ANDRÉ LUIZ SOARES CLINIO DOS SANTOS

VIX – UM FRAMEWORK PARA SUPORTE A OBJETOS VISUAIS INTERATIVOS

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA

Rio de Janeiro, novembro de 1996

André Luiz Soares Clinio dos Santos

VIX – Um Framework para Suporte a Objetos Visuais Interativos

Dissertação apresentada ao Departamento de Informática da PUC-Rio como parte dos requisitos para a obtenção do título de Mestre em Informática: Ciências da Computação.

Orientador: Roberto Ierusalimschy

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, XX de XXXXXXXXXX de 1996 (data da defesa)

À minha família

Agradecimentos

Ao meu orientador e amigo Roberto, que ensinou-me a fazer ciência, mostrando caminhos ao invés de meras soluções.

Aos meus pais e à minha irmã por tudo o que já passamos juntos e pela paciência que tiveram nos momentos mais difíceis da confecção desta tese.

Aos companheiros *Jacksons* — André Costa, Carlos Cassino, Renato Borges e Renato Cerqueira — que, como amigos e companheiros, sempre estiveram junto comigo, contribuindo com idéias e sugestões indispensáveis para este trabalho. Um obrigado também para a nossa "Protegida", Anna M. Hester, que trabalhou no tema.

Aos amigos e amigas da Engenharia de Computação e da Pós-Graduação do Departamento de Informática da PUC-RIO.

Ao professor Marcelo Gattass e a todos do TeCGraf (Grupo de Tecnologia em Computação Gráfica da PUC-Rio) pela força e por acreditarem na importância do tema.

Aos professores e funcionários do Departamento de Informática da PUC-RIO.

Ao CNPq e ao SEPROC/CENPES/PETROBRAS pelo auxílio financeiro.

Resumo

Este trabalho apresenta a modelagem e a implementação de um *framework* para suporte a objetos visuais interativos. Sua arquitetura define um protocolo extensível para a confecção de novos objetos e a definição de novas operações aplicáveis sobre eles.

Para a construção deste *framework*, alguns requisitos são estabelecidos: extensibilidade, reuso, portabilidade e simplicidade conceitual. Tais requisitos foram cumpridos. Além disso, o *framework* possui ampla aplicabilidade para a construção de diversos tipos de ferramentas gráfico-interativas.

Para validar a flexibilidade do *framework*, foram construídos um sistema modular de manipulação direta de objetos, um compositor de objetos hierárquico, um editor gráfico e um *toolkit* de interfaces. Os resultados obtidos no desenvolvimento desses sistemas indicam que o *framework* atendeu os requisitos propostos.

Abstract

This work presents the modeling and the implementation of a framework which supports visual interactive objects. Its architecture defines an extensible protocol for constructing new objects and defining new operations.

For the construction of this framework, some requirements were established: extensibility, reuse, portability and conceptual simplicity. These requirements were accomplished. Moreover, the framework has a wide range of applicability for constructing several types of graphical and interactive systems.

In order to validate the framework's flexibility, a modular system for object direct manipulation, a hierarchical object composer, a graphical editor, and an interface toolkit were constructed. The results obtained from the development of these systems indicate that the framework has accomplished its requirements.

VIX pronuncia-se *vige*.
vige, *sf.* Exclamação típica de espanto da região nordeste do Brasil deteriorada da frase: "Virgem Maria!!!".

Baseado em: *Mini-dicionário de pernambuquês*.
B. Bernardino, Ed. Bagaço, 1994

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

No processo de desenvolvimento de *software*, é importante definir modelos e técnicas com as quais possam ser construídas aplicações com um melhor grau de reuso de código, extensibilidade e modularidade. Essas melhorias têm por objetivo facilitar e dinamizar a construção dos sistemas computacionais, cuja procura cresce bastante.

Do ponto de vista da Engenharia de Software, existe um compromisso de fazer com que a confecção de programas possa ser tratada da mesma forma que em outras engenharias: com prazos menores e melhor definidos, com a cooperação de diversas pessoas, com o reaproveitamento de unidades já existentes, etc. Uma forma de contribuir com esse objetivo vem da linha de reutilização, onde a construção de sistemas pode ser feita a partir de módulos já existentes e devidamente testados.

O reuso desses módulos pode ser atingido pelo uso da filosofia *componentware* [Ude94]. Nesse paradigma, são definidos *componentes de software*, que são unidades computacionais reutilizáveis. Essas unidades oferecem um serviço bem definido e podem ser guardadas para a construção de outros sistemas.

Componentes podem facilitar a construção de programas porque têm como características:

- a possibilidade de serem utilizados em várias aplicações. Desta forma, o programador não precisa implementar totalmente um novo programa, podendo reaproveitar partes já testadas;

- o encapsulamento de suas implementações e o fornecimento de um determinado serviço através de uma interface bem definida. Logo, o *design* das aplicações é forçado a reconhecer os tipos abstratos de dados que representem suas necessidades, aumentando a modularidade dos programas;

- a possibilidade de acréscimo ou substituição destes componentes, o que permite um desenvolvimento de programas em unidades melhor definidas.

Devido a esses aspectos, a adoção de uma filosofia *componentware* pode ajudar no atendimento da demanda por programas, facilitando o trabalho do programador em confeccioná-los.

1.1 Motivação

Na demanda que a informática tem vivenciado, os sistemas gráfico-interativos têm ocupado um espaço importante. Sua importância tem motivado o desenvolvimento de diversas ferramentas que facilitem a construção desse tipo de sistema, como bibliotecas gráficas (p.ex. [Nye90, TeC89, TeC96c]) e de interface (p.ex. [Ope91, Pet, LdFG+96]). Os sistemas de interface suportam, normalmente, um conjunto padronizado de objetos (*widgets*[1]) [BCCI94], oferecendo o que já foi chamado de "um pouco mais que um sistema de menus glorificado" [MR93]. Neste caso, os objetos específicos da aplicação precisam de outro suporte gráfico-interativo para sua construção.

Com a necessidade de utilização de dois sistemas, as aplicações gráfico-interativas lidam, em geral, com uma dicotomia: uma biblioteca se propõe a fornecer primitivas de desenho para a representação visual dos objetos particulares da aplicação, e outra os *widgets* e eventos de interface para dar suporte à interação. Como conseqüência dessa dicotomia, o programador precisa, muitas vezes, combinar esses dois sistemas no nível da aplicação, tendo que lidar com paradigmas diferentes.

Uma forma de minimizar a dicotomia entre os sistemas gráfico e de interface vem do conceito de *objetos visuais interativos* [BCCI94, CI95]. Sucintamente, tais objetos são entidades da aplicação que são exibidas e manipuladas pelo usuário. Além disso, encapsulam, em um único tipo abstrato de dados, suas representações e comportamentos, facilitando sua programação de forma modular e reutilizável. Esta definição é relacionada, por exemplo, com o conceito de *Abstract Data Views* [CILS93].

A implementação do conceito de objetos visuais interativos pode ser feita sob uma arquitetura que defina um protocolo extensível de interação usuário-objetos. Uma vez implementado esse *framework* (arquitetura), um objeto pode ser tratado como um *componente de software* que, uma vez confeccionado, pode ser reutilizado por diversas aplicações. Ao reutilizar comportamentos de objetos, as interações de manipulação direta podem ser tratadas por um protocolo genérico que se aplique a qualquer tipo de objeto. Logo, uma vez construídas essas interações, elas também podem ser guardadas para reuso em outros programas.

Com a confecção deste tipo de *framework*, programas com recursos gráfico-interativos podem ser construídos de forma reutilizável, modular e sem ter que lidar com dois paradigmas distintos. Desta forma, as aplicações podem manipular seus objetos particulares da mesma forma que os elementos de interface, de modo que o conjunto de interações passíveis de serem aplicadas a esses objetos não distingue a natureza deles. Além disso, se esta ferramenta for implementada de forma portátil, os sistemas e aplicações construídos sobre o *framework* serão também portáteis.

1.2 Requisitos

Tendo em vista o estudo e a validação de uma arquitetura que dê suporte à manipulação de objetos visuais interativos, este trabalho tem por objetivo definir um modelo e uma implementação de um *framework* que combine os recursos gráficos e de interface. Desta

[1] O termo *widget* pode ser entendido como elemento de interface

forma, os objetos não necessitam ser implementados por duas ferramentas distintas. Ao integrar os dois sistemas, o tratamento das representações e comportamentos dos objetos podem ser encapsulados em um único tipo abstrato de dados.

Além desse encapsulamento, é fundamental garantir a extensibilidade e a reusabilidade de objetos e operações aplicáveis sobre eles. As novas operações e os objetos, uma vez codificados, podem ser tratados como componentes, de modo que a confecção de novas aplicações não implica na recodificação de interações implementadas anteriormente. Além disso, é possível criar objetos não-convencionais que, uma vez implementados, ficam também disponíveis para reuso em outros programas.

A definição de novos objetos pode estender-se ao ponto de definir *toolkits* de interface. Ao criar novos *widgets* e tratá-los como objetos visuais interativos, pode-se garantir sua extensibilidade, assim como misturar mecanismos de posicionamento direto, via interação, com modos de descrição abstrata. Este recurso é possível porque as operações normalmente aplicadas a objetos particulares da aplicação podem também ser realizadas sobre os *widgets*. Apesar de interessante, esta proposta não incorpora tais recursos nem se compromete na construção do *toolkit*, mas prevê suporte completo para a sua implementação.

Outro requisito fundamental deste trabalho é portabilidade da ferramenta. Essa vantagem vem do fato da construção do *framework* ser feita sobre um sistema gráfico e gerador de eventos mínimo, mas multiplataforma. Assim, pode-se garantir que o *framework*, seus objetos e interações sejam totalmente portáteis.

Apesar de possuir recursos variados, a modelagem e a construção do *framework* visa torná-lo eficiente, de fácil compreensão e mínimo quanto ao consumo de recursos computacionais. Neste aspecto, é fundamental para este trabalho que a definição e a programação dos objetos visuais interativos obedeçam a um paradigma de simples compreensão.

Como características secundárias, o *framework* implementa mecanismos que suavizam o *feedback* com o usuário, evitando problemas de *flickering* quando da manipulação dos objetos. Além da suavização do *feedback*, é implementado um controlador de tempo — mecanismo que permite a definição de eventos síncronos.

1.3 Apresentação

Esta dissertação é dividida em mais quatro capítulos. O segundo capítulo descreve alguns sistemas gráficos e de interface, mostrando que, em maior ou menor grau, existe o tratamento do conceito de objetos visuais interativos.

O terceiro capítulo exibe a modelagem e a arquitetura de **VIX**, um *framework* para suporte a objetos visuais interativos. Quanto à modelagem, são mostrados seus conceitos e funcionalidades. No que diz respeito à arquitetura, é exposto como os conceitos envolvidos são implementados em um filosofia de orientação por objetos.

O quarto capítulo disserta a respeito de aplicabilidades do *framework*. Cada uma de suas seções mostra como algumas ferramentas já tradicionalmente utilizadas podem ser construídas sobre o *framework*, desde bilbiotecas de objetos até *toolkits* de interfaces.

O último capítulo desta dissertação é reservado a algumas considerações finais e conclusões.

Capítulo 2

Sistemas de Interface e Bibliotecas Gráficas

Este capítulo mostra diferentes tipos de ferramentas que propiciam a confecção de aplicações gráfico-interativas.

Na seção 2.1, faz-se uma pequena dissertação a respeito da mudança de filosofia com a introdução de sistemas orientados a eventos. Também é aprofundado o conceito de objeto visual interativo. A partir desse conceito, esta seção disserta sobre como essa definição é, em maior ou menor grau, tratada em algumas plataformas.

Dedica-se a seção 2.2 à exposição de algumas políticas que integram os sistemas gráficos e de interface. Esta seção exemplifica alguns sistemas oriundos dessas propostas de integração.

Finalmente, este capítulo dedica a seção 2.3 para mostrar o desenvolvimento de ferramentas gráfico-interativas no TeCGraf (Grupo de Tecnologia em Computação Gráfica da PUC-Rio).

2.1 Ambientes Gráfico-Interativos

A maioria dos sistemas gráfico-interativos adota o modelo de orientação a eventos, difundido a partir de Smalltalk [GR83]. Segundo essa modelagem, as ações executadas por um programa são ativadas a partir de eventos gerados pelo usuário. Sistemas com esse tipo de interface gráfica têm se popularizado, de forma que essa abordagem tornou-se padrão para o desenvolvimento de aplicações interativas.

No modelo de eventos, a aplicação tem definidas respostas para as ações que o usuário pode gerar (ativação de menus, botões, etc). Como conseqüência desse paradigma, tem-se que o controle da interação homem-máquina sai do programa e passa para o usuário.

As linguagens tradicionais, como C ou Pascal, não oferecem mecanismos de suporte para este novo enfoque. Desta forma, torna-se necessária uma simulação dessa inversão de controle [BCCI94]. Esta geralmente é implementada através do uso de um *loop* principal que lê os eventos e os processa. Neste caso, as próprias aplicações implementam o *loop* ou chamam explicitamente uma outra rotina do sistema de interface, que o faz.

Tipicamente, o fluxo de informações em um programa gráfico-interativo segue um padrão. A cada ação do usuário, são gerados eventos do sistema de interface que passam

o controle para a aplicação. Essa passagem de controle é feita, por exemplo, através do mecanismo de *callbacks*, que são funções pré-determinadas, codificadas pelo programa e que determinam a semântica dos eventos. Respondendo a eventos, a aplicação trata determinados conjuntos dessas ações como formas de manipulação de seus objetos. A interação do usuário com esses objetos da aplicação determina o aparecimento, neste trabalho, do conceito de *objetos visuais interativos*.

Objeto visual interativo é qualquer objeto de um programa que tenha uma representação gráfica e possa ser manipulado interativamente pelo usuário. Esses objetos encapsulam, em um único tipo abstrato de dados, suas representações visuais e seus comportamentos.

As seções seguintes dissertam sobre o tratamento de objetos visuais[1] em duas plataformas: XWindows e Microsoft Windows.

2.1.1 Ambiente XWindows

A plataforma XWindows [SG86] oferece como interface básica para programação a biblioteca Xlib [Nye90]. As aplicações construídas somente sobre Xlib/XWindows implementam suas interfaces sob as abstrações de janelas, primitivas gráficas e eventos.

Normalmente, utiliza-se como suporte gráfico o próprio Xlib e como ferramenta de interação o Intrinsics X Toolkit [AS90]. Com o Intrinsics, pode-se codificar a interface em um nível mais alto de abstração, através do conceito de *widget*. *Widgets* são elementos de interface genéricos que podem ser posteriormente associados a objetos como menus, botões, etc. Esses objetos podem ser agrupados, de modo a definir os diálogos da aplicação.

Apesar de estabelecido o conceito de *widget*, o Intrinsics não implementa nenhum elemento de interface, deixando sua codificação para outra camada. Assim, o Intrinsics tem como finalidade oferecer mecanismos que facilitem a construção de *toolkits* de interface. Exemplos de sistemas construídos sobre o Intrinsics são: Motif [Ope91], Athena Widget Set [Pet], etc. O modelo de aplicação sobre esta biblioteca é ilustrado na figura 2.1.

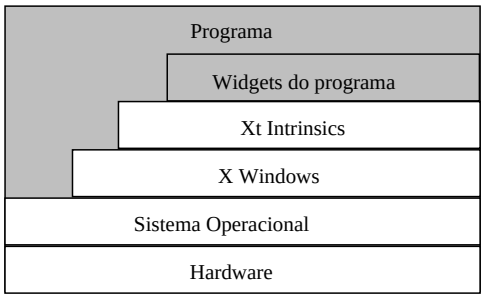


Figura 2.1: Modelo de desenvolvimento Intrinsics

[1] Os termos objeto visual e objeto visual interativo são equivalentes

Motif

Um dos sistemas mais difundidos em ambientes XWindows/Intrinsics é o Motif. Ele é composto de um gerenciador de janelas e um *toolkit* com uma linguagem de descrição de interface (UIL — *User Interface Language*).

O gerenciador de janelas (MWM — *Motif Window Manager*) faz parte do Motif. No entanto, sua discussão foge ao tema deste trabalho. O *toolkit* é formado por um conjunto de *widgets* baseado em Intrinsics e um conjunto de funções para seu tratamento. O UIL, *User Interface Language*, permite especificar o *layout* dos diálogos e associar *callbacks* a funções da aplicação. Estes arquivos são compilados em separado da aplicação e permitem uma rápida prototipação.

Basicamente, um programa construído sobre Motif deve lidar com outras bibliotecas. No mínimo, deve-se trabalhar com Motif e Intrinsics para o tratamento dos elementos de interface, e com Xlib para o desenho de primitivas gráficas. Utilizando essas duas bibliotecas, os objetos específicos da aplicação são, tipicamente, desenhados em uma *drawing area* (*canvas*), de onde o sistema de interface repassa os eventos do usuário. A interação sobre o *canvas* geralmente traz alterações nos atributos desses objetos (cor, posição, etc). Ao tratar essas modificações, a geração do *feedback* implica em uma passagem de controle para o sistema gráfico Xlib, que possui primitivas de desenho. A figura 2.2 ilustra o modelo de desenvolvimento Motif.

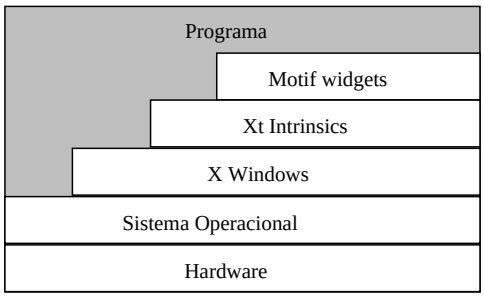


Figura 2.2: Modelo de desenvolvimento Motif

Objetos Visuais em XWindows

O conceito de *widget*, fornecido pelo Intrinsics, oferece mecanismos de tratamento de objetos visuais. *Widgets*, por serem elementos genéricos, podem ser associados a qualquer objeto, inclusive àqueles particulares da aplicação.

Tipicamente, *widgets* são utilizados por sistemas de interface, porque estes determinam um conjunto básico de elementos cuja representação e comportamento já estão parcialmente amarrados. Desta forma, os sistemas de interface gerenciam os elementos mais comumente utilizados e passam para a aplicação o controle de um *widget* especial que lida com seus objetos particulares, normalmente chamado de *canvas*.

Porém, nada impede a definição de um sistema de objetos visuais sobre o Intrinsics. No entanto, a confecção de um novo *widget*, ou objeto, não é uma tarefa simples de ser realizada por programadores não-experimentados. Isto porque o Intrinsics utiliza a

filosofia de orientação por objetos com uma implementação em C, o que acarreta um gerenciamento de classes feito pelo próprio programador.

Por adotar uma filosofia de orientação por objetos em C, o primeiro passo na construção de um novo objeto é a definição de seu posicionamento na hierarquia de classes. Definido o posicionamento, o programador deve declarar estruturas que representem no novo objeto:

- os acréscimos da classe — definem novos atributos do tipo do *widget*, utilizado para a definição a campos estáticos (*static* em C++) e implementação de tabelas virtuais;
- a classe — faz a composição explícita de seus tipos, declarando cada parte herdada que compõe a nova classe;
- os acréscimos do objeto — definem os novos campos a serem acrescentados a cada instância do *widget*.
- o objeto — faz a composição explícita dos *widgets* que compõem o novo objeto.

Para cada novo atributo definido, deve-se definir uma estrutura que forneça informações a seu respeito, como, por exemplo, seu nome, tamanho em memória, tipo, etc. Cada uma dessas estruturas/atributos deve ser colocada em uma lista denominada *resource list*.

Além de definir a estrutura que representa a classe do novo objeto, o programador precisa inicializá-la. A inicialização, feita com a instanciação de uma variável que representa a classe, é obrigatória e deve conter todos os atributos (*resources*) do objeto. Para cada atributo herdado, o programador informa se ele é redefinido ou não, atribuindo-lhe o valor da classe pai ou criando um novo. No caso dos novos atributos incorporados, definem-se, ao final da classe, seus valores.

A inicialização de atributos permite, inclusive, que se possam implementar *callbacks* — "métodos" da classe que permitem a chamada de funções de resposta a eventos. Para implementar esse mecanismo, pode-se associar um determinado conjunto de eventos a uma ação específica. Esta ação é representada por uma *string* que, associada a uma função da classe através do mecanismo de *translations*, permite chamar uma função de resposta. Esta, registrada previamente no Intrinsics, sempre recebe como parâmetro o próprio *widget*, de forma que o programador, nesta ação, pode chamar uma *callback* (um atributo do *widget*).

O Intrinsics define, para cada *widget*, uma janela nativa do sistema hospedeiro (XWindows). Isto faz com que *widgets* possam ser custosos computacionalmente. Com seu uso intensivo, a proliferação de janelas pode determinar um alto consumo de recursos computacionais com, inclusive, efeitos no desempenho.

Uma evidência da possibilidade do alto custo no mapeamento um para um entre *widgets* e janelas nativas vem da definição, no Intrinsics, dos *gadgets*. *Gadgets* são elementos que não possuem uma janela X associada e, conseqüentemente, tratamento de eventos. Logo, possuem algumas restrições, dentre as quais:

- Não recebem foco e, conseqüentemente, eventos de teclado

- Não recebem eventos de mouse, *enterwindow*, *leavewindow*, etc.
- Não têm o conceito de *stacking order*, isto é, não possuem a noção de que um elemento pode estar sobre outro.

Um tratamento adequado para os *gadgets* pode ser feito com a construção de um `CompositeWidget` pai (agrupador de objetos) que receba os eventos e os remeta para um filho, gerenciando o comportamento dos seus *gadgets*. Esses *composites* são responsáveis por um tratamento mais elaborado, pois os eventos que iriam para um filho, a princípio, são enviados ao pai. Desta forma, o *composite parent* deve tomar ações apropriadas para fazer com que seus *gadgets* "atuem" coerentemente quando do aparecimento de eventos.

Existem muitas maneiras de construir *composites* que façam a interação com seus *gadgets*. A especificação deste *widget* deve documentar que convenções ele espera que seus filhos entendam. Um *composite* está somente preparado para tratar filhos de uma determinada classe. Como o Intrinsics é implementado em C, o pai deve fazer uma verificação de tipo antes de permitir a inserção de um filho. Esta verificação ajuda na eliminação de erros de programação e pode ser visualizada no trecho de código abaixo ([AS90, pág.405]). Neste exemplo, permite-se a inserção de filhos que sejam *gadgets* específicos ou *widgets*.

```
if( !XtIsWidget(child) && !XtIsSubClass(child, allowedClass) )
{
    /* Issue error message */
}
```

A confecção desse *composite* pode determinar, também, a base de uma ferramenta de suporte a objetos visuais. No entanto, face ao esforço da confecção de um *framework* sobre o Intrinsics, vale a pena buscar soluções que:

- implementem essa filosofia com uma linguagem orientada por objetos. Desta forma, pode-se facilitar a programação do gerenciamento das "classes" de objetos;
- tenham a possibilidade de serem portadas para plataformas fora do padrão XWindows.

