem familiaridade com esta geometria deve procurar [Sto91, dRS94].

3.2 Modelos de $\mathbb{T}^2$

Geometricamente, o plano projetivo orientado pode ser representado através de um modelo plano e de um modelo esférico, descritos a seguir.

3.2.1 Modelo Plano

O modelo plano consiste de duas cópias de $\mathbb{R}^2$ mais uma circunferência unitária S^1 que representa os pontos no infinito, como ilustra a figura 3.1.

A cópia de $\mathbb{R}^2$ onde $w > 0$ é denominada aquém e a cópia de $\mathbb{R}^2$ onde $w < 0$ é denominada além. Os pontos com coordenada $w = 0$ constituem a reta no infinito, que é representada por Ω.

Quando $w \neq 0$, o ponto $[w, x, y] \in \mathbb{T}^2$ é representado pelo ponto $\left(\frac{x}{w}, \frac{y}{w}\right)$ do aquém se $w > 0$ ou do além se $w < 0$. Quando $w = 0$, com $x \neq 0$ ou $y \neq 0$, o ponto de Ω é representado em S^1 pelas seguintes coordenadas:

$$\left(\frac{x}{\sqrt{x^2 + y^2}}, \frac{y}{\sqrt{x^2 + y^2}}\right)$$

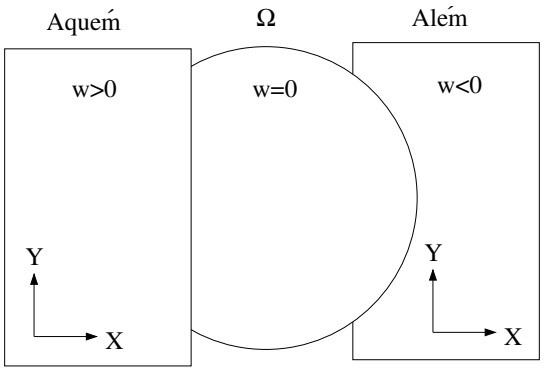

Figura 3.1: Modelo Plano

Um segmento de $\mathbb{T}^2$, entre os pontos p_0 no aquém e p_1 no além, é representado no modelo plano por duas semi-retas euclidianas, uma no aquém e outra no além com os mesmos coeficientes cartesianos e sentidos opostos, e mais um ponto no infinito na direção da semi-reta do aquém, como mostra a figura 3.2, onde o tracejado representa a extensão do segmento para uma reta.

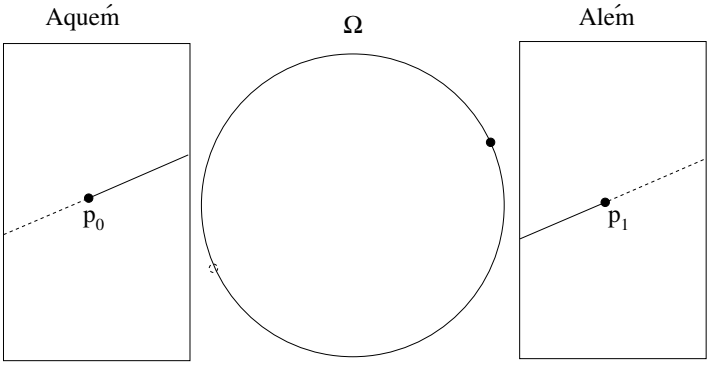

Figura 3.2: Segmento no Modelo Plano

3.2.2 Modelo Esférico

O modelo esférico consiste da superfície da esfera unitária S^2 de $\mathbb{R}^3$ centrada na origem, como ilustra a figura 3.3.

O ponto $[w, x, y] \in \mathbb{T}^2$ é representado pela intersecção da superfície S^2 com o segmento de reta que liga a origem ao ponto $(w, x, y) \in \mathbb{R}^3$, que é o ponto de S^2 com coordenadas:

$$\frac{(w, x, y)}{\sqrt{w^2 + x^2 + y^2}}$$

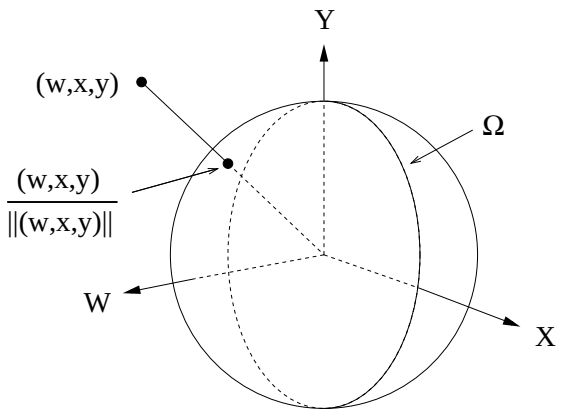

Figura 3.3: Modelo Esférico

O aquém e o além de $\mathbb{T}^2$ correspondem, portanto, aos hemisférios de S^2 com $w > 0$ e $w < 0$ respectivamente. A circunferência de raio máximo dada pela intersecção entre S^2 e o plano $w = 0$ representa os pontos no infinito. No modelo esférico, um par de pontos antipodais é representado por um par de pontos diametralmente opostos sobre S^2.

Um segmento de $\mathbb{T}^2$, entre os pontos p_0 no aquém e p_1 no além, é representado no modelo esférico pelo menor arco de circunferência máxima que passa por p_0 e p_1, como mostra a figura 3.4, onde o tracejado representa a extensão do segmento para uma reta.

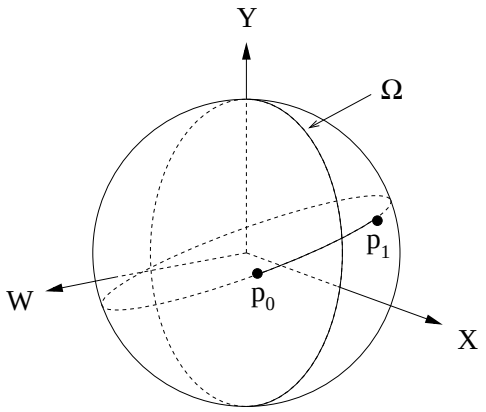

Figura 3.4: Segmento no Modelo Esférico

Capítulo 4

Trabalhos Diretamente Relacionados

Neste capítulo estão descritos dois trabalhos que se relacionam de maneira direta com o presente trabalho, uma vez que seus códigos-fonte fazem parte do $\mathbb{T}^2$ Viewer.

4.1 CGAL

CGAL [CGA96] (Computational Geometry Algorithms Library) é uma biblioteca de algoritmos geométricos e estruturas de dados desenvolvida em C++, de fácil acesso, usada no desenvolvimento de aplicações que necessitem resolver problemas geométricos; possui uma variedade de primitivas geométricas implementadas de maneira eficiente e rigoroso cuidado com a precisão numérica, corretude e robustez dos algoritmos.

A biblioteca CGAL trabalha em $\mathbb{R}^2$ e é composta dos seguintes módulos funcionais:

Kernel: Núcleo contendo primitivas geométricas como ponto, segmento, circunferência e reta. Estão presentes também predicados como teste de posição relativa entre pontos, intersecções e cálculos de distância.

Basic Library: Coleção de algoritmos geométricos comuns, tais como o fecho convexo 2D/3D, triangulação de Delaunay 2D/3D, mapa planar, menor círculo envolvente e estruturas de busca multi-dimensionais.

Support Library: Interfaces de programação para outros módulos, como sistemas de entrada e saída e programas de visualização.

Os visualizadores compatíveis com CGAL, através da `Support Library`, trabalham em $\mathbb{R}^2$ e estão descritos a seguir:

Qt widget: Sistema de visualização feito exclusivamente para CGAL. Não é considerado um visualizador propriamente por não ser um software *standalone*. Pelo contrário,

sempre há a necessidade de se programar seus eventos de interação com o usuário antes que ele possa ser usado. Qt widget possui uma interface de programação clara e minimal e pode ser usado tanto como sistema de entrada como de saída. É feito em C++, com código-fonte aberto e funciona tanto na plataforma Linux/Unix como em Windows.

Geomview: Este visualizador possui interface de programação para vários softwares, incluindo CGAL. O usuário é capaz de interagir com os objetos geométricos através de aproximação (*zoom*) , rotação (*orbit*) e translação (*pan*) em tempo real.

GeoWin: Originalmente feito para a biblioteca LEDA [Gmb98], GeoWin funciona exclusivamente em Windows. Apesar de ser um software de livre uso no meio acadêmico, seu código-fonte é fechado, o que restringe seu desenvolvimento. No entanto, diferentemente do Qt widget, GeoWin possui uma boa gama de eventos de entrada disponíveis, o que facilita o trabalho do programador quando se trata da interação com o usuário.

CGAL é de livre uso para o meio acadêmico, com código-fonte aberto e funciona nas plataformas Linux/Unix e Windows.

4.2 Extensão de CGAL para $\mathbb{T}^2$

Através do trabalho realizado por Oliveira [dO04], a biblioteca CGAL foi estendida para incorporar a geometria projetiva orientada em duas dimensões (descrita no capítulo 3). O conjunto das primitivas geométricas que foram estendidas é constituído por pontos, retas, segmentos, semi-retas, vetores, direções, triângulos, retângulos isotéticos, circunferências e caixas envolventes. Cada primitiva recebeu cuidadosa atenção quanto ao tipo de representação usada, as características especiais apresentadas em $\mathbb{T}^2$ e sua visualização nos modelos plano e esférico.

Foram estendidas também transformações afins, como rotação, translação e escala que operam sobre as primitivas geométricas de $\mathbb{T}^2$. Já a coleção de predicados estendidos abrange testes de intersecção entre as primitivas básicas, localização de pontos em polígonos simples, teste de orientação de trio de pontos, testes envolvendo comparação lexicográfica e ordenação de pontos colineares, comparação de distâncias relativas entre pontos próprios ($w \neq 0$) e impróprios ($w = 0$) entre outros.

O conjunto de algoritmos geométricos que foram inicialmente estendidos é constituído por casco convexo, triangulação arbitrária, triangulação de Delaunay, diagrama de Voronoi, árvore geradora de distância mínima, determinação do maior círculo vazio de sítios com centro próprio, construção do grafo de todos os vizinhos mais próximos, localização

de pontos em subdivisões planares e busca em amplitude com o iso-retângulo. O leitor interessado deve consultar [dO04] para maiores detalhes.

De acordo com a estrutura de módulos funcionais da biblioteca CGAL, apresentada na seção 4.1, o trabalho de extensão de CGAL para $\mathbb{T}^2$ necessitou modificar os módulos funcionais `kernel` e a `basic_library`. O presente trabalho, sendo um visualizador para uma extensão de CGAL, necessitou modificar o módulo funcional `support_library` apenas.

Capítulo 5

Visualizador Geométrico $\mathbb{T}^2$ Viewer

Neste capítulo, estão descritas todas as etapas relevantes à criação do $\mathbb{T}^2$ Viewer. Primeiramente, partindo do ponto de vista mais abstrato, estudaremos os principais objetivos e motivações referentes a criação de um visualizador geométrico, baseando-se em características já vistas em trabalhos anteriores, bem como em funcionalidades inovadoras. A descrição do ambiente de desenvolvimento ajuda a completar, formalmente, a etapa de requisitos.

Em seguida, uma análise mostra como o software é conceitualmente estruturado de modo a satisfazer os requisitos. A etapa de projeto descreve, em detalhes, os módulos funcionais que fazem parte do $\mathbb{T}^2$ Viewer, as relações hierárquicas entre eles e como cada um foi implementado, de maneira a prover uma ou mais funcionalidades ao ambiente. As interfaces gráficas e de programação completam, por fim, a descrição do $\mathbb{T}^2$ Viewer.

5.1 Objetivos

A principal motivação de se construir um visualizador para o plano projetivo orientado é a de se observar as características intrínsecas à geometria projetiva orientada. Através de exemplos práticos, como a execução de algoritmos provenientes da geometria euclidiana, podemos compreender quais os conceitos que a geometria projetiva orientada tem a estender em relação à geometria euclidiana. Por exemplo, a manipulação algébrica de pontos no infinito não é apenas um benefício para os algoritmos, mas também uma característica possível de se visualizar geometricamente.

Trabalhos como [Pin98, Wes99] já puderam demonstrar como algumas propriedades da geometria projetiva orientada podem ser úteis no desenvolvimento de algoritmos. A capacidade de visualizar a evidência de tais propriedades é de grande valor para o aprendizado do aluno e o desenvolvimento da intuição do pesquisador, que pode usar ilustrações para verificar resultados, estabelecer contra-exemplos ou inspirar novas linhas de pesquisa.

Para tanto, algumas das funcionalidades que o $\mathbb{T}^2$ Viewer deve possuir, muitas delas já utilizadas com sucesso em trabalhos anteriores (citados no capítulo 2), estão descritas a seguir:

- Representação das primitivas geométricas de maneira condizente com a geometria;
- Produção de figuras geométricas claras, que não dêem margem a ambigüidades;
- Facilidade de uso;
- Interface de programação bem especificada, consistente e minimal;
- Interface com o usuário compatível com o paradigma de interface do ambiente onde o visualizador funciona;
- Capacidade de mostrar detalhes das figuras geradas (*zoom*);
- Capacidade de mudar o ponto de vista do observador em tempo real, no caso da produção de projeções de figuras geométricas tridimensionais (*orbit*).

Além destas, o $\mathbb{T}^2$ Viewer possui as seguintes funcionalidades exclusivas:

Exibição simultânea dos modelos de $\mathbb{T}^2$: Como vimos na seção 3.2, os modelos plano e esférico possuem uma relação intrínseca entre si, possível de ser visualizada quando exibidos simultaneamente. Isso permite que o usuário observe a mesma figura geométrica representada em modelos distintos, proporcionando melhor compreensão sobre $\mathbb{T}^2$. Geralmente, o modelo plano exibe figuras geométricas de maneira mais usual pois não produz as distorções geradas no modelo esférico, ao passo que este último é capaz de exibir pontos no infinito e representar melhor a topologia de $\mathbb{T}^2$. Como mostram as figuras 3.2 e 3.4, um mesmo segmento é exibido de maneira aparentemente fragmentada no modelo plano e de maneira contínua, no infinito e além dele, no modelo esférico.

Mudança de opacidade do plano e da esfera S^2: Através do controle da opacidade das superfícies plana e esférica, o usuário é capaz de permitir maior ou menor visibilidade através dos modelos de $\mathbb{T}^2$. Torna-se possível, por exemplo, enfatizar a relação entre o aquém e o além (descritos no capítulo 3) ao reduzir-se a opacidade da superfície do modelo em questão; ou então, enfatizar a relação entre os modelos de $\mathbb{T}^2$ reduzindo a opacidade de ambas superfícies. Tornar a superfície do modelo plano completamente opaca significa observar somente o aquém ou o além enquanto que no modelo esférico isso significa observar uma região de $\mathbb{T}^2$ sem observar sua região antípoda. O aumento da opacidade também pode simplificar a visualização de uma figura geométrica densa em primitivas.

Visualização através do centro da esfera S^2: Através da rotação (*orbit*) e aproximação (*zoom*) em tempo real, o usuário é capaz de mover livremente o ponto de vista do observador virtual podendo, inclusive, colocá-lo em posições interiores à esfera S^2. Quando colocado exatamente sobre o centro desta, o usuário poderá observar a perfeita sobreposição das representações, no modelo plano e no modelo esférico, de uma figura geométrica, em virtude de ser este a origem das projeções.

Mudança do ângulo de visão: O ângulo de visão do observador virtual reflete, de maneira natural, o ângulo de visão que o usuário teria caso estivesse observando uma figura tridimensional real. No entanto, em situações como a colocação do ponto de vista do observador sobre o centro da esfera S^2, este ângulo de visão é pequeno demais para se visualizar uma porção significativa da esfera e do plano. Pode-se então aumentar o ângulo de visão, causando um efeito de "grande angular".

Escala da coordenada w: O usuário é capaz de alterar, em tempo real, um valor que multiplica a coordenada homogênea de todos os pontos pertencentes a cada primitiva geométrica. O aumento deste valor faz com que todos os pontos de cada primitiva do aquém sejam levados na direção da origem enquanto que os pontos do além são levados na direção antípoda da origem, facilitando a visualização de figuras geométricas extensas. Os pontos sobre a circunferência Ω permanecem inalterados. Note que o efeito desta escala é meramente visual, não alterando, de fato, a coordenada w de cada ponto da primitiva.

Capacidade de animação: Se imagens apresentam vantagens sobre extensas descrições dissertativas, animações do tipo rotação automática (*orbitting*) apresentam vantagens sobre imagens estáticas, pois evidenciam melhor a tridimensionalidade de uma cena. O usuário é capaz de dar início ao movimento de rotação animada, podendo controlar a direção e a velocidade do movimento. Enquanto isso, o visualizador permanece livre para receber novos eventos, como o acréscimo de novas primitivas, a mudança de opacidade das superfícies dos modelos de $\mathbb{T}^2$, a alteração da escala da coordenada w e a remoção de primitivas. O usuário-programador tem a possibilidade de demonstrar o funcionamento de um algoritmo através da inserção e remoção de primitivas geométricas, intercaladas por intervalos de tempo controlados por ele.

5.2 Ambiente de Desenvolvimento

O software é implementado em C++ utilizando a plataforma Linux. A escolha da linguagem decorre da necessidade de compatibilidade com CGAL aliado às vantagens do

paradigma de orientação a objetos. Qt Designer é usado para criar as interfaces gráficas com o usuário e para prover estrutura de programação integrando o código-fonte, o compilador (GCC) e o tratamento das interrupções. A escolha do Qt Designer decorre de sua compatibilidade com CGAL, como demonstrada pelo Qt widget (mencionado na seção 4.1), visualizador para a biblioteca CGAL feito neste ambiente de programação.

Para que o usuário possa interagir com figuras tridimensionais em tempo real, usamos aceleração gráfica proporcionada por OpenGL. Além de ser amplamente difundido, OpenGL provê numerosos recursos para criação de imagens tridimensionais com a rapidez necessária para aplicações de tempo real. Linux e Qt Designer são amplamente compatíveis com OpenGL.

Observamos que vários dos trabalhos anteriores (descritos no capítulo 2) utilizaram-se de plataformas de desenvolvimento e hardwares de alto custo, de uso dedicado a aplicações gráficas, devido à limitação computacional dos equipamentos mais difundidos da época em que foram desenvolvidos. Isso limitou o desenvolvimento de muitos trabalhos e dificultou o porte destes para outras plataformas, fazendo com que vários trabalhos fossem descontinuados.

Hoje, no entanto, equipamentos de mais baixo custo possuem poder de processamento suficiente para execução de aplicações mais elaboradas e o hardware destinado a aceleração gráfica está mais acessível. Através da plataforma gratuita Linux, do uso de tecnologias de licença aberta (GPL[1] e seus derivados) e do uso de hardware difundido, espera-se facilitar e estimular a manutenabilidade e continuidade de uso deste projeto.

Em razão da escolha deste ambiente de desenvolvimento, seria natural considerar o Geomview (descrito no capítulo 2) como alternativa para prover interface de visualização e manipulação de objetos geométricos ao $\mathbb{T}^2$ Viewer. No entanto, uma análise mais criteriosa revela que essa agregação limitaria diversas de suas funcionalidades.

O projeto que originou o Geomview deixou de existir em 1996 e este software, por conseqüência, foi oficialmente descontinuado. Embora algumas iniciativas voluntárias deram continuidade ao desenvolvimento do Geomview, a generalidade de uso deste software ficou comprometida, pois cada iniciativa direcionou seu desenvolvimento para um uso específico.

Como o Geomview encapsula a interface de programação do hardware gráfico, este não permite usar, por exemplo, comandos OpenGL diretamente. Dessa forma, os recursos para se especificar uma figura geométrica limitam-se aos disponíveis na época em que o Geomview foi descontinuado, afetando a qualidade e o desempenho das imagens produzidas. Novos recursos de hardware gráfico, como *vertex shaders* e *pixel shaders*, não são aproveitados pelo Geomview.

