Carlos Henrique Levy

IUP/LED: Uma Ferramenta Portátil de Interface com Usuário

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

Orientador: Carlos J. P. de Lucena
Co-orientador: Luiz Henrique de Figueiredo

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 26 de Agosto de 1993

À Neisa, à minha filha Isabel
e ao amor que nos une.

Agradecimentos

Ao Prof. Lucena, pela orientação e pela segurança que me proporcionou.

Ao Luiz Henrique de Figueiredo, que participou efetivamente de todo este trabalho, desde o projeto conceitual do IUP/LED, passando pela implementação, até a revisão final desta dissertação, sendo imensurável a sua orientação e dedicação na realização deste trabalho.

Ao Prof. Marcelo Gattass, que sugeriu o tema da dissertação e me proporcionou todos os subsídios para a sua realização.

Ao Ênio Russo, pelas valiosas conversas sobre sistemas de interfaces com usuários e pela revisão minuciosa desta dissertação.

A todo o pessoal do TeCGraf, em especial ao André, que me ajudou na implementação do IUP/LED; ao Waldemar, pela troca permanente de idéias; ao Luiz Cristovão, que foi o primeiro usuário do IUP/LED; ao Camilo Freire, pela amizade; ao Prof. Roberto Ierusalimschy, pelas sugestões apresentadas sobre o algoritmo do cálculo do *layout* concreto.

A Ingrid A. Zech, engenheira do CENPES, pela sua ajuda em descobrir falhas de implementação no IUP/LED.

Ao Prof. Don Cowan, pela gentileza de me trazer alguns trabalhos do Canadá e pela conversa sobre sistemas portáteis de interfaces com usuários.

À minha família, que sempre incentivou meus estudos.

Ao apoio financeiro do CENPES e da Capes.

Resumo

O objetivo desta dissertação é descrever e analisar o projeto e implementação do sistema portátil de interface com usuário IUP/LED, do ponto de vista de portabilidade e de definição de *layouts*. A estratégia de portabilidade adotada pelo IUP/LED permite que as aplicações escolham herdar (ou não) o *look-and-feel* do sistema de interface nativo. Para a especificação de *layouts*, o sistema IUP/LED implementa um modelo de *layout* abstrato baseado no paradigma de *boxes-and-glue* do processador de texto TEX.

Abstract

The objective of this work is to describe and analyze the design and implementation of a portable user interface system called IUP/LED, from the point of view of portability and layout specification. The portability strategy adopted in IUP/LED allows applications the choice of whether to inherit native look-and-feel. For layouts specification IUP/LED implements an abstract model based on the boxes-and-glue paradigm of TEX.

Sumário

Lista de Abreviaturas e Siglas

ACM: *Association for Computing Machinery*

ANSI: *American National Standards Institute*

API: *Application Programmer's Interface*

CGM: *Computer Graphics Metafile*

CIRL: *Coordinate-Independent Resource Language*

DIN: *Deutsches Institut für Normen*

GKS: *Graphical Kernel System*

GL: *Silicon Graphics Graphics Library*

GUI: *Graphical User Interface*

HPGL: *HP Graphics Language*

IntGraf: *Interface Gráfica com Usuário*

ISO: *International Standards Organization*

IUP: *Interface com Usuário Portátil*

LED: *Linguagem de Especificação de Diálogos*

OSF: *Open Software Foundation*

PEX: *PHIGS Extension forXwindows*

PHIGS: *Programmer Hierarchical Interactive Graphical System*

PIWI: *Presentation-Independent Windowed Interface*

SDK: *Software Development Kit*

TeCGraf: *Grupo de Tecnologia em Computação Gráfica*

UIDS: *User Interface Development System*

UIL: *User Interface Language*

UIMS: *User Interface Management System*

UIT: *User Interface Toolkit*

WIMP: *Windows, Icons, Menu and Pointing device*

WYSIWYG: *What You See Is What You Get*

1

Introdução

Neste capítulo, apresentamos inicialmente um pouco da história de interfaces com usuários, dando enfoque ao problema de portabilidade de aplicações gráficas interativas ocasionado pela falta de um padrão de fato para sistemas de interfaces com usuários. Em seguida, apresentamos a motivação para esta dissertação. Por último, apresentamos o objetivo e a estrutura da dissertação.

1.1 O Problema

Até o final da década de 70, os computadores eram utilizados principalmente para resolver problemas de engenharia e problemas administrativos. A preocupação dos programadores concentrava-se na resolução dos problemas, ficando a interação com o usuário em segundo plano [Gattass–Levy (1992)]. A comunicação dos programas com os usuários era bastante rudimentar: a entrada dos dados era geralmente feita através de arquivos textos freqüentemente formatados segundo a conveniência dos programas e não dos usuários, onde o usuário utilizava um editor de texto linha a linha; os resultados de saída eram apresentados em relatórios.

A interação com o usuário estava em segundo plano devido às limitações impostas pelos recursos computacionais da época, que eram limitadas quanto à memória e aos tipos de dispositivos gráficos. Com o avanço tecnológico, estas limitações foram deixando de existir e o uso dos computadores foi se difundindo para outras áreas diferentes da administração e da engenharia, até alcançar o uso doméstico, com a difusão dos micro-computadores. A partir daí, os computadores passaram a ser limitados não pela sua capacidade computacional, mas sim pela capacidade dos programas de se comunicar com os usuários.

No final da década de 70, o centro de pesquisa da Xerox criou o sistema Star baseado na metáfora de uma mesa de trabalho (*desktop*) para fazer a interação homem-máquina. Após conhecer o sistema, em 1984, Steve Jobs criou o sistema de interface que roda, até hoje, nos Macintosh da Apple. O sistema projetado por Steve Jobs também utiliza a metáfora de uma mesa de trabalho, além de utilizar extensivamente recursos gráficos; suas principais características são:

- um dispositivo para apontar, tipicamente um *mouse*;
- menu de barra, que pode aparecer e desaparecer sob controle do *mouse*;
- janelas que exibem o que o computador está fazendo;
- ícones que representam arquivos e diretórios (*folders*);
- caixas de diálogos, botões e muitos outros elementos gráficos de interface representando técnicas de interação.

O sistema do Macintosh popularizou o uso das interfaces gráficas com usuário (GUI) fazendo surgir outros sistemas como: Macintosh Toolbox, MS-Windows, Presentation Manager, Motif, Open Look, HP-NewWave, Gem. Alguns destes sistemas disputam ainda hoje a condição de "padrão de fato".

A grande diversidade e a falta de um padrão de fato para sistemas de interface criam um problema para os programadores que precisam decidir qual sistema deve ser utilizado. A escolha de um sistema errado pode limitar o potencial comercial de uma aplicação a uma determinada plataforma. Por outro lado, se o sistema escolhido se tornar um padrão em GUI, o número de potenciais usuários aumenta e o programa pode ficar a um passo do sucesso.