2.1.2 Ambiente Microsoft Windows

A plataforma Microsoft Windows [Pet90] oferece, como interface básica de programação, o SDK — Software Development Kit [Pet96], que é um conjunto de todas as APIs do Windows. Estas APIs (*Application Programm Interfaces*) são escritas em linguagem C e classificadas de acordo com suas funcionalidades, como, por exemplo: funções de multimídia, controle de processos, gerência de janelas, aplicação de primitivas gráficas (GDI — *Graphical Development Interface*), incorporação de elementos de interface nativos, etc.

Quanto ao tratamento de janelas, o SDK oferece serviços para a sua criação, aplicação de primitivas gráficas da GDI e um tratamento de eventos básicos do usuário. As janelas

podem ser de vários tipos, o que determina sua classe (*window class*). Cada classe possui um único tratador de eventos associado que, após a instanciação da janela, pode ser redefinido. O novo tratador pode, chamando ou não o antigo, fornecer mecanismos de herança no comportamento das janelas. A partir do conceito de janelas, o SDK oferece também um conjunto de elementos de interface nativos como botões, menus, etc. Desta forma, o conceito de janela serve de base para a definição de *widgets* nativos e outros objetos a serem posteriormente definidos.

O uso direto do SDK para programação em Windows pode tornar-se bastante complexo devido ao número de funções disponíveis e mensagens (cerca de 500) enviadas do sistema para as janelas. Além disso, a utilização de uma linguagem procedural como C em um paradigma de programação orientada a eventos com herança pode prejudicar, da mesma forma que no Intrinsics, a legibilidade e simplicidade dos programas. Neste aspecto, existem diversas bibliotecas em C++ que permitem uma maior abstração de particularidades do SDK, como a OWL — Object Windows Library [Inc94] e a MFC — Microsoft Foundation Classes [Mic95a, Mic95b]. Essas duas bibliotecas definem um encapsulamento das entidades definidas pelo SDK, oferecendo um conjunto de classes C++ que mapeia diversos objetos de interface, entidades para desenho, atributos, etc. A OWL, em sua versão 1.0, e a MFC, até sua versão 1.5, mapeavam basicamente os elementos nativos de interface, deixando as outras funcionalidades para serem tratadas sob o próprio SDK. Desta forma, o programdor precisava ainda utilizar dois sistemas distintos para a confecção de aplicações — SDK e OWL/MFC. As versões posteriores destas bibliotecas tenderam a ampliar o conjunto de recursos encapsulados.

No que diz respeito ao encapsulamento de elementos de interface, essas bibliotecas definem hierarquias de classes que agrupam objetos de acordo com seus atributos e comportamentos (métodos). Nessas hierarquias, existem classes que abstraem somente o conceito de janela. Assim, a redefinição e o acréscimo de alguns métodos nestas classes permitem a confecção de novos controles, pela redefinição dos tratadores de eventos.

Objetos Visuais em Windows

O conceito de janelas no Windows oferece, da mesma forma que os *widgets* do Intrinsics, mecanismos para o tratamento de objetos visuais. Por serem objetos genéricos, janelas oferecem o mapeamento de eventos da tela para o objeto. Os objetos podem, então, definir suas representações através das funções gráficas e responder a manipulações do usuário.

Bibliotecas do tipo OWL e MFC oferecem classes genéricas que são abstrações desse mecanismo, permitindo a confecção, por herança, de novos objetos. Isto siginifica que, de maneira análoga aos *widgets* do Intrinsics, esses controles são implementados através de um mapeamento um para um com janelas.

Assim, é possível utilizar o conceito de janela, diretamente do SDK ou não, para tratar objetos visuais interativos. Estes ficam restritos a uma forma retangular e, da mesma forma que o Intrinsics, podem acarretar um consumo elevado de recursos computacionais. Além do consumo de memória, a proliferação de janelas causa um alto consumo de recursos do próprio Windows (chamados de *resources*). O consumo elevado de *resources* pode afetar até mesmo a estabilidade do próprio sistema, fato que foi melhorado com o advento de subsistemas do tipo Win32s e a evolução da plataforma para o Windows 95

e Windows NT [Cer95].

Para minimizar o custo computacional da implementação de um *framework* de suporte a objetos visuais, pode-se definir uma ferramenta que permita a construção de novos objetos sob a filosofia de *gadgets* do Intrinsics. Assim, os objetos não estão amarrados a uma janela retangular nativa e passam a consumir menos *resources* do sistema. A implementação desse tipo de gerenciador de objetos pode seguir a linha dos *composite-widgets* do Intrinsics. Nesse gerenciador, um elemento pai agrupa objetos-filho, de modo a controlar qual deles deve responder a um determinado evento.

Apesar dessa construção ser possível, é interessante buscar soluções que tratem o conceito de objetos visuais mais facilmente e além da plataforma Microsoft Windows.

2.1.3 Abstrações sobre Ferramentas Básicas

Conforme visto anteriormente, as plataformas X Windows e Microsoft Windows oferecem suporte à confecção de objetos visuais, o que não constitui um processo trivial. Essas plataformas oferecem um conjunto de objetos mais normalmente utilizados, permitindo que a aplicação possa confeccionar seus próprios objetos com auxílio de um sistema gráfico.

Neste caso, os programas gráfico-interativos lidam com um sistema gráfico e um de interface, trazendo para o programador a necessidade de gerenciar o fluxo de controle entre esses dois sistemas. Um esquema desta composição na construção de programas gráfico-interativos pode ser visto na figura 2.3.

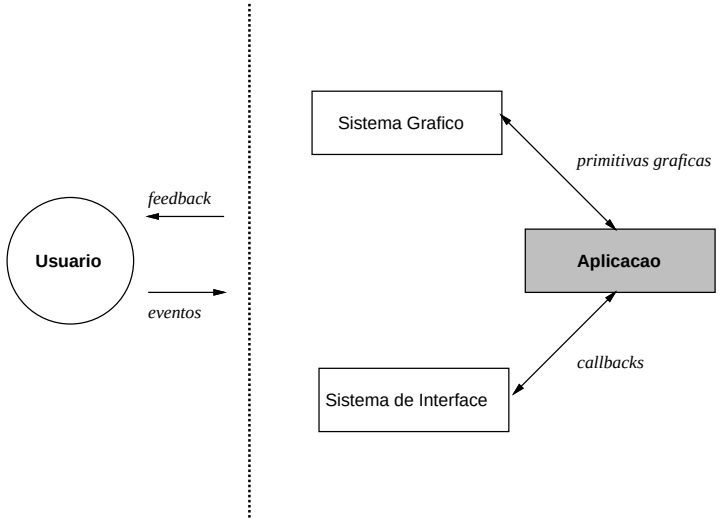


Figura 2.3: Esquema da composição de programas gráfico-interativos com ferramentas básicas

No intuito de facilitar a confecção de novos objetos e a integração dos sistemas gráfico e interativo, existem propostas que visam um melhor tratamento dos recursos desses sistemas. Tais bibliotecas de suporte podem ser compreendidas como abstrações de suas plataformas e seus usos podem ser esquematizados conforme a figura 2.4.

A manipulação é facilitada, como em [Car95, TeC96b], com bibliotecas de filtragem de eventos intermediários do *canvas*, aumentando o nível de abstração no tratamento

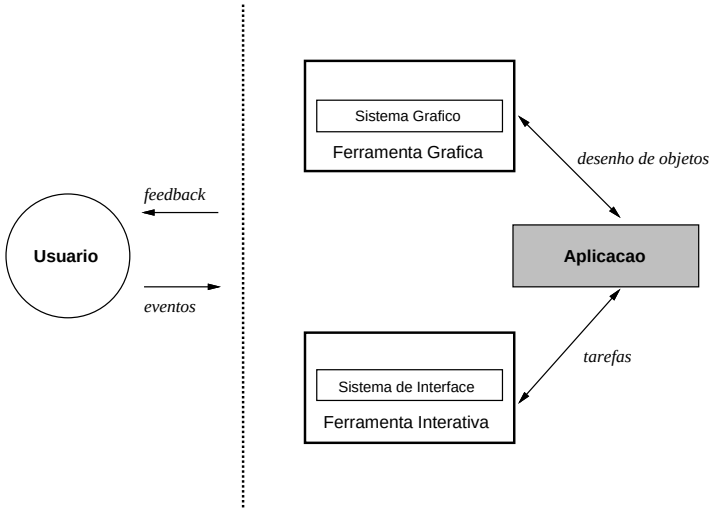


Figura 2.4: Esquema da composição de programas gráfico-interativos com ferramentas de suporte

das ações que o usuário pode realizar. Para tal, a aplicação passa a determinar as tarefas (interações) que o usuário pode fazer em um dado instante. Quando finalizada uma tarefa, a biblioteca retorna os atributos de suas realizações. Uma vez codificada uma tarefa, ela pode ser reutilizada em diversas aplicações, bastando instanciá-la a cada necessidade.

Por outro lado, a representação é facilitada com ferramentas de auxílio para a construção de desenhos, como por exemplo [Rez95]. Neste aspecto, surgiram diversos mecanismos para tal facilitação: representações procedurais através de uma linguagem embutida (*embedded language*), composição via hierarquia de primitivas gráficas, importação de *metafiles* de editores gráficos, etc.

Concluindo, os programas gráfico-interativos normalmente trabalham com *widgets* e objetos gráficos particulares. Os *widgets* são tratados por sistemas de interface, enquanto objetos gráficos particulares da aplicação requerem, em geral, que o programador utilize outra biblioteca. Assim, os programas acabam por ter dois gerenciadores de objetos: um para os *widgets* e outro para suas entidades particulares. Neste aspecto, uma unificação desses gerenciadores pode trazer como benefício o tratamento não-diferenciado dos objetos. Por exemplo, os *widgets* podem ter operações normalmente aplicáveis a objetos particulares da aplicação e vice-versa.

2.2 Propostas de Integração das Ferramentas Básicas

Conforme visto anteriormente, é comum construir programas gráficos interativos sobre dois sistemas: uma biblioteca gráfica e outra de interface. A não integração destes sistemas pode fazer com que o comportamento dos objetos visuais interativos da aplicação seja diferenciado. Por exemplo, os mecanismos de descrição abstrata de *layout*, como *hboxes* e *vboxes* do IUP [LdFG+96], são tipicamente associados a elementos de interface. Por outro lado, as interações de manipulação direta, como seleção e rotação, são

aplicadas normalmente a objetos particulares pela aplicação.

No entanto, existem algumas propostas que permitem a utilização de qualquer tipo de objeto sob uma mesma plataforma, ou seja, sem distinção de sua natureza. Tais sistemas não distinguem a funcionalidade dos objetos, definindo operações que não privilegiam determinados tipos de objetos visuais. Como exemplos desses tipos de sistemas, pode-se citar SUIT e FRESCO.

2.2.1 SUIT

Normalmente, as ferramentas de interface com o usuário oferecem uma variedade de recursos. Todavia, o bom uso dessas ferramentas, segundo [PCD92], necessita que o programador seja um "expert", isto é, que domine não somente a linguagem de programação envolvida mas também as características particulares da ferramenta.

Com a intenção de oferecer uma ferramenta que permita a confecção rápida de interfaces, foi desenvolvido o SUIT [CPP92] — The Simple User Interface Toolkit — que é uma biblioteca C cuja função é oferecer um conjunto de elementos de interface que possa ser manipulado de forma interativa.

O SUIT adota um modelo integrado para a confecção de interfaces, ou seja, ele não oferece uma ferramenta para a descrição da interface e outra biblioteca para a sua manipulação. O modelo SUIT é melhor entendido como um banco de dados que mantém informações a respeito dos "objetos visíveis na tela". Quando uma aplicação é finalizada, o SUIT armazena os atributos dos *widgets* e os recupera em uma execução posterior. Isso significa que o SUIT fornece mecanismos de persistência para os objetos e seus atributos, podendo ambos ser criados, deletados, alterados e, posteriormente, resgatados.

Um exemplo é ilustrado no trecho de código abaixo, onde o programador define uma aplicação com um só botão, que está amarrado a uma *callback* (`Hi`). O botão, a princípio, não tem localização e pode aparecer em qualquer ponto do diálogo. Enquanto a aplicação roda, o usuário pode alterar interativamente o posicionamento do botão até que, uma vez terminado o programa, os atributos do botão são gravados em disco. Na execução seguinte, a função `SUIT_createButton` cria o *widget*, examina o banco de dados e reconstrói os atributos do objeto.

```
void Hi( SUIT_object obj )
{
   printf("Hello, world");
}

SUIT_createButton("Hello", Hi );
```

Os atributos dos objetos são determinados por triplas no formato [`nome`, `tipo`, `valor`]. Desta forma, o estado dos objetos é determinado por uma lista destas triplas, ou seja, a sua *property list*.

Uma vez que o SUIT é entendido como mantenedor de uma lista de objetos e suas respectivas propriedades, ele oferece ferramentas interativas de comandos do usuário sobre os seus *widgets*. Gerado um evento, determina-se para que objeto essa entrada

deve ser enviada. Verifica-se, então, o estado das teclas *shift* e *control*. Caso ambas estejam presionadas, o SUIT interpreta eventos de mouse como tentativas de mudança de posicionamento e tamanho. Caso contrário, tem-se eventos como *click* (pressionamento do botão). Esse mecanismo faz com que não haja noção dos estados da aplicação, *run mode* e *build mode*.

Cada objeto pertence a uma determinanda classe, de modo que o SUIT oferece um mecanismo de herança para os atributos de objetos. Os valores desses atributos podem ser definidos no próprio objeto, por sua classe ou globalmente. Desta forma, quando um programa requisita uma propriedade de um *widget*, o sistema procura na sua *property list*. Caso não haja a tripla correspondente, procura-se na lista da classe. Não exitindo a tripla na classe, o SUIT a procura em uma *global property list*, que pode conter a propriedade requisitada ou gerar um valor *default*.

Uma forma interessante de alterar e visualizar os atributos dos objetos é através do editor de propriedades (*property editor*), cuja ilustração pode ser vista na figura 2.5. Invocando este editor, o usuário pode alterar os atributos de seus objetos durante a execução normal do programa. Na figura, existem três listas de atributos que correspondem às propriedades do objeto, da classe e globais. O usuário escolhe o atributo a ser alterado e, de acordo com o tipo da propriedade, é mostrado um conjunto de *widgets* que determina o seu valor. Por exemplo, um atributo booleano que corresponde a um *toggle*, um atributo definido por uma enumeração que corresponde a um conjunto de *radio-buttons*, etc.

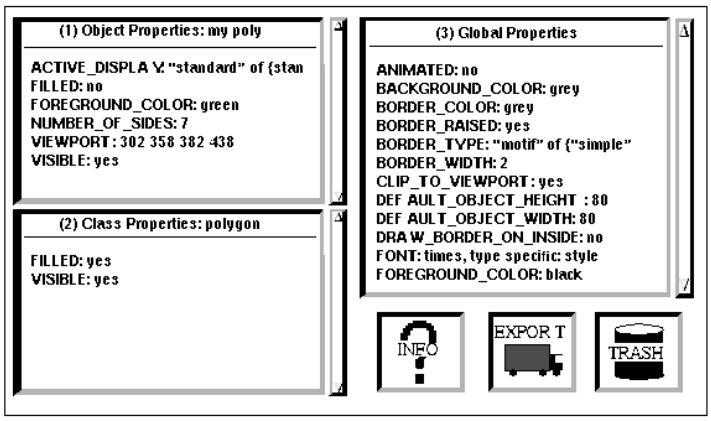


Figura 2.5: Ilustração do editor de propriedades SUIT

Os atributos podem também ser removidos e copiados nas três listas de propriedades, além de exportados. O mecanismo de exportação faz com que os atributos dos objetos possam ser definidos pelo estado de outros *widgets*. Por exemplo, ao exportar o atributo `frame` de um botão, aparece um *toggle* em um diálogo. Quando do acionamento deste *toggle*, o botão passa a ter ou não uma moldura.

Existem *widgets* especiais para a composição do *layout* de um diálogo. Como exemplos temos: o *stacker* que funciona como o `hbox` e `vbox` do TEX, o *pull-down*, que esconde seus filhos até ser pressionado, etc. A composição desses elementos também pode ser feita interativamente, ou seja, através de arrasto (*drag-and-drop*).

No SUIT, os objetos representam tipos abstratos de dados que podem ter diferentes

representações, denominadas *display styles*. Esses estilos podem ser simples ou associados a um conjunto de objetos SUIT. Neste último caso, os objetos desse conjunto são denominados *employees* de um determinado *display style*. A figura 2.6 ilustra este caso. O marcador de velocidades e o termômetro são *display styles* simples pois não possuem *employees*, enquanto que a *scrollbar* apresenta três: duas setas e o elevador.

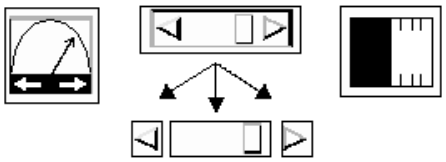


Figura 2.6: Ilustração de *employees* do SUIT

O SUIT apresenta formas no tratamento de *widgets* que não costumam ser oferecidas na maioria dos sistemas. Como exemplo podemos citar: manipulação direta em tempo de execução, alteração dinâmica de suas propriedades, composição por *employees*, etc. Notadamente, esses recursos são tipicamente oferecidos por aplicações finais somente aos seus objetos específicos, não a *widgets* de interface.

Os recursos apresentados no SUIT são bastante interessantes e podem também ser atingidos com a não diferenciação de tipos de objetos, através da definição de um único tipo de objeto: *objetos visuais interativos*, que são gerenciados por um único *framework*.

2.2.2 FRESCO

FRESCO [Chu94] é um sistema de interfaces evoluído do *toolkit* InterViews [MAL88] para o desenvolvimento de aplicações em ambientes de janelas. Sua arquitetura é baseada na filosofia de orientação por objetos e integra objetos que tradicionalmente não são tratados desta forma — objetos gráficos[2] e *widgets* de interface, assim como objetos de composição de *layout*.

A construção de programas é feita com o uso do tipo básico do FRESCO, denominado *glyph*. Existem diversos tipos de *glyphs*: que se desenham, que colocam *frames* em outros objetos, que compõem *layout*, aplicam transformações, etc. Um tipo especial de *glyph* é denominado *viewer*, que trata eventos do usuário como *widgets*, diálogos, editores de desenho e outros.

A ilustração 2.7 mostra um exemplo do uso de diferentes *glyphs*. Nele, dois *character-glyphs* ('G' e 'g') e um *space-glyph* estão compostos em um *hbox-glyph* juntamente com um objeto gráfico (o carro). Como o automóvel também é um *glyph*, o algoritmo de posicionamento do *hbox* trata sem distinção os objetos, colocando-os na localização devida.

O objeto "carro", por sua vez, também pode ser composto por diversos outros *graphical-glyphs* como linhas, polígonos, etc. Qualquer *glyph* pode receber transformações lineares de forma que, se o "carro" for modificado, o algoritmo de alinhamento

[2]Segundo [Chu94], objetos gráficos são aqueles que podem sofrer transformações gráficas.

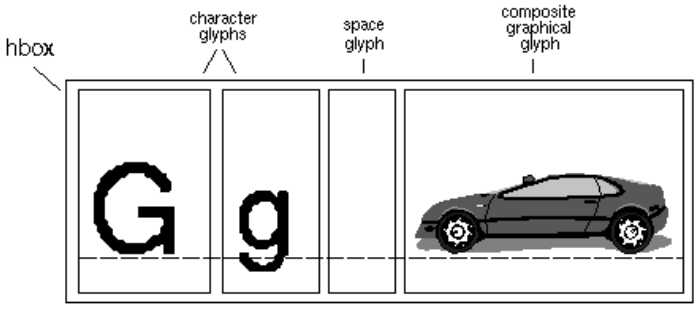


Figura 2.7: Exemplo da composição de glyphs em FRESCO

horizontal automaticamente rearruma o posicionamento dos objetos. A estrutura do exemplo ilustrado pode ser vista na figura 2.8.

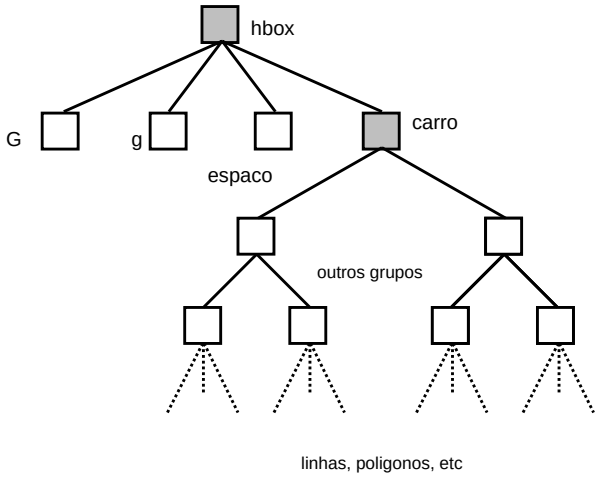


Figura 2.8: Estrutura da composição de glyphs no FRESCO

A mistura na composição de *glyphs* não é limitada apenas a *layout-glyphs*. Os *viewers*, que tratam a entrada de dados, também podem ser misturados e formar uma composição com textos, círculos, etc. A figura 2.9 ilustra esse importante recurso da arquitetura FRESCO, onde aparece a tela do editor Fdraw. Neste programa, o usuário pode criar *graphical-glyphs*, ou seja, linhas e retângulos assim como *layout-glyphs* (*hboxes* e *vboxes*). Além da criação de polígonos e linhas, a figura 2.9 mostra o Fdraw manipulando instâncias de si mesmo. O usuário pode criar e manipular objetos "Fdraw" dentro do próprio editor, pois o programa em si também é um *glyph*. Desta forma, essas instâncias podem ser rotacionadas e transladadas como qualquer outro objeto. Cabe também resaltar que os novos objetos "Fdraw" são funcionais, isto é, podem ser utilizados na instanciação de novos *glyphs*.

O FRESCO inclui também recursos de distribuição e *embedding*. Os recursos de compatilhamento de objetos decorrem do fato de que programas inteiros podem ser tratados como *glyphs*. Por exemplo, um editor de textos pode tratar tanto *glyphs* particulares de seu funcionamento (letras e parágrafos) como um *viewer* que representa uma planilha eletrônica. Como essa planilha é representada como um *glyph*, e os *glyphs* podem ser

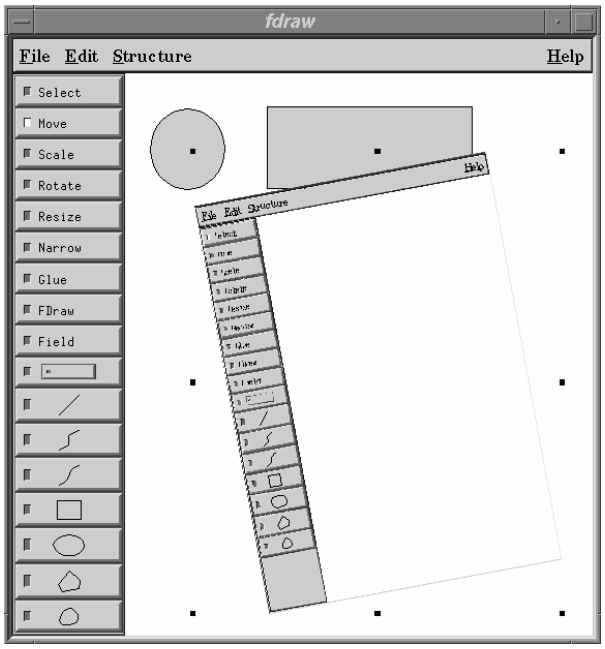


Figura 2.9: Tela do Fdraw — um editor gráfico FRESCO

misturados arbitrariamente, as transformações lineares aplicadas sobre o texto também são aplicadas sobre a planilha que foi incorporada. Apesar de ser algo normalmente esperado pelos usuários, poucos editores oferecem esses recursos, tratando os objetos incorporados como "retângulos opacos e não manipuláveis" [Chu94].