Por conseqüência, o uso do Geomview tornaria impossível a implementação de diversas

[1] GNU General Public License, descrito em maiores detalhes em [GPL91].

funcionalidades inéditas do $\mathbb{T}^2$ Viewer (descritas na seção 5.1) que requerem a mudança simultânea de vários parâmetros de visualização. Um exemplo desta situação seria a mudança, em tempo real, do ponto de vista da imagem enquanto a opacidade de uma superfície presente nela é alterada.

5.3 Requisitos

As seções 5.1 e 5.2 fornecem uma completa descrição das funcionalidades que o visualizador deve possuir assim como especificam o ambiente de desenvolvimento e de uso, tanto em termos de software como de hardware. Estas funcionalidades descrevem vários requisitos desejáveis. Vejamos outros requisitos.

Segundo as especificações de CGAL, um visualizador também deve ter a capacidade de receber entradas através da interface gráfica e transmiti-las para programas do usuário, constituindo um sistema de entrada e saída e não apenas um mecanismo de exposição gráfica das ações de primitivas geométricas.

A interface de programação, além dos atributos citados na seção 5.1, deve ser não orientada a eventos, como veremos a seguir. Isso não é uma restrição imposta pela biblioteca CGAL; pelo contrário, o visualizador Qt widget (descrito na seção 4.1), exclusivo para CGAL, utiliza-se de tal paradigma.

As desvantagens da interface orientada a eventos são as mesmas apresentadas pelo Qt widget: sempre há a necessidade de se programar os eventos de interação com o usuário, de maneira específica para cada programa que o usuário-programador fizer, antes que o Qt widget possa ser usado. Isso exige que o programa do usuário seja estruturado de maneira não linear, podendo ter partes do seu código-fonte executadas em paralelo. Em resumo, o código-fonte do programa do usuário deve sofrer modificações necessárias para ser integrado ao Qt widget e compilado, gerando uma única aplicação. A interface de programação do $\mathbb{T}^2$ Viewer, portanto, além de ser consistente e minimal, deve preservar a estrutura linear e típica dos programas que fazem uso da biblioteca CGAL.

Por fim, notamos que grande parte dos requisitos aqui apresentados têm como objetivo beneficiar o usuário, enquanto os demais beneficiam o usuário-programador (como, por exemplo, os que se referem a interface de programação). Como o usuário típico de CGAL é um usuário-programador, este se beneficiará de todas as funcionalidades.

5.4 Análise

A figura 5.1 mostra um diagrama de alto nível dos módulos conceituais que compõem o $\mathbb{T}^2$ Viewer. Neste diagrama, somente a aplicação do usuário e a biblioteca CGAL não

pertencem ao visualizador (delimitado pelo retângulo pontilhado).

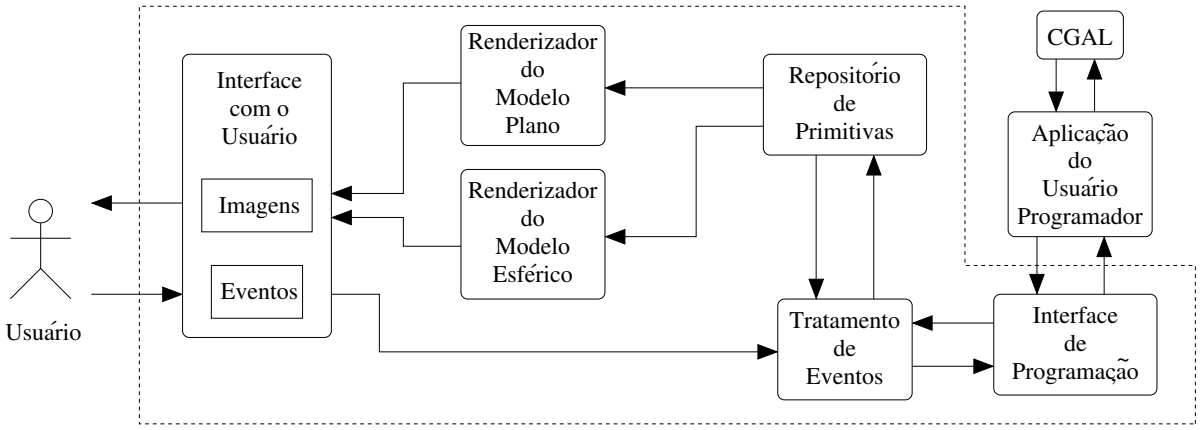

Figura 5.1: Diagrama de Módulos Conceituais

Note que ainda não estamos tratando dos módulos reais que são posteriormente usados para direcionar a implementação. Por este diagrama compreendemos uma estrutura conceitual capaz de mostrar como as diferentes unidades funcionais devem se comunicar a fim de prover o comportamento desejado ao visualizador. Os módulos reais serão descritos em detalhes ao longo da seção 5.5.

Por `eventos` compreende-se todo tipo de entrada produzida pelo usuário em tempo de execução, como seleção de itens nos menus do software, cliques do mouse e acionamento de teclas de atalho no teclado. Por `imagens` compreende-se todo tipo de saída produzida pelo visualizador em tempo de execução, como a produção de figuras geométricas bidimensionais e tridimensionais. Juntos, `eventos` e `imagens` compõem o módulo que corresponde à `interface com o usuário`.

O módulo de `tratamento de eventos` é responsável por interpretar os eventos de entrada, produzidos pelo usuário através da interface gráfica, e executar as ações correspondentes. Também é responsável pela comunicação com o programa do usuário através do envio e recebimento de informações pela `interface de programação` como, por exemplo, o recebimento de primitivas geométricas para serem visualizadas ou o envio de primitivas geométricas, criadas pelo usuário com o auxílio da interface gráfica, para o programa do usuário-programador.

O `repositório de primitivas` é onde todas as primitivas geométricas[2] são armazenadas. A inserção e a remoção de primitivas neste repositório é executada pelo módulo de `tratamento de eventos`. Quando necessário, estas são enviadas aos módulos de renderização que irão produzir figuras geométricas a partir delas e de outros parâmetros

[2]Estruturas de dados da extensão de CGAL para o plano projetivo orientado.

especificados pelo usuário como, por exemplo, o ponto de vista do observador no caso de figuras geométricas tridimensionais.

Existem dois módulos de renderização, um para cada modelo de $\mathbb{T}^2$: o `renderizador do modelo plano` e o `renderizador do modelo esférico`. As representações destes modelos de $\mathbb{T}^2$ estão descritas na seção 3.2.

Nem todos os eventos resultam em mensagens enviadas do visualizador para a aplicação do usuário. Por exemplo, a mudança do ponto de vista da figura geométrica tridimensional não produz nenhuma informação necessária à aplicação do usuário.

Paradigmas e Comunicação Interprocessos

É necessário mencionar aspectos desta análise além dos presentes no diagrama acima. Em particular, tratando-se de um software com interface gráfica, a programação do $\mathbb{T}^2$ Viewer é intrinsecamente orientada a eventos, ao passo que o programa do usuário não o é, como discutido na seção 5.3. A comunicação entre programas com diferentes paradigmas exige uma solução elaborada. Primeiramente, cada programa deve ter compilação e execução independentes a fim de permitir que façam uso de paradigmas distintos. Com isso, cada programa dará origem a um processo independente, sendo que a comunicação entre eles se resume a uma comunicação inter-processos, onde o $\mathbb{T}^2$ Viewer é servidor e os programas do usuário são clientes[3].

As vantagens deste panorama incluem a possibilidade de se deixar o visualizador em execução enquanto o programa do cliente pode ser modificado, compilado e executado diversas vezes. Os resultados de cada execução podem ser visualizados simultaneamente, proporcionando uma comparação entre eles. Outra vantagem é a execução simultânea de diversos programas clientes distintos fazendo uso da mesma instância de visualização. Isso permite que eles utilizem o visualizador para compartilhar estruturas geométricas, o que é particularmente interessante quando se usa um encadeamento de aplicações onde o resultado da execução de um programa é a entrada para o seguinte.

A fim de que a comunicação inter-processos seja estabelecida entre processos de naturezas diferentes, a comunicação entre eles deve ser capaz de proporcionar sincronia e, eventualmente, deve temporariamente interromper a execução do programa do usuário até que um determinado evento ocorra. Suponha, por exemplo, que o programa do usuário solicite a seleção de uma primitiva geométrica, através da interface gráfica, para, então, dar início à execução de um algoritmo geométrico. Assim que o programa requisita a seleção, sua execução é suspensa até que o usuário selecione uma primitiva geométrica. Quando isso acontece, o visualizador resume a execução do programa cliente enviando um sinal indicando a primitiva selecionada.

[3] Motivo pelo qual também são chamados de "programas cliente".

Dessa forma, conseguimos fazer com que o usuário-programador utilize o $\mathbb{T}^2$ Viewer de maneira homogênea ao uso de qualquer mecanismo de entrada e saída presente em CGAL, como a leitura e escrita em arquivos e na linha de comando. Essas operações são feitas com facilidade uma vez que o próprio objeto (representando uma primitiva geométrica) adapta o formato de leitura e escrita dos seus parâmetros para cada meio, não sendo necessário nenhum tipo de conversão do objeto antes que este possa ser enviado ao visualizador, assim como para um arquivo ou para a saída padrão. Um usuário familiarizado com CGAL, portanto, não precisa fazer adaptações na estrutura de seu programa e nem conhecer a lógica interna de funcionamento do $\mathbb{T}^2$ Viewer, o que poupa tempo e esforço de aprendizado.

5.5 Projeto

Nesta seção, descrevemos a modelagem do $\mathbb{T}^2$ Viewer com o objetivo de especificar sua implementação. Para tanto, utilizaremos diagramas na linguagem UML [UML] (Unified Modeling Language), amplamente utilizada no projeto de softwares cuja implementação faz uso do paradigma de orientação a objetos[4].

A modelagem segue a abordagem Top-Down, partindo do nível mais alto de abstração e refinando, sucessivamente, a especificação dos componentes do software. Com freqüência, um componente é dividido em vários módulos que implementam sua funcionalidade. Quando verifica-se que estes módulos são simples demais para serem novamente divididos e contêm detalhes suficientes para serem implementados, dá-se o fim da etapa de modelagem. No presente trabalho, o refinamento seguirá até a etapa imediatamente anterior àquela necessária à implementação, pois esta última necessita de detalhes específicos da linguagem e do ambiente de programação, o que necessitaria de descrições demasiadamente extensas para o presente contexto.

Começaremos pelo Diagrama de Módulos Funcionais, exibido na figura 5.2 na página 43. Diferentemente dos módulos conceituais exibidos na figura 5.1 da seção 5.4, os módulos funcionais podem ser traduzidos para componentes de software capazes de implementar funcionalidades específicas. Ao todo, temos sete módulos funcionais. Ao descrevê-los, citamos funcionalidades especificadas na seção 5.1 e, mais adiante, uma seção dedicada a cada módulo descreverá como suas funcionalidades são implementadas.

renderers: Módulo de renderização, responsável pela geração de todas as imagens produzidas pelo visualizador. Este componente recebe do módulo `models` todas as

[4]Em especial, utilizaremos a categoria *diagrama de classes* dentre as categorias de diagramas de modelagem presentes na linguagem UML.

construções geométricas a serem exibidas e realiza os cálculos de projeção, iluminação, texturização, *blending* e *antialising* (entre outros) de modo a produzir imagens bidimensionais e tridimensionais que representem o ponto de vista do observador virtual. Quando altera-se a posição do observador, este módulo refaz os cálculos de modo a manter a correspondência entre o ponto de vista e a imagem que seria vista por ele. Devido à capacidade de gerar imagens em tempo real, os movimentos de rotação (*orbit*) e de aproximação (*zoom*), comandados pelo usuário, ocorrem de maneira contínua, semelhantemente à interação com objetos reais. A capacidade de animação do tipo rotação automática (*orbitting*) é implementada neste módulo, possibilitando que o usuário dê início ao movimento de rotação, controlando sua velocidade e direção. Sendo possível posicionar o ponto de vista no centro da esfera S^2 e observar a correspondência entre os modelos plano e esférico, este módulo também permite que o ângulo de visão seja aumentado, causando o efeito de "grande angular", de modo a permitir a observação de uma porção significativa dos dois modelos a partir do centro de S^2.

models: Módulo responsável pela produção de construções geométricas, enviadas ao módulo `renderers`. Por construção geométrica, podemos compreender o conjunto de pontos, segmentos, circunferências, curvas, planos, esferas e superfícies que, dispostos de forma conveniente, compõem uma figura geométrica. Além das primitivas geométricas originadas na aplicação do usuário, a própria superfície da esfera S^2 (modelo esférico) e o plano que separa as duas cópias de $\mathbb{R}^2$ (modelo plano) também fazem parte da construção. O módulo `viewers` envia as primitivas geométricas para `models` que as desenha de acordo com o modelo plano e o modelo esférico de $\mathbb{T}^2$ (como descrito na seção 3.2). Este módulo controla a opacidade de S^2 e do plano que separa as duas cópias de $\mathbb{R}^2$, assim como a escala da coordenada w, como descrito na seção 5.1.

primitives: Módulo de repositório das primitivas geométricas, originadas na aplicação do usuário. Toda primitiva recebida pelo visualizador é armazenada em `primitives` para, logo em seguida, ser incluída na figura geométrica final. Eventualmente, a aplicação do usuário poderá requisitar primitivas geométricas do visualizador, que são especificadas ou pela aplicação do usuário através de um identificador (que é produzido toda vez que uma primitiva é enviada ao visualizador) ou pelo usuário através da seleção na tela, com o auxílio do cursor. É o módulo `viewers` que faz a maioria das consultas em `primitives`.

viewers: Módulo responsável por manter uma lista contendo referências a todas as janelas de visualização e seus respectivos contextos. Toda vez que uma nova janela de visualização é aberta, esta é registrada em `viewers` que se encarrega de atualizar seu

contexto com o do visualizador, o que consiste de fornecer todas as primitivas geométricas armazenadas em `primitives` e, também, parâmetros de visualização que o usuário pode especificar através de sua aplicação, como a aproximação (*zoom*) inicial e a posição inicial da figura geométrica (*pan* no modelo plano e *orbit* no modelo esférico). Toda vez que uma nova primitiva geométrica é enviada ao visualizador, cabe a `viewers` atualizar o contexto de todas as janelas de visualização para que estas incluam a nova primitiva.

events: Módulo responsável pelo tratamento dos eventos recebidos pelo visualizador. Estes eventos têm origem tanto na aplicação do usuário como na interação do usuário com o $\mathbb{T}^2$ Viewer através dos seus menus (e teclas de atalho) que é o motivo pelo qual este módulo contém a definição de todos os menus que compõem a interface gráfica do visualizador. Sendo o componente que, de fato, dá início à execução do $\mathbb{T}^2$ Viewer, este também é responsável por instanciar a maioria dos demais módulos e configurar as conexões entre eles.

sockets: Módulo responsável pela comunicação entre o lado cliente, que inclui a aplicação do usuário, e o lado servidor, que inclui o visualizador. Este módulo é chave na questão da comunicação inter-processos, apresentada na seção 5.4, entre softwares cujas execuções seguem paradigmas distintos: o servidor (com interfaces gráficas) é orientado a eventos enquanto o cliente não o é. No lado servidor, `sockets` comunica-se com o módulo `events`, que trata toda troca de informação como eventos gerados pelo programa do usuário. Do lado cliente, `sockets` comunica-se com o módulo `interface`, escondendo da aplicação do usuário todos os detalhes envolvidos na comunicação. Os pacotes de informação, contendo as informações trocadas entre o cliente e o servidor, são definidos no módulo `sockets`.

interface: Módulo que define a interface de programação do visualizador, usada na aplicação do usuário. Todas as funções do $\mathbb{T}^2$ Viewer que o usuário-programador pode fazer uso em seu programa estão reunidas neste módulo. O componente `interface` é o único módulo que não é compilado junto ao lado servidor do visualizador, mas sim junto ao lado cliente (aplicação do usuário); fato que, como explicado na seção 5.4 junto a outras vantagens, possibilita a execução simultânea de diversos programas clientes fazendo uso da mesma instância de visualização, permitindo compartilhar estruturas geométricas através do visualizador. Este componente comunica-se com o lado servidor do visualizador através do módulo `sockets`.

Na figura 5.3 na página 44, as classes que compõem cada módulo funcional estão presentes no diagrama. Para maior clareza, a figura 5.4 na página 45 exibe somente as classes que compõem o $\mathbb{T}^2$ Viewer, sem mostrar os módulos aos quais pertencem. Nestes diagramas,

o componente que representa CGAL aparece como um agregado à aplicação do usuário. Vale notar que CGAL também é agregado a inúmeros outros módulos deste software e, para maior clareza dos diagramas, escolheu-se omitir suas demais ocorrências.

A seguir, veremos como cada módulo é implementado a fim de prover suas funcionalidades. As classes que compõem cada módulo serão exibidas em maiores detalhes e as relações entre classes (e módulos) serão melhor especificadas. Cada classe é composta por diversos atributos (variáveis que contêm o estado de cada objeto instanciado a partir dela) e métodos (funções que atuam nos atributos, modificando-os de maneira a prover o comportamento do objeto). Explicaremos os métodos e atributos mais relevantes para descrever cada classe, omitindo aqueles cuja função não contribui diretamente para o funcionamento do visualizador, como os métodos de inicialização e finalização.

5.5.1 Módulo `renderers`

A figura 5.5 na página 46 mostra as classes que compõem o módulo `renderers`. A classe abstrata `T2renderer` é superclasse das classes `T2renderer2D` e `T2renderer3D`. Como o nome sugere, `T2renderer2D` produz figuras bidimensionais, exibindo apenas o modelo plano de $\mathbb{T}^2$ em verdadeira grandeza, enquanto `T2renderer3D` produz figuras tridimensionais, exibindo os modelos plano e esférico de $\mathbb{T}^2$, observados de qualquer ponto de vista. Os atributos booleanos `showFlatModel` e `showSphericalModel` controlam quais modelos de $\mathbb{T}^2$ serão exibidos em cada janela de renderização. Seus valores são definidos através dos métodos `setShowFlatModel()` e `setShowSphericalModel()` que, sendo visíveis externamente, são chamados pelo módulo `events`.

O método `redraw()` é chamado sempre que ocorrem alterações significativas na imagem, como acréscimo ou deleção de primitivas geométricas, para que a figura final reflita essas modificações. Para tanto, este método utiliza-se de estruturas denominadas *display lists*: listas contendo código OpenGL que, quando executadas pelo hardware gráfico, produzem um componente da imagem final. Cada janela de visualização possui uma referência para um objeto do tipo `T2FlatModel` (`*flatModel`) e, no caso de janelas de visualização tridimensional, outra referência para um objeto do tipo `T2SphericalModel` (`*sphericalModel`). Objetos como estes, provenientes do módulo `models`, são responsáveis por produzir *display lists* contendo instruções para desenhar todas as primitivas geométricas em um determinado modelo de $\mathbb{T}^2$. De acordo com os valores dos atributos booleanos `showFlatModel` e `showSphericalModel`, são executadas, em `redraw()`, as *display lists* provenientes, respectivamente, de `T2FlatModel` e `T2SphericalModel`, contendo instruções para desenhar as primitivas nos modelos plano e esférico de $\mathbb{T}^2$.