Podemos ilustrar a dificuldade em escolher um sistema de interface observando a disputa das duas maiores potências do mundo da micro-informática: Microsoft e IBM. Elas trabalham para tornar seus novos sistemas operacionais, Windows-NT e OS/2 respectivamente, como padrão de fato. Apesar da disputa ser sobre sistemas operacionais, cada um tem sua própria GUI para micro-computadores. Conseqüentemente, o sistema operacional que for vencedor, levará junto o sistema de interface. A Microsoft está tão otimista em relação ao seu novo sistema que já diz que o Windows-NT é o *unix-killer*, isto é, está pretendendo entrar no mercado das *workstations* RISC, onde o Unix é completamente absoluto. A comunidade GUI espera ansiosamente pelo fim dsta disputa, e no momento seu desfecho ainda é imprevisível.

A solução para os desenvolvedores de programas é escolher não um sistema de interface, e sim vários. Contudo, programar um aplicativo interativo que suporte vários sistemas de interface é uma tarefa penosa, pois a experiência nos mostra que aproximadamente metade do código de um programa é responsável pela interação com usuário (Marcus–van Dam [1991]). Isto se deve principalmente à necessidade de controlar vários dispositivos de entrada simultaneamente (e.g., *mouse* e teclado) e de atender às necessidades cognitivas dos usuários. Nota-se que os principais responsáveis pela grande quantidade de código são dependentes da plataforma computacional. Por exemplo, o tratamento do teclado e do *mouse* no IBM-PC/DOS difere do Unix/X11, que

também difere do IBM-PC/MS-Windows. Da mesma maneira, as primitivas gráficas necessárias na geração de um *feedback* são diferentes entre as plataformas. Isto implica na existência de um conjunto de funções de interface diferente para cada ambiente, aumentando significativamente o trabalho de documentação e manutenção. Sem uma ferramenta de interface adequada, o programador da aplicação acabará deixando de lado o objetivo principal do programa para se preocupar com objetos de interface nos diferentes ambientes computacionais.

Existem duas grandes classes de ferramentas de interface: *User Interface Toolkits* (UIT) e *User Interface Management Systems* (UIMS) [Hartson–Hix (1990)]. Um UIT é uma biblioteca de objetos de interface que implementam diferentes técnicas de interação com o usuário. Ferramentas nesta classe estão disponíveis como um conjunto de funções que devem ser chamadas pela aplicação para gerar e controlar o diálogo com o usuário. Alguns exemplos de UIT são: XView [Heller (1990)]; OSF/Motif [OSF (1990)]; OLIT [Sun (1990)]; SDK for MS-Windows 3.1 [Petzold (1990)] e Macintosh User Interface Toolbox [Apple (1985)]. Os UIT freqüentemente oferecem alguns mecanismos para facilitar, essencialmente, a descrição e composição de objetos de interface. Estes mecanismos variam desde simples linguagens de descrição (*resource languages*) até editores gráficos com interfaces por manipulação direta. É importante ressaltar que a gerência da interação com o usuário tem que ser programada pela aplicação, ainda que através do *toolkit*

Um UIMS é conjunto de programas interativos de alto nível para projetar, prototipar, executar, avaliar e manter interfaces com usuários, tudo integrado sob uma interface de desenvolvimento de diálogos simples [Hartson–Hix (1989)]. Alguns exemplos de UIMS são: University of Alberta UIMS [Green (1985)]; DMS [Hartson et al. (1984)]. Os UIMS encapsulam os UIT, permitindo, além da descrição e composição dos objetos de interface, a especificação do controle da seqüência de interação com usuário [Marcus (1991)]. Os UIMS pressupõem que o desenvolvimento de uma aplicação seja feito por dois especialistas trabalhando em conjunto: um na área do problema em questão e outro na área de interface com usuário. O primeiro especialista resolve o problema computacional da aplicação. O segundo se preocupa com fatores humanos, relacionados com aspectos psicológicos, cognitivos, ergonométricos e linguísticos, de modo a proporcionar uma verdadeira interação entre usuário e aplicação [Hartson–Hix (1990)].

Muitos dos sistemas de interface comerciais, como o Visual Basic [Microsoft (1992)], suportam apenas a construção de diálogos, não permitindo a definição do controle da seqüência de interação, o qual tem que ser programado pela aplicação.

Portanto, estes sistemas não podem ser classificados como UIMS, apesar de serem sistemas integrados. O trabalho de Figueiredo et al. (1992) propõe uma ferramenta para geração automática de interfaces para a captura de dados sobre desenhos para programas de simulação e otimização em engenharia. Esta ferramenta, apesar de não ser uma aplicação integrada em um mesmo ambiente, pode ser considerada um UIMS, pois suporta todos os aspectos de um projeto de interface com usuário.

Recentemente, uma nova geração de UIMS, integrados com bases de conhecimentos [Hix 1990], está surgindo: são os UIDS - *User Interface Development Systems*. Os UIDS utilizam uma base de conhecimento sobre técnicas de projetos de interfaces e princípios de projeto de *software* que auxiliam o processo de especificação da interface [Lucena et al. (1990)]. Esta integração permite que o próprio usuário do programa seja o especialista de interface, ao contrário dos UIMS, que necessitam de um especialista em sistema de *software.*

Estas ferramentas auxiliam bastante na construção de programas interativos, mas não abordam dois aspectos importantes do problema:

- não dão suporte para aplicações multi-plataforma. As ferramentas geralmente são específicas para um determinado ambiente computacional;
- não oferecem um modelo para definição de *layout* que seja natural, exigindo que o usuário defina coordenadas e tamanhos para cada objeto de interface. Isto pode ser minimizado utilizando editores gráficos interativos para construção de *layouts* por manipulação direta. Entretanto, estes editores não retiram da aplicação a responsabilidade de recalcular as posições e tamanhos de cada objeto de interface, quando o usuário final promove uma alteração no tamanho do diálogo.

Como os UIMS são desenvolvidos sobre os UIT, a construção de um UIT portátil e que possua um modelo *layout* abstrato é a base para a construção de ferramentas de interface com usuário que contemplem estes importantes aspectos.

1.2 Motivação

O grupo TeCGraf desenvolve, entre outras coisas, programas gráficos interativos para diversas áreas da engenharia. O principal cliente do TeCGraf é a PETROBRÁS, que possui um parque de computadores bastante diversificado. Esta diversificação exige que as aplicações desenvolvidas pelo TeCGraf sejam multi-plataforma, isto é, possam ser executadas em diferentes ambientes computacionais.

Para resolver o problema de portabilidade das rotinas gráficas, o Centro de Pesquisas da PETROBRÁS (CENPES) solicitou ao TeCGraf a utilização do padrão internacional para sistemas gráficos GKS [ANSI (1985)]. Como esta decisão foi tomada em 1986, em paralelo à homologação do GKS pela ANSI, o CENPES e o TeCGraf resolveram absorver por completo a tecnologia de sistemas gráficos e desenvolveu o GKS/puc [TeCGraf (1989)], seguindo rigorosamente as normas ANSI e ISO.