Por ser *OMG CORBA compliant* [Obj93], o FRESCO oferece recursos para a distribuição de objetos. Deste modo, os objetos são capazes de se comunicar através de diversos espaços de endereçamentos, isto é, pela rede. Assim, o programador utiliza, para a confecção de objetos, a linguagem IDL (*Interface Definition Language*) para fazer uma interface de alto nível do objeto. Posteriormente, tal objeto pode ser implementado em C, C++, Smalltalk e outras linguagens.

Outra forma de tratar objetos é através do Dish [PL94], um interpretador de *scripts* Tcl [Ous94]. Como a linguagem Tcl é interpretada, é possivel confeccionar protótipos e programas mais rapidamente.

2.3 O TeCGraf

O TeCGraf (Grupo de Tecnologia em Computação Gráfica da PUC-Rio) é um projeto de pesquisa e desenvolvimento em computação gráfica que mantém convênios com diversos parceiros industriais. As aplicações desenvolvidas por esse grupo têm necessitado cada vez mais de um suporte apropriado para interfaces gráficas interativas. Entretanto, os serviços oferecidos pelos *toolkits* tradicionais não têm maiores compromissos com a portabilidade, requisito fundamental para o grupo.

A necessidade da programação de sistemas gráfico-interativos de maneira simples e portátil levou o TeCGraf a desenvolver ferramentas que possibilitam criar sistemas multiplataformas. Estes, ao serem empregados nos mais diversos ambientes, se adaptam

como se tivessem sido desenvolvidos especialmente para eles. Como exemplo, podemos citar os sistemas gráficos GKS/puc, de larga utilização devido à sua grande portabilidade, e o CD (*Canvas Draw*) [TeC96c], substituto do GKS/puc, que tem a vantagem de tratar de imagens e ser mais simples de usar.

Também foram construídos sistemas de interface, dentre os quais destaca-se o IUP [LdFG+96] (Interface com o Usuário Portátil). O IUP oferece uma solução bastante atraente para a utilização dos *widgets* mais comumente empregados. Seu compromisso com a portabilidade tem assegurado o desenvolvimento de programas que rodam, sem alterações, nas plataformas XWindows/Motif, Microsoft Windows, MS-DOS e Macintosh. Além disso, os aplicativos tornam-se bem mais fáceis de serem mantidos, pois o IUP define uma abstração muito simples, porém abrangente, dos elementos de interface.

No entanto, o IUP estava tornando-se custoso quanto à incorporação de novos elementos. Para a construção de novos objetos não nativos do IUP, o programador necessitava de bibliotecas gráficas como o CD sobre um *canvas* IUP. Este é o caso de *widgets* mais complexos como `matrix`, `dial` e `gauge`. Essa solução, apesar de bem sucedida, não permitia a utilização desses objetos sobre o mesmo paradigma dos elementos nativos IUP, dificultando seu uso em novas aplicações.

Uma proposta de reestruturação do IUP foi apresentada em [Cer95]. Foi mostrada a viabilidade da inclusão de novos elementos, através da utilização de um *canvas* como gerador de eventos e superfície de visualização do novo *widget*. Essa solução deu origem à CPI (*Control Program Interface*) [Cer96], que permite o desenvolvimento de novos controles e o seu tratamento sob o mesmo paradigma IUP.

Tal solução garante a extensibilidade dos objetos visuais do IUP. No entanto, essa mesma proposta verifica que os objetos não fazem o tratamento de certas operações como *drag*, rotação, seleção, etc. Desta forma, continua existindo uma separação entre os tipos de objetos — específicos da aplicação e de interface. Essa dicotomia tem reforçado a concepção de um *framework* que trate objetos visuais independentemente de seu tipo e que possa ser a base de um sistema de interface.

O desenvolvimento de ferramentas de suporte à manipulação direta e de integração dos recursos gráfico-interativos tem se mostrado fator importante para as futuras aplicações do TeCGraf. Por isso, o grupo vem trabalhando sobre este problema há algum tempo. Por não haver soluções concensuais, diversos autores experimentaram soluções variadas, com diferentes ênfases. De maneira geral, as técnicas adotadas para solucionar o problema de acoplamento e flexibilidade da biblioteca foram: o uso de linguagens orientadas por objetos e de liguagens de configuração.

No que diz respeito a essas ferramentas, as seções seguintes descrevem resumidamente algumas das propostas.

2.3.1 UAI

UAI [CCCI94, BCCI94] constituiu uma experiência de integração dos recursos gráfico-interativos com suporte ao conceito de objetos ativos. Esta biblioteca é um *framework* de suporte a objetos visuais acoplado a primitivas gráficas de alto nível. A ferramenta constitui-se de um conjunto de classes C++ que pode ser dividido em:

- um pequeno conjunto de classes básicas, que fazem a interface com o ambiente nativo. O sistema é todo construído sobre esta base, garantindo uma boa portabilidade pela facilidade como podem ser confeccionados novos *drivers*;

- um conjunto de classes que suporta operações gráficas diversas. Cada classe implementa uma operação específica, como *clipping*, transformações geométricas (rotação, escala etc), *patterns* etc;

- um conjunto de classes para suporte a "objetos visuais", ou seja, objetos que têm uma representação visual. Essas classes facilitam a implementação de operações como manipulação direta dos objetos, agrupamento, etc. e podem constituir a base para um sistema de interface.

A UAI adota um modelo de orientação a eventos onde, do ponto de vista da aplicação, inexiste *loop* de eventos e programa principal (função `main`). Assim, a aplicação perde o conceito de "modo", não tendo controle sobre a ação executada a cada momento e necessitando de um mecanismo de temporização. O temporizador possibilita a geração de eventos em intervalos regulares de tempo, permitindo que uma aplicação possa realizar tarefas de forma assíncrona com os eventos de interface.

Como em aplicações UAI não há um programa principal, são necessários construtores globais para instanciar objetos. Isto porque, no início do programa, ainda não existem objetos, e a aplicação estaria impedida de se inicializar. Para contornar esse problema, uma aplicação deve declarar um objeto global que a representa. O construtor desse objeto chama um método definido pela aplicação, de modo que esta possa fazer suas inicializações.

Uma vez inicializada uma aplicação UAI, ela passa a criar objetos e inseri-los na interface do programa. A partir daí, os objetos são capazes de reagir a eventos, comportando-se como *Visual Objects* (VOs) que são posicionados em *Visual Spaces* (VSs). O conceito de VO modela os objetos gráficos ativos, oferecendo métodos de desenho e de resposta a ações do usuário. Por outro lado, o conceito de VS modela o espaço onde os VOs são posicionados.

Outros conceitos importantes foram derivados de VO e VS, tais como grupos e filtros. O conceito de filtro serve para modelar objetos que fazem interface entre um VS e um VO. Desta forma, ele repassa para o seu VO associado os eventos gerados em seu VS, podendo dar algum tratamento especial a esses eventos. A comunicação inversa também é válida, isto é, as requisições feitas por um VO a seu VS podem também ser filtradas por esse tipo de objeto, ou passadas adiante.

O conceito de grupo modela um tipo de objeto que contém uma lista de VOs e está também associado a um VS. Este VS o trata como se ele fosse um único VO, sinalizando os eventos do usuário. O grupo repassa os eventos para um subconjunto de seus VOs, selecionando quais os VOs que devem receber as mensagens. Da mesma forma que um filtro, um grupo é visto por seus VOs como se fosse um único VS, que atende todas as suas requisições.

Devido à natureza híbrida de filtros e grupos, esses objetos podem ser encadeados entre diversos VOs e VSs, compondo diversos níveis de filtragens. Esse encadeamento é ilustrado na figura 2.10.

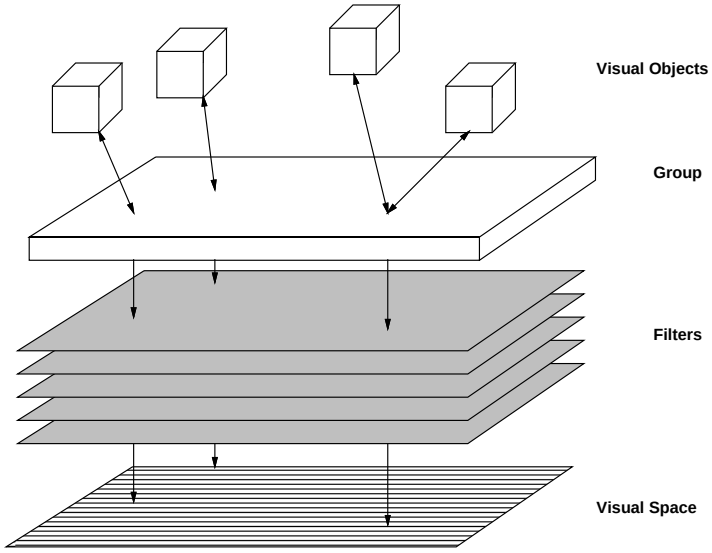


Figura 2.10: Esquema de encadeamento de VOs, VSs, grupos e filtros na UAI

A definição do "comportamento" dos objetos visuais mostrou-se tema bastante abrangente. Neste aspecto, a UAI oferece a possibilidade da customização, fora do *framework*, de seus objetos. Para isso, foi adotada uma linguagem embutida (*embebeded language*) onde o usuário pudesse definir seu objeto. A linguagem Lua [IdFF96] foi escolhida por ser de sintaxe simples, de fácil entedimento e por ter o paradigma da orientação por objetos, que se mostrou bastante adequado a este tipo de problema. Assim, a UAI incorpora objetos visuais Lua, que são objetos visuais descritos em Lua que fazem do *framework* um servidor de eventos e de desenho, dando maior flexibilidade à ferramenta.

Para oferecer os serviços de um sistema gráfico, a UAI modela alguns conceitos necessários à criação e manipulação de recursos gráficos. O primeiro destes conceitos é o de "janela" (*window*), que modela a área sobre a qual são aplicadas as primitivas gráficas. Este conceito mapeia uma janela do sistema hospedeiro e a abstração da área de trabalho UAI, que oferece transformações de *pixels* para milímetros e vice-versa.

No entanto, o conceito de *window* não oferece flexibilidade a ponto de permitir que a aplicação trabalhe no universo que mais lhe convenha. Para eliminar essa deficiência, criou-se o conceito de *canvas*, que estende o conceito de *window* e possibilita a utilização de um sistema de coordenadas que se adapte melhor à aplicação. Deste modo, o *canvas* trata os conceitos de *world coordinates* e sua visualização por meio de uma "lente" (*viewport space*), conceitos já bastantes difundidos em computação gráfica [FvDFH91].

Para que as primitivas gráficas pudessem ser traçadas sobre a área de trabalho, fez-se necessária a definição de um novo conceito na UAI: o *pen*. Este conceito, de maneira genérica, pode ser encarado apenas como uma "especificação" das primitivas gráficas passíveis de serem aplicadas. Desta maneira, torna-se possível a criação de diversas implementações de *pens*, que podem ser agrupadas em duas categorias: os que efetivamente desenham sobre áreas de trabalho e os que recebem primitivas e as repassam, após aplicar operações específicas, a outros *pens*. Isto confere também aos *pens* a capacidade de atuarem como filtros.

Assim, pode-se combinar diversos *pens* sob a forma de uma *pipeline*, de forma a conseguir efeitos tais como: *clipping*, rotação, translação etc. É interessante observar que esses efeitos podem ser combinados através do encadeamento correto de *pens*. Um exemplo ilustrativo é mostrado a seguir, na figura 2.11.

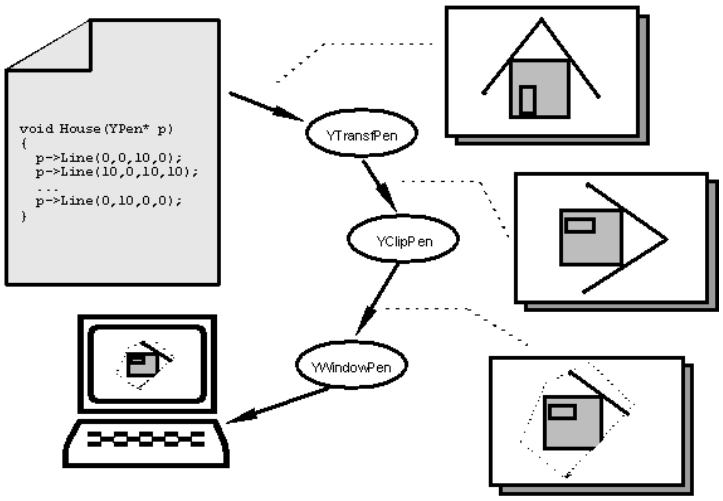


Figura 2.11: Esquema da composição de pens em UAI

2.3.2 TCHE

A utilização de Lua para a definição de comportamentos de objetos na UAI mostrou-se um recurso muito interessante. Desse interesse surgiu a idéia de construir um ambiente de manipulação de objetos visuais onde, além da definição de seus comportamentos e de suas representações visuais, possa-se alterar interativamente seus atributos em tempo de execução. A partir desta implementação foi desenvolvido o sistema TCHE [CI95], que propicia a edição dos objetos via *browsers* e a digitação de comandos Lua em um console.

Deste modo, o TCHE amplia a concepção UAI de extensibilidade de objetos visuais, pois, além dos objetos poderem ser programados fora da ferramenta, eles podem ser codificados dentro do próprio ambiente e automaticamente incorporados ao sistema. Neste sentido, foi feita uma experiência em edição de textos no TCHE, onde criou-se um objeto que representava um editor de textos — TEXT [Cli95]. Os arquivos criados no editor serviam como descrições de objetos que eram incorporados e manipulados na ferramenta.

Os novos objetos criados no ambiente TCHE, dos mais simples como botões até os mais complexos como um editor de textos completo, são tratados da mesma forma, isto é, podem ser manipulados e ter seus atributos alterados de forma interativa (figura 2.12). A alteração interativa desses atributos não traz como conseqüência novos esforços de codificação para o programador do objeto. Isto vem do fato de que o TCHE utiliza um *browser* capaz de tratar qualquer objeto. Esta construção altamente genérica foi possível

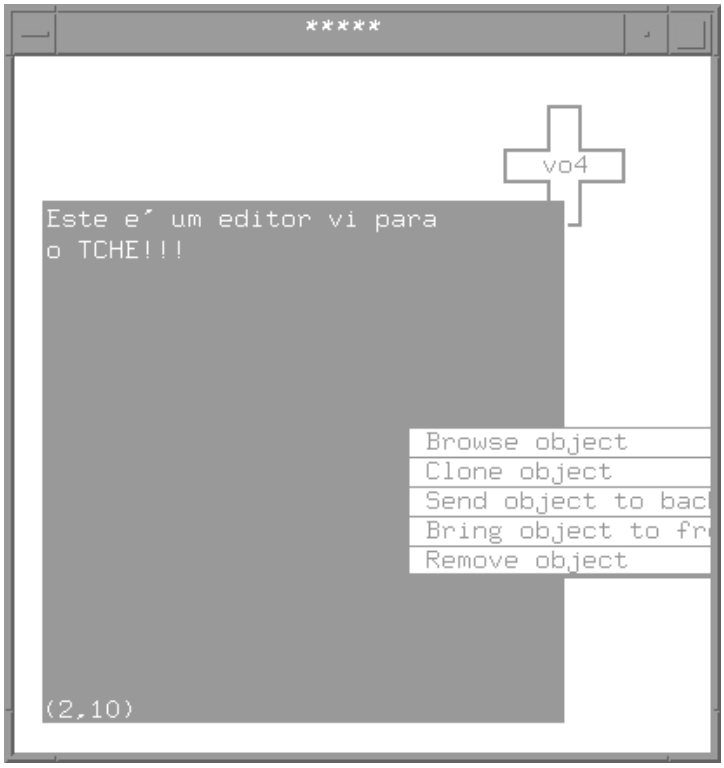


Figura 2.12: Incorporação de um objeto editor de textos em TCHE

graças aos mecanismos reflexivos de Lua, que têm se mostrado, em diversas aplicações, um recurso bastante poderoso.

Outro recurso interessante do TCHE foi a incorporação de mecanismos gráficos de *double buffering*, que suavizam muito o *feedback* com o usuário do sistema. Neste aspecto, o TCHE trata de maneira bastante elegante aspectos de *feedback* e redesenho do *canvas*, não necessitando de desenhos em *xor* quando da manipulação de objetos.

2.3.3 Interact

Tendo em vista a necessidade de sofisticação para a implementação de interfaces com manipulação direta em programas IUP/CD, foi desenvolvido o Interact [Car95]. Esta ferramenta propõe um modelo que define um conjunto genérico de tarefas de interação, onde apenas o resultado final das interações é passado para a aplicação. Todos os estados transientes são tratados somente pela ferramenta de interação, aumentando o nível de abstração do programador na gerência dos eventos de *mouse*/teclado e *feedbacks* com o usuário.

O modelo de interação propõe que um programa de aplicação possa especificar uma tarefa a ser executada e só se preocupar com o seu resultado. Todos os eventos intermediários do *canvas* IUP que ocorram durante a interação ficam a cargo do Interact. Assim, uma tarefa de interação, ao ser associada a um *canvas*, torna-se ativa e passa a receber todos os eventos de *mouse* e teclado. Um esquema da composição de programas Interact é ilustrado na figura 2.13

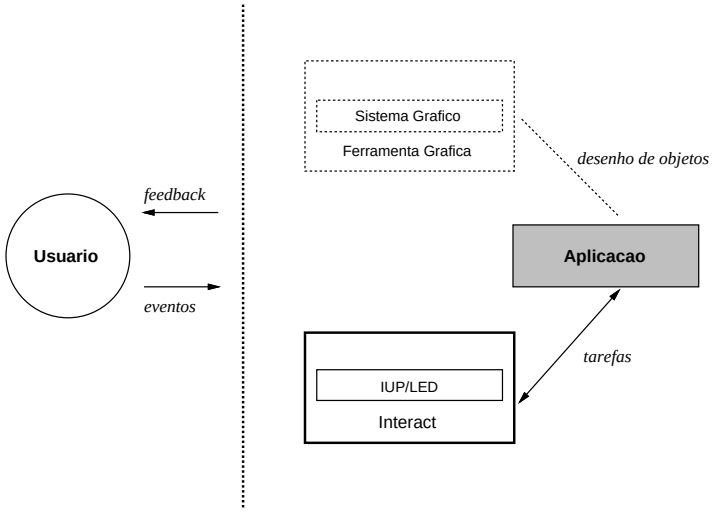


Figura 2.13: Esquema da composição de programas gráfico-interativos com o Interact

Para proporcionar uma maior flexibilidade, o modelo pressupõe que a aplicação possa suspender temporariamente a execução de uma tarefa, associando outra ao *canvas* e retornando à anterior assim que a última terminar. Por exemplo, a tarefa de criação de uma *polyline* pode ser interrompida enquanto o usuário navega pelo desenho, tornando visível a posição do novo ponto a ser inserido na *polyline*. Logo, cada *canvas*, em um dado momento, só pode estar executando uma tarefa de interação por vez [Car95, pág.29]. Nada, entretanto, impede de se ter diversas tarefas suspensas em um *canvas*, prontas para ganharem execução do ponto onde foram suspensas.

A implementação do modelo utiliza os conceitos de programação orientada por objetos. Neste aspecto, o Interact define duas classes principais:

- *canvas* — é a abstração do conceito de área de desenho, onde a aplicação exibe seus objetos. O canvas é responsável pela captura e tratamento de todos os eventos gerados pelo sistema de interface, tais como movimento do cursor, pressionamento de teclas (*mouse* e teclado), etc;

- *task* — é a abstração das tarefas de interação, tais como captura de uma linha, posicionamento, seleção e transformação de objetos, controle do nível de *zoom*, etc.

O Interact implementa a maior parte das tarefas mais comumente utilizadas. Apesar disso, o modelo define formas através das quais o programador pode criar suas próprias tarefas específicas. Este recurso importante vem do fato do Interact utilizar, em sua codificação, mecanismos de orientação por objetos, que fornecem mecanismos de herança e *late-binding*.

As classes que definem os *tasks* possuem grande parte de seus métodos virtuais, de modo que seu comportamento padrão pode ser alterado. Como a maioria das classes do Interact são abstratas, uma aplicação normalmente utiliza essa ferramenta derivando suas classes e implementando os métodos que estão faltando ou precisam ser redefinidos.

O Interact classifica suas tarefas em três categorias:

- tarefas de *construção*, que permitem a criação de novos objetos (linhas, polígonos, etc);
- tarefas de *seleção e transformação*, que alteram propriedades de objetos já existentes;
- tarefas de *visualização*, que controlam a área visível no *canvas*.

Abordando as formas de interação mais comuns, o Interact permite o desenvolvimento de aplicações gráfico-interativas com um melhor grau de reuso. Como exemplo, pode-se citar seu uso no programa SigDraw [TeC96d], um editor gráfico para gerenciamento de energia em sistemas elétricos de potência.

2.3.4 GLB

A GLB [Rez95] é uma biblioteca de classes C++ para a definição de objetos gráficos de modo hierárquico. Inicialmente, a ferramenta foi construída sobre o sistema gráfico GKS/puc e evoluiu, mais tarde, para o CD. Foi adotada uma linguagem embutida denominada Lurdes[3], que permite o tratamento desses objetos fora do ambiente C++.

Cada objeto gráfico GLB pode ser desenhado em qualquer superfície de visualização controlada pelo sistema gráfico. Os objetos recebem dados da aplicação que determinam a alteração de seus atributos ou a criação de novos elementos. Esta manipulação de objetos pode ser feita em Lurdes ou C++, através de métodos destas entidades.

Existem dois modos de trabalho na GLB: *desenho corrente* e de *tratamento de itens*. No primeiro tipo, a biblioteca mantém os elementos que compõem o desenho. Esta manutenção é feita de forma que todos os itens gráficos criados são automaticamente inseridos em um desenho corrente, não trazendo para a aplicação preocupações quanto ao armazenamento e controle de ponteiros para esses novos objetos.

No segundo modo, a GLB não exerce nenhum controle sobre os itens que compõem o desenho, de modo que a aplicação trata individualmente cada item gráfico. Logo, quando é criado um objeto gráfico, o programador deve guardar a referência para este novo elemento, assim como saber desenhá-lo.