Os métodos virtuais `setAlpha()`, `setRadius()` e `setCenter()` servem para ajustar, respectivamente, a escala da coordenada w, a aproximação (*zoom*) e a centralização

(*pan*/*orbit*) da imagem na tela. Eles são chamados em resposta a eventos gerados tanto pela interface gráfica como pela interface de programação. A implementação desses métodos ocorre de maneira diferente em cada subclasse. Por exemplo, em `T2renderer2D`, `setCenter()` desloca a imagem do modelo plano de $\mathbb{T}^2$ na tela (*pan*) enquanto que, em `T2renderer3D`, o mesmo método provoca a rotação (*orbit*) da imagem do modelo esférico de $\mathbb{T}^2$ na tela. O resultado é que a mesma região da figura geométrica final é exibida em ambos tipos de janelas de visualização. O método `setAlpha()` envia um valor de escala da coordenada w (denominado, aqui, de `alpha`) para o módulo `models`, que produzirá as novas *display lists* refletindo esta modificação.

O módulo `renderers` é responsável por receber os eventos originados pelo usuário através do uso do cursor nas janelas de visualização. É importante mencionar que estes eventos não incluem a interação do usuário com os menus do $\mathbb{T}^2$ Viewer, o que é feito pelo módulo `events`. Os eventos gerados nas janelas de visualização são interpretados de acordo com o modo de entrada ativo no momento. O usuário pode alternar o modo de entrada ativo através dos menus do visualizador. Na janela de visualização bidimensional, os métodos que ativam um modo de entrada são:

- `setSelectionMode()`: Ativa o modo de seleção de primitivas geométricas com o cursor.

- `setZoomMode()`: Ativa o modo de ajuste de aproximação (*zoom*), variando a proximidade entre o observador virtual e o desenho através do movimento vertical do cursor (o que é feito com o auxílio do método `setRadius()`).

- `setPanMode()`: Ativa o modo de ajuste de centralização da imagem (*pan*), deslocando a posição da imagem na tela de acordo com o movimento do cursor (o que é feito com o auxílio do método `setCenter()`).

- `setFrontBackGradientMode()`: Ativa o modo de ajuste interativo da opacidade de primitivas geométricas no aquém e no além do modelo plano de $\mathbb{T}^2$. O movimento vertical para cima do cursor faz com que as primitivas do além tornem-se gradualmente mais transparentes, enquanto que o movimento vertical para baixo produz efeito semelhante no aquém. Na posição verticalmente central, a opacidade das primitivas no aquém e no além é plena.

- `setPlaneGradientMode()`: Ativa o modo de ajuste interativo da opacidade do plano que separa o aquém do além no modelo plano de $\mathbb{T}^2$. O movimento vertical para cima faz com que o plano torne-se mais opaco enquanto o movimento vertical para baixo produz o efeito contrário.

Na janela de visualização tridimensional, temos vários dos modos de entrada que estão presentes na janela de visualização bidimensional, pois esta também exibe o modelo plano de $\mathbb{T}^2$. Os métodos que ativam um modo de entrada na janela de visualização tridimensional são:

- `setSelectionMode()`: Ativa o modo de seleção de primitivas geométricas com o cursor.

- `setZoomMode()`: Ativa o modo de ajuste de aproximação (*zoom*), variando a proximidade entre o observador virtual e o desenho através do movimento vertical do cursor (o que é feito com o auxílio do método `setRadius()`).

- `setWScaleMode()`: Ativa o modo de ajuste da escala da coordenada w. Neste modo, manipula-se o valor (`alpha`) que multiplica a coordenada homogênea de todos os pontos pertencentes a cada primitiva geométrica do modelo esférico de $\mathbb{T}^2$. O movimento vertical para baixo do cursor provoca o aumento deste valor que faz com que todos os pontos de cada primitiva do aquém sejam levados na direção da origem enquanto que os pontos do além são levados na direção antípoda da origem. O movimento vertical para cima produz o efeito contrário e a posição verticalmente central anula este efeito (`alpha`=1).

- `setOrbitMode()`: Ativa o modo de rotação (*orbit*) interativa da figura geométrica tridimensional. O movimento do cursor direciona a rotação da figura, de modo semelhante a interação com objetos reais. O usuário pode também dar início ao movimento de rotação animada (*orbitting*) neste modo de entrada. Para isso, ele deve rotacionar a figura de maneira usual com a única diferença de que deve soltar o botão do cursor com este ainda em movimento. A figura geométrica continuará a rotacionar com a mesma direção e velocidade com que o usuário deu início ao movimento.

- `setFovyMode()`: Ativa o modo de ajuste do ângulo de visão do observador virtual. O movimento vertical para baixo aumenta o ângulo de visão, reduzindo o tamanho aparente das primitivas geométricas. O movimento contrário produz o efeito inverso.

- `setFrontBackGradientMode()`: Ativa o modo de ajuste interativo da opacidade de primitivas geométricas no aquém e no além do modelo plano de $\mathbb{T}^2$. O movimento vertical para cima do cursor faz com que as primitivas do além tornem-se gradualmente mais transparentes, enquanto que o movimento vertical para baixo produz efeito semelhante no aquém. Na posição verticalmente central, a opacidade das primitivas no aquém e no além é plena.

- `setPlaneGradientMode()`: Ativa o modo de ajuste interativo da opacidade do plano que separa o aquém do além no modelo plano de $\mathbb{T}^2$. O movimento vertical para cima faz com que o plano torne-se mais opaco enquanto o movimento vertical para baixo produz o efeito contrário.

- `setS2GradientMode()`: Ativa o modo de ajuste interativo da opacidade da superfície S^2 do modelo esférico de $\mathbb{T}^2$. O movimento vertical para cima faz com que S^2 torne-se mais opaca enquanto o movimento vertical para baixo produz o efeito contrário.

Por fim, os atributos numéricos `cameraX`, `cameraY` e `cameraZ` registram a posição do observador virtual. Na janela de visualização bidimensional, o valor da terceira coordenada da posição do observador é constante, não necessitando do atributo `cameraZ`.

5.5.2 Módulo `models`

A figura 5.6 na página 47 mostra as classes que compõem o módulo `models`. A classe abstrata `T2Model` é superclasse das classes `T2FlatModel` e `T2SphericalModel`; esta última associada à classe `T2Texture`.

Este módulo é responsável pela criação de estruturas denominadas *display lists*: listas contendo código OpenGL que, quando executadas pelo hardware gráfico, produzem componentes da imagem final. O agrupamento de uma longa seqüência de comandos OpenGL em uma *display list* é opcional, mas oferece um desempenho superior. Em situações como a rotação interativa ou automática (*orbit*/*orbitting*), a figura geométrica tridimensional precisa ser continuamente redesenhada para refletir a constante mudança do ponto de vista do observador virtual. Através do uso de *display lists*, toda a construção geométrica presente na figura pode ficar armazenada diretamente na memória do hardware gráfico. Desta maneira, a execução da *display list* evita o tempo gasto no reenvio de toda a construção geométrica, do visualizador para o hardware gráfico, toda vez que a figura final precisar ser redesenhada devido a mudança do ponto de vista. Como resultado, as imagens são produzidas mais rapidamente, tornando o movimento de rotação mais suave e permitindo que a unidade de processamento central realize outras tarefas.

Os objetos da classe `T2FlatModel` compilam *display lists* que, quando executadas, produzem desenhos contendo todas as primitivas geométricas representadas no modelo plano de $\mathbb{T}^2$. O mesmo ocorre com os objetos da classe `T2SphericalModel` em relação ao modelo esférico de $\mathbb{T}^2$. Essas *display lists* contêm, além das primitivas geométricas, superfícies que ilustram os modelos de $\mathbb{T}^2$, como a superfície que separa as duas cópias de $\mathbb{R}^2$ no modelo plano de $\mathbb{T}^2$ e a superfície S^2 do modelo esférico de $\mathbb{T}^2$.

Todas as *display lists* produzidas em `models` são executadas, através do método `runDisplayList()`, sempre que o método `redraw()`, de `renderers`, é chamado. Esta

estrutura, no entanto, é estática, precisando ser recompilada toda vez que uma nova primitiva for adicionada ou removida. A compilação de uma *display list* segue os seguintes passos:

1. Iniciar a compilação da *display list*. Todo código OpenGL que vier em seguida a esta etapa é armazenado na *display list* ao invés de ser executado pelo hardware gráfico;

2. Percorrer todas as primitivas geométricas armazenadas no módulo `primitives`, executando, em cada uma, o método `draw()`, cuja função é desenhar a primitiva através de métodos abstratos de `T2Model`;

3. Finalizar a compilação da *display list*.

A primeira e a última etapas deste processo utilizam-se, respectivamente, dos métodos `beginDisplayList()` e `endDisplayList()` de `T2Model`. Na segunda etapa, o método `draw()` de cada primitiva geométrica pode fazer uso dos seguintes métodos abstratos de `T2Model`:

- `Point_T2()`: Desenha um ponto de $\mathbb{T}^2$ a partir das coordenadas w, x e y.

- `Segment_T2()`: Desenha um segmento não-degenerado de $\mathbb{T}^2$ a partir das coordenadas w, x e y dos seus extremos.

- `Circle_T2()`: Desenha uma circunferência própria e não-degenerada de $\mathbb{T}^2$ a partir das coordenadas w, x e y do centro e das coordenadas w e x do raio.

- `Line_T2()`: Desenha uma reta de $\mathbb{T}^2$ que passa por dois pontos, especificados por suas coordenadas w, x e y.

Uma primitiva geométrica é degenerada quando algum de seus parâmetros, sua medida ou posição se dá em situação de fronteira de domínio, como uma reta ou segmento de reta entre dois pontos coincidentes ou uma circunferência com raio nulo. Cabe ao método `draw()`, de cada primitiva geométrica, usar estes quatro métodos abstratos de `T2Model` para representar todas as formas possíveis que a primitiva pode ter. Por exemplo, uma circunferência em $\mathbb{T}^2$ pode estar sob três formas distintas:

Raio finito: Caso não-degenerado da circunferência, onde esta é desenhada através de `Circle_T2()`;

Raio nulo: Caso degenerado, onde a circunferência é representada por um ponto, desenhado através de `Point_T2()`;

Raio infinito: Caso degenerado, onde a circunferência é representada pela reta no infinito, desenhada através de `Line_T2()`.

Esses quatro métodos abstratos de `T2Model` são suficientes para representar todas as primitivas geométricas suportadas pelo $\mathbb{T}^2$ Viewer, em todas as suas formas. Diagramas, triangulações e outras estruturas geométricas de maior complexidade são compostas por primitivas geométricas mais simples, desenhadas de maneira usual.

Cada um destes quatro métodos abstratos é implementado de maneira diferente de acordo com o modelo de $\mathbb{T}^2$. Por exemplo, `Segment_T2()` é implementado em `T2FlatModel` de maneira a produzir um segmento de reta que une dois pontos enquanto que, em `T2SphericalModel` o mesmo método produz o menor arco de circunferência de raio máximo sobre S^2 que une dois pontos. Desta maneira, o tratamento das degenerações ocorre de maneira transparente e homogênea nos dois modelos de $\mathbb{T}^2$.

Os métodos abstratos de `T2Model` também recebem, como parâmetros, as cores correntemente usadas para desenhar primitivas (armazenadas em `frontColor`, `backColor` e `sphereColor`) e o valor booleano `selected`, que indica se a primitiva geométrica foi selecionada pelo usuário, com o cursor, através da interface gráfica. Ambos parâmetros são provenientes do módulo `primitives`.

O atributo numérico `wScale` é usado para implementar a escala da coordenada w, multiplicando a coordenada homogênea de todos os pontos pertencentes a todas primitivas geométricas. Este valor (também denominado de `alpha`) é inicialmente unitário, sendo posteriormente alterado pelo módulo `renderers`. Quando isso ocorre, a *display list*, devido à sua natureza estática, precisa ser recompilada para que possa refletir a nova escala da coordenada w dos pontos pertencentes às primitivas.

O modelo plano de $\mathbb{T}^2$ é visualizado por ambas janelas de visualização (bidimensional e tridimensional). Na janela de visualização tridimensional, no entanto, o plano cujos lados representam as duas cópias de $\mathbb{R}^2$ é recortado, tendo sua área limitada ao interior de um quadrado de dimensões fixas. Isso evita que a imagem do plano sobreponha, em demasia, a imagem da esfera S^2, quando observada de determinados pontos de vista. Para tanto, os objetos do tipo `T2FlatModel`, que produzem *display lists* para ambas janelas de visualização, utilizam-se do atributo booleano `infinite` para limitar, condicionalmente, a área do plano. A opacidade deste é controlada através do método `setPlaneGradient()` e a opacidade de S^2 é alterada através de `setS2Gradient()`. Junto com `setFrontBackGradient()`, que controla a opacidade de primitivas geométricas no aquém e no além do modelo plano de $\mathbb{T}^2$, estes três métodos implementam o ajuste interativo de opacidade, ajustados através dos modos de entrada do módulo `renderers`.

As *display lists* produzidas por objetos do tipo `T2SphericalModel` contêm, além das construções geométricas, três eixos, com rótulos "W", "X" e "Y", que representam a base ortogonal do espaço onde o modelo esférico de $\mathbb{T}^2$ é desenhado, na janela de exibição

tridimensional. A exibição desses eixos é opcional, sendo controlada através do método `setShowArrows()`, que é manipulado pelo módulo `events`. Os rótulos dos eixos estão definidos em `T2Textures` sob a forma de texturas.

5.5.3 Módulo `primitives`

A figura 5.7 na página 48 mostra as classes que compõem o módulo `primitives`, cuja função é servir de repositório para todas as primitivas geométricas enviadas ao visualizador. Cada primitiva é armazenada em um objeto que funciona como um *container*, contendo, também, atributos como a cor com que a primitiva será desenhada, seu número de identificação e um valor booleano que indica se ela foi selecionada pelo usuário.

Para cada tipo de primitiva geométrica[5] existe uma subclasse específica da classe pai `T2Container`, cujos objetos armazenam primitivas pertencentes a este tipo. Os métodos de `T2Container` são:

- `getPrimitiveID()`: Retorna o valor do atributo numérico `primitiveID`, que identifica univocamente a primitiva geométrica enviada ao visualizador. Este valor é usado pela aplicação do usuário para referenciar primitivas geométricas armazenadas no visualizador.

- `draw()`: Método responsável por desenhar a primitiva geométrica armazenada em instâncias das subclasses de `T2Container`. A implementação deste método abstrato, nas subclasses, faz uso somente dos quatro métodos abstratos de desenho de `T2Model` (explicados na subseção 5.5.2). Como os métodos de desenho de `T2Model` são especializados tanto para o modelo plano como para o modelo esférico de $\mathbb{T}^2$, uma única implementação de `draw()` é capaz de desenhar a primitiva em ambos modelos de $\mathbb{T}^2$. Em cada subclasse de `T2Container`, a implementação de `draw()` reflete, também, todas as degenerações inerentes às primitivas armazenadas por instâncias desta subclasse.

- `setSelected()`: Define o valor da variável booleana `selected`, que indica se a primitiva geométrica foi selecionada pelo usuário, através da interface gráfica.

- `getSelected()`: Retorna o valor da variável booleana `selected`, que indica se a primitiva geométrica foi selecionada pelo usuário, através da interface gráfica.

Cada subclasse de `T2Container` possui um atributo para armazenar a primitiva geométrica correspondente ao tipo da subclasse. Devido a diversidade de tipos numéricos, as

[5] Consideram-se aqui as primitivas geométricas da extensão de CGAL para $\mathbb{T}^2$ ([dO04]).

subclasses de `T2Container` são implementadas através de *templates*, onde todos os atributos numéricos são declarados com um tipo numérico parametrizável. Dessa forma, uma única implementação de classe é capaz de cobrir todas as implementações possíveis se precisássemos produzir uma classe para cada tipo numérico. As subclasses de `T2Container` são:

- `T2PointContainer`: Classe cuja instância armazena uma primitiva geométrica do tipo `Point_T2` (ponto). Para o desenho desta, a implementação de `draw()` usa apenas o método `Point_T2()` de `T2Model`.

- `T2SegmentContainer`: Classe cuja instância armazena uma primitiva geométrica do tipo `Segment_T2` (segmento). Para o desenho desta, a implementação de `draw()` usa o método `Segment_T2()` se o segmento não for degenerado. Caso as extremidades sejam coincidentes, o segmento é representado por um ponto, desenhado por `Point_T2()` de `T2Model`.

- `T2LineContainer`: Classe cuja instância armazena uma primitiva geométrica do tipo `Line_T2` (reta). Para o desenho desta, a implementação de `draw()` usa apenas o método `Line_T2()` de `T2Model`.

- `T2RayContainer`: Classe cuja instância armazena uma primitiva geométrica do tipo `Ray_T2` (semi-reta). Para o desenho desta, a implementação de `draw()` usa apenas o método `Segment_T2()` de `T2Model`.

- `T2CircleContainer`: Classe cuja instância armazena uma primitiva geométrica do tipo `Circle_T2` (circunferência), especificada através de dois pontos: um no centro e outro sobre a circunferência. Para o desenho desta, a implementação de `draw()` usa o método `Circle_T2()` de `T2Model` se a circunferência não for degenerada (ambos pontos são próprios). Caso o raio seja nulo (os pontos de entrada são coincidentes), a circunferência é representada por um ponto, desenhado por `Point_T2()` de `T2Model`. Caso o centro seja um ponto próprio e o ponto sobre a circunferência seja impróprio (raio infinito), esta é representada pela reta do infinito, desenhada por `Line_T2()` de `T2Model`. Por fim, caso o centro da circunferência seja impróprio e o ponto sobre esta seja próprio (raio infinito), a primitiva é representada por duas meia-retas, uma no aquém e outra no além, ambas perpendiculares à direção dada pelo centro e localizadas à mesma distância deste. Cada meia-reta é desenhada através de duas chamadas a `Segment_T2()` de `T2Model`.

- `T2TriangleContainer`: Classe cuja instância armazena uma primitiva geométrica do tipo `Triangle_T2` (triângulo). Para o desenho desta, o triângulo é representado por três segmentos, desenhados por `Segment_T2()` de `T2Model`. As possíveis

degenerações do triângulo são automaticamente tratadas como degenerações dos segmentos que o compõem.

- `T2Iso_rectangleContainer`: Classe cuja instância armazena uma primitiva geométrica do tipo `Iso_rectangle_T2` (retângulo isotético), especificada através de um ponto de $\mathbb{T}^2$ e dois comprimentos de $\mathbb{T}^1$. Para o desenho desta, o retângulo é representado por quatro segmentos, desenhados por `Segment_T2()` de `T2Model`. As possíveis degenerações do retângulo são automaticamente tratadas como degenerações dos segmentos que o compõem.

- `T2BboxContainer`: Classe cuja instância armazena uma primitiva geométrica do tipo `Bbox_T2` (caixa envolvente), especificada através de um ponto de $\mathbb{T}^2$ e dois comprimentos de $\mathbb{T}^1$. Para o desenho desta, o retângulo que corresponde à caixa envolvente é representado por quatro segmentos, desenhados por `Segment_T2()` de `T2Model`. Esta primitiva não cruza a reta do infinito e suas possíveis degenerações são automaticamente tratadas como degenerações dos segmentos que a compõem.

A classe `T2Primitives` é instanciada produzindo um único objeto no $\mathbb{T}^2$ Viewer, responsável por possuir e manter uma lista de todas as primitivas geométricas enviadas ao visualizador. Esta lista consiste de uma estrutura de dados do tipo lista ligada de objetos de `T2Container`. Além de tratar da inserção e remoção de primitivas, o objeto de `T2Primitives` possui um método (`drawOnModel()`) responsável por percorrer todas as primitivas geométricas armazenadas, executando, em cada uma, seu método `draw()`. Este procedimento é coordenado pelo módulo `viewers` e faz parte da construção das *display lists* pelo módulo `models` (explicado na subseção 5.5.2). Os métodos de `T2Primitives` são:

- `clear()`: Apaga todas as primitivas geométricas armazenadas no atributo `container` (lista ligada de objetos de `T2Container`) e reinicia o valor do atributo numérico `newPrimitiveID`, usado para atribuir um número de identificação unívoco a cada nova primitiva armazenada.