Para sistemas de interfaces não havia – e ainda não há – padrões internacionais. Desta forma, o TeCGraf acabou criando o IntGraf [TeCGraf (1991)], um sistema portátil de interface gráfica sobre o GKS/puc. O IntGraf resultou de um sistema de interface desenvolvido especialmente para o programa FMAT-2D, que está ilustrado na Figura 1. Como este programa teve uma boa aceitação dos usuários, o TeCGraf decidiu transformar as rotinas de interface do FMAT-2D em um sistema de interface: o IntGraf.

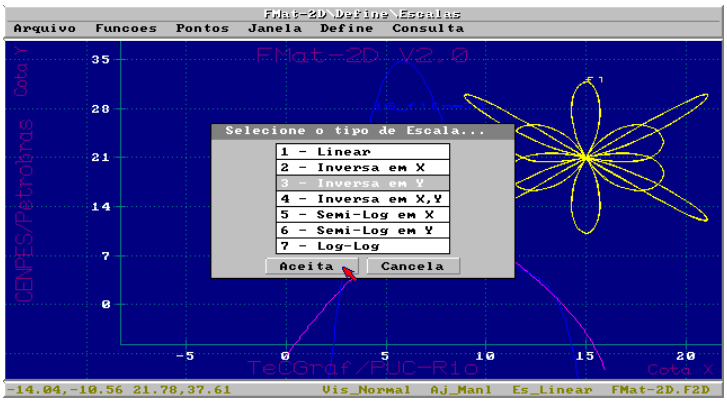
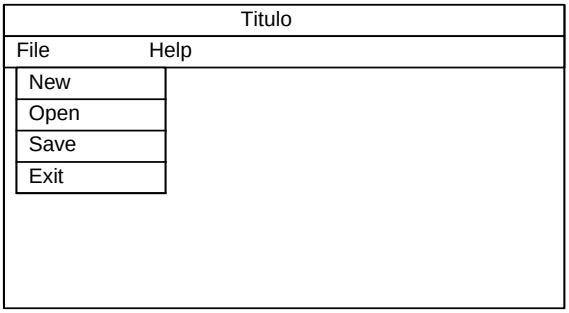


Figura 1: interface do FMAT-2D

O IntGraf é composto de um gerador de menu – `criamenu` – e de um *toolkit* de objetos de interface gráfica com aparência 3-D, que possibilitam a interação com o usuário de uma forma clara e harmoniosa. O `criamenu` gera o código de programa necessário para a construção do menu de barra no topo da tela (Figura 2-a), a partir de uma descrição hierárquica, sob a forma de árvore (Figura 2-b). As folhas da árvore representam funções da aplicação, que são ativadas quando são selecionadas. A árvore de menu pode ser definida e experimentada, antes mesmo de se escrever as funções da aplicação, permitindo assim, a prototipagem rápida de interfaces.



(a) aparência

```
#funcao_de_inicializacao
#funcao_de_finalizacao
#funcao_de_tela
Titulo                #info {
    File {
    new         #fnew
    open        #fopen
    save        #fsave
    exit        #fexit }
    Help              #fhelp )
```

(b) descrição textual

Figura 2: descrição do menu de barra no IntGraf.

O IntGraf foi desenvolvido para ser simples e permitir um rápido aprendizado por parte do programador da aplicação. A biblioteca de funções de interface contém somente o conjunto mínimo necessário para geração de diálogos simples, porém poderosos. Os seguintes objetos estão disponíveis:

- menu de barra;
- botão;
- entrada de dados, um campo por linha;
- seleção única/múltipla;
- seleção de arquivo;
- alarme;
- mensagem.

Atualmente, o IntGraf está instalado em vários ambientes, destacando-se: VAX/VMS, PC-DOS, Unix/X11, IBM/MVS. Todas as instalações possuem o mesmo código, escrito em C, com exceção do IntGraf para o IBM/VMS, o qual foi todo re-escrito, em FORTRAN, para contemplar eficientemente os terminais 3278 do tipo *cell-units*. Esta versão do IntGraf está descrita em Derraik (1992).

O IntGraf conseguiu alcançar os seus objetivos, garantindo assim o seu sucesso perante os programadores da aplicação e os usuários finais. Porém, com o seu uso difundido, surgiram vários pedidos de alteração e inclusão de novas funções, tais como:

- composição de objetos de interface;
- criação de novos objetos de interface (e.g., imagens, cursores);
- maior acesso ao teclado;
- gerenciamento de *help*;
- permitir o *look-and-feel* nativo além do *look-and-feel* fixo.

Alguns destes pedidos são simples de serem implementados e alteram pouco a estrutura interna do IntGraf, mas alguns outros, como compor objetos de interface e permitir *look-and-feel* nativo, não podem ser realizados sem uma re-estruturação na base do IntGraf. Esta dificuldade ocorre porque o IntGraf foi retirado do FMAT-2D e não incluiu um projeto conceitual de interface com usuário na sua elaboração. Isto obrigou o desenvolvimento de um novo projeto de sistema portátil de interface com usuário; esta dissertação descreve o projeto e a implementação deste novo sistema.

Um ponto importante que deve ser observado é o prejuízo da íntima ligação do IntGraf com o GKS/puc. Embora o IntGraf tenha sido desenvolvido para dar suporte a aplicações gráficas, o seu sucesso perante os programadores e os usuários finais fez com que aplicações não gráficas, basicamente *front-ends* para aplicações de bancos de dados, passassem a utilizá-lo como ferramenta de interface. Estas aplicações, por não utilizarem diretamente o sistema gráfico, ficam penalizadas com a carga de código desnecessária que o GKS/puc acrescenta nas suas aplicações. Não obstante, como é mostrado no próximo capítulo, apesar do GKS/puc ser um padrão internacional, ele não se tornou um padrão de fato, e outros sistemas gráficos estão ocupando o seu lugar. Com a ausência de um único sistema gráfico universalmente aceito, a separação do sistema de interface do sistema gráfico se tornou um pré-requisito importante neste novo projeto.

1.3 Objetivo

O objetivo desta dissertação é descrever o projeto e a implementação de um sistema portátil de interface com usuário (IUP/LED) que substitui o sistema atual (IntGraf) utilizado pelo TeCGraf e analisar comparativamente os dois sistemas.

O IUP/LED foi projetado para solucionar os problemas encontrados com o IntGraf, preservando as suas duas principais qualidades: facilidade de uso, tanto pelos programadores quanto pelos usuários finais da aplicação; e portabilidade para diferentes ambientes computacionais. As principais características do IUP/LED são:

- permitir composição de diálogos, através da descrição de um *layout* abstrato;
- ter uma linguagem de especificação de diálogos (LED) compilada em tempo de execução da aplicação;
- permitir tanto *look-and-feel* nativo quanto *look-and-feel* fixo;
- separar o sistema de interface do sistema gráfico;
- ser expansível.