Os itens gráficos que compõem um desenho podem ser de três tipos básicos:

- *primitivas* — são itens monolíticos, isto é, são elementos atômicos que não podem ser divididos e servem para compor objetos mais complexos. Como exemplo, podemos citar: linhas, *boxes*, *splines*, textos, etc;
- *grupos* — formam conjuntos de outros itens, podendo inclusive conter outros grupos de modo a formar uma hierarquia;
- *clones* — são referências a outros itens. Estes "ponteiros" permitem que um mesmo item gráfico tenha diferentes matrizes de transformação, fornecendo diversas visões desse elemento com economia de memória.

No modo desenho corrente, o conceito de *drawing* representa um desenho na GLB que gerencia seus itens. *Drawings* oferecem os seguintes recursos:

[3]Lurdes é uma especialização da linguagem Lua.

- controle do desenho corrente;
- métodos para adição e remoção de itens gráficos;
- exibição do desenho em uma superfície de visualização;
- controle de seleção de desenhos. Neste caso, é interessante utilizar as tarefas de seleção do Interact para captura de área ou ponto;
- armazenamento e recuperação de desenhos no formato Lurdes.

Segundo a proposta GLB em [Rez95, pág.22], a configuração ideal para a construção de editores gráficos de domínios específicos é feita conforme a ilustração 2.14. Tal configuração é muito boa porque explora as capacidades de criação e manipulação de objetos gráficos da GLB e utiliza os recursos de manipulação direta do Interact e os serviços de interface do IUP. Com base na ilustração 2.14, pode-se admitir que, em geral, os programas que utilizem a GLB são compostos conforme a ilustração 2.15.

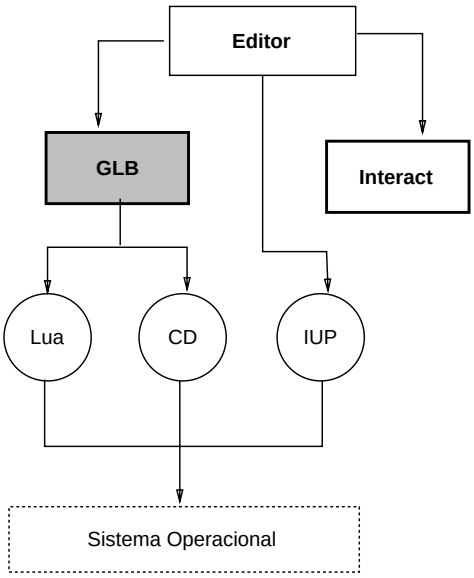


Figura 2.14: Configuração ideal para a construção de um editor gráfico usando a GLB

A GLB, por oferecer recursos para a representação de objetos, tem sido utilizada em diversas aplicações do TeCGraf. Como exemplo, podemos citar o PETROX, um editor de simulação de processos químicos, e o TeCDraw, um editor gráfico [TeC94].

2.3.5 EDG

A confecção de ferramentas gráficas e de interação tem facilitado muito a programação de sistemas no TeCGraf. Notadamente, destacam-se a GLB, por seu suporte à manipulação de objetos, o Interact, por seu sofisticado tratamento do *canvas*, e o IUP, por seu suporte aos *widgets* de interface.

No entanto, a crescente demanda por programas com suporte gráfico-interativo tem acarretado o aparecimento de programadores ocasionais, isto é, especializados na área de

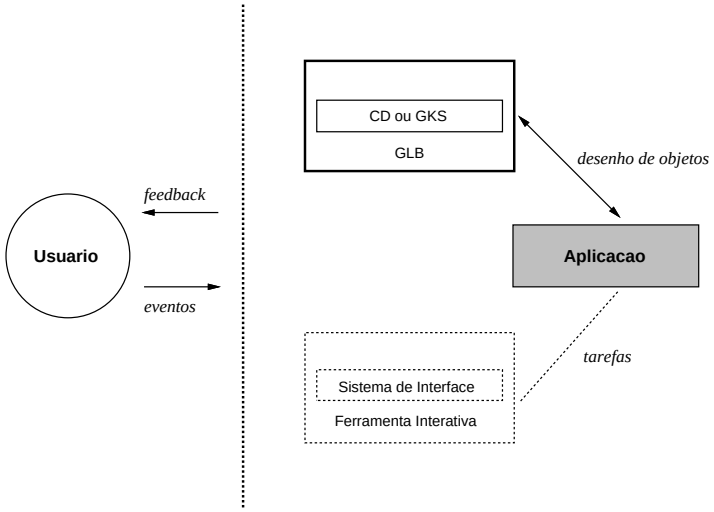


Figura 2.15: Esquema da composição de programas gráfico-interativos com a GLB

conhecimento em que atuam (Química, Engenharias, etc). Estes novos programadores precisam de um suporte gráfico poderoso, mas não podem lidar com *toolkits* de interface e bibliotecas gráficas, que são normalmente complexos e requerem um conhecimento mais específico de programação [FFG95].

No esforço de atender a estes programadores ocasionais, que não conhecem príncipios de Engenharia de Software, o TeCGraf desenvolveu um sistema de suporte ao desenvolvimento de programas gráfico-interativos com um elevado nível de abstração. O estudo dessa nova ferramenta deu origem ao sistema EDG [FFG95], que integra dois conjuntos de objetos:

- *elementos de interface* — implementam abstrações sobre objetos de interface do sistema IUP. Permitem a criação de diálogos através da composição de elementos como listas, botões, etc.

- *objetos gráficos* — definem um conjunto de abstrações das primitivas GLB como elipses, textos e linhas, que podem ser descritas e instanciadas diretamente por uma linguagem embutida ou por um editor de desenhos.

O EDG utiliza a linguagem embutida Lua para a descrição das interfaces e dos objetos gráficos da aplicação. A utilização desta linguagem foi motivada pelo fato de Lua ser procedural e contar com um poderoso mecanismo de descrição de dados, controle de fluxo, definição de funções, etc. Utilizando Lua, o programador descreve seu programa gráfico-interativo e conta ainda com os recursos inerentes à própria linguagem como: tratamento de I/O, facilidades na manipulação de *strings* e funções matemáticas, etc. Mesmo sendo bastante abrangente, Lua possui uma sintaxe simples e uma semântica de fácil compreensão.

A manipulação de primitivas gráficas em conjunto com *widgets* de interface resulta em importantes recursos para a associação de desenhos com capturas de dados. Isto tem se mostrado muito útil em aplicações de Engenharia, onde entidades de desenhos estão relacionadas com dados da aplicação. Por exemplo, pode-se associar uma cota de uma

planta com o valor desta propriedade na aplicação. Quando um *click* é feito sobre a cota, cria-se um diálogo que altera esse valor e seu respectivo desenho.

Um recurso extremamente relevante no tratamento dos objetos gráficos do EDG é a capacidade de confecção de objetos tanto por codificação Lua quanto por uso de um editor — o TeCDraw [TeC94]. A possibilidade do programador usar um editor gráfico para criar suas representações é uma característica importante do EDG, pois possibilita uma programação facilitada para hierarquização, criação e definição de detalhes gráficos.

O TeCDraw é um editor que possui os recursos normalmente encontrados neste tipo de programa, tais como desenho de linhas, polígonos, etc. Essas primitivas gráficas podem ser agrupadas e identificadas como *entidades*, sendo manipuladas posteriormente no ambiente EDG. Essa identificação é feita com a atribuição de um nome simbólico, que passa a representar um objeto gráfico que possui diversos métodos para manipulação (transformações, mudanças de cor, etc).

A comunicação entre o TeCDraw e o EDG é facilitada pelo uso da própria sintaxe Lua como formato do arquivo exportado pelo editor. Assim, carregar um objeto gráfico do TeCDraw significa simplesmente interpretar um arquivo Lua. Além disso, a geração de *metafiles* legíveis em sintaxe Lua permite que se modifique, via editores de texto convencionais, um objeto já criado. Uma ilustração da integração TeCDraw/EDG pode ser vista na figura 2.16. A representação criada é salva em um *metafile* na sintaxe Lua e associada ao elemento `estrela` (parte do arquivo gerado pode ser visto na figura 2.17).

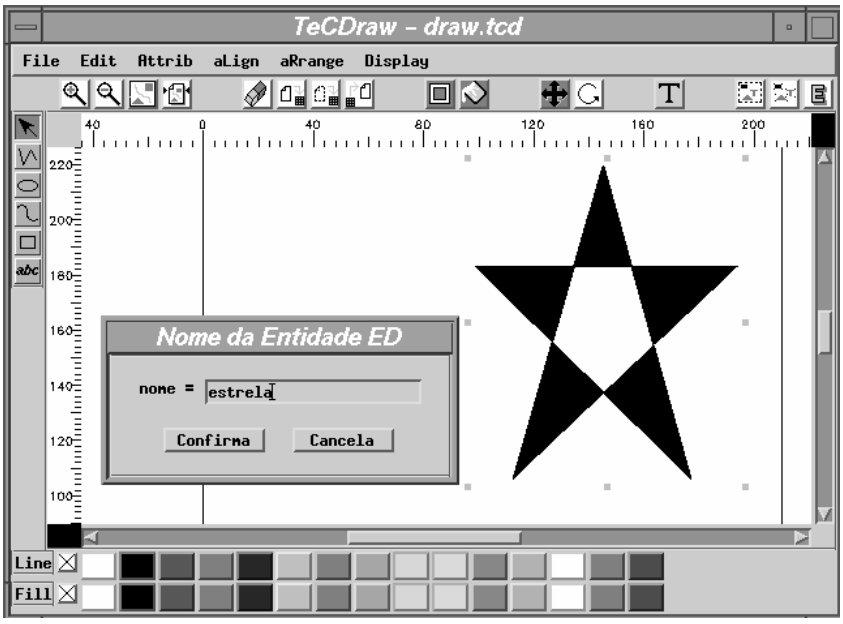


Figura 2.16: Criação de uma representação para um objeto usando o TeCDraw

A interação no EDG é feita através do tratamento de eventos sobre os objetos gráficos. Esses eventos são "filtrados" em seus detalhes (*button down*, *mouse motion*, *button up*) e passados ao programador em um nível abstrato, como *click* e *drag*. Sempre que um objeto for apontado ou arrastado, o sistema EDG executa um método correspondente do objeto gráfico. A combinação desses eventos permite ao programador a elaboração de tarefas mais complexas como transformações lineares, seleção, *reshape*, etc.

Os recursos de interação e representação de objetos no EDG possibilitam a confecção de objetos de interface próprios. Para isso, o programador deve construir a representação do novo *widget*, via código Lua ou TeCDraw, e determinar as *callbacks* que correspondem às ações do usuário. Este tipo de programação determina uma integração de *representação* e *comportamento* em um único tipo abstrato de dados, trazendo à tona o conceito de *objetos visuais interativos*.

No que diz respeito à codificação de novos objetos, o EDG permite que eles possam ser incorporados a qualquer outra aplicação construída sobre ele. Para tal, basta que os arquivos Lua contendo a representação e o comportamento do objeto sejam carregados pelo programa. Deste modo, pode-se incorporar objetos de diversas aplicações, bastando somente gravar e ler os arquivos Lua que os descrevem. Um esquema da incorporação de representações gráficas pode ser visto na figura 2.17.

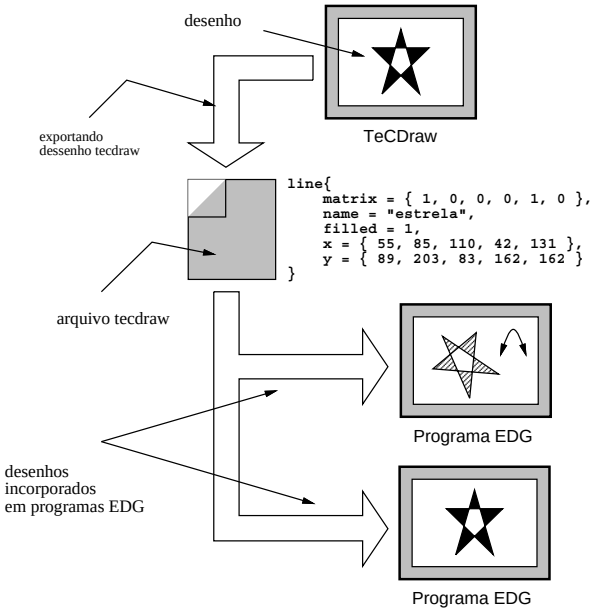


Figura 2.17: Esquema da incorporação de representações gráficas no EDG

Notadamente, o EDG oferece um suporte ao tratamento de objetos visuais interativos, integrando objetos de interface e particulares da aplicação. Estes serviços têm se mostrado tão úteis e práticos que diversas aplicações do TeCGraf utilizam essa ferramenta. Dentre inúmeros exemplos, podemos citar o PGM — Programa Gráfico Mestre [TeC93], um editor/visualizador de perfis verticais contínuos de acompanhamento geológico. Um esquema de uma aplicação EDG pode ser visualizado na figura 2.18.

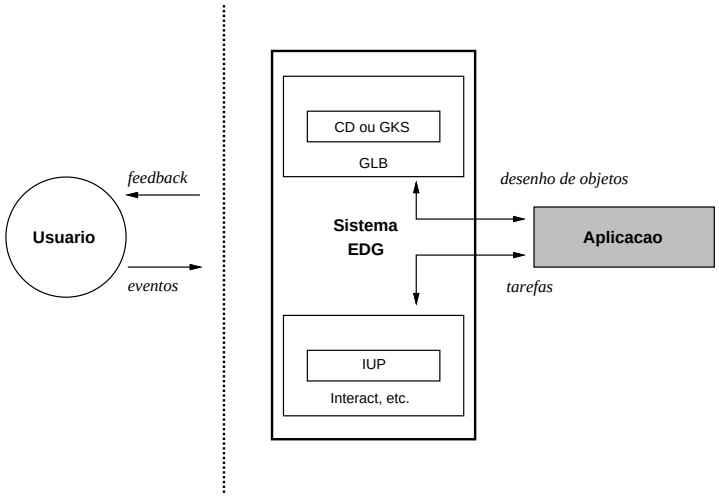


Figura 2.18: Esquema da composição de programas gráfico-interativos com o EDG

Capítulo 3

VIX, um Framework

Este capítulo tem por finalidade exibir a modelagem e a implementação de **VIX**. Neste sentido, a seção 3.1 define os conceitos envolvidos no modelo **VIX**. Este modelo utiliza a filosofia de orientação por objetos, que se repercute na adoção de uma linguagem compatível com esta filosofia.

No que diz respeito à implementação do *framework*, a seção 3.2 mostra como a modelagem proposta foi implementada. Neste aspecto, essa seção explica a implementação de **VIX** em C++ e Lua.

3.1 Modelo conceitual

O suporte para objetos visuais interativos é feito com o aproveitamento de diversos conceitos e recursos oriundos de outros sistemas. Notadamente, a modelagem conceitual deste *framework* é bastante baseada em UAI (seção 2.3.1), que utiliza recursos típicos de orientação por objetos. As melhorias a partir do modelo UAI resultam de um amadurecimento que seus autores reconheceram como importante. Esta evolução surgiu principalmente da necessidade de tornar a modelagem UAI mais prática, isto é, passível de ser utilizada como ferramenta para futuras aplicações.

Na UAI, a escolha do sistema gráfico-interativo básico implicou na confecção de *drivers* para as diversas saídas gráficas (*pens*) e na abstração dos variados sistemas de janelas (*windows*). No **VIX**, foram aproveitados os diferentes dispositivos de saída do CD [TeC96c], da mesma forma que a abstração de janelas foi feita com o uso do CD *standalone*. Assim, pôde-se concentrar um maior esforço na modelagem e implementação do *framework*, reaproveitando recursos de geração de eventos, janelas e primitivas gráficas dos sistemas nativos.

Sendo baseada em UAI, a modelagem **VIX** utiliza uma filosofia de orientação por objetos que se repercute na adoção de uma linguagem OO. Os recursos provenientes deste tipo de linguagem foram limitados somente a herança, polimorfismo e *late-binding*, de forma a não limitar a aplicabilidade da modelagem a uma linguagem específica.

O tratamento dos objetos visuais também necessitou do aproveitamento de recursos já difundidos em diversas ferramentas do TeCGraf. Em destaque, podemos citar o Interact, a GLB e o EDG. O Interact por seu tratamento sofisticado às manipulações que o usuário pode exercer, a GLB pelo seu suporte ao tratamento de objetos hierárquicos e o EDG

por sua integração de objetos gráficos e de interface sob a linguagem Lua.

A modelagem conceitual do **VIX** trata um conjunto de conceitos necessários para cobrir todas as funcionalidades propostas por esta dissertação, ou seja, o suporte a uma programação baseada em objetos visuais interativos. Procurou-se fazer com que este conjunto fosse o mais simples possível, facilitando sua aceitação.

Objetos Visuais — Visual Objects (VO)

Da mesma forma que em UAI, o **VIX** utiliza o termo *visual object*, ou simplesmente *VO*, para tratar objetos visuais. O conceito de VO, inspirado inicialmente em *Abstract Data Views* [CILS93], modela objetos gráficos ativos que oferecem métodos de desenho e de resposta a ações do usuário.

O tratamento desses objetos não é restrito somente aos objetos gráficos de uma aplicação. Pode-se entender como VO qualquer objeto que possua uma representação e um comportamento. A definição deste comportamento dá suporte à manipulação desses elementos, de forma que pode-se definir também *widgets* de interface e até mesmo elementos mais sofisticados como "painéis de controle" ou subprogramas. Por construção, estes elementos de interação possuem, além de versatilidade, a possibilidade de serem estendidos com facilidade por mecanismos de herança.

Diferentes tipos de VOs podem ser conectados ao *framework* devido à sua interface bem definida. A facilidade com que esses novos elementos podem ser inseridos vem do tamanho dessa interface, que é mínima para cobrir suas funcionalidades. A funcionalidade mínima de um VO é a sua capacidade de responder a determinadas ações e saber representar-se graficamente. Desta forma, sua interface contém somente ações de respostas ao usuário e de redesenho, ou seja, *feedback* de suas alterações. Estas capacidades podem ser visualizadas na figura 3.1.

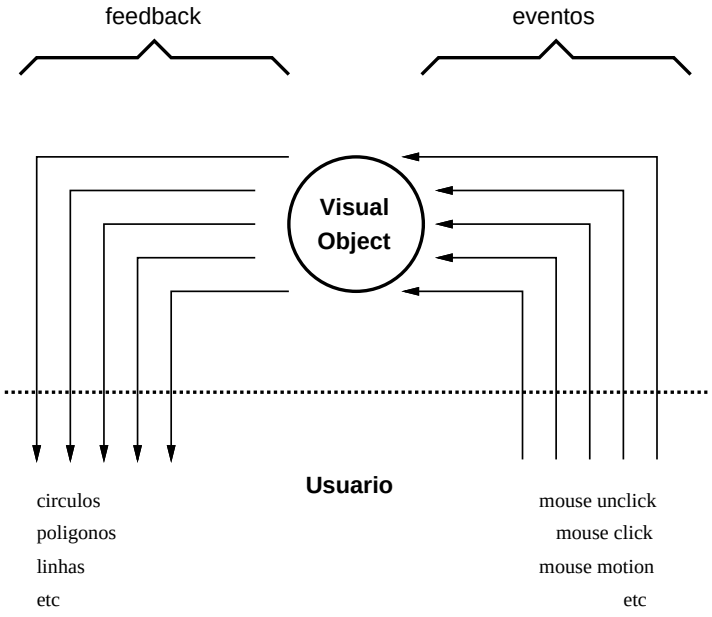


Figura 3.1: Esquema de um objeto visual (*visual object*)

A definição do comportamento de um VO pode ser facilitada por mecanismos de herança, característica intrínsica a linguagens orientadas por objetos. Deste modo, podem-se definir hierarquias de objetos visuais nas quais a aplicação pode reaproveitar a maior parte de um VO já definido. Essa característica faz com que o *framework* ofereça um modelo que suporte o desenvolvimento de objetos reutilizáveis.

Dado que um objeto visual é determinado por seu comportamento e representação, é necessária a existência de uma entidade que transmita os eventos do usuário e ofereça recursos para que o objeto seja desenhado. Em outras palavras, é preciso que haja uma entidade que transmita *inputs* do sistema nativo e ofereça mecanismos de aplicação de primitivas gráficas.

Espaços Visuais — Visual Spaces (VS)

O conceito de *visual space*, ou VS, mapeia entidades que transmitem eventos para VOs e fornecem uma superfície de visualização onde eles são posicionados. Os VSs devem ser encarados como elementos de ligação entre o usuário e o VO que ele está manipulando, de forma que o usuário enxerga e manipula seu objeto através de recursos do VS.

O conceito de *window* é um tipo de VS. *Windows* mapeam janelas do sistema nativo, repassando os eventos do usuário/sistema nativo e oferecendo uma superfície para aplicação de primitivas gráficas. Uma ilustração do relacionamento entre uma *window* e um VO pode ser vista na figura 3.2. Como *inputs*, uma *window* transmite eventos para o VO (*mouse clicks*, *button press*, etc). Como *output*, a *window* fornece a área sobre a qual agem as primitivas gráficas, ou seja, uma janela do sistema hospedeiro.

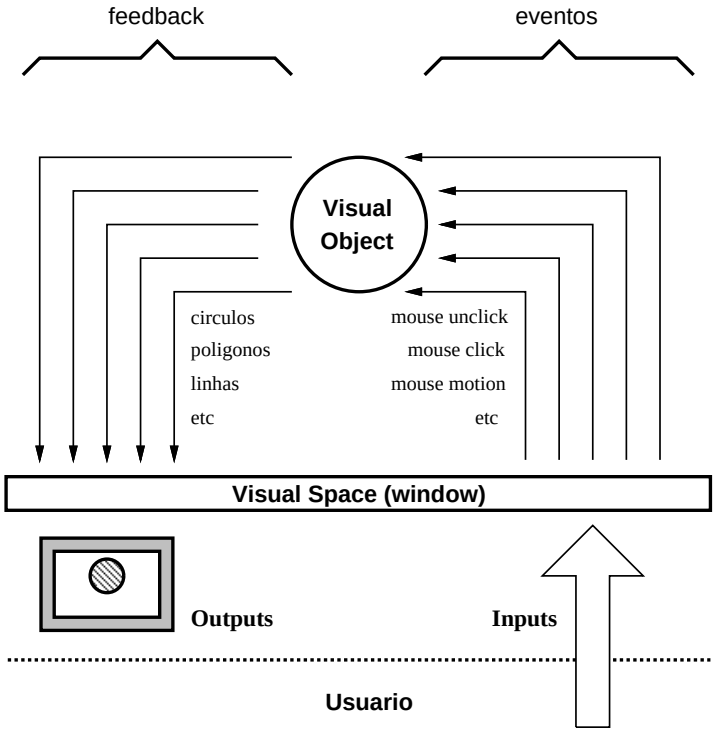


Figura 3.2: Esquema do relacionamento entre um objeto visual e uma janela

A comunicação entre o VS e seu VO pode ser exemplificada da seguinte forma: a cada evento gerado, o VS envia para o seu objeto essa informação. O VO realiza uma interpretação própria destes eventos, o que pode ocasionar alterações de suas propriedades (posição, cor, etc.). Depois de alteradas as propriedades do VO, ele requisita ao VS que invalide alguma região da superfície do espaço visual. A invalidação dessa região ocasiona a geração de um evento de *repaint* para o objeto. Este exemplo mostra os dois tipos básicos de comunicação VO↔VS: passagem de eventos VS→VO e invalidação de áreas VS←VO.