- `drawOnModel()`: Método que percorre todas as primitivas geométricas armazenadas no atributo `container`, executando, em casa uma, o método `draw()`. Este procedimento é coordenado pelo módulo `viewers` e faz parte da construção das *display lists* pelo módulo `models`.

- `setColor()`: Define as cores com que as novas primitivas geométricas serão desenhadas. O $\mathbb{T}^2$ Viewer usa três cores distintas para desenhar primitivas: uma para cada lado do plano que separa as duas cópias de $\mathbb{R}^2$ do modelo plano de $\mathbb{T}^2$ e outra

para as primitivas desenhadas no modelo esférico de $\mathbb{T}^2$. Através da aplicação do usuário-programador, é possível à aplicação do usuário redefinir cada uma das três cores.

- `resetColor()`: Atribui as cores padrão para desenho das primitivas geométricas.
- `addPrimitive()`: Este método adiciona uma nova primitiva geométrica à lista `container`. Para isso, alguns parâmetros deste método identificam o tipo de primitiva geométrica a fim de instanciar o objeto correspondente (de um dos subtipos de `T2Container`) ao qual a primitiva será inserida. O valor do atributo numérico `newPrimitiveID` define o número de identificação unívoco deste novo objeto. O valor de `newPrimitiveID` é incrementado logo em seguida.
- `deletePrimitive()`: Remove da lista `container` o objeto cujo número de identificação for igual ao especificado através do parâmetro deste método.
- `unSelectAllPrimitives()`: Percorre todos os objetos da lista `container`, atribuindo o valor *falso* para o atributo booleano `selected` de cada objeto. Este método é usado durante a seleção de primitivas pelo usuário.
- `setSelectedPrimitive()`: Altera o atributo booleano `selected` do objeto da lista `container`, especificado através do número de identificação. Este método é usado durante a seleção de primitivas na tela pelo usuário.
- `getNumberOfPrimitives()`: Retorna o número de primitivas geométricas armazenadas. Este método é usado durante a seleção de primitivas pelo usuário.

O módulo `primitives` é manipulado exclusivamente pelo módulo `viewers`.

5.5.4 Módulo `viewers`

A figura 5.8 na página 49 mostra as classes que compõem o módulo `viewers`, cuja função é fazer com que todas as janelas de visualização exibam imagens atualizadas e consistentes entre si. Para isso, `viewers` mantém uma lista de referência a todas as janelas de visualização e seus respectivos contextos. Quando ocorre uma mudança na construção geométrica, como o acréscimo ou remoção de primitivas ou a seleção de primitivas pelo usuário, aquela lista é percorrida e, para cada janela de visualização, o contexto é atualizado com o auxílio do módulo `primitives`. Como resultado, qualquer alteração na construção geométrica reflete-se simultaneamente em todas as janelas de visualização instanciadas. Quando uma nova janela é aberta pelo usuário, `viewers` atualiza seu contexto, deixando-a consistente com as demais.

A lista que `viewers` contém é uma estrutura de dados do tipo lista ligada de objetos da classe `Viewer`. Estes objetos são simples, possuindo apenas os seguintes atributos públicos:

- `viewerID`: Atributo numérico que identifica univocamente a janela de visualização. Este número é usado para identificar a procedência de eventos gerados pela interface gráfica e para auxiliar o módulo `viewers` a remover o nó, da lista ligada, correspondente à janela de visualização quando esta é fechada pelo usuário.

- `flatModel`: Referência ao objeto que é instância da classe `T2FlatModel` do módulo `models`. Este objeto está presente tanto nas janelas de visualização bidimensional como nas de visualização tridimensional e é responsável por gerar *display lists* contendo primitivas geométricas representadas no modelo plano de $\mathbb{T}^2$. Esta referência permite que `viewers` requisite a reconstrução da *display list* referente ao modelo plano, com o auxílio de `primitives`, atualizando seu conteúdo.

- `sphericalModel`: Referência ao objeto que é instância da classe `T2SphericalModel` do módulo `models`. Este objeto está presente somente nas janelas de visualização tridimensional e é responsável por gerar *display lists* contendo primitivas geométricas representadas no modelo esférico de $\mathbb{T}^2$. Esta referência permite que `viewers` requisite a reconstrução da *display list* referente ao modelo esférico, com o auxílio de `primitives`, atualizando seu conteúdo.

- `renderer`: Referência ao objeto que é instância de uma das subclasses de `T2Renderer` do módulo `renderers`. Este objeto é responsável por executar as *display lists* provenientes dos objetos referenciados por `flatModel` e por `sphericalModel` (este último presente somente em janelas de visualização tridimensional) produzindo a imagem final. O módulo `viewers` utiliza a referência `renderer` para chamar os métodos virtuais `setAlpha()`, `setRadius()` e `setCenter()` de `T2Renderer`, que servem para ajustar, respectivamente, a escala da coordenada w, a aproximação (*zoom*) e a centralização (*pan*/*orbit*) da imagem na tela. Quando algum destes três ajustes é requisitado pela aplicação do usuário, `Viewers` realiza a chamada correspondente em todas as janelas de visualização instanciadas, de modo a refletir o ajuste globalmente.

Um única instância de `T2Viewers` é usada no visualizador e este objeto recebe chamadas exclusivamente do módulo `events`. Os métodos de `T2Viewers` são:

- `addViewer()`: Método que adiciona as referências aos objetos instâncias das classes `T2Renderer`, `T2FlatModel` e `T2SphericalModel`, provenientes da nova janela

de visualização, à lista ligada de `viewers`. Essas referências são fornecidas através dos parâmetros deste método. O atributo numérico `viewerID` recebe o valor de `newViewerID`, sendo este último incrementado logo em seguida. O objeto referenciado por `renderer`, da nova janela, recebe chamadas aos métodos virtuais `setAlpha()`, `setRadius()` e `setCenter()` para que a visualização reflita os valores iniciais, definidos pela aplicação do usuário, para ajustar, respectivamente, a escala da coordenada w, a aproximação (*zoom*) e a centralização (*pan/orbit*) da imagem na tela. Estes valores são armazenados nos atributos numéricos `currentRadius`, `currentCenterW`, `currentCenterX`, `currentCenterY` e `currentAlpha`.

- `removeViewer()`: Remove o nó da lista ligada de referências às janelas de visualização, cujo `viewerID` é igual ao da janela de visualização que está sendo fechada pelo usuário.

- `clear()`: Remove todos os nós da lista ligada de referências às janelas de visualização. Este método é chamado durante o encerramento da execução do visualizador.

- `setRestore()`: Restaura todas as janelas de visualização para seus ajustes iniciais, feitos pela aplicação do usuário, de escala da coordenada w, aproximação (*zoom*) e centralização (*pan/orbit*) da imagem na tela. Isso é feito percorrendo-se todos os nós da lista ligada (de referência às janelas de visualização), executando em cada um, respectivamente, os métodos `setAlpha()`, `setRadius()` e `setCenter()` do objeto referenciado por `renderer`. Como parâmetros destes métodos, são passados os valores armazenados nos atributos numéricos `currentRadius`, `currentCenterW`, `currentCenterX`, `currentCenterY` e `currentAlpha`.

- `setViewersRadius()`: Percorre todos os nós da lista ligada de referências às janelas de visualização, executando o método `setRadius()` do objeto referenciado por `renderer`. O método `setViewersRadius()` é chamado quando a aplicação do usuário requisita o ajuste de aproximação (*zoom*).

- `setViewersCenter()`: Percorre todos os nós da lista ligada de referências às janelas de visualização, executando o método `setCenter()` do objeto referenciado por `renderer`. O método `setViewersCenter()` é chamado quando a aplicação do usuário requisita o ajuste de centralização (*pan/orbit*) da imagem na tela.

- `setViewersAlpha()`: Percorre todos os nós da lista ligada de referências às janelas de visualização, executando o método `setAlpha()` do objeto referenciado por `renderer`. O método `setViewersAlpha()` é chamado quando a aplicação do usuário requisita o ajuste da escala da coordenada w.

- `draw()`: Principal método da classe `T2Viewers`, cuja função é coordenar o processo de criação das *display lists* nos objetos de `T2Models`, com o auxílio do módulo `primitives`, em cada janela de visualização. Quando ocorre uma mudança na construção geométrica, como o acréscimo ou remoção de primitivas ou a seleção de primitivas pelo usuário, este método é chamado com a finalidade de atualizar o contexto de todas as janelas de visualização instanciadas.

O código em C++ que implementa `draw()` está exibido logo abaixo:

```
void T2Viewers::draw()
{
    for ( listIt it = viewers.begin(); it != viewers.end(); it++ ) {
        if ( it→flatModel != NULL ) {
            it→flatModel→beginDisplayList( it→render );
            repository→drawOnModel( it→flatModel );
            it→flatModel→endDisplayList( it→render );
        };
        if ( it→sphericalModel != NULL ) {
            it→sphericalModel→beginDisplayList( it→render );
            repository→drawOnModel( it→sphericalModel );
            it→sphericalModel→endDisplayList( it→render );
        };
        it→render→redraw();
    };
}
```

5.5.5 Módulo `events`

A figura 5.9 na página 50 mostra as classes que compõem o módulo `events`, cuja finalidade é tratar todos os eventos que o visualizador recebe e, também, definir as interfaces gráficas onde o usuário interage diretamente. Os eventos tratados por este módulo têm origem na aplicação do usuário e na interface gráfica. Eventos provenientes do sistema operacional são tratados automaticamente pelo Qt Designer (explicado na seção 5.2) para que a aplicação seja portável para outras plataformas suportadas pelo Qt Designer.

A classe `T2ViewerControl` é responsável pela janela principal do $\mathbb{T}^2$ Viewer (janela de controle). A partir desta, o usuário pode instanciar janelas de visualização bidimensional e tridimensional, remover todas as primitivas geométricas inseridas, restaurar todas as janelas de visualização para seus ajustes iniciais (feitos pela aplicação do usuário) e finalizar a execução do visualizador. Por ser a classe central do visualizador, o objeto do tipo

`T2ViewerControl` é o primeiro objeto a ser instanciado durante a inicialização do $\mathbb{T}^2$ Viewer, sendo, portanto, responsável pela inicialização dos demais módulos da aplicação, como `primitives` e `viewers`. Este objeto também é responsável pela comunicação com a aplicação do usuário, através do módulo `sockets`, cabendo a ele interpretar os pacotes recebidos e criar os pacotes a serem enviados. A interpretação dos pacotes recebidos representa o tratamento de todos os eventos originados na aplicação do usuário e, para isso, a instância de `T2ViewerControl` comunica-se com os diversos módulos que compõem o $\mathbb{T}^2$ Viewer.

As classes `T2Viewer2D` e `T2Viewer3D` são responsáveis pelas janelas de visualização bidimensional e tridimensional. Cada janela contém uma instância de uma das subclasses de `T2Renderer` (`T2Renderer2D` para `T2Viewer2D` e `T2Renderer3D` para `T2Viewer3D`) do módulo `renderers` (descrito na subseção 5.5.1). Janelas de visualização são instanciadas pelo usuário através da janela de controle. Nelas, o usuário pode controlar diversos aspectos da visualização, como a aproximação (*zoom*) e a centralização (*pan*/*orbit*) da imagem, através do acionamento dos ícones presentes nos menus das interfaces gráficas descritas por `T2Viewer2D` e `T2Viewer3D`. Nestes menus há ícones que representam modos de operação da janela de visualização, como a seleção de primitivas geométricas na tela, o ajuste da escala da coordenada w e o ajuste do ângulo de visão do observador virtual. Como esses modos de operação são exclusivos, somente um ícone permanece ativado por vez (dentre aqueles que representam modos de operação). Há ícones, no entanto, que não possuem este caráter exclusivo, pois representam a visualização ou não dos modelos de $\mathbb{T}^2$ e a presença ou não dos eixos que representam a base ortogonal do espaço onde o modelo esférico de $\mathbb{T}^2$ é desenhado.

Os métodos da classe `T2ViewerControl` são:

- `launch2DViewerButton()`: Instância um objeto da classe `T2Viewer2D`, abrindo uma janela de visualização bidimensional na tela, e acrescenta uma referência deste ao módulo `viewers` (através do método `addViewer()`). Em seguida, chama o método `draw()` de `viewers` para que o contexto da nova janela seja atualizado.

- `launch3DViewerButton()`: Instância um objeto da classe `T2Viewer3D`, abrindo uma janela de visualização tridimensional na tela, e acrescenta uma referência deste ao módulo `viewers` (através do método `addViewer()`). Em seguida, chama o método `draw()` de `viewers` para que o contexto da nova janela seja atualizado.

- `restoreAllButton()`: Restaura todas as janelas de visualização para seus ajustes iniciais, feitos pela aplicação do usuário, de escala da coordenada w, aproximação (*zoom*) e centralização (*pan*/*orbit*) da imagem na tela. Isso é feito através da chamada ao método `setRestore()` do módulo `viewers`.

- `clearButton()`: Apaga todas as primitivas geométricas armazenadas no módulo `primitives` (através do método `clear()`).
- `receiveControlConnection()`: Método chamado sempre que uma nova aplicação cliente conecta-se ao visualizador, retornando, através de um pacote de dados, o número de identificação a ser atribuído à aplicação cliente. Este valor provém do atributo numérico `newClientID`, sendo, este último, incrementado logo em seguida.
- `receiveControl()`: Recebe um pacote de controle e executa sua requisição. As requisições de controle permitem à aplicação do usuário definir aspectos de visualização (como aproximação e centralização da imagem), aspectos de desenho (como a definição da cor das primitivas geométricas) e aspectos de contexto (como a deleção de todas as primitivas geométricas armazenadas).
- `receiveData()`: Recebe um pacote de dados e executa sua requisição. Pacotes de dados são usados para enviar e receber primitivas geométricas, junto com meta-informações que complementam sua descrição. Através destes pacotes, o visualizador recebe as primitivas geométricas que deverão ser desenhadas e envia à aplicação do usuário primitivas requisitadas por ela.

Os principais atributos de `T2ViewerControl` são referências a módulos (`primitives`, `viewers` e `sockets`). Os métodos de `T2Viewer2D` são:

- `saveImageButton()`: Grava a imagem produzida pelo visualizador em um arquivo de imagem. Os formatos possíveis para este arquivo são: BMP, JPEG, PBM, PGM, PNG, PPM, XBM e XPM.
- `restoreButton()`: Restaura a janela de visualização bidimensional para seus ajustes iniciais, feitos pela aplicação do usuário, de escala da coordenada w, aproximação (*zoom*) e centralização (*pan*/*orbit*) da imagem na tela. Isso é feito através da chamada ao método `setRestore()` do módulo `viewers`.
- `selectButton()`: Ativa o modo de seleção de primitivas geométricas na tela.
- `zoomButton()`: Ativa o modo de ajuste de aproximação da imagem (*zoom*).
- `panButton()`: Ativa o modo de ajuste de centralização da imagem (*pan*/*orbit*).
- `frontBackRangeButton()`: Ativa o modo de ajuste interativo da opacidade de primitivas geométricas no aquém e no além do modelo plano de $\mathbb{T}^2$.
- `planeGradientButton()`: Ativa o modo de ajuste interativo da opacidade do plano que separa o aquém do além no modelo plano de $\mathbb{T}^2$.

Os objetos do tipo `T2Viewer2D` possuem atributos de referência a um objeto da classe `T2Renderers2D` (módulo `renderers`) e a outro objeto da classe `T2FlatModel` (módulo `models`). Os métodos de `T2Viewer3D` são:

- `saveImageButton()`: Grava a imagem produzida pelo visualizador em um arquivo de imagem. Os formatos possíveis para este arquivo são: BMP, JPEG, PBM, PGM, PNG, PPM, XBM e XPM.
- `restoreButton()`: Restaura a janela de visualização tridimensional para seus ajustes iniciais, feitos pela aplicação do usuário, de escala da coordenada w, aproximação (*zoom*) e centralização (*pan*/*orbit*) da imagem na tela. Isso é feito através da chamada ao método `setRestore()` do módulo `viewers`.
- `selectButton()`: Ativa o modo de seleção de primitivas geométricas na tela.
- `zoomButton()`: Ativa o modo de ajuste de aproximação da imagem (*zoom*).
- `wScaleButton()`: Ativa o modo de ajuste da escala da coordenada w.
- `orbitButton()`: Ativa o modo de rotação (*orbit*) interativa da figura geométrica tridimensional.
- `fovyButton()`: Ativa o modo de ajuste do ângulo de visão do observador virtual.
- `frontBackGradientButton()`: Ativa o modo de ajuste interativo da opacidade de primitivas geométricas no aquém e no além do modelo plano de $\mathbb{T}^2$.
- `planeGradientButton()`: Ativa o modo de ajuste interativo da opacidade do plano que separa o aquém do além no modelo plano de $\mathbb{T}^2$.
- `s2GradientButton()`: Ativa o modo de ajuste interativo da opacidade da superfície S^2 do modelo esférico de $\mathbb{T}^2$.
- `showFlatModelButton()`: Exibe ou não o modelo plano de $\mathbb{T}^2$.
- `showSphericalModelButton()`: Exibe ou não o modelo esférico de $\mathbb{T}^2$.
- `showArrowsButton()`: Exibe ou não os eixos que representam a base ortogonal do espaço onde o modelo esférico de $\mathbb{T}^2$ é desenhado.

5.5.6 Módulo `sockets`

A figura 5.10 na página 51 mostra as classes que compõem o módulo `sockets`, cuja finalidade é prover a comunicação entre a aplicação do usuário-programador e o visualizador. Seu funcionamento baseia-se no uso de *sockets*, estruturas de comunicação inter-processos freqüentemente usada entre aplicações locais e remotas. Através desta estrutura, a aplicação cliente (aplicação do usuário) constitui um canal persistente de comunicação TCP/IP com a aplicação servidora (visualizador). Todos os dados que constituem a comunicação são serializados antes de serem enviados pelo canal e desserializados no outro extremo. Serialização consiste em enfileirar todos os atributos que constituem o estado de um objeto de maneira facilmente reversível. Tal método é padrão nas estruturas de dados do CGAL e isso foi mantido na extensão de CGAL para $\mathbb{T}^2$ ([dO04]).

O uso de *sockets* permite, além da comunicação, a sincronia entre um programa orientado a eventos (geralmente aqueles que possuem interface gráfica, como o $\mathbb{T}^2$ Viewer) e outro que não o é (aplicações de console, como a aplicação do usuário-programador). Como mencionado nas seções 5.3 e 5.4, manter a diferença entre as naturezas das aplicações cliente e servidora oferece vantagens. Graças à implementação independente de plataforma da tecnologia de *sockets*, provida pelo ambiente Qt Designer (descrito na seção 5.2), o módulo `sockets` é capaz de produzir eventos sempre que um pacote de comunicação chega ao servidor, ao passo que é capaz de interromper a execução do cliente quando este requisita alguma informação do servidor (até que a informação chegue). Para tanto, `sockets` provê duas implementações distintas para a realização da comunicação inter-processos: a primeira, junto ao módulo `events`, funciona como os demais módulos do servidor, trocando mensagens e tratando eventos, enquanto a segunda, junto ao cliente, através do módulo `interface`, funciona de maneira linear (não paralela). O lado servidor é implementado pelas classes `T2ServerSocket` e `T2ListenSocket`, enquanto o lado cliente é implementado pelo componente `QSocketDevice`, provido pelo ambiente Qt Designer.

Existem dois tipos de pacotes de comunicação no $\mathbb{T}^2$ Viewer: um para dados e outro para controle. Pacotes de dados são usados para enviar e receber primitivas geométricas[6], junto com meta-informações que complementam sua descrição. Através destes pacotes, o visualizador recebe as primitivas geométricas que deverão ser desenhadas e envia à aplicação do usuário primitivas requisitadas por ela. Pacotes de controle permitem à aplicação do usuário definir aspectos de visualização (como aproximação e centralização da imagem), aspectos de desenho (como a definição da cor das primitivas geométricas) e aspectos de contexto (como a deleção de primitivas geométricas armazenadas).