1.4 Estrutura da dissertação

Neste Capítulo 1, apresentamos os problemas, as motivações e os objetivos desta dissertação, além de descrever a estrutura de capítulos desta monografia.

No Capítulo 2, apresentamos a diferença entre ser portável e ser portátil. Em seguida, descrevemos e discutimos alguns problemas que dificultam a construção de programas portáteis. Também descrevemos estratégias de desenvolvimento, dando ênfase às estratégias de portabilidade para aplicações gráficas interativas. Finalmente, apresentamos várias estratégias de desenvolvimento de ferramentas de interface e discutimos a estratégia adotada no desenvolvimento do IUP/LED. Este capítulo é uma expansão do artigo de Figueiredo et al. (1993).

No Capítulo 3, definimos o que são diálogos e descrevemos os elementos de interface que se tornaram padrão de fato entre os sistemas de interface. Em seguida, analisamos, do ponto de vista de especificação de *layouts*, os sistemas de interface: Motif, MS-Windows, Macintosh Toolbox e XView. Por último, discutimos as formas de integração entre os diálogos e as aplicações.

No Capítulo 4, começamos apresentando os objetivos do *toolkit* IUP e da linguagem de especificação de diálogos (LED) e os conceitos envolvidos. Em seguida, descrevemos detalhadamente a LED e apresentamos superficialmente o IUP. Finalizando o capítulo, discutiremos os principais aspectos da implementação do IUP/LED. Deixamos para o capítulo seguinte uma análise crítica desta ferramenta portátil de interface com usuário.

No Capítulo 5, discutimos algumas experiências no uso do IUP/LED, apontamos algumas contribuições do IUP/LED para o desenvolvimento de programas portáteis, comparamos o IUP/LED com o IntGraf e com o CIRL/PIWI, um outro sistema portátil de interface com usuário desenvolvido na Universidade de Waterloo. Finalizando o capítulo, fazemos um resumo dos resultados da dissertação e propomos temas para trabalhos futuros.

2

Estratégias de portabilidade

Hoje em dia, uma importante característica desejada em um programa é que ele possa ser executado em diversos ambientes computacionais (e.g., MS-DOS, MS-Windows, OS/2, Macintosh, Motif/X11, Open Look/X11, IBM VM/CMS, VAX/VMS). Sendo estes ambientes bastante diferentes entre si, os programadores podem ter certa dificuldade em alcançar esta característica se não utilizarem ferramentas apropriadas para desenvolvimento de programas portáteis. Neste capítulo, discutiremos obstáculos e estratégias no desenvolvimento de programas portáteis, dando ênfase às aplicações gráficas interativas.

"Portar" um programa significa fazer as modificações necessárias para que este programa possa ser executado em um ambiente diferente do ambiente no qual o programa foi originalmente desenvolvido. Portar é mais que simplesmente "transportar", que significa, neste contexto, levar código e dados fisicamente de uma máquina para outra. (Isso pode não ser trivial, mesmo nestes tempos de redes e *open systems*.)

Estritamente falando, toda aplicação é portável: basta reprogramá-la, mantendo a funcionalidade. Entretanto, fazemos aqui a distinção entre "portável" e "portátil": "portável" significa poder ser portado; "portátil" significa poder ser *facilmente* portado. Este sentido técnico de "portátil" coincide com o sentido comum, que é ser facilmente transportado. Assim, um programa é portátil quando o esforço de reprogramação necessário para movê-lo para um novo ambiente computacional não é substancial [Cowan—Wilkinson (1984)].

Os principais obstáculos potenciais à portabilidade são [Blackham (1988)]:

- Diferenças de *hardware*: uma diferença bastante comum é a ordem dos *bytes* numa *word*. Alguns sistemas armazenam primeiro o *byte* menos significativo e depois o mais significativo (Sun Sparc) e outros (INTEL 80x86) na ordem inversa. Esta diferença deve ser levada em consideração nos projetos de arquivos binários.
- Diferenças dos sistemas operacionais: existem sistemas com (Unix) e sem (PC-DOS) multiprocessamento preemptivo, além dos sistemas que permitem multiprocessamento cooperativo (MS-Windows e Macintosh). Alguns sistemas de arquivo (Macintosh) não diferenciam letras maiúsculas das minúsculas, e outros sim (Unix). Até sistemas que seguem o mesmo padrão básico, como é o

caso do Unix-BSD e do Unix-SYSTEM V, possuem diferenças que, à primeira vista, parecem pequenas, mas que podem dificultar a migração de um sistema para o outro, se certos cuidados não forem tomados [Frey (1988)].

- Diferenças nos compiladores das linguagens de programação: no compilador C da Microsoft, o tamanho *default* do tipo `int` é de 16 bits, enquanto que no compilador da Watcom é de 32 bits. No desenvolvimento de programas que manipulam arquivos binários, esta diferença pode trazer sérias complicações. Outra diferença que pode passar desapercebida é que cada compilador oferece uma biblioteca de funções proprietárias, que não são compatíveis entre si. Estas bibliotecas devem ser usadas com cuidado para não penalizar a portabilidade do programa.
- Capacidade dos dispositivos gráficos: alguns dispositivos suportam cores (alguns permitem 256 cores simultâneas, outros apenas 16); alguns possuem co-processadores gráficos, enquanto outros apenas oferecem um mapa de *pixels*.
- API dos sistemas de interface utilizado pela aplicação: Como veremos mais adiante, os sistemas de interface possuem programação bastante diferentes, apesar do resultado final ser equivalente.

Uma discussão mais detalhada destes fatores foi dada por Durance (1990).

O impacto na mudança de sistemas operacionais e linguagens de programação pode ser minimizado usando-se padrões de fato, como Posix [IEEE (1988)] e ANSI C [Kernighan–Ritchie (1988)]. Se ainda assim houver código dependente, deve-se usar a estratégia mais simples de portabilidade: isolar o código dependente da aplicação e documentá-lo com relação à sua funcionalidade, para facilitar uma posterior implementação em outra plataforma.

Os problemas encontrados com a diversidade dos dispositivos gráficos e com os sistemas de interfaces são estudados com mais detalhes na próxima seção.

2.1 Portabilidade de aplicações gráficas

No início, os programas gráficos tinham acesso direto aos dispositivos, muitas vezes se comunicando com o *hardware* por meio de rotinas de “baixo nível”, cujas chamadas estavam espalhadas pelo código. Estes programas eram fortemente não portáteis, no sentido de Cowan–Wilkinson (1984). Rapidamente, notou-se que esta estratégia não era adequada para a criação de programas portáteis, pois não era simples usar outros dispositivos gráficos. As dificuldades principais eram que os dispositivos

oferecem serviços diferentes e a estrutura das aplicações era muito influenciada pelo dispositivo.