Conforme visto anteriormente, um *visual space* fornece mecanismos para que o objeto seja desenhado. O VS, conceitualmente, não serve para a aplicação das primitivas gráficas, mas oferece um *dispositivo* para esse desenho.

Dispositivos — Devices

Dispositivos são entidades que equivalem aos *drivers* do *framework*, sendo responsáveis pela implementação real dos métodos de desenho.

Quando um objeto precisa ser desenhado, basta aplicar seu "método" de redesenho em um dispositivo qualquer. No caso de um VO que está sendo manipulado em uma janela, o VS fornece um *window device*, que permite ao objeto ser redesenhado nesta janela.

Além do *window device*, existem outros tipos de dispositivos, que permitem a impressão e exportação da representação de um objeto. Como o *framework* foi implementado sobre o CD [TeC96c], tem-se a possibilidade de geração de diversas saídas gráficas. Cada uma das saídas CD é mapeada por um *device* **VIX**. Como exemplos de dispositivos, podemos citar: CGM (*Computer Graphics Metafile*), PostScript, janela nativa (*CD standalone canvas*), IUP *canvas*, etc. A figura 3.3 ilustra exportação de um VO em diversos *devices*.

Como um dispositivo é um mapeamento de um *driver* CD, os recursos oferecidos por *devices* são os mesmos do CD, acrescidos da incorporação de uma filosofia de orientação por objetos. Basicamente, cada dispositivo oferece recursos de:

- cerceamento ou *clipping*;
- aplicação de primitivas gráficas como: linhas, arcos, marcas, retângulos, setores circulares, textos, etc;
- controle de atributos de primitivas como: estilos de linhas, cores, tamanhos, espessuras, padrões de preenchimentos, etc;
- gerenciamento de cores;
- tratamento de imagens;
- transformação de coordenadas.

Com esta interface, os dispositivos oferecem todos os recursos pertencentes ao CD, que são suficientes para tratar a representação de objetos.

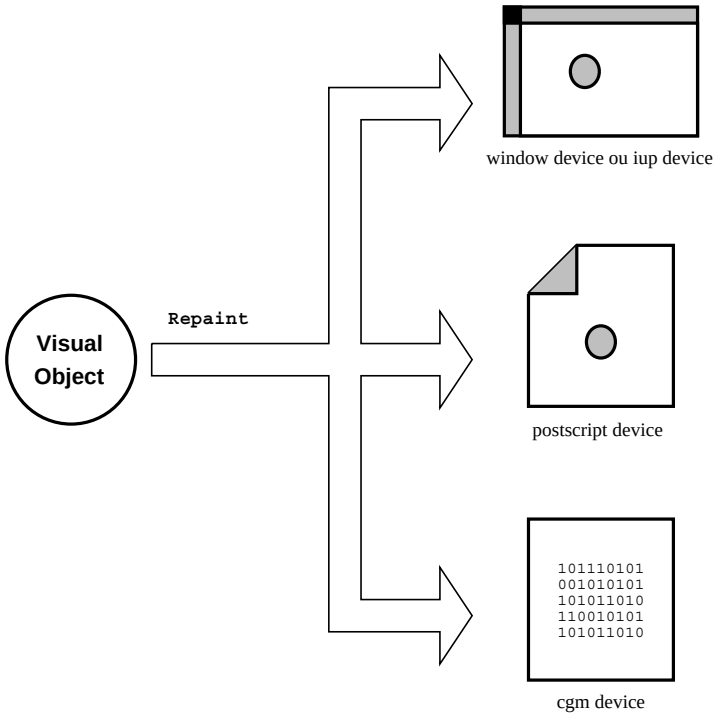


Figura 3.3: Esquema do desenho de um VO em diversos dispositivos

Filtros — Filters

Filtros são entidades derivadas dos conceitos de VO e VS, que servem para modelar objetos que fazem interface entre um espaço visual e um objeto. Um filtro repassa para o seu VO associado os eventos gerados em seu VS, podendo dar algum tratamento especial a esses eventos. A comunicação inversa também é válida, ou seja, as requisições feitas por um VO a seu VS podem também ser filtradas por esse tipo de objeto, ou passadas adiante.

Graças aos mecanismos de orientação por objetos, os filtros normalmente apenas repassam os eventos para o seu VO e as requisições para o seu VS, comportando-se como um filtro "transparente". No entanto, este filtro pode ser especializado para fazer aquilo que lhe compete, mexendo apenas na parte que diz respeito a esse novo tratamento.

A conexão entre VOs e VSs não está limitada a um par [VO,VS] nem restrita ao repasse de somente eventos nativos, conforme mostra a figura 3.4. Os filtros podem ser utilizados para multiplexar eventos entre diversos objetos e ampliar o comportamento dos objetos. Isto é feito por seus dois tipos básicos: *grupos* e *filtros de comunicação*.

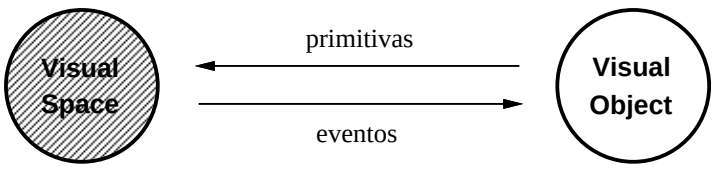


Figura 3.4: Esquema de um par [VO,VS]

O conceito de grupo modela um tipo de objeto que contém uma coleção de VOs e está associado a um único VS. Este VS trata o objeto visual como se ele fosse um único VO, sinalizando as ações do usuário. A função básica do grupo é repassar os eventos para seus VOs, selecionando quais deles são adequados para receber estas ações. Assim como um filtro, um grupo é visto por seus VOs associados como se fosse um VS, que atende todas as suas requisições.

Desta forma, pode-se ter diversos objetos visuais em um mesmo VS, bastando colocar um grupo entre entre o *visual space* e os diversos *visual objects*. Grupos armazenam e gerenciam uma lista de seus VOs-filho, o que pode ser visto na figura 3.5.

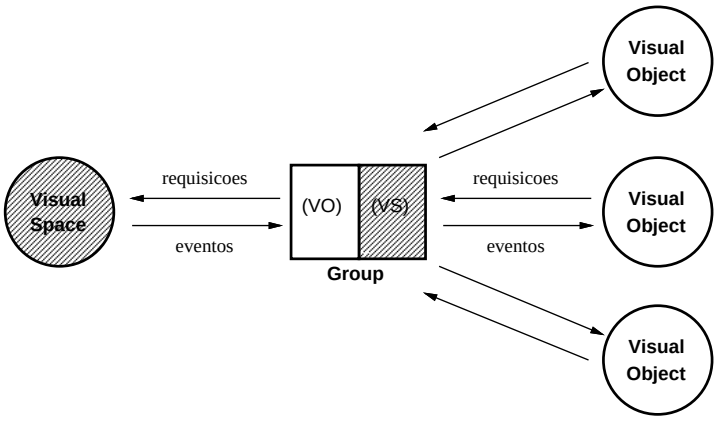


Figura 3.5: Esquema de um grupo com diversos VOs no **VIX**

Outro conceito introduzido pelo *framework* é o de filtro de comunicação ou simplesmente filtro. Com este recurso, é possível filtrar eventos e requisições da comunicação VO↔VS. Uma ilustração deste novo potencial pode ser vista na figura 3.6.

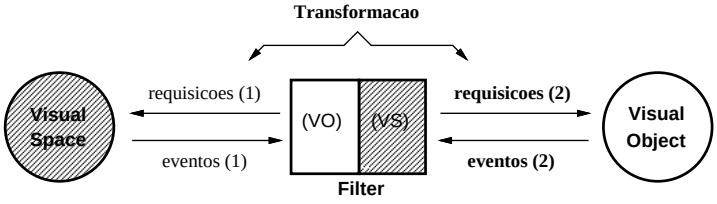


Figura 3.6: Esquema da utilização de filtro em **VIX**

Redespachando eventos do VS para o seu VO, os filtros permitem a expansibilidade das ações passíveis de aplicação nos objetos. Por exemplo, um filtro pode acumular eventos de *button press* e *mouse move* e enviar uma requisição de translação ao seu VO, caso o botão tenha sido pressionado sobre ele.

Desta forma, os filtros garantem a possibilidade de extensão dos comportamentos dos VOs, de forma que os objetos não tratam somente eventos básicos do sistema nativo. Os filtros podem detectar um conjunto de eventos, combiná-los e enviar "mensagens" para os VOs, caracterizando algum comportamento extendido do objeto.

Os filtros são VOs e VSs simultaneamente e compõem uma interface entre estas entidades. Assim, podem ser colocados inúmeros filtros entre VOs e VSs, compondo

uma *hierarquia dinâmica* de entidades **VIX**, conforme a figura 3.7. Com a definição desta hierarquia, pode-se, entre outras coisas:

- colocar vários filtros atuando simultaneamente, formando uma *pipeline* de filtros;
- criar vários grupos de objetos que contêm filtros diferentes, de forma a aplicar determinadas operações a somente um subconjunto dos VOs existentes.

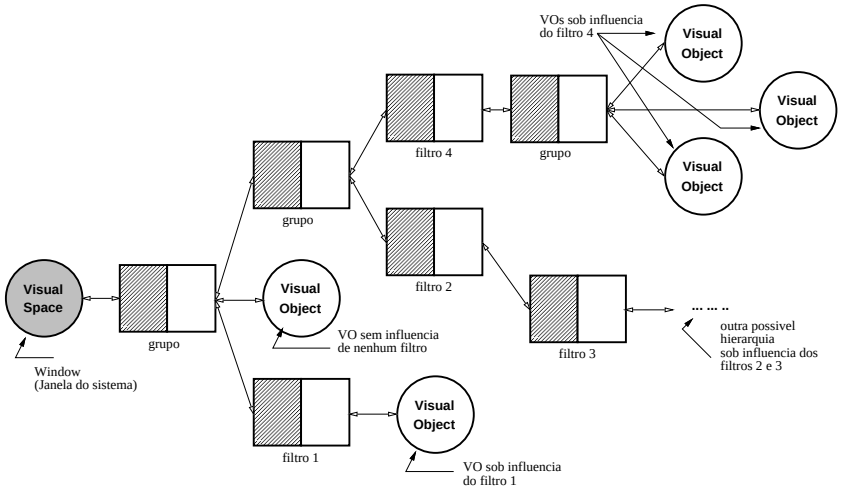


Figura 3.7: Exemplo de uma composição de hierarquia dinâmica em **VIX**

Ao estender o comportamento dos VOs, é necessário, a princípio, garantir que o objeto visual saiba responder às ações enviadas pelo seu VS. Este fato sugere que a interface deste VO tenha que conter "métodos" de resposta a esses novos eventos, o que pode ser resolvido com mecanismos de herança. Neste caso, o programador teria que criar um VO especializado que implementasse esse novo comportamento.

Esta solução não é adequada pois pode acarretar hierarquias de objetos extremamente complexas, à medida que novos filtros são criados. Isto porque os filtros só tratariam VOs específicos, que possuem em sua interface esse método de resposta. Para evitar esse problema, é necessário estabelecer de um novo conceito: *mensagens*. Através desse mecanismo, qualquer objeto básico **VIX** pode trocar dados com outro.

Mensagens — Messages

Mensagens são entidades **VIX** com a finalidade de definir um protocolo de transmissão de dados entre objetos do *framework*. Com este mecanismo, pode-se garantir a extensibilidade dos comportamentos dos objetos sem "poluir" suas interfaces com métodos específicos assumidos pelos filtros ou qualquer outra entidade **VIX**.

Uma mensagem, em seu sentido mais amplo, é um conceito que apenas define o protocolo de troca de dados. Logo, ela não sabe nada a respeito do conteúdo que envia nem sobre os dados que retorna. Quando o programador precisa enviar dados, ele cria um novo tipo de mensagem e a especializa. Essa especialização é feita com o uso de mecanismos de herança, acrescentando os dados a serem transmitidos e recebidos.

Enviada uma mensagem, o objeto a recebe no seu *tratador* (*message handler*), que consta de sua interface. Tal tratamento é feito utilizando-se aspectos polimórficos de mensagens, isto é, tratando uma mensagem genérica como se fosse do tipo esperado. Para fazer esta "conversão", o objeto precisa identificar o tipo da mensagem recebida, de modo que possa transformá-la corretamente. Isto é feito introduzindo-se, na interface das mensagens, funções que permitem esta correta identificação.

O mecanismo de mensagens foi construído de forma que VOs e VSs possam recebê-las. Assim, tanto *visual objects* como *visual spaces* possuem tratadores. Como qualquer objeto **VIX** pode receber mensagens, os tratadores podem ser de dois tipos: *VS handlers* e *VO handlers*.

Dado que entidades **VIX** estão sujeitas ao recebimento de mensagens, os filtros também podem recebê-las. Conforme visto anteriormente, estas entidades são VOs e VSs simultaneamente, logo possuem os dois tipos de tratadores. Por *default*, os filtros simplesmente repassam as mensagens para seus VSs ou VOs, atuando de forma análoga quando da transmissão de eventos em um filtro transparente.

A figura 3.8 ilustra o mecanismo de mensagens. Neste exemplo, um VS envia uma mensagem para o seu VO. Esta mensagem é repassada por filtros e grupos até atingir um objeto que a trata.

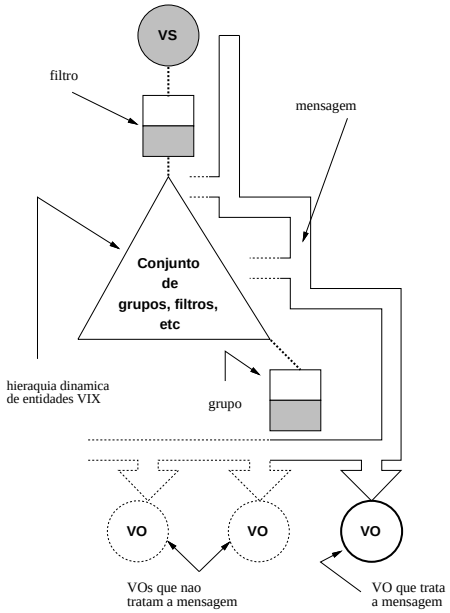


Figura 3.8: Exemplo do envio de mensagens em **VIX**

Temporizadores — Timers

Uma vez que o **VIX** adota um modelo de orientação a eventos, suas aplicações perdem o conceito de "modo", ou seja, não têm controle sobre qual ação será executada a cada momento. No sentido de permitir que os programas **VIX** possam definir eventos síncronos, foi definido o conceito de *temporizador*. Este mecanismo possibilita que um

determinado evento seja gerado em intervalos regulares de tempo, permitindo que uma aplicação possa realizar tarefas assincronamente com eventos de interface.

Os *timers* podem ser utilizados pela definição de três atributos: um intervalo de tempo, um *flag* de repetição e uma *callback*. O intervalo define os instantes em que o *timer* passa o controle para a aplicação. O *flag* indica se a passagem do controle para a aplicação deve ser cíclica ou não. No primeiro caso, o programa é chamado em intervalos regulares de tempo até a desativação do temporizador. Caso contrário, o timer chama o programa apenas uma única vez após transcorrido o intervalo. A *callback* é um método do *timer* que, redefinido pelo programador, define o que a aplicação deseja fazer no(s) instante(s) específico(s) em que recebe o controle. Uma ilustração do recurso de temporização pode ser visto na figura 3.9.

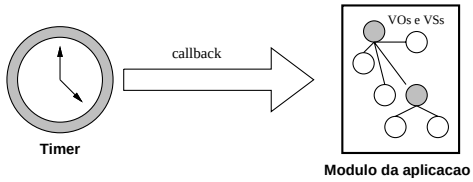


Figura 3.9: Esquema da utilização de um temporizador **VIX**

3.2 Arquitetura do sistema

A implementação da modelagem descrita anteriormente necessita de uma linguagem de programação que suporte recursos de orientação por objetos. Como o modelo **VIX** se limitou somente à utilização de recursos de herança, polimorfismo e *late-binding*, a aplicabilidade do modelo necessita apenas de uma LOO que tenha estes recursos[1].

Neste trabalho, foi adotada a linguagem C++ [Str92] porque, embora sujeita a algumas críticas [Wal93], é mais utilizada na construção de sistemas. Seu maior uso se deve às suas variadas implementações em diversos compiladores e plataformas.

Além da utilização de C++, a implementação do *framework* conta com o uso de uma linguagem de configuração, tornando possível a especificação e criação de objetos visuais fora do *framework*. Assim, as propriedades da aplicação podem ser descritas por uma linguagem de configuração auxiliar.

A idéia é fornecer ao usuário ou a um programador menos experimentado meios de especificar características de objetos visuais através de arquivos de configuração. A linguagem utilizada nestes arquivos foi Lua [IdFF96], que é procedural e com facilidades para descrição de objetos estruturados. Por ser uma linguagem embutida, Lua não tem a noção de programa principal. Além disto, ela provê um mecanismo de construtores, permitindo que sejam chamadas funções quando da criação de uma estrutura (tabela). Tais características a tornam perfeitamente compatível com a filosofia **VIX**pois são fortemente baseadas em construtores para a inicialização e a composição de objetos.

[1]Segundo [Mey88], qualquer linguagem orientada por objetos deve ter mecanismos de herança, polimorfismo e *late-binding*.

Para confecção do *framework*, foi adotada uma estratégia de decomposição. Desta forma, alguns recursos disponibilizados são destacados do **VIX** e podem ser utilizados em conjunto com outros sistemas. Esta decomposição faz com que os serviços provenientes deste trabalho sejam divididos em quatro subprodutos, detalhados a seguir:

- um *binding* CD para C++ e Lua.

  A implementação do conceito de *device* está diretamente relacionada com os diversos *canvases* oferecidos pelo CD. Neste aspecto, foi definido um conjunto de classes que mapeiam dispositivos CD para *devices* **VIX**. No entanto, o uso deste sistema gráfico em C++ não é de interesse exclusivo do **VIX**. Assim, foi definido um primeiro produto: Um *binding* CD/C++.

  Além de C++, o **VIX** permite a utlização de seus recursos sob Lua. Logo, foi interessante também a construção de um *binding* CD/Lua.

- um controlador de tempo genérico.

  Os recursos oferecidos pelo temporizador podem também não ser de interesse exclusivo do **VIX**. Para permitir o uso de um *timer* em outros sistemas, foi definido um subproduto que gerencia eventos de temporização.

  Esta ferramenta de controle de tempo está implementada sob *bindings* C++ e Lua.

- **VIX**, o *framework*.

  O terceiro e principal estágio de construção deste trabalho consiste na implementação de um *framework* de suporte a objetos visuais interativos de acordo com a modelagem já descrita. Esta implementação consiste em uma série de classes que oferecem os serviços propostos no modelo.

- **VIX**Lua, o *framework* em Lua.

  A utilização do *framework* como servidor de um arquivo Lua necessita que os serviços e entidades **VIX** estejam disponíveis nesta linguagem. Para permitir essa flexibilidade, foi definido também um binding VIX/Lua.

Para cada subproduto definido na implementação do **VIX**, existe um conjunto de classes C++ responsável por esse serviço. As próximas seções explicam a arquitetura de classes do *framework*.

3.2.1 Os Bindings CD/C++ e CD/Lua

No intuito de oferecer os serviços do CD, foi necessária a definição de uma classe que possuísse tais recursos — `Device`. Esta classe possui em sua interface todas as rotinas que o CD exporta, implementando o conceito de dispositivos **VIX**.

Os métodos de `Device` apenas repassam para o CD as requisições feitas ao dispositivo. Isso porque o CD tem o conceito de dispositivo corrente ou ativo. Assim, uma vez feita uma requisição ao *device*, basta garantir que ele esteja ativo e chamar a rotina CD correspondente. A princípio, pode-se deixar para o programador a responsabilidade da ativação do *device* antes do seu uso, da mesma forma que seu binding C.

Classes e dispositivos	
`StdDevice`	janela CD-Standalone
`IupDevice`	IUP *canvas*
`PsDevice`	*Postscript*
`ImgDevice`	*CD server image*

Tabela 3.1: Tabela de classes e dispositivos do binding CD/C++

No entanto, a existência de contextos globais ou correntes não é muito adequada. Isto porque, ao chamar um método de um *device*, é razoável que esse método esteja sendo aplicado ao seu dispositivo CD correspondente — e não a algum outro dispositivo ativo ou corrente. Para contornar esse problema, o *binding* CD/C++ garante que, ao requisitar serviços de um *device*, o dispositivo estará ativo. Desta forma, não existe a necessidade de funções do tipo `cdActivate`.

A classe `Device`, ao ter a interface dos dispositivos **VIX**, representa um dispositivo genérico. Um *device* possui dois atributos que definem a qual *CD-canvas* ele está associado: um `cdCanvas*` e o seu `void* data` (vide manual do CD [TeC96c]). Para utilizar os diversos tipos de saída, é necessária a especialização de `Device` para os diferentes *drivers*, o que é feito através do mecanismo de herança.

As subclasses de `Device` possuem construtores que preenchem corretamente os atributos do *CD-canvas*, de forma que seus métodos são sempre aplicados no próprio dispositivo. Desta forma, tem-se a hierarquia de classes da tabela 3.1 ilustrada pela figura 3.10.

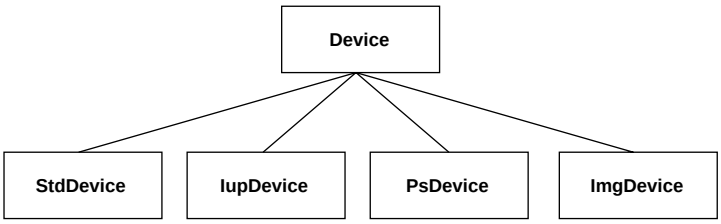


Figura 3.10: Hierarquia de classes de dispositivos **VIX**

Em especial, as classes `StdDevice` e `IupDevice` possuem métodos de resposta a eventos. Isto porque esses dispositivos oferecem também *inputs* do usuário. Ao construir *devices* desses dois tipos, o programador pode requisitar que os eventos sejam associados a métodos específicos de resposta. Esta requisição de registro é feita, por *default*, nos construtores pelo parâmetro `regtype`, que pode assumir os valores `REGISTER` e `NO_REGISTER`. O trecho de código abaixo mostra a interface dos construtores desses dipositivos.

```
StdDevice( char* title, int sx, int sy, int regtype = REGISTER );
IupDevice( Ihandle* handle, int regtype = REGISTER );
```

Além das rotinas do CD, cada um dos dispositivos possui também rotinas de mudança de referencial em *pixels* e funções WD. Os métodos de mudança de referencial somente auxiliam o programador a desenhar com uma translação (em *pixels*) requisições CD. Os métodos WD compõem uma extensão para o tratamento de coordenadas específicas da aplicação.