Os métodos de `T2ListenSocket` são:

- `newClient()`: Método chamado sempre que uma nova aplicação cliente conecta-

[6]Estruturas de dados da extensão de CGAL para o plano projetivo orientado.

se ao servidor. Este método instância um objeto do tipo `T2ServerSocket` para atender, exclusivamente, às requisições do novo cliente.

Os métodos de `T2ServerSocket` são:

- `newData()`: Método chamado sempre que pacotes (dados ou controle) chegam ao servidor. Este método emite um sinal (evento) para o módulo `events` indicando a chegada de dados para serem lidos e interpretados.
- `connectionClosed()`: Método chamado quando o cliente finaliza sua execução. Como os objetos de `T2ServerSocket` são específicos para cada cliente, a chamada a este método causa a destruição do objeto, no servidor, referente ao cliente que encerra sua conexão.

Os atributos do pacote de dados, `T2DataPacket`, são:

- `clientID`: Atributo numérico que identifica a aplicação cliente.
- `primitiveID`: Atributo numérico que identifica a primitiva geométrica.
- `request`: Atributo de tipo enumerado que especifica a requisição da aplicação cliente. Seus valores podem ser: `Output` para enviar primitivas geométricas ao visualizador, `Input` para receber primitivas geométricas do visualizador, `Delete` para remover uma primitiva geométrica armazenada no visualizador e `Wait` para requisitar uma breve pausa na execução da aplicação cliente (o que é interessante para apresentações de algoritmos).
- `primitive`: Atributo de tipo enumerado que especifica o tipo de primitiva geométrica a ser enviada ao visualizador (ou recebida deste). Seus valores podem ser: `PointT2`, `SegmentT2`, `TriangleT2`, `IsorectangleT2`, `BboxT2`, `CircleT2`, `LineT2` e `RayT2`.
- `number`: Atributo de tipo enumerado que especifica o tipo numérico usado na primitiva geométrica a ser enviada ao visualizador (ou recebida deste). Seus valores podem ser: `int`, `float`, `double`, `MP_Float`, `Quotient<int>`, `Quotient<float>`, `Quotient<double>` e `Quotient<MP_Float>`.
- `data`: Atributo do tipo cadeia de caracteres usado para transportar a primitiva geométrica em sua forma serializada.

Os atributos do pacote de controle, `T2ControlPacket`, são:

- `clear`: Atributo booleano que especifica se todas as primitivas geométricas armazenadas no visualizador devem ser removidas.
- `radius`: Atributo numérico que especifica a aproximação da imagem na tela (*zoom*).
- `centerW`, `centerX`, `centerY`: Atributos numéricos que especificam a centralização da imagem na tela (*pan/orbit*).
- `alpha`: Atributo numérico que especifica a escala da coordenada w.
- `colorTgt`: Atributo de tipo enumerado que especifica em qual superfície as primitivas geométricas serão desenhadas com a cor especificada pelo atributo `color`. Os valores de `colorTgt` podem ser: `front` e `back` para o aquém e o além, respectivamente, do modelo plano de $\mathbb{T}^2$ e `sphere` para o modelo esférico de $\mathbb{T}^2$.
- `color`: Atributo numérico que especifica a cor com que serão desenhadas as novas primitivas geométricas enviadas ao visualizador. Este atributo especifica a cor das primitivas que serão desenhadas na superfície indicada por `colorTgt`.
- `resetTgt`: Indica qual dos parâmetros de controle devem ser restaurados para seus valores iniciais, especificados pela aplicação do usuário no início de sua execução. Os parâmetros são: `radius` para restaurar a quantidade de aproximação inicial da imagem na tela (*zoom*), *center* para restaurar a centralização inicial da imagem na tela (*pan/orbit*), `alpha` para restaurar a escala inicial da coordenada w e `colors` para restaurar as cores iniciais com as quais as primitivas geométricas enviadas serão desenhadas.

5.5.7 Módulo `interface`

A figura 5.11 na página 52 mostra as classes que compõem o módulo `interface`, que constitui a interface de programação (API) do $\mathbb{T}^2$ Viewer, através da qual a aplicação do usuário tem acesso às funcionalidades do visualizador. A maioria dos métodos da classe `T2Viewer`, referenciada diretamente pela aplicação cliente, tratam do envio de primitivas geométricas ao visualizador. Uma parte menor dedica-se ao recebimento de primitivas do visualizador e do controle de ajustes visuais.

A interface de programação utiliza o módulo `sockets` para estabelecer a comunicação entre o processo resultante da execução da aplicação do usuário com o processo referente ao visualizador (orientado a eventos). Alguns métodos de `sockets` são bloqueantes porque interrompem a execução da aplicação do usuário sempre que esta requisita alguma primitiva geométrica do visualizador ou um pequeno intervalo de tempo de espera (útil para

apresentações de algoritmos). Quando o visualizador retorna da requisição bloqueante, a aplicação do usuário resume sua execução.

Os métodos de `T2Viewer` são:

- `clear()`: Apaga todas as primitivas geométricas armazenadas no visualizador.
- `reset()`: Restaura diversos parâmetros para seu valores iniciais, especificados pela aplicação do usuário no início de sua execução. Os parâmetros são: `radius` para restaurar a quantidade de aproximação inicial da imagem na tela (*zoom*), `center` para restaurar a centralização inicial da imagem na tela (*pan/orbit*), `alpha` para restaurar a escala inicial da coordenada w e `colors` para restaurar as cores iniciais com as quais as primitivas geométricas enviadas serão desenhadas.
- `setRadius()`: Ajusta a aproximação da imagem na tela (*zoom*).
- `setCenter()`: Ajusta a centralização da imagem na tela (*pan/orbit*).
- `setAlpha()`: Ajusta a escala da coordenada w.
- `setColor()`: Especifica a cor com que serão desenhadas as novas primitivas geométricas enviadas ao visualizador.
- `operator≪(PointT2)`: Envia uma primitiva geométrica do tipo `Point_T2`, de CGAL estendido para $\mathbb{T}^2$, ao visualizador e retorna o seu número de identificação.
- `operator≪(SegmentT2)`: Envia uma primitiva geométrica do tipo `Segment_T2`, de CGAL estendido para $\mathbb{T}^2$, ao visualizador e retorna o seu número de identificação.
- `operator≪(TriangleT2)`: Envia uma primitiva geométrica do tipo `Triangle_T2`, de CGAL estendido para $\mathbb{T}^2$, ao visualizador e retorna o seu número de identificação.
- `operator≪(Iso_rectangleT2)`: Envia ao visualizador uma primitiva geométrica do tipo `Iso_rectangle_T2`, de CGAL estendido para $\mathbb{T}^2$, e retorna o seu número de identificação.
- `operator≪(BboxT2)`: Envia uma primitiva geométrica do tipo `Bbox_T2`, de CGAL estendido para $\mathbb{T}^2$, ao visualizador e retorna o seu número de identificação.
- `operator≪(CircleT2)`: Envia uma primitiva geométrica do tipo `Circle_T2`, de CGAL estendido para $\mathbb{T}^2$, ao visualizador e retorna o seu número de identificação.
- `operator≪(LineT2)`: Envia uma primitiva geométrica do tipo `Line_T2`, de CGAL estendido para $\mathbb{T}^2$, ao visualizador e retorna o seu número de identificação.

- `operator≪(RayT2)`: Envia uma primitiva geométrica do tipo `Ray_T2`, de CGAL estendido para $\mathbb{T}^2$, ao visualizador e retorna o seu número de identificação.
- `deletePrimitive()`: Apaga uma primitiva geométrica, armazenada no visualizador, especificada através do seu número de identificação.
- `wait()`: Suspende a execução da aplicação do usuário-programador por um período de tempo especificado através de um parâmetro numérico. Esta função é útil para apresentações de algoritmos.

O código em C++ a seguir é um exemplo de uma aplicação do usuário-programador que, através da interface de programação do $\mathbb{T}^2$ Viewer, realiza o *input* e o *output* de algumas primitivas geométricas.

```
int main() {
        T2Viewer w;
        w.clear();

        Point_T2<Signed_homogeneous <double> > p1( 1, -2, 1 );
        Point_T2<Signed_homogeneous <double> > p2( rando() );
        w ≪ p1; // Output of point p1
        w ≪ p2; // Output of point p2
        Segment_T2<Signed_homogeneous <double> > s( p1, p2 );
        w ≪ s; // Output of segment connecting p1 and p2

        w.draw(); // Draw primitives

        Line_T2<Signed_homogeneous<int> > l;
        w ≫ l; // Input of line
        Circle_T2<Signed_homogeneous<Quotient<float> > > c;
        w ≫ c; // Input of circle

        return 0;
}
```