Seguindo o paradigma de programação estruturada, uma das soluções adotadas foi o isolamento das rotinas específicas de acesso ao dispositivo em módulos chamados *drivers*. Este isolamento não só torna a manutenção mais simples, como também implica na criação de um protocolo de comunicação entre a aplicação e o *driver*. Este protocolo, chamado *application programmer's interface* (API), induz uma metáfora potencialmente portátil para dispositivos gráficos. Este tipo de arquitetura para aplicações gráficas está ilustrado na Figura 2.1

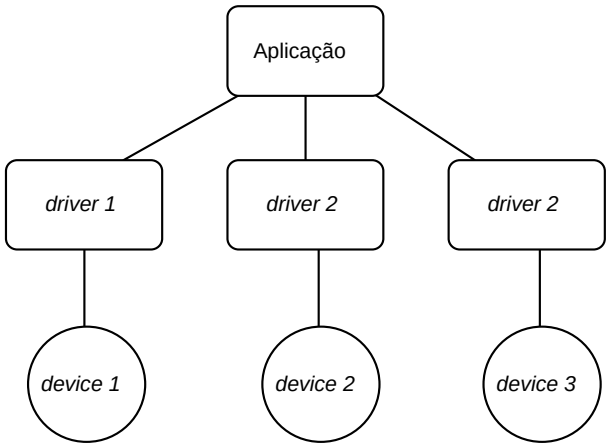


Figura 2.1: aplicações gráficas com *drivers*

Esta solução seria bastante razoável se os fabricantes de equipamentos distribuíssem os *drivers*, mesmo que as suas API's não fossem as mesmas, pois o desenvolvimento de um *driver* não é trivial. É preciso que o programador seja um especialista no equipamento que o *driver* irá controlar. Analisando o desenvolvimento de um programa gráfico que é executado apenas em micro-computadores compatíveis com IBM-PC sob o sistema operacional PC-DOS, veremos que, no mínimo, o programa deve suportar:

- placa de vídeo VGA;
- impressora matricial padrão EPSON;
- *mouse* como dispositivo de entrada.

Entretanto, é desejável que o programa também suporte:

- placa de vídeo SVGA (existem vários fabricantes, cada um com sua arquitetura proprietária);
- impressora Laser (Postscript e HPGL);
- padrão CGM, para permitir a exportação do desenho para outros programas.

Fica claro que o esforço dispendido pelo programador da aplicação em escrever os diferentes *drivers* não é pequeno; sem contar a manutenção e documentação que todos estes *drivers* vão exigir ao longo da existência da aplicação. Outro fator que torna esta estratégia difícil de ser estritamente seguida pelos programadores é o surgimento de novas tecnologias que precisam ser incorporadas à aplicação para que esta possa se manter atualizada e competitiva.

Antes do aparecimento dos micro-computadores, na pré-história da computação gráfica, o fabricante de *plotters* CalComp desenvolveu um *driver*, chamado de UHCBSS**,** para acessar qualquer um dos seus equipamentos. Este *driver* acabou se tornando, na época, um padrão de fato, que até hoje ainda é utilizado.

Para combater a proliferação de *drivers* incompatíveis entre si, foram criados grupos de trabalho no âmbito de organizações como ACM, ISO, DIN e ANSI para a elaboração de padrões internacionais, como CORE, GKS e PHIGS. Estes sistemas gráficos padrões não só forneciam uma metáfora portátil para dispositivos, como definiam a semântica de dispositivos virtuais, descrevendo o comportamento tanto da saída gráfica quanto dos métodos de entrada. A arquitetura de aplicações que usam sistemas gráficos está ilustrada na Figura 2.2.

No entanto, os padrões propostos não se tornaram padrões de fato, por vários motivos:

- as implementações dos padrões não tinham um desempenho adequado;
- os fabricantes de *hardware* não aceitaram mudar a interface com os seus produtos;
- era possível ter implementações parciais dos padrões;
- os padrões continham exceções e *escape functions* que permitiam utilizar melhor certos dispositivos, mas que acabavam por limitar a portabilidade.
- a metáfora de interação era rígida.

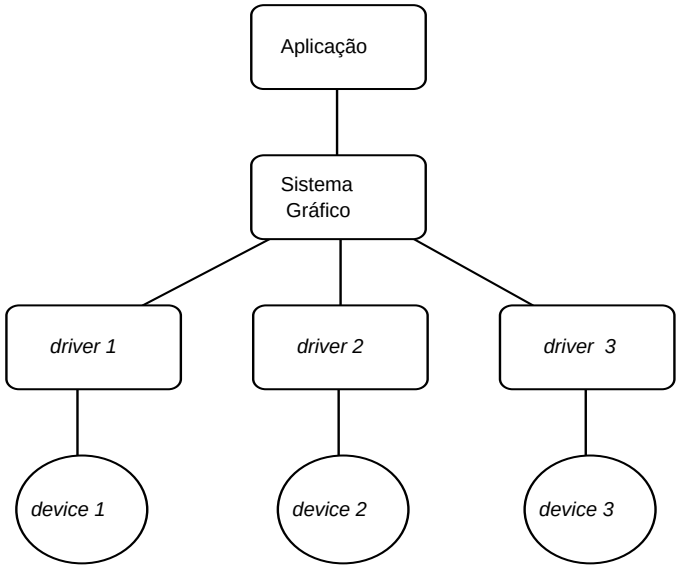


Figura 2.2: aplicações gráficas com sistemas gráficos

No início da década de 80, o que mais se aproximou de uma aceitação universal foi o **GKS-2D**. Entretanto, este sistema gráfico não se tornou um padrão de fato por que:

- as implementações eram muito grandes para os equipamentos da época;
- as duas transformações de ponto flutuante, devido aos seus dois sistemas de coordenadas, limitavam o seu desempenho;
- não contemplava aplicações tridimensionais.

Exatamente para tentar contemplar aplicações tridimensionais, surgiu em seguida uma proposta para um **GKS-3D**, logo superada pelo **PHIGS**. Entretanto, o **PHIGS** também não se estabeleceu como um padrão de fato. As principais razões foram:

- a falta de modelos de iluminação e de superfícies;
- uma séria incompatibilidade com a arquitetura do sistemas de janelas X, que é o padrão de fato para *workstations*, já que o núcleo do protocolo X não permite acesso às placas 3D, indispensável a um desempenho adequado.

Como resultado, surgiram o **PHIGS**+ (PHIGS *Plus Lumier und Surfaces*) e a extensão **PEX** para o protocolo X. O principal obstáculo para o sucesso do **PHIGS+** com a extensão **PEX** é a modelagem obrigatória de dados: no **PHIGS**, o programador

de aplicação é forçado a modelar as cenas tridimensionais numa estrutura que é interna ao sistema gráfico. Esta estrutura de dados não é nem necessária nem suficiente para um grande número de aplicações, que preferem sistemas gráficos que não requerem esta modelagem artificial, sendo simplesmente sistemas de desenho em três dimensões.