Cada um destes *devices* pode ser também instanciado em Lua. Para isto, existe um módulo auxiliar Lua que descreve a interface dos dispositivos e associa seus métodos a chamadas do binding C++. Além de descrever e associar *devices*, este arquivo contém funções para a redefinição de duas *fallbacks* Lua[2]:

- `gc`: para que destrutores das classes C++ sejam chamados quando da coleta de lixo sobre suas respectivas tabelas;
- `index`: para a implementação de herança, onde a classe pai é indicada pelo campo `parent` da tabela.

Da mesma forma que o *binding* C++, alguns dispositivos Lua podem ter *inputs*. O dispositivo do tipo `TpStdDevice` (CD *standalone*) possui também métodos de resposta a esses eventos. No entanto, o *device* `TpIupDevice` (IUP) não possui esses métodos. Isto porque assume-se que um *binding* IUP/Lua já faça a associação desses eventos, o que é razoável pois ele não poderia desprezar esta funcionalidade do *canvas*. Deste modo, o *device* pode ser utilizado em conjunto com outros sistemas gráfico-interativos Lua, como por exemplo, IUP-Lua [Gor95], ED [Fil94], etc.

3.2.2 O Temporizador

O recurso de temporização é oferecido pela classe `Timer`. Sua interface possui, além de um construtor e um destrutor, três métodos:

- `Start` — serve para ativação do *timer*, informando o intervalo de repetição e se a geração de *callbacks* é cíclica;
- `Stop` — desativa o *timer*;
- `TicTac` — como método virtual puro, precisa ser redefinido quando é feita uma instanciação. Este método é chamado a cada instante definido pelo intervalo, de forma a passar o controle para a aplicação.

A biblioteca de gerenciamento de temporização possui uma lista ordenada com os *timers* ativos. Constantemente, essa lista é verificada e, caso o primeiro elemento já tenha "estourado" o seu tempo, seu método `TicTac` é chamado. Ao chamar a *callback* de um *timer*, a biblioteca verifica se este é cíclico. Se este for o caso, o elemento é reposicionado adequadamente na lista. Caso contrário, é desativado.

A verificação periódica da lista de *timers* precisa ser feita por um *loop* que fique constantemente analisando o estado dos temporizadores, o que poderia ser resolvido

[2]O uso destas *fallbacks* é importante para a adoção de uma filosofia de orientação por objetos, podendo ser melhor entendida no manual de Lua [IFC95].

pela própria biblioteca. No entanto, os sistemas gráfico-interativos precisam ter, normalmente, um *loop* de controle de eventos, o que invalida a existência prévia de outro *loop*.

Para contornar este problema, a biblioteca de *timers* exporta a função de verificação da lista. Assim, se o sistema gráfico-interativo permitir o registro de novas funções no *loop* de eventos, o programador pode chamar o verificador da lista dentro desse *loop*, possibilitando o uso de temporização. Por exemplo, é possível chamar o verificador do *timer* na *iup idle function* do IUP [LdFG+96], registrador de eventos do CD *standalone* [TeC96c], *loop* de eventos de sistemas nativos como Xlib, etc.

Os mecanismos de temporização podem também ser tratados em Lua. A implementação do *binding timer-Lua* segue o mesmo padrão definido para os *devices*.

3.2.3 O Framework

As principais classes do *framework* implementam os conceitos de VO e VS. A classe `VixTypeVO` define a interface básica de todos os objetos visuais. Ela possui todos os métodos virtuais puros, de forma a definir somente uma especificação para os VOs. Esta especificação define, para a interface de VO, alguns métodos obrigatórios, como:

- tratador de mensagens de VO (`MsgHandlerVO`);
- *callbacks* básicas do sistema, conforme detalhado na tabela 3.2;
- métodos de auxílio gráfico como detecção de pontos no interior de objetos (`Pick`) e determinação de área de ocupação (`BoundingBox`);
- manipulação de objetos como troca de espaço visual (`ChangeVS`), determinação de foco de teclado (`GetFocusedObject`) e detecção de VOs-folha (`GetLeaves`), isto é, que estão no final da hierarquia dinâmica.

Como todos os métodos desta classe são virtuais puros, o usuário teria que, a princípio, implementar todos os seus métodos quando desejasse criar seu próprio VO. Para melhorar este trabalho de redefinição, exite uma subclasse de `VixTypeVO`, chamada `VixVO`, que já implementa grande parte desses métodos.

`VixVO` redefine todas as *callbacks*, exceto a de *repaint*, e o tratador de mensagens para que não façam nada. Desta forma, o programador não precisa tratar os métodos se não for utilizá-los. Como um objeto visual "pensa" que ele é o único VO de seu VS, `VixVO` também implementa, por *default*, os métodos `GetFocusedObject` e `GetLeaves` retornando sempre o `this`, ou seja, o próprio VO. Desta forma, tem-se a hierarquia da figura 3.11:

Um *visual space* também possui uma classe que contém somente métodos virtuais puros — `VixTypeVS`. Esta classe define a interface básica de qualquer VS, determinando a existência de algumas funcionalidades básicas:

- tratamento de mensagens (`MsgHandlerVS`);
- controle de cor de fundo (`GetBgColor` e `SetBgColor`);

Callbacks básicas	
`CallbackEnter`	entrada do *mouse* sobre o objeto
`CallbackLeave`	saída do *mouse* de sobre o objeto
`CallbackDestroy`	fechamento do VS do objeto
`CallbackFocusIn`	recebimento do foco
`CallbackFocusOut`	perda do foco
`CallbackResize`	redimensionamento do tamanho do VS do objeto
`CallbackRepaint`	necessidade de redesenho do VO
`CallbackMove`	movimentação do *mouse* sobre o objeto
`CallbackClick1`	pressionamento do botão esquerdo do *mouse* sobre o objeto
`CallbackClick2`	pressionamento do botão central do *mouse* sobre o objeto
`CallbackClick3`	pressionamento do botão direito do *mouse* sobre o objeto
`CallbackDrag1`	movimentação do *mouse* com botão esquerdo pressionado
`CallbackDrag2`	movimentação do *mouse* com botão central pressionado
`CallbackDrag3`	movimentação do *mouse* com botão direito pressionado
`CallbackDblclick1`	*double click* do botão esquerdo do *mouse*
`CallbackDblclick2`	*double click* do botão central do *mouse*
`CallbackDblclick3`	*double click* do botão direito do *mouse*
`CallbackUnclick1`	despressionamento do botão esquerdo do *mouse*
`CallbackUnclick2`	despressionamento do botão esquerdo do *mouse*
`CallbackUnclick3`	despressionamento do botão esquerdo do *mouse*
`CallbackKey`	pressionamento de alguma tecla (objeto que tem o foco)

Tabela 3.2: Tabela de *callbacks* básicas de objetos visuais

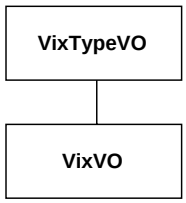


Figura 3.11: Hierarquia de classes de objetos visuais **VIX**

- pedidos de invalidação parcial ou total de sua área de desenho (`Redraw` e `RedrawAll`);
- troca de VO (`ChangeVO`);
- controle do estado de teclas especiais quando do aparecimento de eventos (`IsShift` e `IsControl`);
- requisições de *devices* para desenho (`GetDevice`).

Assim como no caso da implementação de VOs, existe uma subclasse `VixVS` que implementa a maior parte dos métodos existentes da interface VS. `VixVS` possui atributos que controlam todo o gerenciamento básico de cores, teclas especiais, geração de eventos para o VO, etc. Desta forma, a construção de novos VSs não implica no retratamento destas funcionalidades. São exemplos de VS implementados sobre a classe `VixVS`: `VixWindow` e `VixIupWindow`, que implementam o conceito de janela nas plataformas CD *standalone* e IUP, respectivamente. Desta forma, tem-se a hierarquia da figura 3.12:

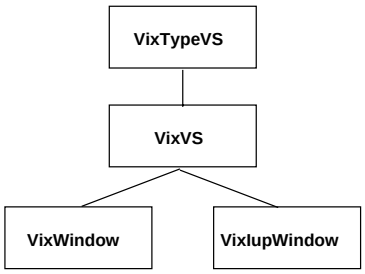


Figura 3.12: Hierarquia de classes de espaços visuais **VIX**

Além da definição de VO e VS, foi necessária a conceituação de filtros. Estes elementos têm a capacidade de atuar como VSs e VOs, servindo de interface para outros filtros, *visual objects* e *visual spaces*. Como um filtro é um VO e um VS simultaneamente, a implementação deste conceito é feita com recursos de herança múltipla. A classe `VixTypeFilter`, que implementa filtros, é uma subclasse de `VixTypeVO` e `VixTypeVS`, de forma a conter uma interface com métodos virtuais puros de VO e de VS. Esta herança pode ser visualizada na figura 3.13.

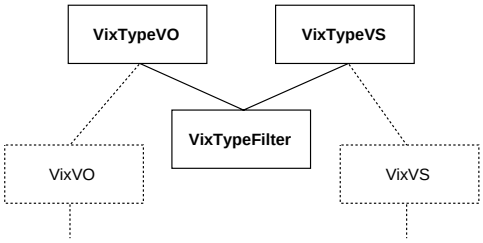


Figura 3.13: Hierarquia da classe de filtro genérico no **VIX**

Dois tipos de filtros foram necessários quando da definição da arquitetura **VIX** — filtros de comunicação e grupos. Estes subtipos de filtros são implementados com subclasses de `VixTypeFilter`: `VixGroup` e `VixFilter`. Estas heranças podem ser visualizadas na figura 3.14.

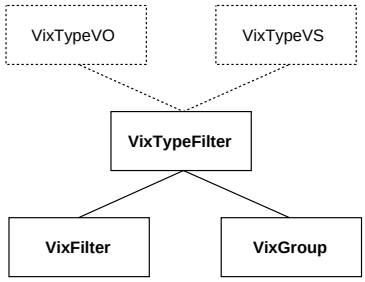
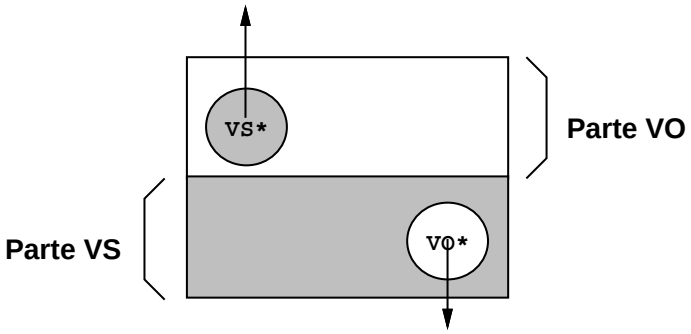


Figura 3.14: Hierarquia de classes de filtros e grupo no **VIX**

Os filtros de comunicação somente repassam as requisições do seu VS para o seu VO e vice-versa. Assim, objetos da classe `VixFilter` atuam como filtros transparentes, que simplesmente chamam os mesmos métodos que foram requisitados ao seu VS ou VO. Este comportamento é ilustrado na figura 3.15. Nesta ilustração, o evento de *left button click*, vindo de um VS, é repssado para o VO, assim como uma requisição de invalidação de área (`RedrawAll`), feita pelo VO, é retransmitida ao VS.

```
void VixFilter::CallbackClick1( VixTypeVS*, int x, int y )
{
  if ( vo ) vo->CallbackClick1( this, x, y );
}
```




```
void VixFilter::RedrawAll()
{
  if ( vs ) vs->RedrawAll();
}
```

Figura 3.15: Ilustração da hierarquia dinâmica em filtros **VIX**

Conforme visto nos trechos de código da figura 3.15, a amarração de um filtro na hierarquia dinâmica de objetos é feita por ponteiros: um para VS (`vs`) e outro para VO (`vo`). O controle desses ponteiros é feito pelos métodos herdados `ChangeVO` e `ChangeVS`, que são redefinidos.

Os grupos atuam de forma semelhante aos filtros. A transmissão de requisições do VO para o VS é feita da mesma forma que nos filtros. Porém, a comunicação VS→VO é redefinida para comportar o gerenciamento de diversos objetos visuais. A classe `VixGroup`, que implementa grupos, possui um campo que é uma lista de VOs (atributo `vos`) e armazena os diversos objetos sob sua gerência. Tal gerência de VOs implica na repassagem de eventos e mensagens aos seus objetos-filho adequados.

Para fazer a retransmissão de eventos e mensagens, `VixGroup` tem a noção de VO corrente e de VO focado. O objeto corrente, ou que possui o *grab*, é aquele que recebe os eventos de *mouse* como *click*, *unclick*, etc., e o objeto com o foco recebe os eventos de teclado (*key pressed*). Os conceitos de *current* e *focused object* são tratados por atributos internos desta classe. Esta política de *focused* e *current* pode ser redefinida posteriormente por especializações de `VixGroup`, de acordo com a necessidade.

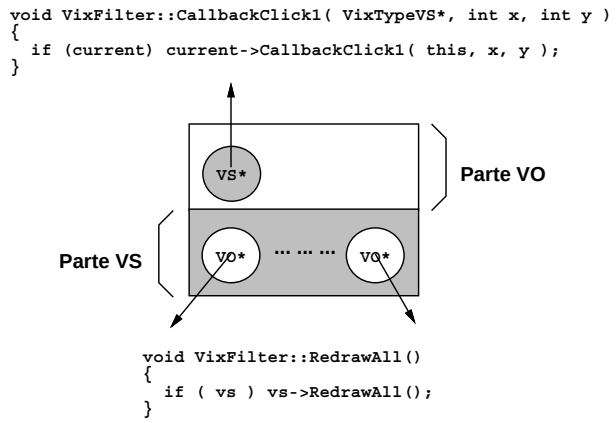


Figura 3.16: Ilustração da hierarquia dinâmica em grupos **VIX**

A classe `VixMessage` é responsável pela implementação do conceito de mensagens, cujo envio é feito pelos métodos `SendVO` ou `SendVS`. A aplicação do método *send* em uma mensagem implica no envio desta para o tratador do objeto envolvido. Os *handlers*, por sua vez, retornam se a mensagem foi tratada ou não, o que é definido pelas constantes `MSG_NOT_PROCESSED` e `MSG_PROCESSED`. O trecho de código abaixo ilustra esse comportamento do método `SendVO` de `VixMessage`.

```
short int VixMessage ::  SendVO( VixTypeVO* vo )
{
    return vo->MsgHandlerVO( this );
}
```

Conforme visto na arquitetura **VIX** (seção 3.2), as mensagens possuem métodos que permitem sua identificação. Isto é necessário pois os tratadores recebem como parâmetro objetos do tipo `VixMessage`, que não possuem conteúdo. Uma mensagem com conteúdo é, necessariamente, uma subclasse de `VixMessage` que acrescenta dados e redefine os métodos de identificação. Identificar mensagens é importante, pois os tratadores precisam de algum mecanismo seguro para fazer a conversão explícita de `VixMessage` para o tipo desejado.

Os métodos de identificação são: `MsgId` e `GetMessageId`. Estas duas funções devem retornar uma mesma *string* (`char*`), que é o identificador de mensagem. `MsgId` é um método estático da classe, logo não precisa de um objeto associado para ser chamado. `GetMessageId` não é estático e só pode ser chamado por uma instância da classe. Desta

forma, o tratator de mensagens não precisa conhecer a *string* de identificação da mensagem para reconhecê-la. Basta comparar o identificador da classe (`MsgId`) com o da mensagem recebida (`GetMessageId`). O trecho de código abaixo mostra o tratamento de uma mensagem do tipo `MyNewMessage`. Neste exemplo, um VO chamado `MyNewVO` identifica a mensagem, converte-a seguramente ao tipo desejado e faz seu tratamento.

A identificação da mensagem pode ser feita sem o uso funções do tipo `strcmp`. Para isso, o programador da mensagem faz o retorno do mesmo ponteiro em `MsgId` e `GetMessageId`, o que é razoável pois as *strings* são iguais. Desta forma, o tratador da mensagem pode simplesmente utilizar uma comparação dos ponteiros na identificação, o que evita o uso de funções de comparação de *strings*.

```
short int MyNewVO::MsgHandlerVO( VixMessage* msg )
{
    if ( msg->GetMessageId() == MyNewMessage::MsgId() )
    {
         MyNewMessage* m = (MyNewMessage*) msg;
         /* Trata adequadamente MyNewMessage */
         return MSG_PROCESSED;
    }
    /* Tenta identificar e tratar outras mensagens */
    return MSG_NOT_PROCESSED;
}
```

3.2.4 O Framework em Lua

Da mesma forma que os recursos de dispositivos e temporizadores, os serviços e conceitos **VIX** estão disponíveis em um *binding* Lua. A exportação destes recursos implica em um novo conjunto de classes que amarra objetos C++ a tabelas (*Lua tables*).

A classe que oferece este serviço de objetos visuais Lua é a `VixLuaVO` que, como especialização de `VixTypeVO`, define a interface de VO nos seguintes aspectos:

- associa a criação de uma tabela Lua `VixLuaVO` à chamada do construtor C++ `VixLuaVO`;
- associa a chamada do *garbage colector*, em uma tabela `VixLuaVO`, à destruição do objeto C++ correspondente.
- repassa eventos recebidos nos métodos C++ para campos da tabela Lua associada (vide tabela 3.2).
- associa os métodos de VO de *pick* e *bounding box* a funções Lua correspondentes.

Com a possibilidade de construir VOs em Lua, os programas **VIX** podem executar módulos Lua de dentro de módulos C++, definindo e instanciando novos objetos. Este

recurso é importante pois permite a customização de objetos fora do *framework*, de forma que as aplicações **VIX** podem incorporar objetos definidos fora dela.

A classe `VixLuaVO` garante a incorporação de objetos visuais Lua no ambiente C++. Para fazer a operação inversa, **VIX** redefine a *index fallback* de Lua. Esta nova função testa se o dado que está sendo acessado é do tipo `userdata` (um ponteiro C++). Se este for o caso, a *fallback* envia uma mensagem chamada `LuaGetFieldMsg`, que sugere ao objeto que ele retorne um dado Lua que represente o atributo acessado. Caso o objeto saiba informar a respeito desse dado, ele preenche um campo com esse valor e retorna `MSG_PROCESSED`. Caso contrário, retorna somente `MSG_NOT_PROCESSED`. Neste caso, o não tratamento da mensagem implica em um erro de programação, o que gera a interrupção do programa.

Desta forma, os objetos confeccionados em C++ e Lua podem "conviver" sob os mesmos espaços visuais e filtros. A importação de objetos Lua em C++ (C++←Lua) é feita por classes especializadas (figura 3.17) e a incorporação de objetos C++ por Lua (Lua←C++) é feita com *fallbacks* e mensagens.

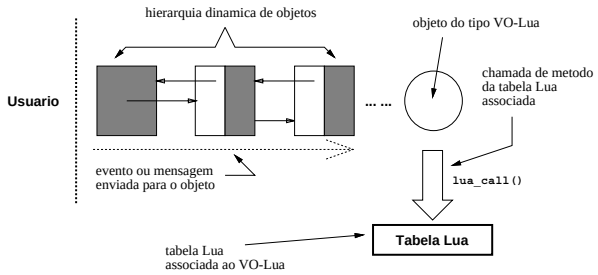


Figura 3.17: Esquema da incorporação de objetos C++←Lua

Da mesma forma que os VOs, VSs podem ser também tratados em Lua através de um mecanismo semelhante ao feito para os VOs. No caso dos VSs, o tratador da mensagem `LuaGetFieldMsg`, chamada na *fallback*, retorna os atributos referentes aos *visual spaces*. Para a incorporação C++←Lua, este mecanismo é feito por classes especializadas, como `VixLuaWindow` e `VixLuaIupWindow`, que tratam janelas nativas e IUP, respectivamente.

Integrando os serviços de VOs e VSs, os filtros e grupos também podem ser tratados em Lua, em um mecanismo análogo ao utilizado por VOs e VSs. Esta incorporação é feita por subclasses e mensagens oriundas de *fallbacks*, que garantem a comunicação C++↔Lua.

Capítulo 4

Aplicações do Framework

Conforme visto anteriormente, a confecção deste *framework* de suporte a objetos visuais foi dividido em quatro partes: *bindings* CD/C++ e CD/Lua, temporizadores, VIX/C++ e VIX/Lua. Os dois primeiros itens possuem aplicabilidades imediatas e seus usos podem ser feitos em conjunto com bibliotecas já existentes, como, por exemplo, IUP, IUP/Lua, ED, etc.

O *framework*, por outro lado, permite a construção de novas ferramentas que facilitam o desenvolvimento de programas gráfico-interativos. Neste aspecto, este capítulo mostra diversas aplicabilidades de **VIX**. Essas ferramentas, por serem construídas sobre os conceitos básicos de *visual object* e *visual space*, podem naturalmente ser utilizadas como componentes, possibilitando bom grau de reuso.

A seção 4.1 ilustra a potencialidade do conceito de filtros. Uma biblioteca padrão de filtros pode ser construída para fornecer diversos tipos de manipulação sobre objetos visuais. Essas manipulações podem também ser utilizadas como tarefas do usuário sobre o VS, de forma análoga ao Interact [Car95].

A seção 4.2 mostra a confecção de um *toolkit* de interfaces Lua. Neste sistema, a confecção de novos objetos visuais se estende a *widgets*, que podem ser manipulados em qualquer VS — em diálogos, *canvases*, etc.

Dedica-se a seção 4.3 a expor a confecção de um editor gráfico simplificado. Este editor utiliza o sistema de interfaces e a biblioteca de filtros descritos anteriormente.

Por último, a seção 4.4 disserta sobre a construção de um sistema de manipulação de objetos gráficos hierárquicos. Neste aspecto, pode-se construir um conjunto de VOs que tratam elementos primitivos para a composição de objetos, como linhas, retângulos, etc. Tais elementos podem ser agrupados de forma hierárquica (conceito de grupo) e responder a eventos e mensagens, de forma a atuarem de forma análoga à GLB [Rez95].

4.1 Um Pacote de Filtros de Interação

Dada a capacidade dos filtros de interceptar e transformar eventos e mensagens dos objetos, é possível confeccionar de um pacote de filtros responsável pelas operações mais comumente aplicadas sobre os VOs. A elaboração desta biblioteca facilita a programação de programas gráfico-interativos, pois o programador não precisa mais recodificar essas operações. Os filtros, uma vez implementados, ficam disponíveis para outras aplicações.

Deste modo, as operações aplicáveis sobre objetos podem ser tratadas como componentes, formando um conjunto extensível e reutilizável de filtros.

As operações implementadas por filtros podem também atuar como tarefas no estilo Interact. Assim como *tasks* atuam entre o *IUP canvas* e a aplicação, os filtros podem ser colocados entre um VS-pai (janelas **VIX**) e seus objetos visuais. O Interact permite que apenas uma tarefa por vez esteja ativa no canvas, o que é razoável face à existência de somente um *mouse* ou teclado. Já o modelo **VIX**, por permitir uma *pipeline* de filtros, oferece mecanismos de atuação de diversas tarefas simultaneamente. Esta simultaneidade da operação dos filtros acarreta uma maior flexibilidade para a manipulação destas entidades. Como vantagens desta flexibilidade, podem-se citar:

1. Combinação de tipos distintos de tarefas.
   O modelo Interact permite que se possa suspender temporariamente a execução de uma tarefa, associando outra ao *canvas* e retornando à anterior assim que a última terminar. Isto é interessante pois uma tarefa de movimentação de objetos pode ser suspensa enquanto o usuário navega pelo desenho, tornando visível a posição onde deseja colocar o objeto. Após definir a nova *viewport*, a aplicação pode retornar à movimentação do desenho.