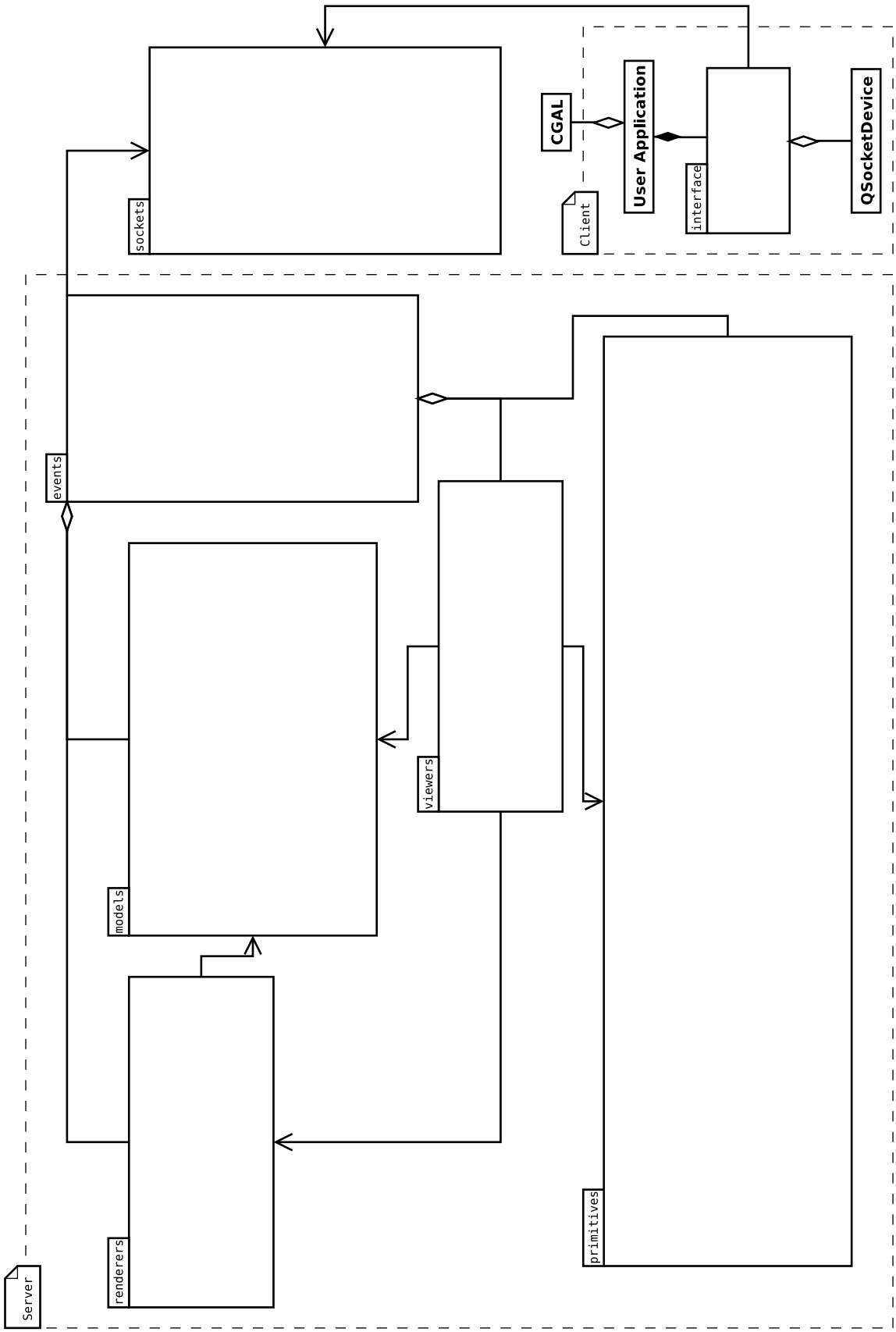

Figura 5.2: Diagrama de Módulos Funcionais

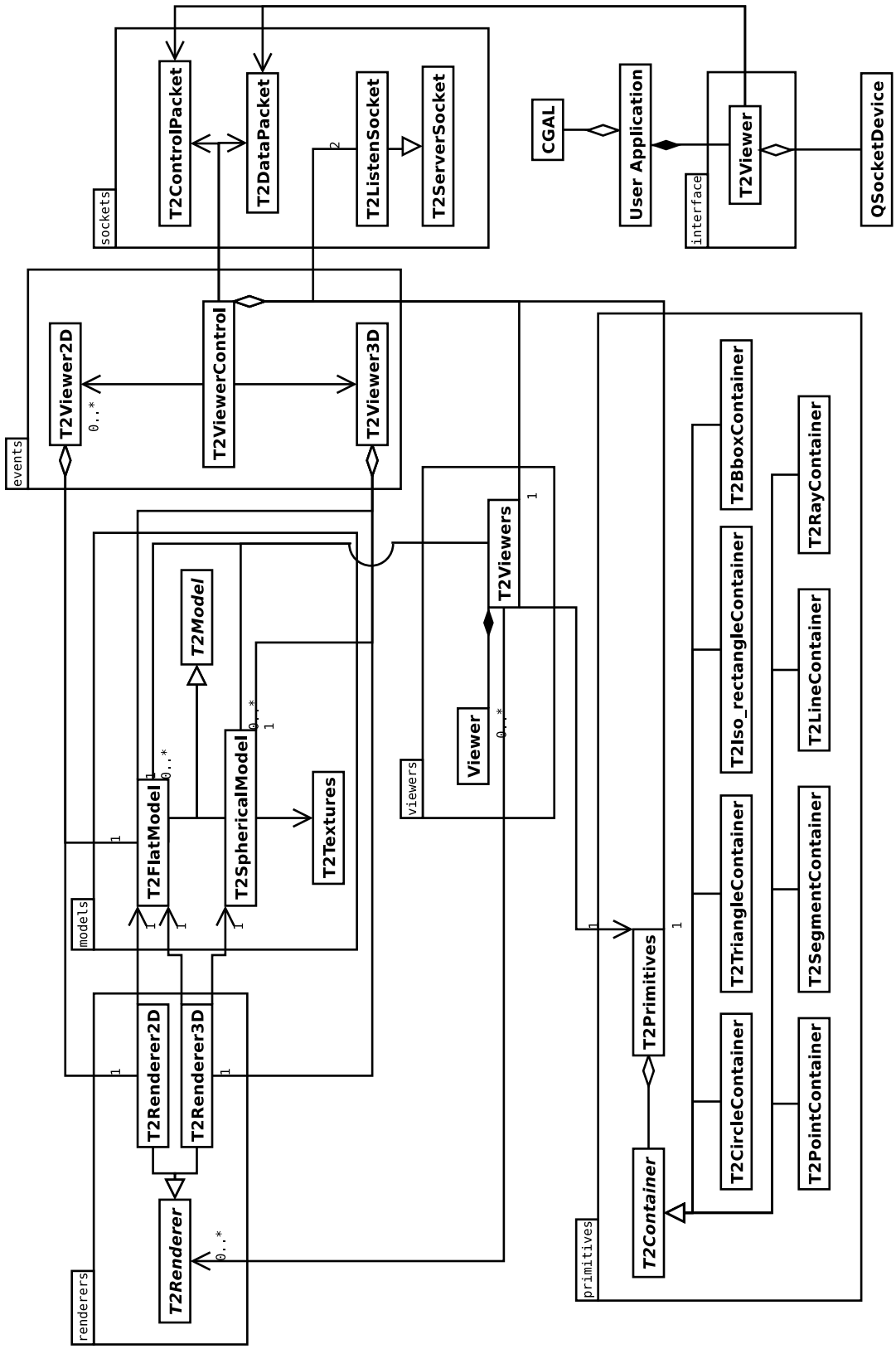

Figura 5.3: Diagrama de Módulos Funcionais e Classes

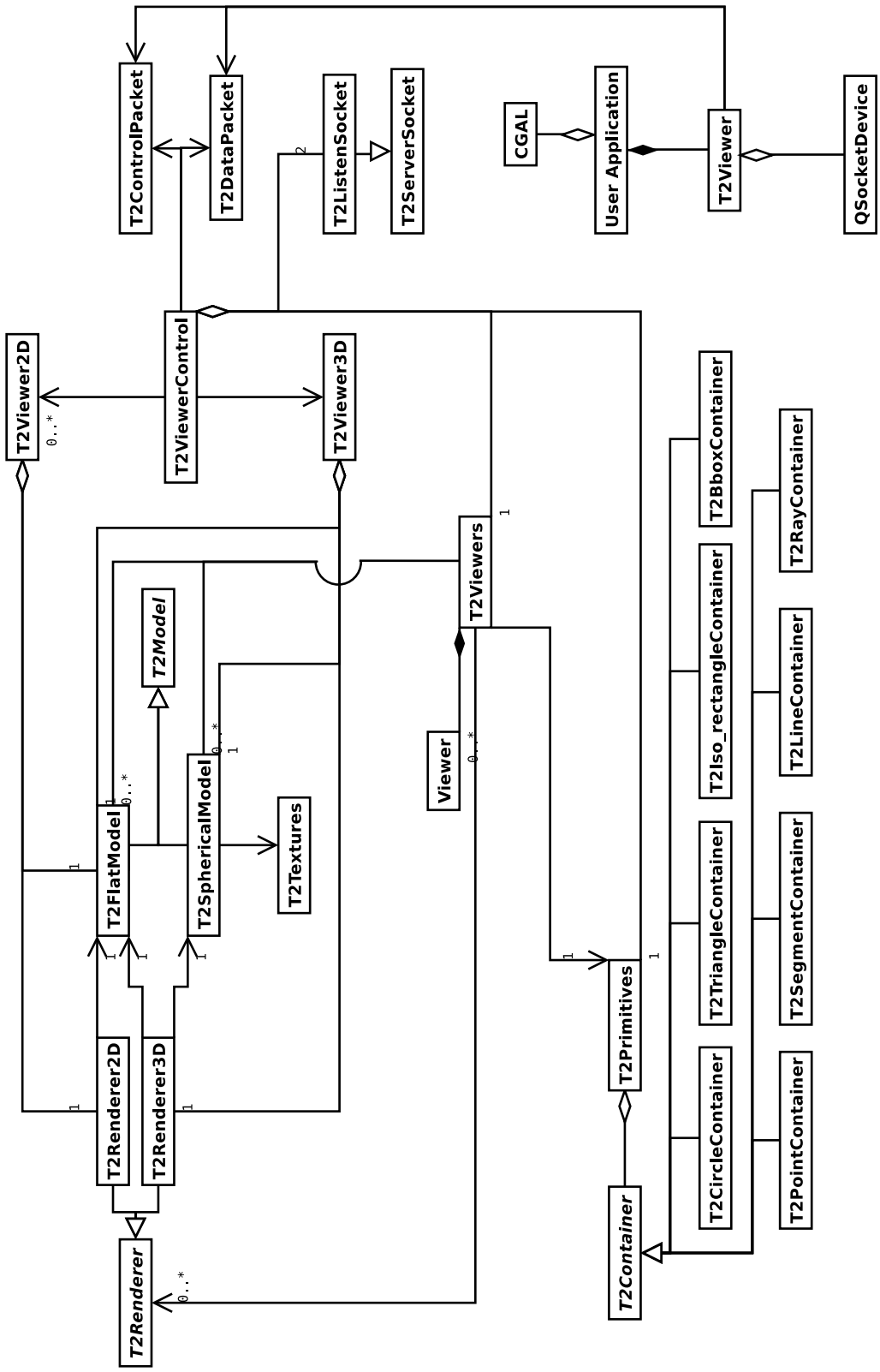

Figura 5.4: Diagrama de Classes

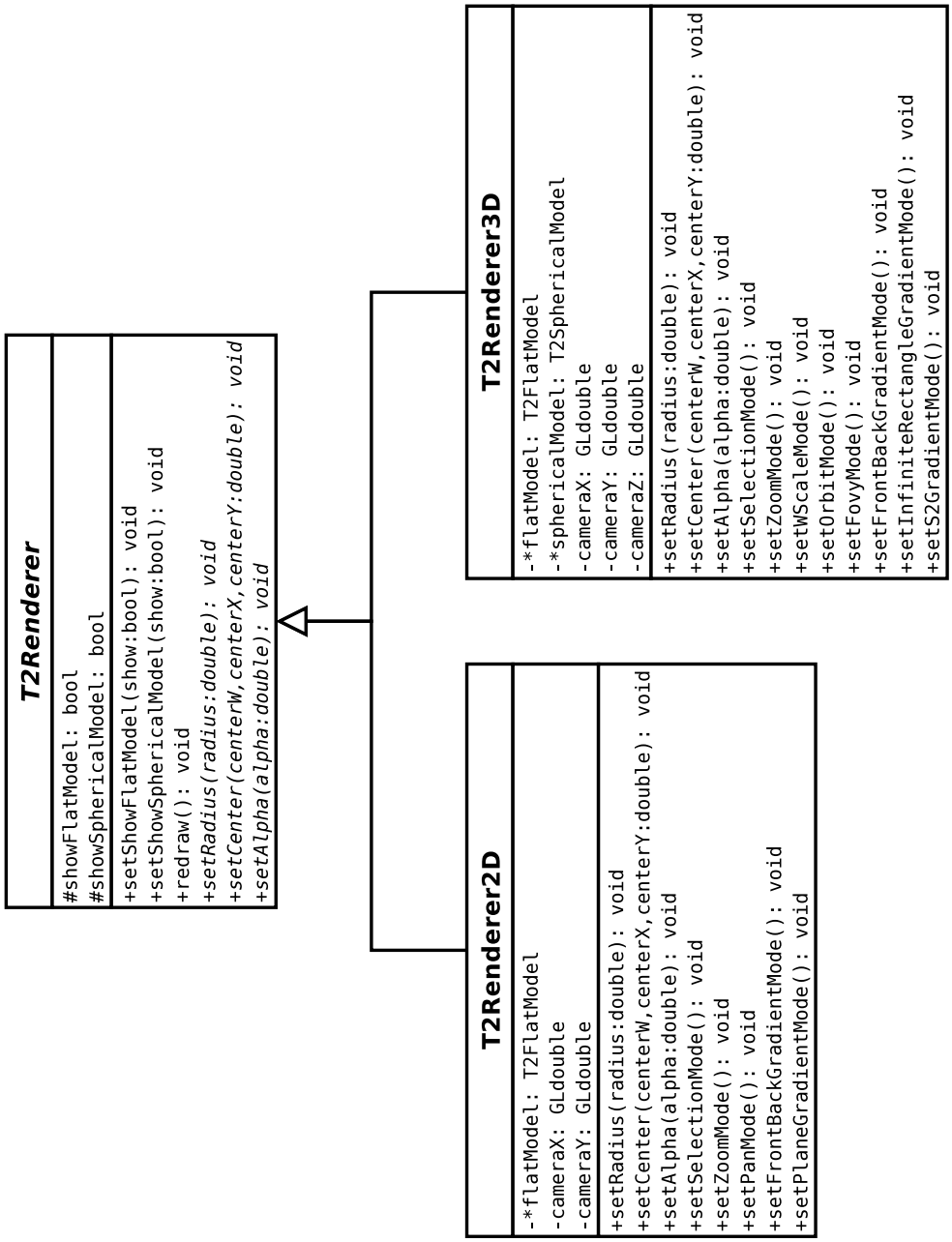

Figura 5.5: Módulo `renderers`

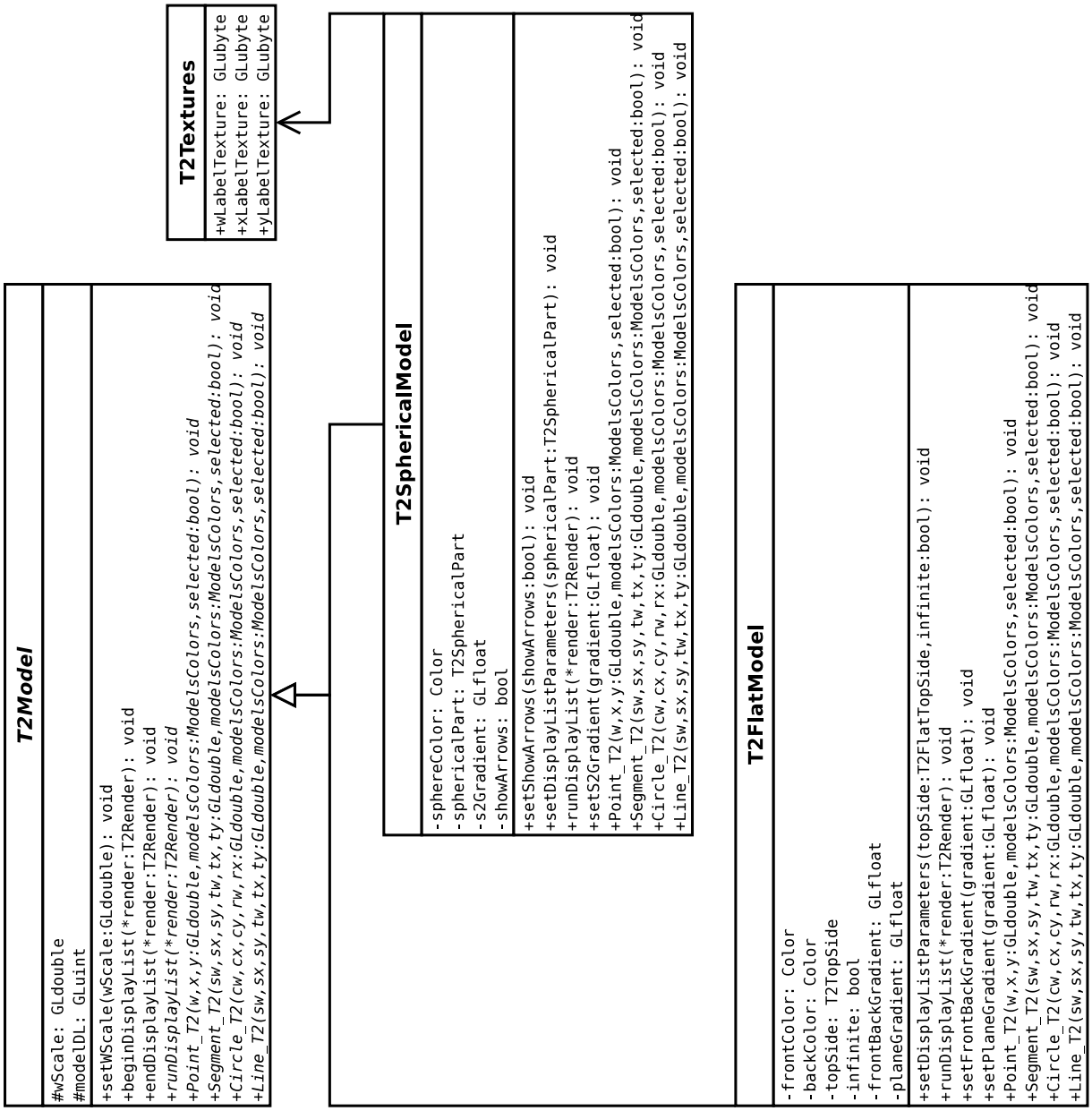

Figura 5.6: Módulo `models`

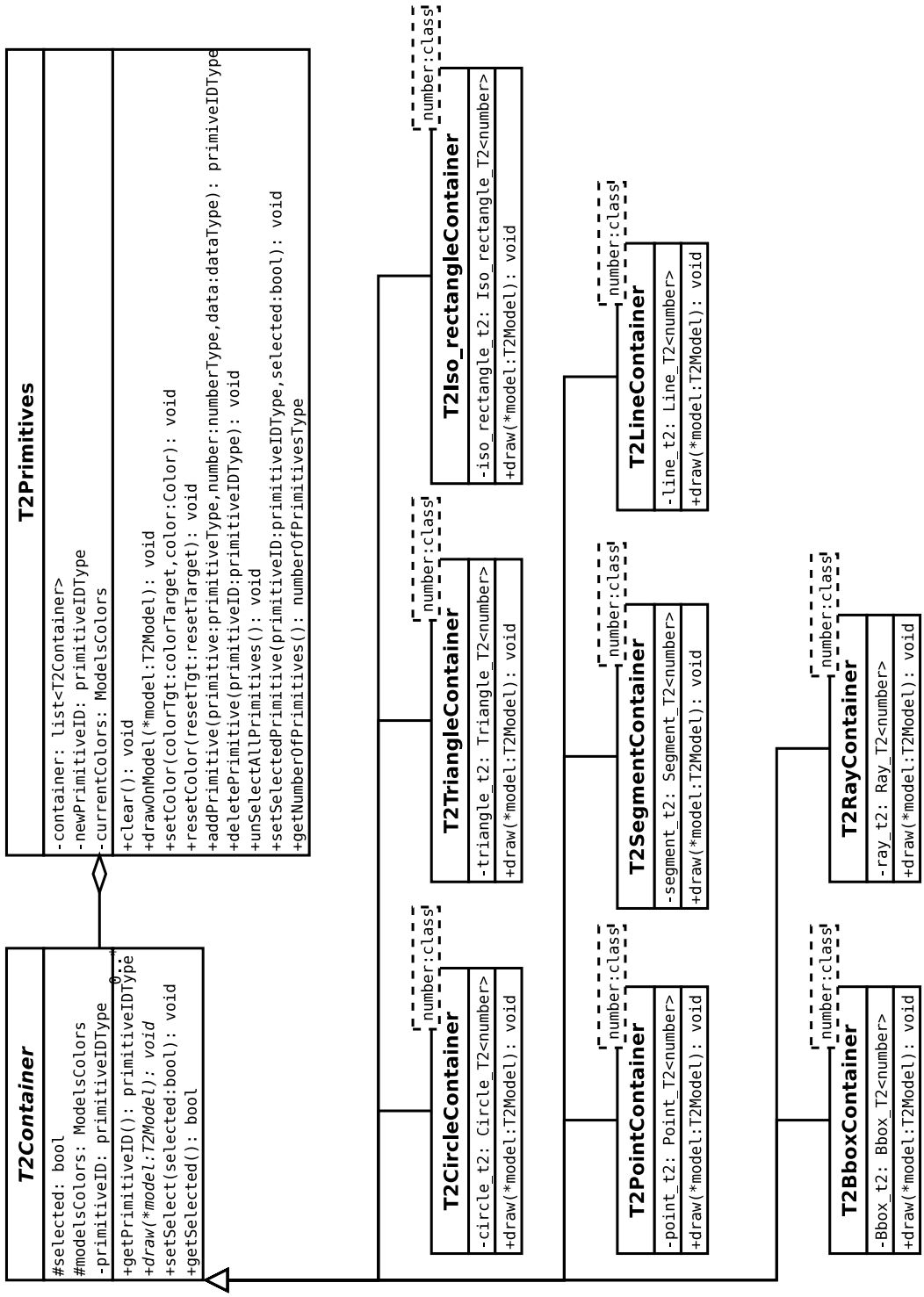

Figura 5.7: Módulo `primitives`

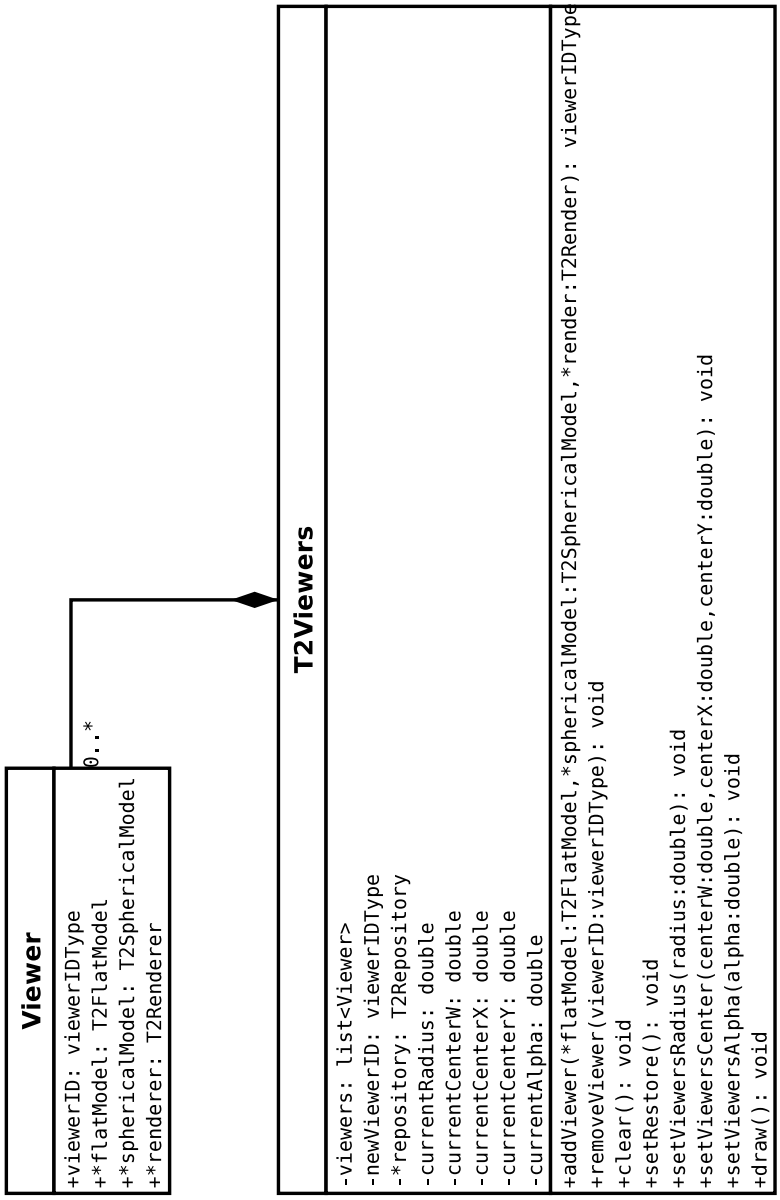

Figura 5.8: Módulo `viewers`

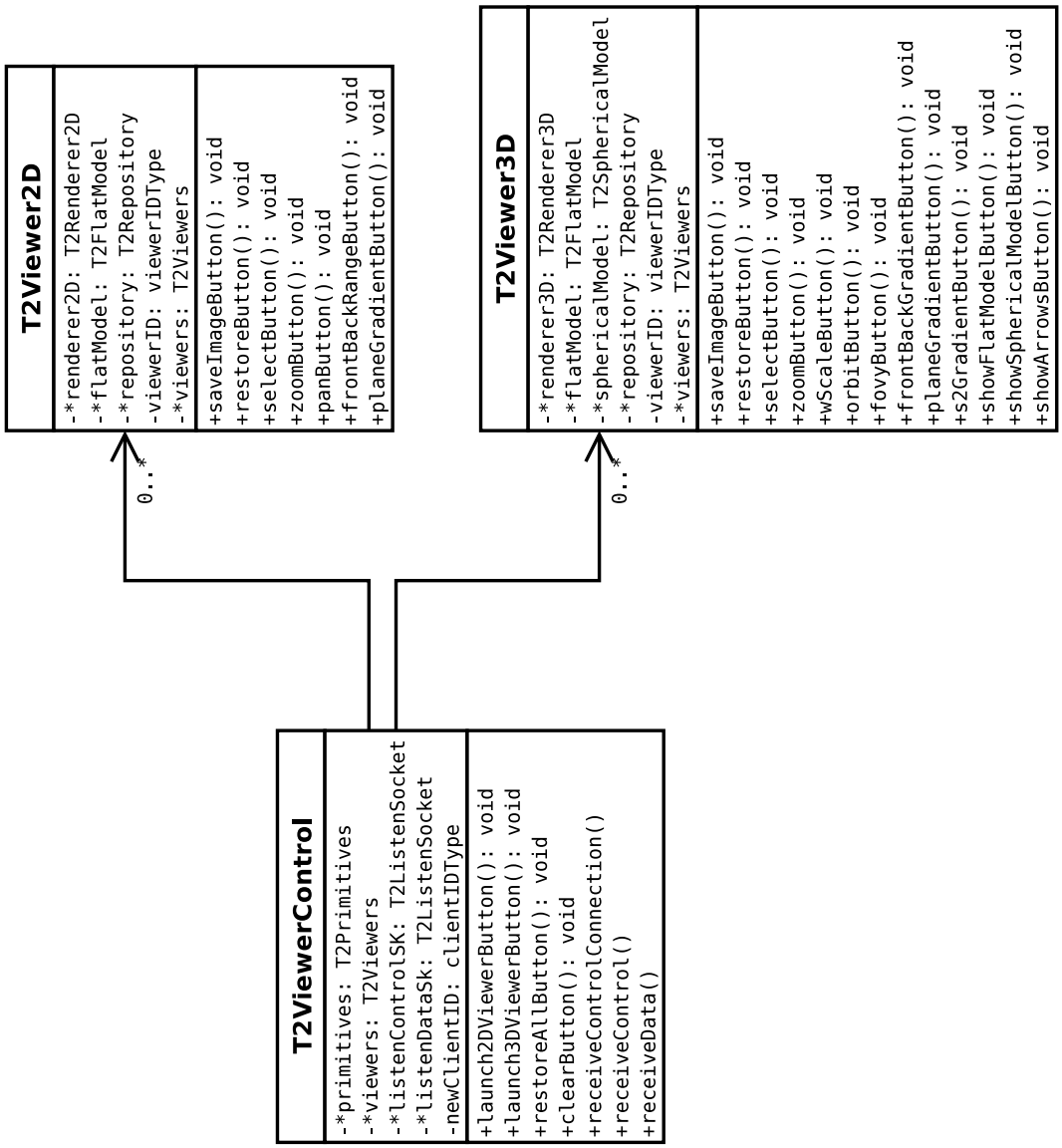

Figura 5.9: Módulo `events`

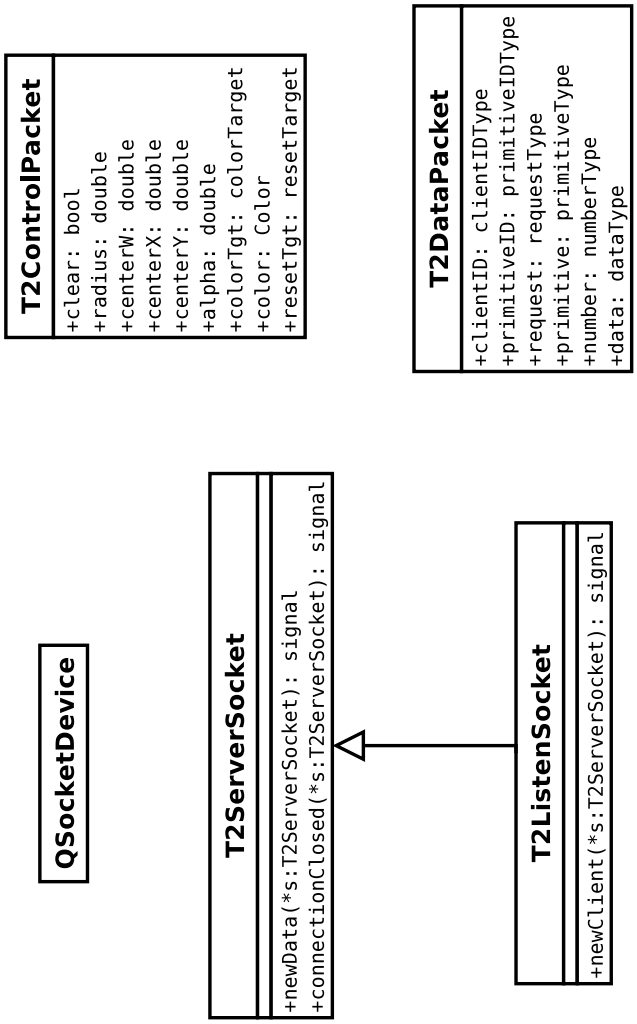

Figura 5.10: Módulo `sockets`

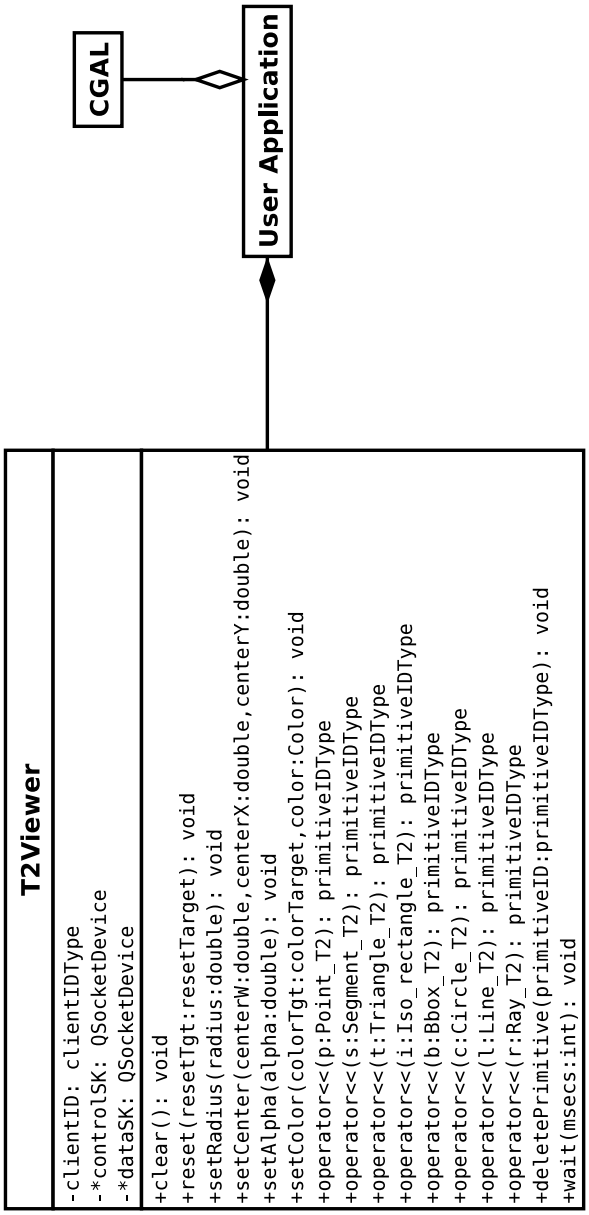

Figura 5.11: Módulo `interface`

5.6 Interface Gráfica

Nesta seção são apresentadas algumas imagens, capturadas durante a execução do $\mathbb{T}^2$ Viewer, com a finalidade de exibir a interface gráfica que compõe este visualizador. Primeiramente, damos ênfase às imagens renderizadas por este ambiente, mostrando o resultado da execução de algoritmos geométricos conhecidos, presentes na extensão de CGAL para $\mathbb{T}^2$ (explicado na seção 4.2). Em seguida, o mecanismo de entrada (*input*) através da interface gráfica é exibido, mostrando como o usuário pode interagir com algoritmos geométricos durante a execução dos mesmos. Por fim, os menus são apresentados e suas funcionalidades são descritas sucintamente.

A figura 5.12 mostra uma triangulação de Delaunay, exibida no modelo plano de $\mathbb{T}^2$. Neste modelo, por padrão, as primitivas geométricas no aquém são desenhadas em amarelo e as no além em verde.

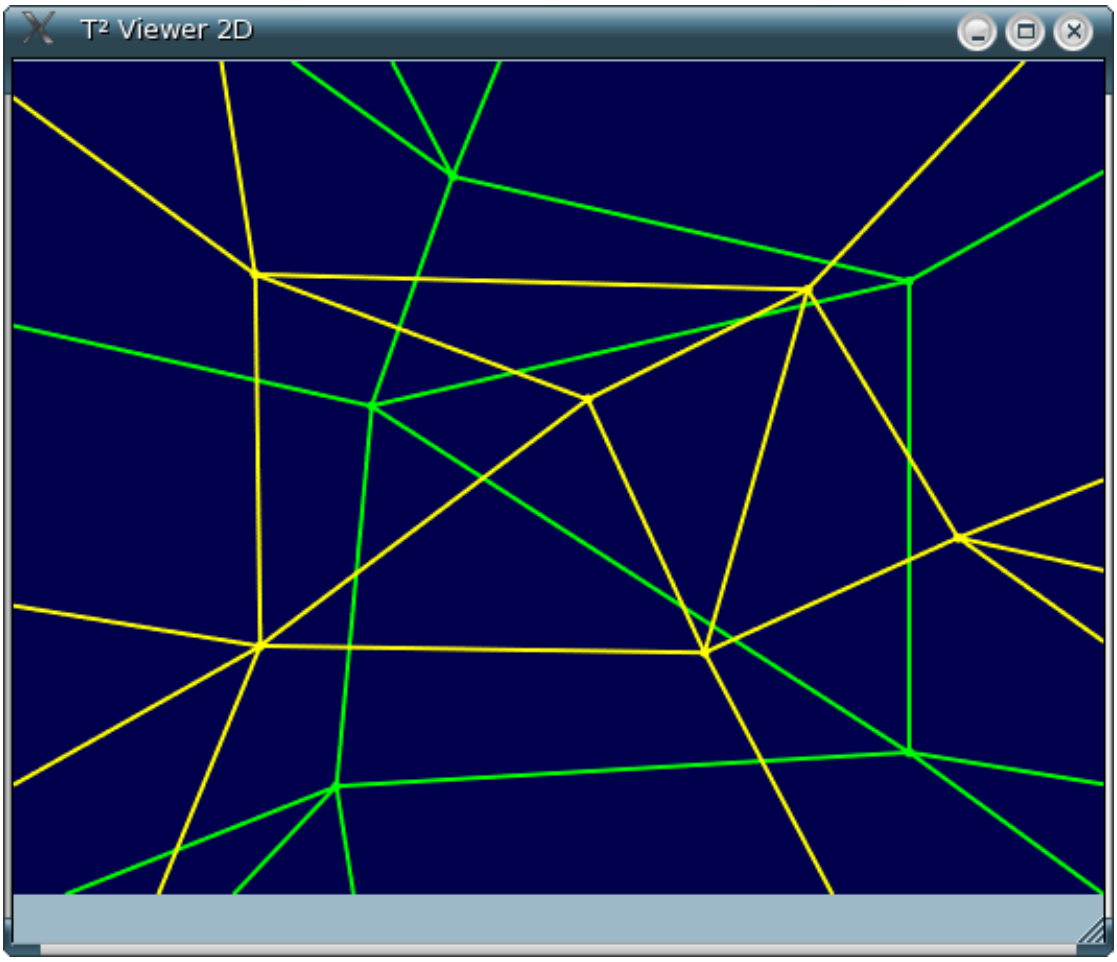

Figura 5.12: Triangulação de Delaunay no modelo plano de $\mathbb{T}^2$

A figura 5.13 mostra um diagrama de Voronoi, exibido no modelo esférico de $\mathbb{T}^2$. A imagem à esquerda mostra um diagrama de vizinho mais próximo e, como todos os sítios estão no aquém, a imagem à direita mostra que o diagrama estende-se através da reta do infinito (representada pela circunferência em vermelho) formando o diagrama de vizinho mais distante dos antípodas dos sítios, o que evidencia umas das vantagens da dualidade presente no plano projetivo orientado.

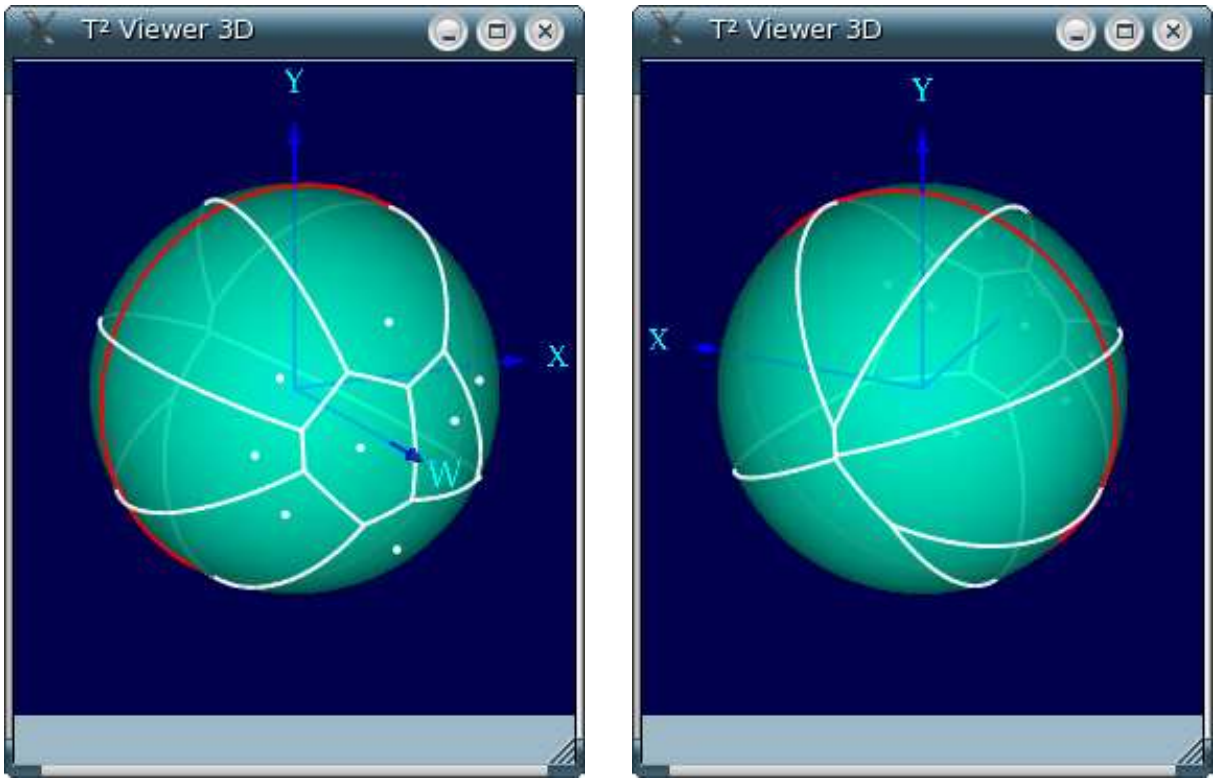