O exemplo típico destes sistemas de desenho tridimensionais é a **GL**, cuja plataforma nativa eram as máquinas da Silicon Graphics. Recentemente, a biblioteca **GL** foi revista numa tentativa de exportar esta arquitetura para outras plataformas, resultando na **OpenGL** [Neider et al. (1993)], que tem grandes chances de se tornar um padrão de fato para sistemas gráficos tridimensionais. Entretanto, **OpenGL** ainda está em fase de consolidação, com poucas implementações fora da plataforma Silicon Graphics.

Uma arquitetura alternativa para aplicações gráficas que manipulam modelos representados por estrutura de dados complexas é mostrada na Figura 2.3, na qual sistemas gráficos abstratos manipulam objetos usando diretamente a estrutura de dados da aplicação. Esta arquitetura é bastante adequada para aplicações bidimensionais; para aplicações tridimensionais, esta arquitetura só é viável se os sistemas gráficos abstratos estiverem estreitamente ligados às aplicações, pois um desempenho adequado ainda depende do *hardware*.

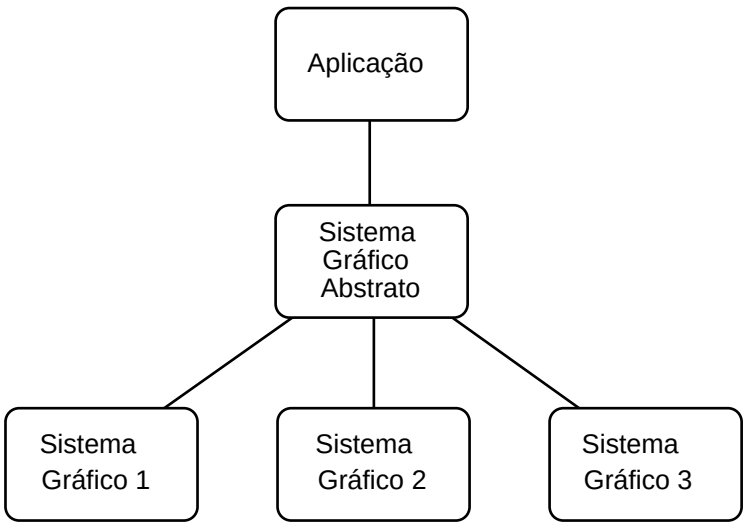


Figura 2.3: arquitetura com sistemas gráficos abstratos

2.2 Portabilidade de interfaces com usuários

A grande variedade de sistemas de interfaces com usuários (e.g., *Microsoft Windows*, *Presentation Manager*, *Macintosh ToolBox*, *Motif*, *Open Look*) torna difícil o desenvolvimento de programas interativos portáteis. Esta dificuldade não é imediatamente aparente, pois todos os sistemas de interfaces gráficas são do tipo *WIMP*, isto é, implementam a metáfora de *desktop* usando *windows*, *icons*, *menu* e *pointing device*. Assim, as aplicações têm uma aparência final (*look-and-feel*) semelhante nos vários sistemas. Entretanto, a programação de interfaces para aplicações nos vários ambientes, embora conceitualmente quase idêntica, se mostra, na prática, extremamente complexa devido às muitas diferenças nos vários *toolkits*, obrigando o programador da aplicação a ser um especialista em cada um dos sistemas.

Este problema é análogo ao problema de controle de dispositivos gráficos, que vimos na seção anterior, embora as seguintes diferenças devam ser notadas:

- embora os serviços básicos de interface estejam presentes em todos os sistemas, uma solução portátil deve ser capaz de prover *look-and-feel* "nativo" e não um somente *look-and-feel* fixo;
- o fluxo de informação em sistemas gráficos é principalmente uni-direcional — da aplicação para o dispositivo — enquanto o fluxo em sistemas de interface é por natureza bidirecional, já que a função da interface é exatamente intermediar a comunicação entre o usuário e a aplicação;
- a disposição dos objetos de interface é um problema de arranjo bidimensional, enquanto os objetos gráficos geralmente têm uma geometria herdada dos modelos mantidos pela aplicação.

Apesar disso, é possível copiar a solução clássica e usar *drivers* para controlar os diversos sistemas de interface, já que os objetos básicos são equivalentes nas várias plataformas. Porém, esta solução obriga os desenvolvedores de aplicações a criarem uma API própria e desenvolver os *drivers* para os sistemas de interface que pretendem suportar. Se escrever um *driver* para um dispositivo gráfico já é um trabalho grande, o desenvolvimento para um sistema de interface é bem maior, tornando inviável a adoção desta solução.

A solução ideal seria utilizar um sistema de interface que fosse padrão de fato. Como esse sistema ainda não existe, a solução, então, seria utilizar um padrão internacional, mas que também não existe. A única saída do desenvolvedor de aplicativos para evitar a dependência de plataformas computacionais é utilizar

ferramentas proprietárias de interfaces portáteis (e.g., IUP/LED, CIRL/PIWI [Cowan et al. (1992)], XVT [Rochkind (1989)]). A desvantagem desta solução é que a aplicação, apesar de se tornar independente de dispositivo, fica dependente do fabricante da ferramenta utilizada, já que esta, não sendo padrão internacional, não pode ser desenvolvida por outros fabricantes.

Existem várias estratégias para construir uma ferramenta de interface portátil. A forma mais simples é desenvolver uma ferramenta que seja o denominador comum entre todos os sistemas de interfaces suportados. Primeiramente, estudam-se todos os sistemas de interface, e somente as características comuns a todos são implementadas na ferramenta. Apesar de serem de fácil construção, as aplicações que utilizam esta estratégia geralmente possuem uma interface pobre, pois recursos de cor e fonte de caracteres podem não estar disponíveis. Uma outra desvantagem é que a aplicação tem que programar mecanismos sofisticados, que não aparecem em todos os sistemas de interface, como, por exemplo, a *file selection box*.

Uma estratégia um pouco mais sofisticada é portar um sistema de interface para diferentes ambientes, sem contudo utilizar o sistema de interface nativo. Uma característica desta solução é que as aplicações ficam com o mesmo *look-and-feel* em todos os ambientes computacionais. Isto pode ser vantajoso quando o usuário utiliza a mesma aplicação em diferentes ambientes, mas quando a aplicação é utilizada por um usuário em uma única máquina, o *look-and-feel* da aplicação provavelmente não será consistente com o *look-and-feel* das outras aplicações da máquina, o que poderá confundir o usuário, levando-o a uma má utilização da aplicação, aumentando sua frustração e diminuindo sua produtividade. A arquitetura desta estratégia está ilustrada na Figura 2.4.