   O modelo **VIX**, ao permitir o encadeamento de filtros, garante que os filtros de movimentação de objetos e rolagem do *canvas* possam estar atuando simultaneamente. Um filtro pode controlar se o usuário faz um *drag* para fora do *canvas* e outro verifica um *drag* por cima de objetos. Estes filtros combinados permitem que, quando um objeto é arrastado para fora do *canvas*, haja um *scroll* contínuo (via *timer*) e automático da janela.

   Esta funcionalidade **VIX** mostra que pode ser interessante combinar de tarefas de visualização com os demais tipos.

2. Modularização de filtros.
   Algumas restrições sobre o *canvas* podem ser modularizadas se houver maneiras de compor filtros. Um exemplo deste recurso é a restrição de *snap to grid*. É possível confeccionar um filtro que receba os eventos do *mouse* e os repasse ao VO com deslocamentos referentes aos ajustes da grade. Deste modo, outras tarefas, como capturas de retângulos e linhas, translações e cizalhamentos de objetos, etc., podem reutilizar essa restrição, bastando colocar o *snap* como VS desses filtros. Além disso, outros filtros, que não estiverem abaixo, na hierarquia dinâmica do *snap*, não sofrem interferência de ajustes de *grid*. Este exemplo pode ser visualizado na figura 4.1.

Diversas experiências foram realizadas na confecção de filtros para verificar a sua adequabilidade. Os filtros que oferecem serviços de manipulação sobre VOs trabalham na comunicação de sentido VS→VO. Verificou-se que uma estratégia interesante para a confecção deste tipo de filtro é a sua decomposição em um núcleo funcional, uma interface com o *framework* e um desenhador de *feedback*. O núcleo funcional contém rotinas do filtro necessárias à implementação do serviço oferecido. Essas rotinas são chamadas por funções da interface com o *framework*, definindo como os recursos do

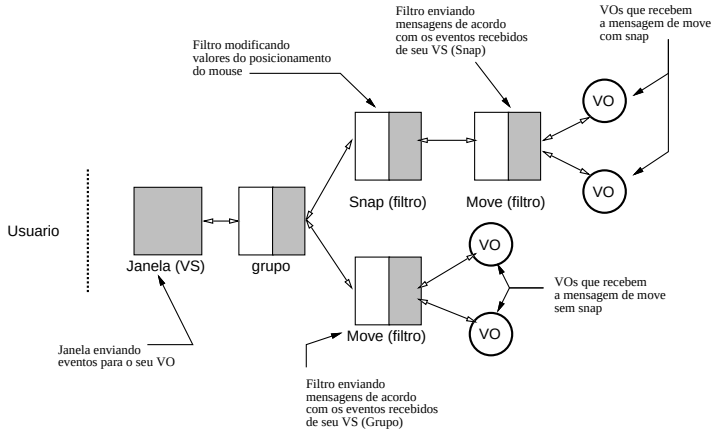


Figura 4.1: Composição do filtro de restrição com outros filtros **VIX**

núcleo são utilizados. O resultado da manipulação é, então, transmitido ao desenhador de *feedback*, que interage com o usuário.

Um exemplo da utlização desta filosofia pode ser visto na confecção de um filtro de seleção. O núcleo deste filtro contém métodos que inserem, consultam e retiram VOs de uma lista de objetos selecionados (`SetSelected`, `IsSelected`, `AddFence`, etc). Esses métodos são oferecidos pela classe `Select`, que não é um filtro e somente fornece serviços de seleção. A interface é representada por uma classe denominada `FtSelection`, que herda de `VixFilter` e `Select`. Assim, `FtSelection` é um filtro que pode ser colocado na hierarquia dinâmica de objetos **VIX** e também um selecionador de objetos. Os métodos herdados de `Select` compõem o núcleo do filtro e os adquiridos de `VixFilter` formam sua interface com o **VIX**.

Esta divisão é importante porque os métodos da interface determinam, neste exemplo, como é feita a seleção de objetos. Assim, a definição de novos filtros de seleção pode reutilizar os serviços do núcleo. A figura 4.2 ilustra uma possível hierarquia de filtros de seleção. `FtSelection` seleciona objetos com o botão esquerdo do *mouse* enquanto que `FtSelection2` os seleciona com o botão direito. Ao herdarem de `Select`, estes filtros reutilizam serviços de seleção com interfaces distintas.

O *feedback* de `FtSelection` é feito em dois momentos: definição de *fences* de seleção múltipla e marcação de objetos selecionados. As marcas dos objetos são feitas por uma pequena estrutura que informa os objetos das primitivas gráficas que devem ser adicionadas em suas representações (p.ex. marcas nas bordas do objeto). Essa estrutura é enviada em uma mensagem quando da seleção de um objeto. A definição do *fence* é tratada dentro do núcleo da seleção pela reutilização do núcleo do filtro de captura de retângulo.

Este reuso foi possível porque os filtros de captura também foram construídos de modo particionado. A classe `CapTwoPoints` oferece um núcleo de captura de dois pontos através de métodos do tipo `Begin`, `Move`, `End`, etc. Esta mesma classe oferece a função `Feedback`, que é responsável pelo desenho temporário da captura e pode ser redefinido para representar linhas e retângulos nas mais diversas formas (pontilhado, tracejado, etc). O núcleo e o *feedback* estão na classe `CapTwoPoints`, que oferece tanto os serviços de núcleo quanto de representação da captura. Assim, o filtro de seleção reutiliza este

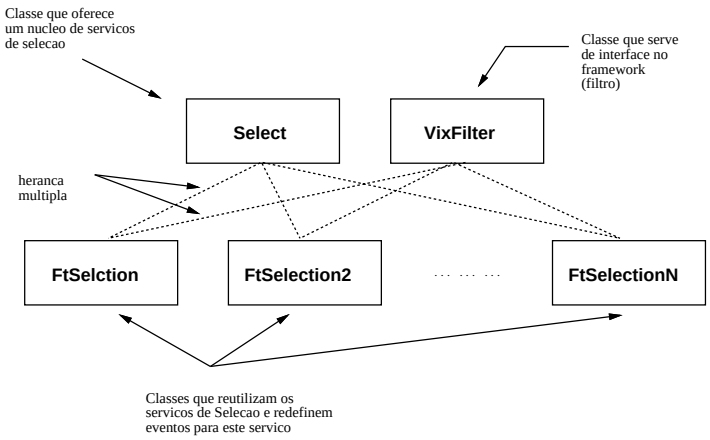


Figura 4.2: Possível hierarquia de filtros de seleção em **VIX**

núcleo para a definição do *fence*.

Um filtro exclusivo para a captura de retângulos (`FtDrag1CapRect`) é também implementado pela herança de `VixFilter` para ter uma interface **VIX**. No entanto, diferentemente do filtro de seleção, a junção dos serviços de captura e interface em `FtDrag1CapRect` foi feita por composição de objetos, ou seja, esta classe possui um campo que aponta para um objeto `CapTwoPoints`. Essa composição é interessante pois permite que diversas interfaces de filtros (*click-drag-release*, *click-move-click*, etc.) possam ser combinadss com diferentes núcleos sem a necessidade de uma hierarquia de classes. Essa composição pode ser visualizada na figura 4.3.

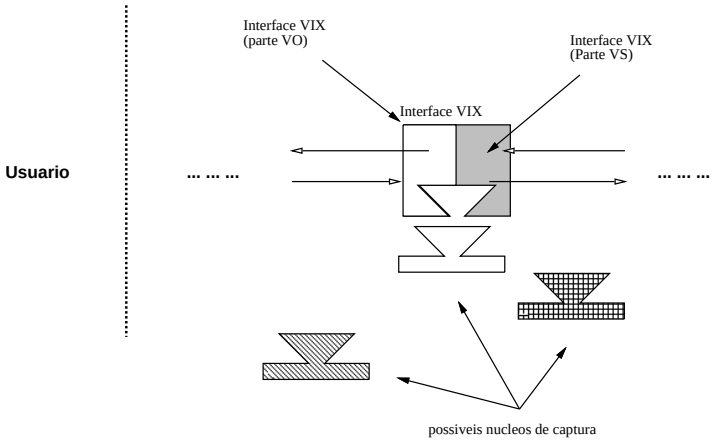


Figura 4.3: Composição de filtros de capturas em **VIX**

Além de seleção e capturas, os filtros podem também ser utilizados para a aplicação de transformações lineares. Neste caso, os filtros não conhecem os atributos que representam as coordenadas dos objetos, não podendo alterá-las quando da aplicação da operação. Logo, esses filtros somente informam os VOs que o usuário requisitou uma determinada transformação, o que é feito com o envio de mensagens. São exemplos deste tipo de filtro: `FtMove` e `FtRotate`, que enviam mensagens de translação e rotação quando ocorrem eventos de *click* e *drag* do mouse. O núcleo deste tipo de filtro consta, basicamente, de

uma rotina que envia a mensagem "`MOVE`" ou "`ROTATE`" para o VO. A interface, como todos os filtros, é feita pela herança de `VixFilter`, enquanto que o *feedback* fica a cargo dos objetos visuais que estão sendo manipulados (invalidação de regiões da tela).

Conforme visto anteriormente, o encadeamento de filtros permite a modularização de alguns serviços de interação. Como exemplo, a classe `FtBackSnapGrid` filtra todos os eventos de *mouse* colocando-os sob a atração de uma grade (*snap to grid*). Desta maneira, filtros como capturas e translações podem reutilizar essa restrição sem modificações em suas classes, bastando colocar um objeto `FtBackSnapGrid` como espaço visual da captura ou translação, conforme ilustrado na figura 4.1.

Esta moduralização e a simultaneidade da atuação dos filtros faz com que seus posicionamentos na hierarquia dinâmica seja bastante importante. Por exemplo, o filtro `FtFollow` serve somente para desenhar marcas nas extremidades do VS que indicam a localização do *mouse*. Para tal, este filtro atualiza, a cada evento de *mouse move*, sua representação gráfica (*cross-hair*). Desta forma, se houver um filtro de *snap* como VS do *follow*, a representação gráfica deste último filtro fica também ajustada pelo *grid*. Por outro lado, se o filtro de *snap* for o VO do filtro de *follow*, o *cross-hair* não é afetado pelo *snap*; embora as operações (filtros) a partir de `FtBackSnapGrid` o sejam.

Apesar de sua abrangência para a manipulação de objetos na comunicação VS→VO, os filtros podem também ser utilizados na comunicação VS←VO, isto é, podem reinterpretar algumas requisições de um objeto ao seu espaço visual. Neste aspecto, um filtro *anti-flickering* é um bom exemplo desse tipo de recurso.

A classe `FtFrameRatio` implementa um filtro que captura todas as requisições de invalidação da superfície de visualização e chamadas de *repaint* dos objetos. Ao capturar essas requisições, `FtFrameRatio` faz com que as primitivas gráficas de redesenho sejam aplicadas em um *buffer off-screen*. Esta "bufferização" é feita pela chamada das rotinas de *repaint* dos VOs, passando como argumento um dispositivo do tipo imagem (`ImageDvc`). Depois de desenhados os objetos, a imagem é aplicada sobre a janela que, ao receber várias primitivas gráficas simultaneamente, faz com que a atualização da janela fique melhorada, minimizando problemas de *flickering*.

A utilização do filtro `FtFrameRatio` pode ser interessante quando ele é utilizado em conjunto com filtros de manipulação direta de objetos. Isto porque, ao responder mensagens de transformações, os VOs alteram seus atributos e requisitam ao seu VS a invalidação de alguma área, através dos métodos `Redraw` ou `RedrawAll`. Os objetos da classe `FtFrameRatio`, ao receber esta requisição, podem então redesenhar rapidamente e de uma só vez a área da tela inválida, dispensando o uso de mecanismos de *drafts* em *xor*. Um esquema desta utilização do filtro `FtFrameRatio` pode ser visualizado na figura 4.4.

Outro recurso interessante do filtro *anti-flickering* é a possibilidade de redesenho de toda a janela em intervalos regulares de tempo. Isto é possível porque a classe `FtFrameRatio` herda de `Timer`, além de `VixFilter`. A *callback* do temporizador nesta classe redesenha toda a janela regularmente, de forma que as atualizações nessa janela não são dependentes das chamadas dos métodos `Redraw` ou `RedrawAll`. Logo, torna-se possível programar objetos visuais sem preocupar-se com a atualização do VS. Os objetos simplesmente alteram suas propriedades e `FtFrameRatio` se responsabiliza em dar o *feedback* automaticamente para o usuário.

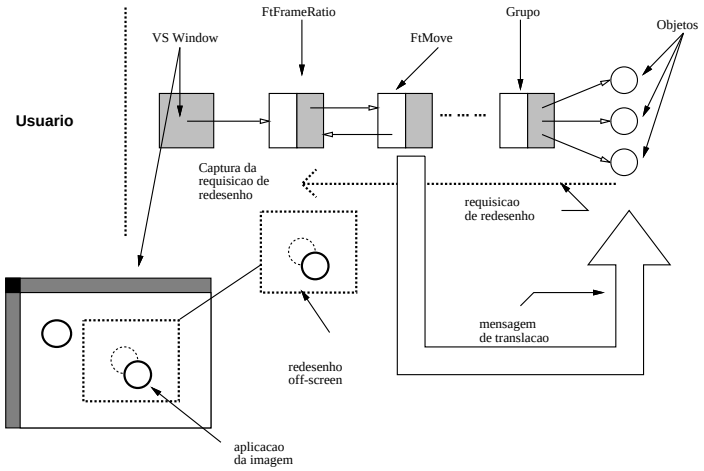


Figura 4.4: Esquema da utilização do filtro `FtFrameRatio`

Os filtros têm sido utilizados para a implementação de diversos recursos gráfico-interativos: tarefas de construção, visualização e seleção no estilo Interact, restrições de eventos, suavização de *feedback*, etc. Desta maneira, essas entidades têm se mostrado bastante flexíveis para a implementação de diversos serviços de forma modular, extensível e reutilizável.

4.2 Um Toolkit de Interfaces com o Usuário

Conforme mencionado nos requisitos deste trabalho (seção 1.2), a extensibilidade e reusabilidade de objetos visuais pode ser aplicada na confecção de *toolkits* de interface. Estes *toolkits* definem VOs que oferecem os serviços dos *widgets* mais comumente utilizados. Além de oferecer um conjunto pré-determinado de elementos de interface, esse *toolkit* pode garantir a extensibilidade, a capacidade de manipulação direta e um bom grau de reuso dos seus elementos.

A extensibilidade é garantida pela possibilidade da construção de novos VOs. Assim como os *visual objects*, é possível confeccionar de novos *widgets* pela simples redefinição de alguns métodos de objetos mais genéricos.

A capacidade de manipulação direta vem do fato de que tais *widgets*, sendo VOs, recebem eventos e mensagens vindos de filtros como seleção, translação, rotação, etc. Assim, estes elementos de interface podem responder adequadamente às mensagens, comportando-se da mesma forma que os objetos gráficos da aplicação.

A reusabilidade de objetos visuais é algo intrínsico à arquitetura **VIX**, pois, uma vez confecionado este VO, ele fica disponível para outras aplicações. Conseqüentemente, é possível reutilizar os *widgets* construídos como especializações de VOs, conforme espera-se de um sistema de interfaces.

Como exemplo deste tipo de sistema de interface, pode-se citar o Tk/VIX[1][dC96]. Neste *toolkit*, os *widgets* são especializações de objetos visuais que podem ser colocados em qualquer VS, recebendo eventos e mensagens. Ao estarem sobre qualquer espaço

[1] O nome Tk/VIX é provisório até a publicação deste documento.

visual, eles podem "conviver" com objetos gráficos da aplicação. A figura 4.5 ilustra essa potencialidade. Neste exemplo, tem-se um diálogo com menus, botões e um *canvas*. Dentro deste *canvas*, existem diversos elementos de interface que podem ser manipulados da mesma forma que os objetos gráficos comuns.

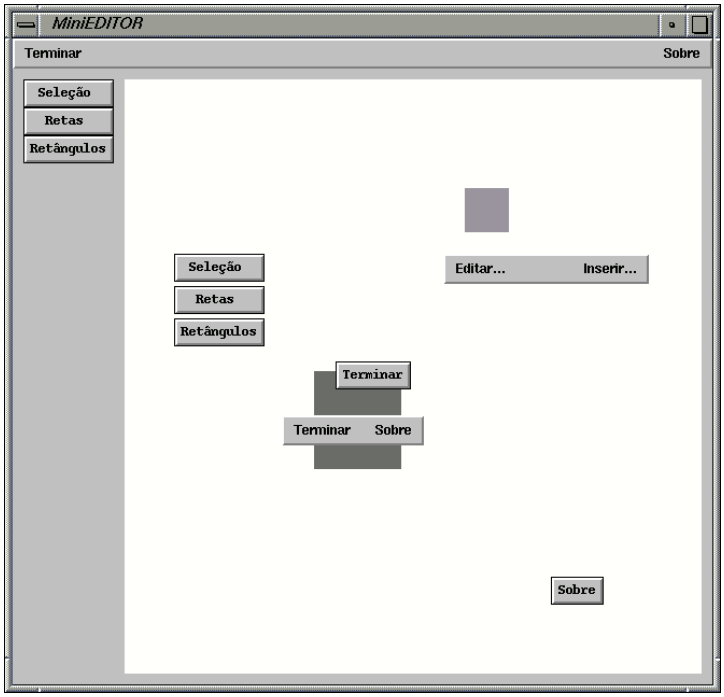


Figura 4.5: Exemplo de aplicação construída com o Tk/VIX

Apesar da possibilidade de manipulação direta sobre os *widgets*, é interessante permitir a composição de diálogos de forma mais abstrata. Muitas vezes, o posicionamento dos elementos não é feito por coordenadas absolutas, mas por descrições de composição. Como exemplos destas descrições abstratas podemos citar os conceitos de *hbox* e *vbox* do IUP e do TEX.

Para suportar este recurso, o Tk/VIX utiliza o conceito de grupo para compor objetos. Tais grupos podem determinar o posicionamento de seus filhos com métodos específicos de *layout*. Como esses métodos podem ser redefinidos, pode-se estabelecer uma variedade de formas composição.

A implementação dos *widgets* em Tk/VIX é feita com base nos VOs-Lua, de forma que a construção de novos elementos é facilitada pela simplicidade de Lua para a gerência de atributos, criação de tabelas, etc. A especificação de diálogos Tk/VIX também é feita com a sintaxe Lua, explorando a potencialidade desta linguagem para a descrição de dados.

Como exemplo dessa potencialidade, o código abaixo define um diálogo simplificado com um botão e um *label*. O botão com o texto "Ok" termina o programa pela chamada da função `exit`. A figura 4.6 ilustra o diálogo correspondente à descrição do exemplo.

```
d1 = Dialog {width = 300, height = 100, title = "Bye"}

v1 = Vbox {d1}
    Fill {v1}
    h0 = Hbox {v1}
         Fill {h0}
         Label {h0; text = "Press 'OK' to exit program"}
         Fill {h0}
    Fill {v1}
    h1 = Hbox {v1}
         Fill {h1}
         Button {h1; label = "Ok ", button_cb = exit}
         Fill {h1}
    Fill {v1}
```

Figura 4.6: Exemplo de diálogo Tk/VIX

Assim, o **VIX** pode ser utilizado como base de um *toolkit* de interface que permite a manipulação direta sobre seus *widgets*, além de sua programação e descrição em uma linguagem simples porém poderosa — Lua.

4.3 Um Editor de Primitivas Gráficas

Nas seções 4.1 e 4.2, foi vista a possibilidade do uso do *framework* para a construção de *toolkits* de interface e de um sistema de filtros para a manipulação de objetos. No entanto, a utilização do *framework* não pôde ficar restrita somente à construção de novas ferramentas. Foi preciso construir uma aplicação que utilizasse a filosofia proposta neste trabalho, isto é, verificar a adequabilidade da confecção de programas gráfico-interativos com ferramentas *VIX-oriented*.

A aplicação escolhida para testes foi o Ed-VIX, um editor simplificado para o desenho de primitivas gráficas[2]. O editor utilizou o Tk/VIX para a instanciação de *widgets* de interface e alguns filtros descritos na seção 4.1 para a manipulação de objetos.

O editor consiste em um *canvas* e uma *toolbar* de trabalho. A *toolbar* consiste em um diálogo com um conjunto de botões Tk/VIX, onde o usuário seleciona ações, escolhe cores

[2]O Ed-VIX foi construído em conjunto com Anna Magdalena Hester para o IV PIBIC — Plano Institucional de Bolsas de Iniciação Científica do CNPq, 1996

e aplica algumas operações sobre as primitivas. O *canvas* é destinado à manipulação dos objetos visuais que representam retângulos e linhas. Por serem VOs, essas primitivas podem ser manipuladas pela atuação dos filtros.

As operações de seleção, translação e criação de retângulos e linhas foram implementadas com os filtros descritos anteriormente. As ações de *"send to back"* e *"send to front"*, comuns em editores gráficos, são tratadas por um grupo que controla todos os VOs, manipulando a ordem visual dos objetos. O *feedback* da movimentação dos VOs foi tratado pelo filtro *anti-flickering* com o timer desligado, pois os objetos não precisavam ser redesenhados constantemente.

Uma ilustração do Ed-VIX pode ser vista na figura 4.7. Nesta figura, diversos objetos foram instanciados, alterados com relação às cores, selecionados e transladados.