Figura 5.13: Diagrama de Voronoi no modelo esférico de $\mathbb{T}^2$

A figura 5.14 mostra o mesmo diagrama de Voronoi da figura 5.13, exibido, simultaneamente, em ambos modelos de $\mathbb{T}^2$. A opacidade das primitivas geométricas do além foi reduzida no modelo plano para enfatizar a projeção, das primitivas localizadas no aquém, do plano na esfera, na direção do centro de S^2.

Na figura 5.15, o mesmo diagrama de Voronoi, exibido em ambos modelos de $\mathbb{T}^2$, é observado através de dois pontos de vista interiores à esfera S^2. Na imagem da esquerda, o observador virtual situa-se próximo ao centro da esfera e, portanto, o desenho das primitivas geométricas nos dois modelos é quase coincidente. Na imagem da direita, o observador virtual está no centro de S^2 e, portanto, o desenho das primitivas de ambos modelos é perfeitamente coincidente. Para melhor visualização, o ângulo de visão foi aumentado para 120° nestes pontos de vista e a opacidade das primitivas geométricas do

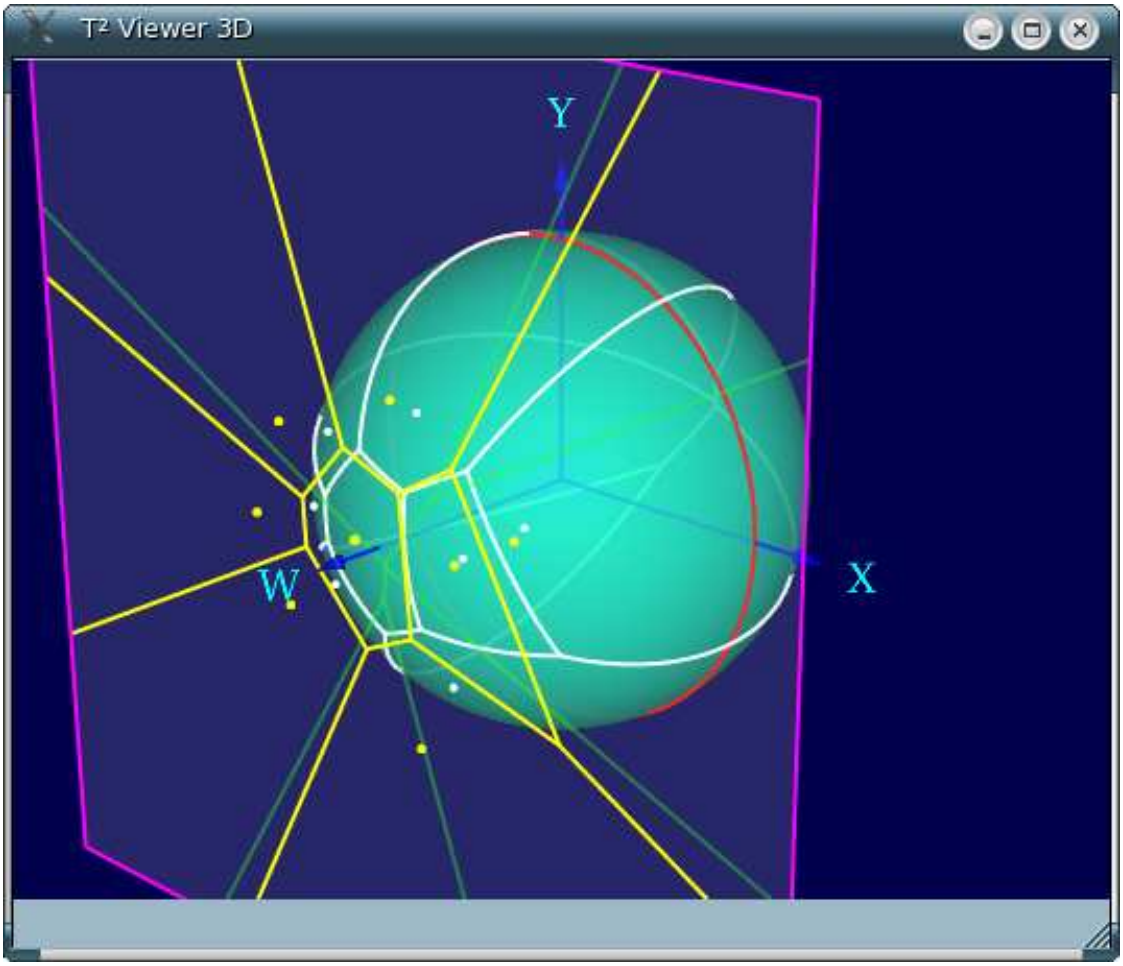

Figura 5.14: Diagrama de Voronoi em ambos modelos de $\mathbb{T}^2$

além foi completamente reduzida no modelo plano.

A figura 5.16 mostra o *input* de primitivas geométricas através da interface gráfica. Nesta figura, a imagem da esquerda mostra um segmento de $\mathbb{T}^2$ com um dos extremos no aquém e outro no além. Uma vez que a função de criação de segmentos está ativada, o usuário pode especificar, com o cursor, os extremos do segmento que deseja inserir. Neste processo, tanto pontos especificados no modelo plano quanto no modelo esférico são válidos. Em ambos modelos, enquanto se pressiona a tecla "Shift", o visualizador considerará como ponto de inserção o ponto antípoda ao especificado pelo cursor. Este recurso é particularmente interessante para *inputs* no modelo plano, pois pontos antipodais são coincidentes neste modelo. Na imagem da esquerda, podemos notar como o segmento aparece de maneira aparentemente descontínua no modelo plano enquanto aparece como um único arco no modelo esférico de $\mathbb{T}^2$. A imagem da direita mostra outros tipos de primitivas geométricas que o usuário pode inserir através da interface gráfica. A circunfe-

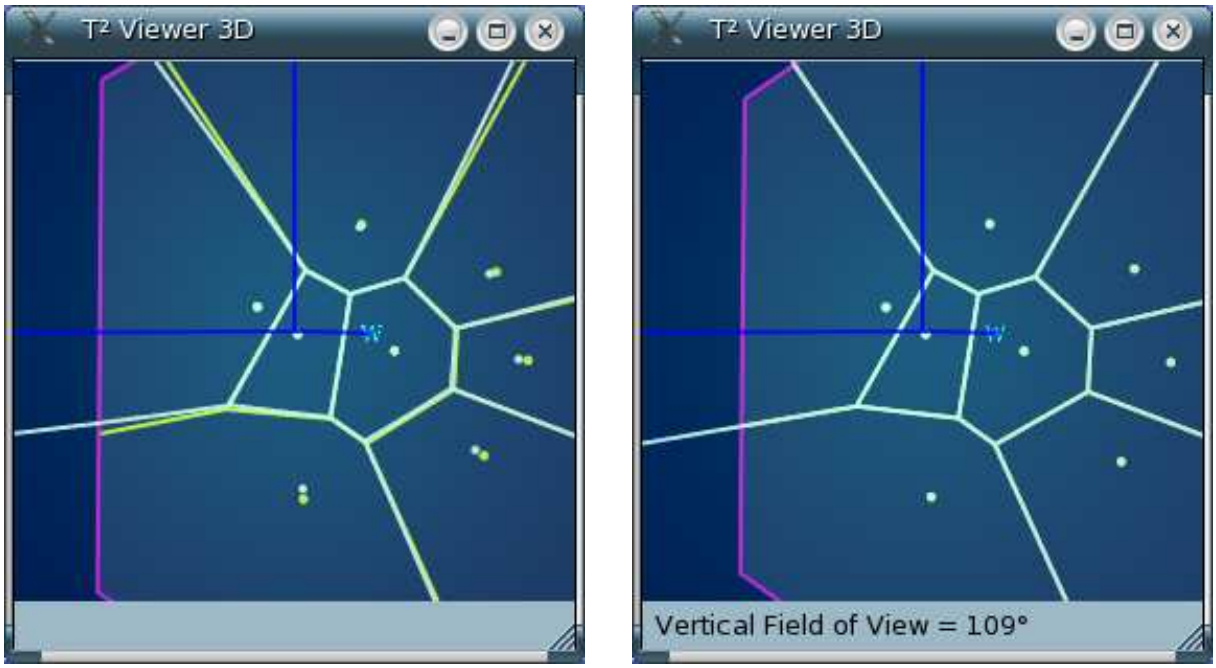

Figura 5.15: Diagrama de Voronoi em ambos modelos de $\mathbb{T}^2$. Pontos de vista interiores a S^2