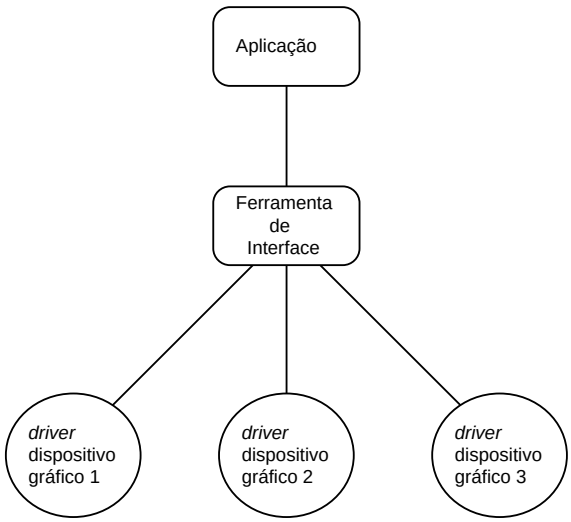


Figura 2.4: arquitetura de uma ferramenta que porta um sistema de interface.

A criação de um *toolkit* virtual, que implementa uma metáfora portátil de interface com usuário, é uma boa estratégia no desenvolvimento de ferramentas portáteis. Um *toolkit* virtual mapeia os elementos de interface nos elementos do sistema nativo do ambiente em que ele está sendo executado. Desta forma, a aplicação herda o *look-and-feel* do sistema nativo. Esta solução resolve o problema dos usuários de uma máquina só, aqueles que executam várias aplicações em uma mesma máquina, pois todas as aplicações terão o mesmo *look-and-feel*. Com isso, os usuários podem utilizar, em outras aplicações, técnicas já aprendidas em uma aplicação, aumentando sua motivação e conseqüentemente sua produção. Em contrapartida, o usuário de uma única aplicação em diferentes máquinas será prejudicado, pois ele terá que aprender o funcionamento do programa em vários ambientes, podendo confundi-lo. Isto também pode levar a uma má utilização da aplicação e a um decréscimo de produtividade. A arquitetura de um *toolkit* virtual é ilustrada na Figura 2.5

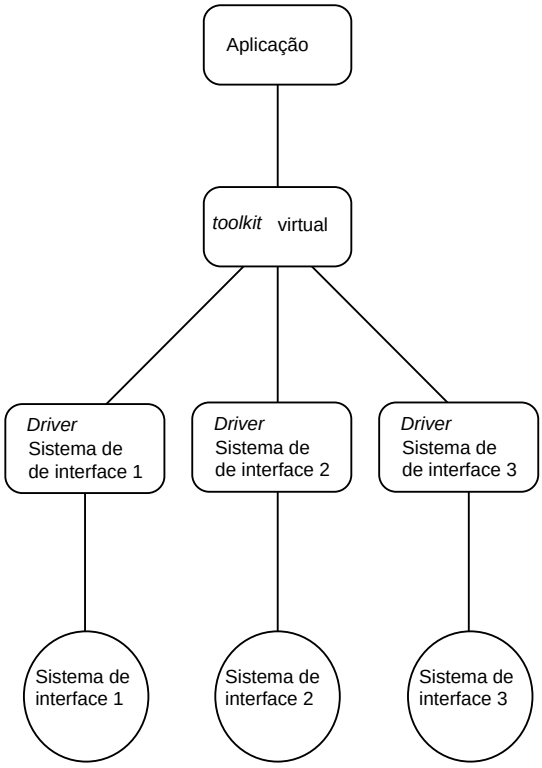


Figura 2.5: arquitetura de um *toolkit* virtual.

O problema principal desta estratégia é criar a metáfora portátil. Uma tal metáfora é dada pelo *toolkit* IUP (Interface com o Usuário Portátil), que descreveremos mais adiante no Capítulo 4. Este *toolkit* permite tanto *look-and-feel* nativo quanto *look-*

and-feel fixo, pois, além de *drivers* para os vários sistemas de interface disponíveis, foi escrito um sistema de interface totalmente portátil. Como um sistema de interface é uma aplicação puramente gráfica, podemos aplicar as estratégias, descritas na seção anterior, na construção deste sistema portátil. Desta forma, combinamos a estratégia de implementar um *toolkit* virtual com a estratégia de portar um sistema de interface. A arquitetura de uma aplicação IUP é ilustrada na Figura 2.6; o ramo destacado representa a estratégia de portar um sistema de interface totalmente.

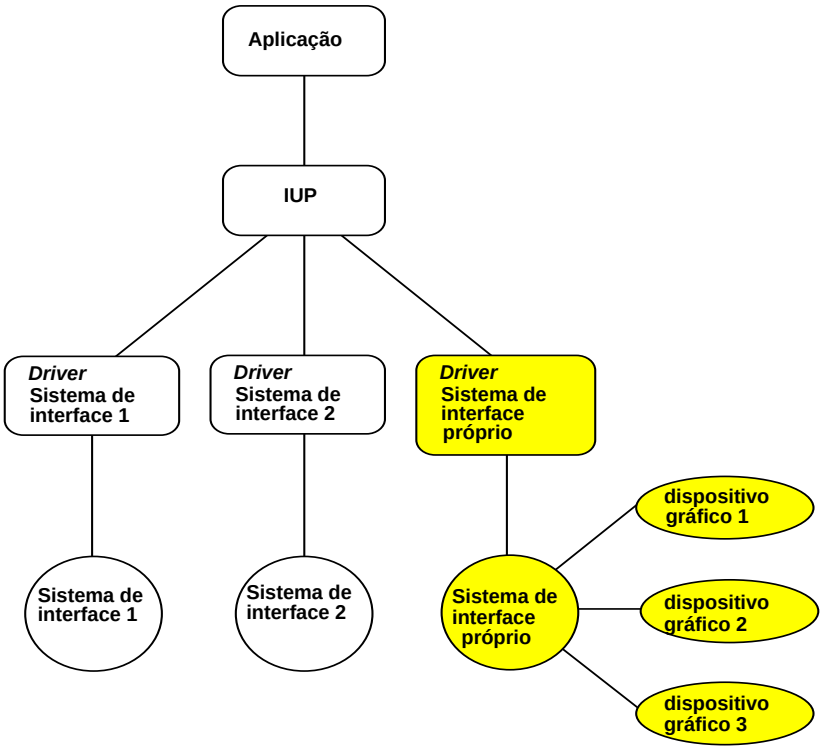


Figura 2.6: arquitetura de uma aplicação IUP.

3

Diálogos

Um diálogo é “a troca de informação entre o usuário e o sistema, dentro de um contexto espacial limitado” [Marcus (1992)]. Por abuso de linguagem, chamaremos de “diálogo” ao grupo de objetos de interface que estão no “contexto espacial limitado”.

Um aspecto de fundamental importância na programação de interfaces com o usuário é a descrição do *layout* de diálogos: qual é o método pelo qual os elementos de interface são criados, agrupados e dispostos geometricamente em diálogos?