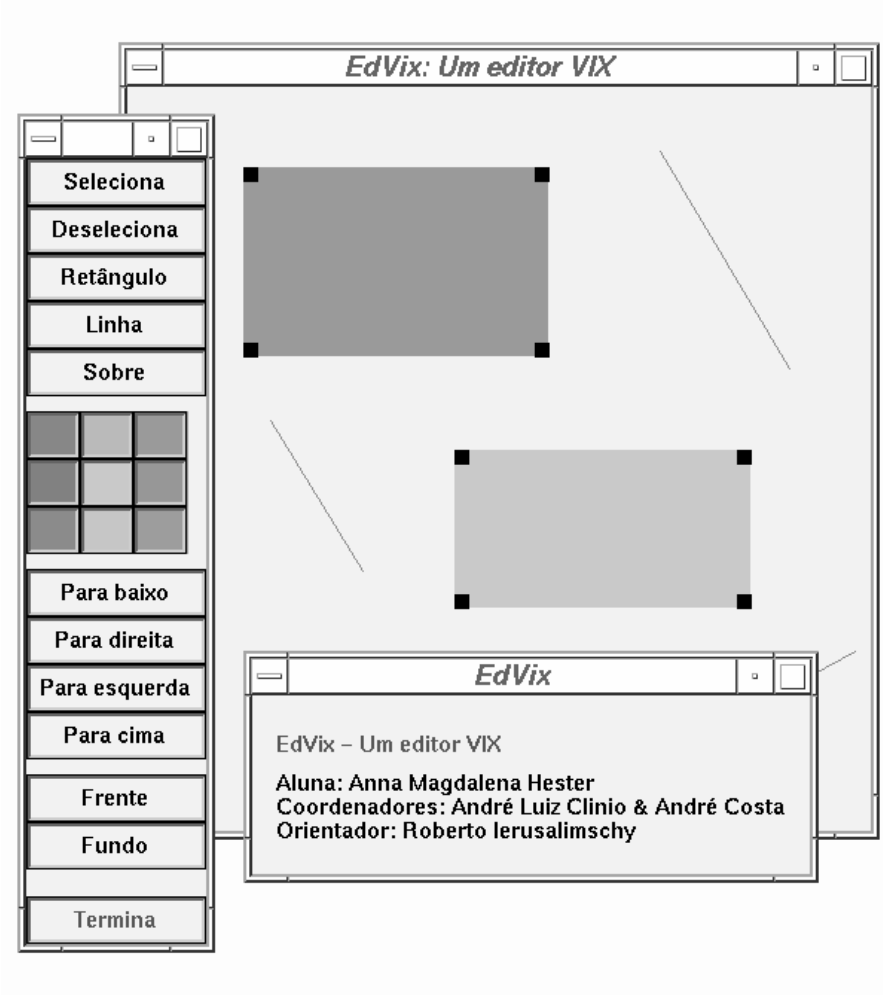


Figura 4.7: Tela de editor Ed-VIX

4.4 Um Sistema para Composição de Objetos

A definição e a instaciação de objetos visuais em Lua permite a customização de objetos fora da linguagem C++. Por ser reflexiva, Lua permite ainda que sua sintaxe possa ser

utilizada em arquivos de descrição de objetos, de forma que as descrições possam ser compartilhadas por várias aplicações [FFG95].

Uma estratégia interessante para abordar este recurso vêm da linha de composição de objetos hierárquicos, onde tem-se como exemplo a GLB [Rez95]. Nesta linha, o **VIX** pode ser utilizado também para definir um conjunto de primitivas atomizadas, que podem ser agrupadas hierarquicamente e ter um conjunto de ações associadas.

Para construir tal tipo de ferramenta, foram utilizadas especializações de VO-Lua que representam *splines*, linhas, *boxes*, textos, etc. Estes VOs são instanciados por construtores que têm os mesmos nomes das entidades GLB. Desta forma, o arquivo gerado por aplicações GLB, dentre elas o próprio TeCDraw [TeC94], pode ser capturado por uma aplicação **VIX**.

Esta experimentação deu origem ao GLB/VIX, que permite a execução de um arquivo de descrição do TeCDraw e, posteriormente, sua visualização em uma janela. Na figura 4.8, pode-se ver uma *VS window* exibindo um desenho feito no TeCDraw.

Figura 4.8: Janela **VIX** exibindo um arquivo importado do TeCDraw

Da mesma forma que na GLB, o **VIX** permite a manipulação da hierarquia de objetos. Métodos do tipo *link* e *unlink* podem ser mapeados para chamadas `ChangeVO`, de forma a permitir a inserção e retirada de partes (itens) dos desenhos. Outros métodos, como `Read` e `Save`, podem ser tratados diretamente com o uso da reflexibilidade de Lua, não sendo primordial a sua existência no VO Lua básico.

O VO básico não prevê funções GLB do tipo `GetClassName`, pois assume-se que o *framework* deve tratar da mesma forma qualquer VO, independentemente do tipo. Esta identificação dinâmica dos tipos de objetos pode ser feita, no **VIX**, com o mecanismo de

Principais funções do profile		
Tempo gasto (%)	Nome da função	Observações
25.1	`lua_execute`	Função Lua para execução de código
21.2	`mcount`	Função interna do profiler
6.0	`luaY_parse`	Função do parser Lua
3.7	`do_call`	Função do parser Lua
3.6	`present`	Função Lua
3.5	`lua_equalObj`	Função Lua
3.3	`luaY_lex`	Função de análise léxica Lua
3.2	`pushsubscript`	Função Lua de consulta de atributos
2.7	`hashindex`	Função Lua

Tabela 4.1: Profile típico da execução de desenhos GLB/VIX

mensagens. Logo, se a implementação de algum sistema sobre o **VIX** quiser fazer esse tipo de verificação, ela deve usar esse mecanismo.

A criação de objetos visuais Lua implica na alocação de memória em C++ e Lua. O espaço destinado em C++ implementa o mapeamento dos recursos de VO entre as duas linguagens. Já a memória Lua é utilizada para armazenar atributos dos objetos. Isto significa que, diferentemente da GLB, não há duplicação de atributos em C++ e Lua, o que melhora a manutenção da biblioteca.

Na implementação de objetos *GLB-like*, optou-se por tratar este tipo de VO somente em Lua, o que poderia trazer um *overhead* de desempenho durante o desenho dos VOs. Para contornar esse problema, foi criado um pequeno *stub* C++ que, consultando os atributos dos objetos, os desenha, ou seja, os métodos Lua `CallbackRepaint` dos VOs chamam funções C++ de redesenho. Esta medida melhorou o tempo de desenho em aproximadamente 40%, e comprova que Lua tem influência decisiva no desempenho do redesenho de objetos.

Apesar da melhora, constatou-se visualmente que o tempo de redesenho de objetos **VIX** ainda está bem acima do tempo dos objetos GLB. Para buscar uma explicação para este problema, foram feitos diversos *profiles*. Constatou-se que a maior parte do tempo era gasto com a consulta de atributos e a execução de código Lua. A tabela 4.1 ilustra um dos *profiles* realizados para aplicações GLB/VIX que lêem um arquivo TeCDraw e o exibem. Neste exemplo, foram desenhadas aproximadamente 130 *polylines*, conforme a figura 4.8.

Comprovou-se mais uma vez que a utilização de Lua como servidor de atributos causa um *overhead* no desenho das primitivas gráficas. Este *overhead* não é encontrado na GLB, porque ela copia os atributos de Lua para C++ no instante da criação dos objetos.

Com a finalidade de não duplicar atributos e não comprometer de desempenho do sistema, está prevista uma nova implementação da GLB dentro do TeCGraf. Esta nova implementação tem por base o armazenamento dos atributos somente em C++. Neste caso, a manipulação de atributos em Lua é feita com a utilização de *fallbacks*. Os objetos Lua são do tipo `userdata`, que, quando indexados como tabelas, chamam funções

específicas de tratamento. Esse mecanismo é utilizado em outros sistemas, como o IUP-Lua [Gor95].

Qualquer que seja a solução, o **VIX** permite a construção de sistemas do tipo GLB, seja qual for a implementação adotada. Os objetos visuais Lua fornecem uma interface natural para os VOs que podem ser instanciados e tratados em Lua. Caso uma aplicação não queira tratar os atributos em Lua, pode-se utilizar diretamente o VO/C++ e exportar seus serviços de outra forma: via duplicação de atributos, uso de *fallbacks*, etc. Deste modo, o **VIX** pode ser utilizado para a confecção de sistemas de composição de objetos com diferentes abordagens.

Uma característica interessante da utilização de VOs como base de um sistema para a composição de objetos é o fato de que VOs sabem responder mensagens, o que garante extensibilidade às ações que eles podem tomar, não ficando restritos a *click*, *move*, *rotate*, etc. Esses objetos podem receber, inclusive, mensagens de filtros, pois estes são *visual spaces*.

Assim, um sistema de composição de objetos pode naturalmente reutilizar descrições e incorporar um conjunto de operações de outro sistema. Desta forma, o **VIX** pode ser utilizado como *framework* integrador de sistemas de filtros/tarefas e de composição de objetos, estendendo tanto objetos como operações do usuário.

Capítulo 5

Conclusões

O **VIX** define um modelo e uma implementação de um *framework* de suporte a objetos visuais interativos. Como característica básica, essa modelagem integra os recursos gráficos e de interface em uma única ferramenta, não havendo a necessidade do uso de dois sistemas distintos para a confecção dos objetos.

A arquitetura e a implementação propostas neste trabalho cumprem os requisitos expostos na seção 1.2. Estes requisitos são:

- Extensibilidade e reusabilidade de objetos.

  Os objetos que implementam o relacionamento homem-máquina nas aplicações **VIX** podem ser tratados como componentes. Os conceitos de VO e filtros (seção 3.1), que tratam respectivamente os objetos visuais e suas operações, possuem a capacidade de ser estendidos, reutilizados e redefinidos. Tal capacidade garante que sistemas gráfico-interativos construídos sobre **VIX** tenham um bom grau de reuso, modularidade e extensibilidade.

- Aplicabilidade na construção de *toolkits* de interface

  Conforme visto na seção 4.2, o **VIX** pode ser aplicado na confecção *toolkits* de interface como o Tk/VIX, que define VOs que atuam como *widgets*. A aplicabilidade de operações de manipulação direta sobre os *widgets* advem do fato de que estes elementos são VOs, podendo tratar eventos e mensagens vindos de qualquer filtro, conforme ilustrado na figura 4.5.

- Portabilidade

  O *framework* **VIX**, conforme exposto na seção 3.2, foi construído totalmente sobre um *binding* C++ e Lua do CD *standalone*, que é multiplataforma. Assim, os objetos visuais e as interações (filtros) podem ser portados entre diversas plataformas.

- Simplicidade

  A simplicidade do modelo conceitual **VIX** deriva do fato de que todos os conceitos inerentes ao tratamento de objetos visuais vêm do pequeno conjunto formado por objeto visual, espaço visual, dispositivos e mensagens — grupos e filtros são VOs e VSs simultaneamente.

- Suavização de *feedback* e recursos de temporização

  O recurso de temporização é oferecido pelo conceito de *timer*, apresentado na seção 3.1. A suavização do *feedback* com o usuário decorre da implementação do filtro *anti-flickering*, descrito na seção 4.1.

Além de preencher os requisitos propostos, **VIX** é um *framework* flexível, podendo ser utilizado na confecção de diversos tipos de ferramentas gráfico-interativas portáteis. Como evidências destas utilizações, quatro ferramentas são apresentadas no capítulo 4 desta dissertação:

1. um pacote extensível de interações, constituído de um conjunto de filtros que atuam de forma semelhante às *tasks* do Interact [Car95] (seção 4.1).

2. um *toolkit* de interfaces Lua como o Tk/VIX [dC96], descrito na seção 4.2.

3. um editor de primitivas gráficas — o programa descrito na seção 4.3.

4. um compositor de objetos gráficos hierárquicos onde, conforme exposto na seção 4.4, é possível utilizar a abordagem GLB [Rez95].

Ao permitir a construção de novas ferramentas portáteis sob a filosofia **VIX**, o programador sabe, de antemão, que os objetos visuais têm a capacidade de receber requisições (mensagens) de operações definidas em um outro sistema de interação. Por outro lado, ao construir um sistema de interação, sabe-se que existe uma interface mínima que os objetos visuais podem responder. Logo, o **VIX** pode ser utilizado como um sistema integrador das ferramentas 1, 2 e 4 descritas acima, sem impor que esses sistemas sejam confeccionados em uma única biblioteca e que algum deles preveja características particulares do outro.

Além desta integração, é possível que esses sistemas sejam bibliotecas distintas e independentes. Mesmo sendo independentes, eles são compatíveis por construção, de forma que o programador não necessita adaptá-los no momento de suas utilizações. Comprovando esta integração sem dependência, nota-se a possibilidade de utilizar filtros, sem modificações, nas ferramentas 2, 3 e 4 descitas acima.

Aplicabilidades no TeCGraf

Conforme discutido anteriormente, a flexibilidade e a integração do **VIX** permite a confecção de diversos tipos de ferramentas gráfico-interativas. A construção destas ferramentas têm aplicabilidades dentro do TeCGraf. Em um primeiro estágio, o temporizador e os *bindings* CD/C++ e CD/Lua, por terem sido desenvolvidos como subprodutos (seção 3.2), podem ser reaproveitados em outros sistemas. Como exemplos, estão sendo estudadas a incorporação do *timer* no IUP [LdFG+96] e a disponibilização dos *bindings* CD.

Além disso, com a integração das ferramentas 1 e 4, também é possível utilizar o **VIX** como um sistema integrador da GLB e do Interact, facilitando a confecção de aplicações que utilizem essas duas bibliotecas. Esta facilitação decorre do fato de que as aplicações

não precisam gerenciar o fluxo de controle entre a interação (Interact) e a representação de seus objetos (GLB) (ver figura 2.4).

Com essa integração e com o uso da linguagem embutida Lua, os recursos da GLB e do Interact podem ser tratados em outros sistemas além do IUP e do CD *standalone*, como ED [Fil94], IUP/Lua [Gor95] e Tk/VIX. Desta forma, é possível confeccionar de um *toolkit* Lua com as diversas facilidades do EDG [FFG95]. A construção deste *toolkit* pode ser feita de diversas maneiras: IUP/Lua + VIX/Lua, ED + VIX/Lua ou Tk/VIX + VIX/Lua. No último caso, torna-se possível a manipulação direta dos *widgets*. Isto decorre da integração do sistema de interface e gráfico, conforme descrito na seção 4.2.

Sugestões para trabalhos futuros

Com a extensibilidade e o agrupamento de objetos visuais, pode-se confecionar, de maneira análoga à CPI [Cer96] (seção 2.3), elementos de interface mais complexos como painéis de controle ou subprogramas. Estes *"VO-wares"* podem ser instanciados dentro de outros programas, sendo manipulados com filtros de interação e comportando-se de maneira similar aos *program-glyphs*[1] do FRESCO [Chu94]. Neste aspecto, a confecção do TEXT [Cli95] em TCHE [CI95] comprova essa possibilidade[2], conforme visto na seção 2.3.2.

Em um horizonte mais distante, avalia-se a adaptação do modelo **VIX** para a confecção de objetos visuais 3D pela substituição do *binding* CD por um G3D [TeC96a]. Com a existência desses VOs, surge a posibilidade de construir um *toolkit* de interfaces 3D.

De acordo com aspectos apresentados, o **VIX** e os recursos introduzidos por ele possuem uma ampla variedade de aplicações, definindo uma ferramenta flexível para o tratamento do conceito de *objetos visuais interativos*.

[1] Por exemplo, o FDraw da seção 2.2.2.

[2] UAI, TCHE e **VIX** possuem modelagens conceituais muito semelhantes.

Referências Bibliográficas

[AS90] P. Asente e Ralph Swick. *X Window System Toolkit*. Digital Press, 1990.

[BCCI94] R. Borges, C. Cassino, R. Cerqueira, e R. Ierusalimschy. UAI — um framework para suporte a objetos visuais. Em *VIII Simpósio Brasileiro de Engenharia de Software*, páginas 79–89, Curitiba — PR, Brasil, 1994.

[Car95] Marcelo Medeiros Carneiro. Interact: um modelo de interação para editores gráficos. Dissertação de mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro — RJ, Brasil, 1995.

[CCCI94] A. Costa, A. Clinio, C. Cassino, e R. Ierusalimschy. UAI — uma biblioteca gráfico-interativa portável orientada por objetos. Em *XIII Concurso de Trabalhos de Iniciação Científica da SBC*, páginas 767–775, Caxambu — MG, Brasil, 1994. (Menção Honrosa).

[Cer95] Renato Cerqueira. Seminários do Grupo de Tecnologia em Computação Gráfica — TeCGraf, 1995.

[Cer96] Renato Cerqueira. *Manual de Utilização do IUP/CPI — Control Programm Interface*. TeCGraf - Grupo de Tecnologia em Computação Gráfica (PUC-Rio), Rio de Janeiro, Brasil, 1996. on-line manual at `http://www.icad.puc-rio.br/~rcerq/iup/`.

[Chu94] Steve Churchill. Fresco tutorial. *C++ Report*, 6(8):17–21, October 1994.

[CI95] A. Carregal e R. Ierusalimschy. Tche — a visual environment for the Lua language. Em *VIII Simpósio Brasileiro de Computação Gráfica*, páginas 227–232, São Carlos — SP, Brasil, 1995.

[CILS93] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, e T. M. Stepien. Abstract data views. *Structured Programming*, 14(1):1–13, 1993.

[Cli95] André Luiz Clinio. TEXT — um manipulador abstrato de textos lua. Projeto final de programação, PUC-Rio, Rio de Janeiro, Brasil, 1995.

[CPP92] Matthew Conway, Randy Pausch, e Kimberly Passarella. A Tutorial for Suit - The Simple User Interface Toolkit. Virginia, United States of America, 1992. `http://www.cs.virginia.edu/~suit/`.

[dC96] André Oliveira da Costa. Tk/VIX — um toolkit de interfaces lua (*título provisório*). Dissertação de mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro — RJ, Brasil, 1996. *documento em fase de confecção até a publicação deste trabalho.*

[FFG95] Waldemar Celes Filho, Luiz Henrique Figueiredo, e Marcelo Gattass. EDG: Uma ferramenta para criação de interfaces gráficas interativas. Em *VIII Simpósio Brasileiro de Computação Gráfica*, páginas 241–248, São Carlos — SP, Brasil, 1995.

[Fil94] Waldemar Celes Filho. *ED — Manual do Usuário.* TeCGraf - Grupo de Tecnologia em Computação Gráfica (PUC-Rio), Rio de Janeiro, Brasil, 1994.

[FvDFH91] J. D. Foley, A. v. Dam, S. K. Feiner, e J. F. Hughes. *Computer Graphics: principles and practice.* Addison-Wesley, 1991.

[Gor95] Tomas Guisasola Gorham. IUP/LUA — uma interface lua para o sistema IUP. Monografias em ciência da computação, PUC-Rio, Rio de Janeiro — RJ, Brasil, 1995.

[GR83] A. Goldberg e D. Robson. *Smalltalk-80 : The Language and its Implementation.* Addison-Wesley, 1983.

[IdFF96] R. Ierusalimschy, L.H. de Figueiredo, e W. Celes F. Lua: An extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[IFC95] R. Ierusalimschy, L. H. Figueiredo, e W. Celes. Reference manual of the programming language Lua version 2.1. Monografias em Ciência da Computação 08/95, PUC-Rio, Rio de Janeiro – RJ, Brazil, 1995. (available by ftp at `ftp.inf.puc-rio.br/pub/docs/techreports`).

[Inc94] Borland International Inc. Borland C++ 4.5 — Object Windows Library Help, 1994.

[LdFG+96] Carlos Henrique Levy, Luiz Henrique de Figueiredo, Marcelo Gattass, Carlos Lucena, e Don Cowan. IUP/LED: a portable user interface development tool. *Software: Practice & Experience*, 26(7):737–762, 1996.

[MAL88] J. M. Vlissides M. A. Linton, P. R. Calder. InterViews: a C++ graphical interface toolkit. Technical Report CSL-TR-88-358, Stanford University, 1988.

[Mey88] B. Meyer. *Object Oriented Software Construction.* Prentice Hall, 1988.

[Mic95a] Microsoft. *Microsoft Foundation Class Library Reference*, volume 3. Microsoft Press, second edition, 1995.

[Mic95b] Microsoft. *Microsoft Foundation Class Library Reference*, volume 4. Microsoft Press, second edition, 1995.

[MR93] A. Morse e G. Reynolds. Overcoming current growth limits in UI development. *Communications of the ACM*, 36(4):73–81, 1993.

[Nye90] A. Nye. *The Definitive Guides to the X Window System: Xlib Programming Manual for Version 11.* O'Reilly & Associates, Inc, 1990.

[Obj93] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, October 1993.

[Ope91] Open Software Foundation, New Jersey, USA. *OSF/Motif Programmer's Guide*, 1991.

[Ous94] J. Ousterhout. *Tcl and Tk Toolkit.* Addison-Wesley, 1994.

[PCD92] Randy Pausch, Matthew Conway, e Robert DeLine. Lessons Learned from SUIT - The Simple User Interface Toolkit. *ACM Transactions on Office Information Systems*, 10(4):320–344, October 1992.

[Pet] Chris D. Peterson. Athena Widget Set — C language interface. parte da documentação do X11R6 — `ftp://ftp.x.org/pub/R6.1/xc/doc/hardcopy/Xaw/widgets.PS.Z`.

[Pet90] C. Petzold. *Programming Windows: the Microsoft Guide to Writing Aplications for Windows 3.* Microsoft Press, 1990.

[Pet96] Charles Petzold. *Programming Windows 95: The Definitive Developer's Guide to the Windows 95 API.* Microsoft Press, fourth edition, 1996.

[PL94] D. Pan e L. Linton. Dish: A dynamic invocation shell for FRESCO. Em *Proceedings of the 1994 Tcl/Tk Workshop*, 1994.

[Rez95] Neil Armstrong Rezende. GLB: Uma ferramenta para manipulação de objetos gráficos procedurais. Dissertação de mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro — RJ, Brasil, 1995.

[SG86] R. Scheifler e J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[Str92] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, second edition, 1992.

[TeC89] TeCGraf - Grupo de Tecnologia em Computação Gráfica (PUC-Rio), Rio de Janeiro, Brasil. *Manual de Utilização do GKS/puc versão 3.0*, 1989.

[TeC93] TeCGraf - Grupo de Tecnologia em Computação Gráfica (PUC-Rio), Rio de Janeiro, Brasil. *PGM — Programa Gráfico Mestre (Manual do Usuário)*, 1993.

[TeC94] TeCGraf - Grupo de Tecnologia em Computação Gráfica (PUC-Rio), Rio de Janeiro, Brasil. *TECDRAW - Manual do Usuário*, 1994.

[TeC96a] TeCGraf - Grupo de Tecnologia em Computação Gráfica (PUC-Rio). *G3D — Uma Biblioteca Gráfica para Visualização Tridimensional Baseada em OpenGL*, 1996.

[TeC96b] TeCGraf - Grupo de Tecnologia em Computação Gráfica (PUC-Rio), Rio de Janeiro, Brasil. *IMO (*manual ainda não disponível*)*, 1996.

[TeC96c] TeCGraf - Grupo de Tecnologia em Computação Gráfica (PUC-Rio), Rio de Janeiro, Brasil. *Manual de Utilização do Canvas Draw versão 1.0*, 1996.

[TeC96d] TeCGraf - Grupo de Tecnologia em Computação Gráfica (PUC-Rio), Rio de Janeiro, Brasil. *SigDraw — Manual do Usuário*, 1996.

[Ude94] J. Udell. Componentware. *Byte*, 19(5):46–56, 1994.

[Wal93] Jim Waldo, editor. *The Evolution of C++ — Language Design in the Marketplace of Ideas*. USENIX — The MIT Press, 1993.

VIX - Um Framework para Suporte a Objetos Visuais Interativos

Dissertação de Mestrado apresentada por **André Luiz Soares Clinio dos Santos**, no dia, ao Departamento de Informática da PUC-Rio e aprovada pela Comissão Julgadora, formada pelos seguintes professores:

Prof. Roberto Ierusalimschy
Orientador
Departamento de Informática - PUC-Rio

Prof. Marcelo Gattass
Departamento de Informática - PUC-Rio

Profa. Clarisse Sieckenius de Souza
Departamento de Informática - PUC-Rio

Visto e permitida a impressão.
Rio de Janeiro, de de 1996.

Coordenador dos Programas de Pós-Graduação e
Pesquisa do Centro Técnico Científico