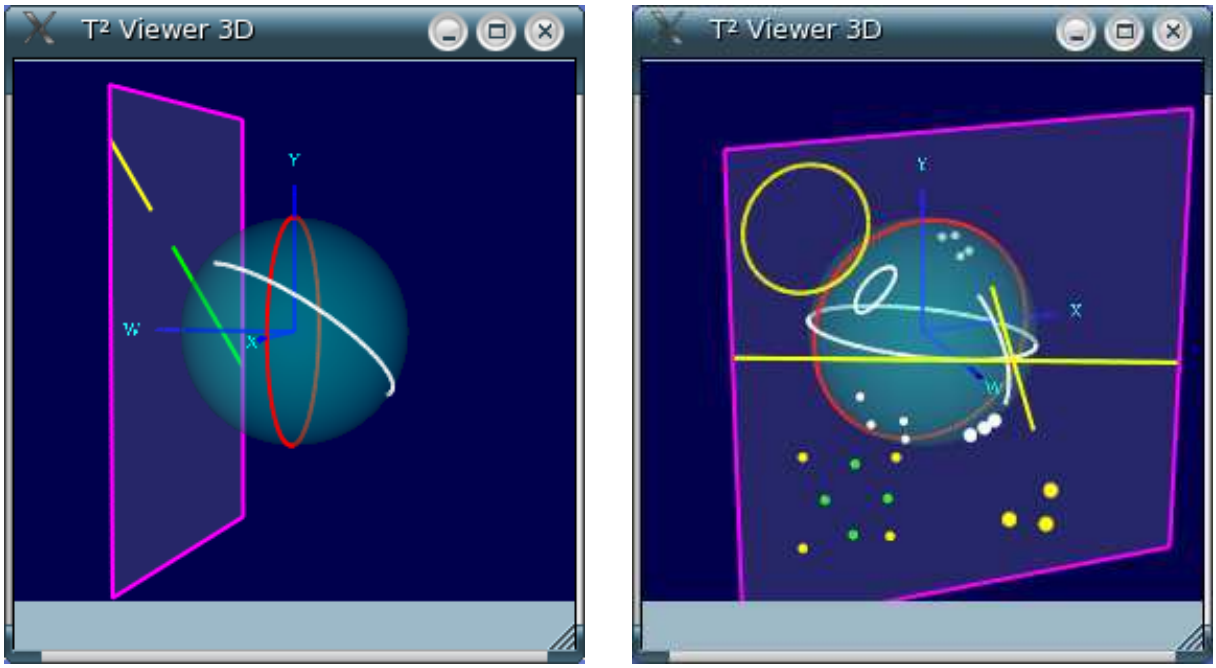

Figura 5.16: *Input* de primitivas geométricas no $\mathbb{T}^2$ Viewer

rência é especificada através de dois pontos (um no centro e outro sobre a circunferência) e a reta é especificada por dois pontos pelos quais esta passa. Os três pontos presentes na região inferior direita da figura aparecem em destaque por terem sido selecionados pelo usuário, o que pode ser feito de duas maneiras: seleção direta (clicando-se o cursor sobre a primitiva geométrica) ou através de um retângulo de seleção (especificado pelo cursor, através de dois pontos). Sempre que selecionada, a primitiva geométrica será desenhada com uma espessura maior. Ainda nesta imagem, podemos notar, como explicado na seção 3.2, que retas são representadas por circunferências de raio máximo sobre S^2 enquanto que segmentos são representados como arcos destas, no modelo esférico. Também observamos que circunferências são exibidas de maneira usual no modelo plano porém apresentam forma aparentemente distorcida no modelo esférico, principalmente quando estas situam-se nas proximidades da reta do infinito (representada pela circunferência em vermelho).

A figura 5.17 mostra o menu de modos de operação. Com exceção da primeira função (1) todos os botões descrevem modos mutuamente exclusivos, de maneira que somente um deles é ativado por vez. Com um modo ativo, o usuário pode utilizar sua função através do uso do cursor sobre a imagem. Uma breve descrição destas funções encontra-se a seguir (mais detalhes estão na subseção 5.5.1).

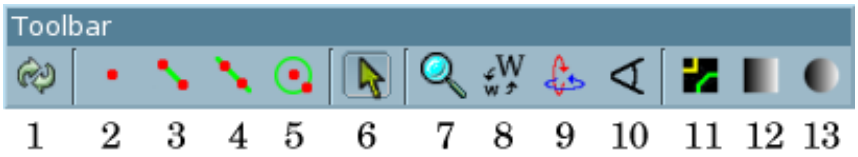

Figura 5.17: Menu de modos de operação

1. Restaura a posição inicial do ponto de vista, assim como o ângulo de visão inicial.
2. Ativa o modo de inserção de pontos de $\mathbb{T}^2$.
3. Ativa o modo de inserção de segmentos de $\mathbb{T}^2$.
4. Ativa o modo de inserção de retas de $\mathbb{T}^2$.
5. Ativa o modo de inserção de circunferências de $\mathbb{T}^2$.
6. Ativa o modo de seleção de primitivas na tela.
7. Ativa o modo de ajuste de aproximação (*zoom*).
8. Ativa o modo de escala da coordenada w (disponível somente nas janelas de renderização tridimensional).

9. Ativa o modo de rotação interativa ou automática (*orbit*/*orbitting*) da imagem tridimensional (disponível somente nas janelas de renderização tridimensional).

10. Ativa o modo de ajuste interativo do ângulo de visão do observador virtual (disponível somente nas janelas de renderização tridimensional).

11. Ativa o modo de ajuste interativo da opacidade de primitivas geométricas no aquém e no além do modelo plano de $\mathbb{T}^2$.

12. Ativa o modo de ajuste interativo da opacidade do plano que separa o aquém do além no modelo plano de $\mathbb{T}^2$.

13. Ativa o modo de ajuste interativo da opacidade da superfície S^2 do modelo esférico de $\mathbb{T}^2$ (disponível somente nas janelas de renderização tridimensional).

A figura 5.18 mostra o menu de exportação de imagens, disponível em ambas janelas de renderização, e o menu de exibição dos modelos de $\mathbb{T}^2$, disponível somente nas janelas de renderização tridimensional. Uma breve descrição das funções destes menus encontra-se a seguir (mais detalhes sobre estes menus estão, respectivamente, nas subseções 5.5.5 e 5.5.1).

Figura 5.18: Menu de exportação de imagens e menu de exibição dos modelos de $\mathbb{T}^2$

1. Grava a imagem produzida pelo visualizador em um arquivo de imagem. Os formatos disponíveis para este arquivo são: BMP, JPEG, PBM, PGM, PNG, PPM, XBM e XPM.

2. De maneira semelhante à função anterior, grava a imagem produzida pelo visualizador, porém, com o fundo da imagem na cor branca. Esta função é interessante para a exportação de imagens com a finalidade de serem incluídas em artigos impressos.

3. Exibe ou não o modelo plano de $\mathbb{T}^2$ (disponível somente nas janelas de renderização tridimensional).

4. Exibe ou não o modelo esférico de $\mathbb{T}^2$ (disponível somente nas janelas de renderização tridimensional).

5. Exibe ou não os eixos que representam a base ortogonal do espaço onde o modelo esférico de $\mathbb{T}^2$ é desenhado (disponível somente nas janelas de renderização tridimensional).

Capítulo 6

Trabalhos Futuros

O capítulo 5 descreve, inicialmente, os principais objetivos do $\mathbb{T}^2$ Viewer. No decorrer de suas seções, o leitor pode notar como cada objetivo foi associado a uma funcionalidade e esta, por fim, implementada, percorrendo-se assim o caminho desde o nível mais alto de abstração até um nível mais baixo, próximo ao da implementação do visualizador.

O critério de escolha dos objetivos apresentados na seção 5.1 tem por finalidade a criação de um ambiente de visualização capaz de facilitar a observação das características intrínsecas à geometria projetiva orientada; o que é de grande valor para o usuário-programador da extensão de CGAL para $\mathbb{T}^2$. Temos, no entanto, a possibilidade de acrescentar novas funcionalidades relacionadas a objetivos não explorados pelo presente trabalho, expandindo a capacidade do $\mathbb{T}^2$ Viewer para além de uma ferramenta de visualização.

Tornar o $\mathbb{T}^2$ Viewer um ambiente de animação de algoritmos é uma possibilidade factível, e pode ser feita através da inserção de novas propriedades para a visualização de primitivas geométricas. Alterando-se, ao longo do tempo, os valores destas propriedades, é possível produzir uma animação que descreve, visualmente, a atuação de algoritmos nas estruturas de dados (primitivas geométricas). Dentre as novas propriedades, podemos considerar, por exemplo, o uso de tracejado, o uso de cores com diferentes níveis de translucidez e a inserção e remoção de primitivas geométricas através de algum tipo de destaque visual (como *blinking*). Devido às características de projeto do $\mathbb{T}^2$ Viewer, o acréscimo destas funcionalidades pode ser feito de maneira local, alterando-se apenas alguns módulos (como `models`) e acrescentando outros. O uso da aceleração gráfica OpenGL proporciona numerosas variações no desenho de primitivas geométricas, facilitando a implementação de novas propriedades de visualização. A necessidade de eventos relacionados ao tempo é um recurso usado no $\mathbb{T}^2$ Viewer para coordenar a rotação automática (*orbitting*) da figura geométrica na tela e pode ser usado para coordenar a animação de algoritmos. Para agilizar o processo de criação de animações de algoritmos, pode ser interessante acrescentar

ao $\mathbb{T}^2$ Viewer o suporte a linguagens script, como Python [Pyt98], Perl [Per87] ou Lua [Lua93]. Através delas é possível criar e alterar uma seqüência visual que descreve os passos de um algoritmo, durante sua execução, sem a necessidade de uma nova compilação a cada alteração. Deve-se mencionar, no entanto, que organizar estas propriedades visuais de maneira a representar, intuitivamente, as operações que os algoritmos geométricos realizam não é uma tarefa trivial e requer um profundo estudo nesta área específica.

O $\mathbb{T}^2$ Viewer pode ser modificado de modo a utilizar a estrutura de dados *half-edge* para armazenar todas as primitivas geométricas no visualizador, ao invés de utilizar objetos do tipo `T2Container`, como explicado na subseção 5.5.3 (módulo `primitives`). A vantagem de tal abordagem é tornar uniforme tanto a armazenagem das primitivas como suas rotinas de desenho. Atualmente, em `primitives`, cada tipo de primitiva geométrica é armazenada em um objeto específico, subtipo de `T2Container`. Neste subtipo, a rotina de desenho é capaz de desenhar especificamente a primitiva contida. Caso um novo tipo de primitiva geométrica seja criada para a extensão de CGAL para $\mathbb{T}^2$, será necessário criar mais um subtipo de `T2Container` e implementar sua rotina de desenho de modo específico à primitiva contida. Com o uso da *half-edge*, todas as primitivas geométricas mais básicas e estruturas geométricas de nível mais alto (como diagramas e triangulações) são armazenadas de maneira topologicamente uniforme. Um percurso sistemático pela *half-edge* é capaz de extrair todas as primitivas geométricas básicas, independentemente de sua pertinência ou não a uma estrutura geométrica de nível mais alto. Desta maneira, a programação das rotinas de desenho fica orientada somente às primitivas básicas, não sendo mais necessário criar rotinas específicas para desmembrar cada estrutura geométrica de nível mais alto, para que estas possam ser desenhadas. Com esta modificação, o $\mathbb{T}^2$ Viewer aumentará sua escalabilidade.

Outra possibilidade para trabalhos futuros no $\mathbb{T}^2$ Viewer está na exploração de modelos alternativos para a visualização de $\mathbb{T}^2$. Os modelos plano e esférico de $\mathbb{T}^2$, explicados na seção 3.2, desempenham um importante papel na compreensão e desenvolvimento da intuição do usuário que já possui bons conhecimentos da geometria euclidiana e pretende se aprofundar no plano projetivo orientado. No entanto, ao passo que o modelo plano exibe figuras geométricas de maneira mais usual em relação a $\mathbb{R}^2$, este é incapaz de representar, de maneira visualmente contínua, primitivas geométricas que atravessam o infinito. O modelo esférico, por sua vez, não tem esta limitação, porém representa a figura geométrica com aparente distorção nas proximidades do infinito. Permanece em aberto o desafio de se criar um modelo de visualização para $\mathbb{T}^2$ capaz de incorporar apenas as vantagens dos dois modelos mencionados aqui. A implementação de um novo modelo para $\mathbb{T}^2$, no entanto, não oferece grandes dificuldades, bastando que o novo modelo seja um subtipo de `T2Model`, no módulo `models` (subseção 5.5.2).

O projeto modular e a escolha do ambiente de desenvolvimento (seção 5.2) tiveram

por finalidade a escalabilidade e a durabilidade. Acredita-se que estes fatores facilitem e estimulem trabalhos futuros envolvendo o $\mathbb{T}^2$ Viewer.

Capítulo 7

Conclusões

Na seção 5.1 estão descritas várias funcionalidades incorporadas ao $\mathbb{T}^2$ Viewer, com a finalidade de torná-lo um eficiente visualizador para a extensão de CGAL ao plano projetivo orientado. Dentre as funcionalidades comuns às presentes em trabalhos anteriores (descritos no capítulo 2), destacam-se a capacidade de mostrar detalhes das figuras geradas (*zoom*), a capacidade de mudar o ponto de vista do observador (*orbit*) em tempo real e a capacidade de rotação animada do ponto de vista do observador (*orbitting*). Esta última facilita a compreensão da figura geométrica tridimensional pelo usuário, devido ao uso do movimento para ilustrar a profundidade da cena.

No entanto, os benefícios de maior destaque se concentram nas funcionalidades inéditas deste visualizador. A exibição simultânea dos modelos de $\mathbb{T}^2$ permite ao usuário observar as relações intrínsecas entre estes; além disso, o ajuste da transparência das superfícies dos modelos plano e esférico torna possível enfatizar a figura geométrica representada em um dos modelos ou em ambos. A liberdade no posicionamento do ponto de vista do observador virtual e a possibilidade de alteração do ângulo de visão deste fazem com que seja possível observar a figura geométrica por posições interiores à esfera S^2 onde, no centro desta, a imagem da figura geométrica representada no modelo plano sobrepõe-se à imagem da mesma figura representada no modelo esférico.

A escala da coordenada w mostrou-se eficiente na visualização de figuras geométricas extensas, no modelo esférico de $\mathbb{T}^2$. Com este recurso, o usuário é capaz de observar a região central da figura, com valores pequenos para a escala da coordenada w (próximos à unidade), ou observar a região periférica da figura com valores grandes para esta escala. Neste último caso, o recurso reduz a aparente distorção presente nas proximidades do infinito, no modelo esférico de $\mathbb{T}^2$.

As funcionalidades aqui apresentadas contribuem para a compreensão do plano projetivo orientado (e suas propriedades) pelo usuário, facilitando o aprendizado de $\mathbb{T}^2$. No entanto, para usuários já habituados a esta geometria, o $\mathbb{T}^2$ Viewer também oferece bene-

fícios, como a ilustração de construções geométricas, a depuração visual da implementação de algoritmos e a facilidade que o ambiente oferece para gerar entradas para algoritmos geométricos, através de sua interface gráfica, dando assim amplo suporte para a criação de novos algoritmos para $\mathbb{T}^2$, provenientes ou não de algoritmos para $\mathbb{R}^2$.

Os requisitos de software e de hardware descritos na seção 5.2 visam a continuidade de uso deste projeto. Através da plataforma gratuita Linux, do uso de tecnologias de licença aberta (GPL e seus derivados), do uso de hardware difundido e do projeto modular, espera-se facilitar e estimular a manutenabilidade e continuidade de uso do $\mathbb{T}^2$ Viewer. Neste sentido, até mesmo uma eventual necessidade de se portar o ambiente para outras plataformas, como Windows e Macintosh, é facilitada pelos softwares e tecnologias de desenvolvimento multi-platarforma, como OpenGL e Qt Designer.

Durante a elaboração deste trabalho, dois vídeos foram feitos demonstrando as características e funcionalidades do $\mathbb{T}^2$ Viewer. Ambos foram submetidos e aprovados, respectivamente, nos seguintes simpósios: SIBGRAPI [OdRSD04] e 21st Annual ACM Symposium on Computational Geometry [OdRSD05]. O vídeo submetido ao SIBGRAPI está disponível, junto com o código-fonte do $\mathbb{T}^2$ Viewer e outras informações, em `http://www.ic.unicamp.br/~rezende/T2Viewer`.

Referências Bibliográficas

[Bro91] M. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. http://research.compaq.com/SRC/zeus/home.html, 1991.

[CGA96] CGAL - Computational Geometry Algorithms Library. http://www.cgal.org, 1996.

[dO04] A. G. de Oliveira. Implementação do Plano Projetivo Orientado na Biblioteca CGAL. Master's thesis, UNICAMP - Universidade Estadual de Campinas, 2004.

[dRJ93] P. J. de Rezende and W. R. Jacometti. Animation of Geometric Algorithms using GeoLab. *Proceedings of the Ninth Annual Symposium on Computational Geometry*, 1993.

[dRS94] P. J. de Rezende and J. Stolfi. *Fundamentos da Geometria Computacional*. UFPE-DI, IV Escola de Computação, 1994.

[Geo92] Geomview. http://www.geomview.org, 1992.

[Gmb98] Algorithmic Solutions Software GmbH. LEDA - Library for Efficient Data types and Algorithms. http://www.algorithmic-solutions.com/enleda.htm, 1998.

[Gon96] C. N. Gon. Computação Exata em Geometria Projetiva Orientada e Tratamento de Degenerações. Master's thesis, UNICAMP - Universidade Estadual de Campinas, http://www.dcc.unicamp.br/~rezende/geopro, 1996.

[GPL91] GNU General Public License. http://www.gnu.org/copyleft/gpl.html, 1991.

[Lee95] D. T. Lee. GeoSheet. http://www.ece.northwestern.edu/~dtlee/theory/geosheet_doc.html, 1995.

[Lua93] Lua Programming Language. http://www.lua.org, 1993.

[OdRSD04] A. G. Oliveira, P. J. de Rezende, and F. P. Selmi-Dei. An Extension of CGAL to the Oriented Projective Plane $\mathbb{T}^2$ and its Dynamic Visualization System. *SIBGRAPI*, 2004. `http://www.inf.ufpr.br/sibgrapi2004`.

[OdRSD05] A. G. Oliveira, P. J. de Rezende, and F. P. Selmi-Dei. An Extension of CGAL to the Oriented Projective Plane $\mathbb{T}^2$ and its Dynamic Visualization System. *Annual ACM Symposium on Computational Geometry*, 2005. `http://www.socg05.org`.

[Per87] Perl Programming Language. `http://www.perl.org`, 1987.

[Pin98] G. A. Pinto. Generalizações do Diagrama de Voronoi construídas através de Cônicas no Plano Projetivo Orientado. Master's thesis, UNICAMP - Universidade Estadual de Campinas, 1998.

[Pyt98] Python Programming Language. `http://www.python.org`, 1998.

[Sac90] J. Sack. Computational Geometry Workbench. `http://www.scs.carleton.ca/~csgs/resources/cg.html`, 1990.

[Sto91] J. Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations*. Academic Press, Inc., 1991.

[UML] Unified Modeling Language. `http://www.uml.org`.

[Wes99] R. B. Westrupp. Diagramas de Voronoi de Ordem k na Geometria Projetiva Orientada. Master's thesis, UNICAMP - Universidade Estadual de Campinas, 1999.

[XTa90] XTango algorithm animation system. `http://www.cc.gatech.edu/gvu/softviz/algoanim/xtango.html`, 1990.

[XYZ90] XYZ GeoBench. `http://www.schorn.ch/geobench/XYZGeoBench.html`, 1990.