Neste capítulo, apresentamos e discutimos os elementos básicos de interação e os métodos de descrição de *layouts* de diálogos. Estes métodos variam desde descrição textual até programas interativos que especificam diálogos por manipulação direta. Finalizando o capítulo, discutiremos as formas de integração entre os diálogos e a aplicação.

3.1 Elementos de Interface

Elementos (ou objetos) de interfaces são entidades que de alguma forma interagem com o usuário, seja informando alguma resposta da aplicação, através de som, imagem ou texto; ou capturando um dado, através do teclado, do *mouse*, ou da voz. Não iremos descrever todos os elementos de interfaces existentes, mas somente aqueles que se tornaram um padrão de fato entre os *toolkits* estudados: Motif, Open Look/XView, MS-Windows e Macintosh Toolbox.

3.1.1 *Dialog*

Este elemento é uma área retangular da tela que contém outros elementos de interface e os gerencia na interação entre a aplicação e o usuário final. A Figura 3.1 ilustra uma tela com vários *dialogs.*

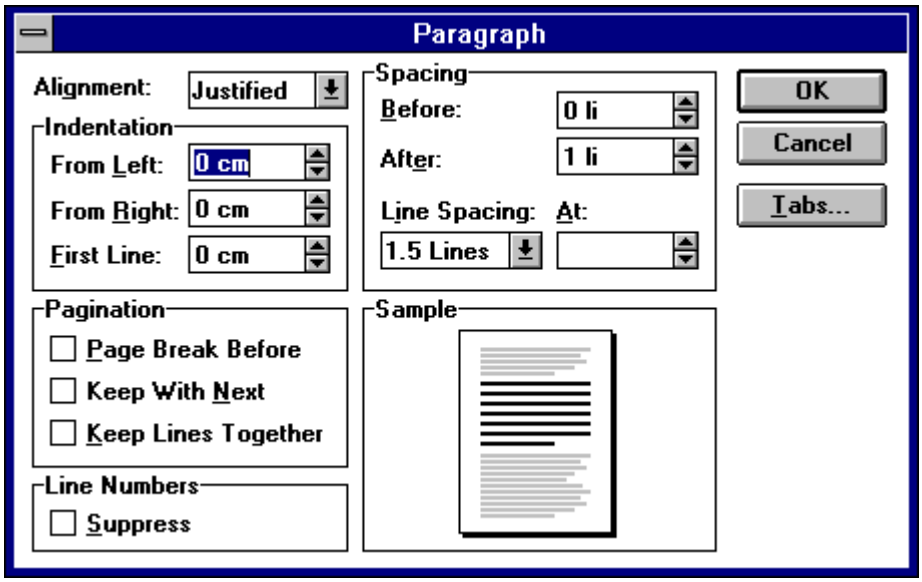


Figura 3.1: elemento de interface *dialog* do MS-Windows.

Alguns *toolkits* permitem vários diálogos simultaneamente na tela. Entretanto, apenas um deles pode estar ativo, isto é, interagindo com o usuário final. Os diálogos podem ser de dois tipos: modais e não modais. Os diálogos **modais** são aqueles que obrigam o usuário final a interagir exclusivamente com ele, e somente após o encerramento dessa interação é que o usuário pode interagir com outro diálogo. Os diálogos **não modais** não impõem nenhuma restrição, permitindo ao usuário interagir com vários diálogos ao mesmo tempo.

Os diálogos não modais são a base para as interações assíncronas, que representam a tendência atual [Hartson–Hix (1989)]. Este tipo de interação apresenta várias tarefas simultâneas para o usuário: é ele quem escolhe a seqüência de execução. Por outro lado, os diálogos modais forçam o diálogo tipo seqüencial, onde o sistema apresenta um diálogo por vez. (Os diálogos do sistema IntGraf, descrito na seção 1.2, são todos modais.)

3.1.2 *Menu*

Um *menu* é uma apresentação visual de operações que o usuário pode executar. Geralmente, um *menu* é formado por uma estrutura hierárquica, onde um ou mais itens do *menu* podem abrir um outro *menu.* Em alguns sistemas, estes itens têm uma apresentação visual diferente dos itens folhas. A Figura 3.2 ilustra o elemento de interface *menu.*

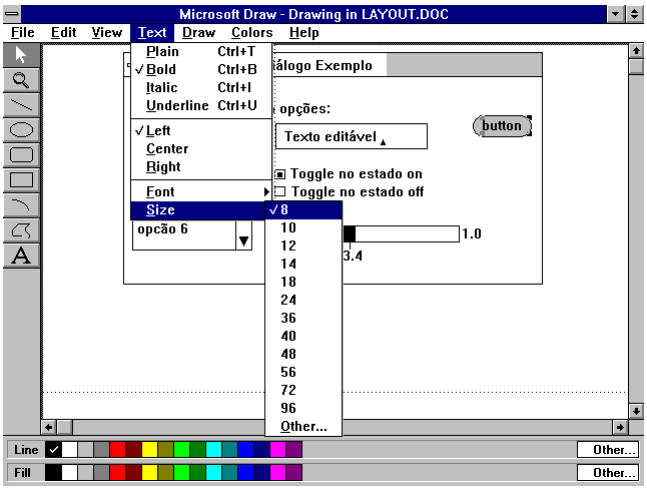


Figura 3.2: elemento de interface *menu* do MS-Windows.

O *menu* ilustrado na Figura 3.2 é um menu de barra. Ele é bastante utilizado em sistemas como o Motif e MS-Windows. Já o padrão de interface Open Look [Sun (1991)] propõe a utilização do menu *popup*. Este tipo de menu aparece para o usuário na posição corrente do *mouse* quando uma determinada ação é gerada pelo usuário, geralmente quando o botão da direita do *mouse* é pressionado. O menu *popup* fica na tela até um item folha ser selecionado, ou até o usuário cancelar a operação.

3.1.3 *Canvas*

Este elemento delimita uma área onde a aplicação possui total controle sobre os eventos que nela ocorrem. A aplicação utiliza o *canvas* para exibir os seus resultados, geralmente através de chamadas a um sistema gráfico. O *canvas* é, portanto, o elemento de integração entre o sistema de interface e o sistema gráfico.

3.1.4 *Label*

Este elemento é utilizado para informar algo para o usuário. Sua apresentação para o usuário pode ser um texto ou uma imagem (ver Figura 3.3).

3.1.5 *Button*

Este elemento de interface representa uma ação da aplicação. Para executar a ação, o usuário deve selecionar o *button*, isto é, posicionar o *mouse* sobre o *button* e acionar um botão do *mouse* (geralmente, o botão mais à esquerda). Acionar um botão do *mouse* significa duas ações: pressionar e soltar. A apresentação de um *button* para o usuário final, pode ser através de um texto ou de uma imagem (ver Figura 3.3).

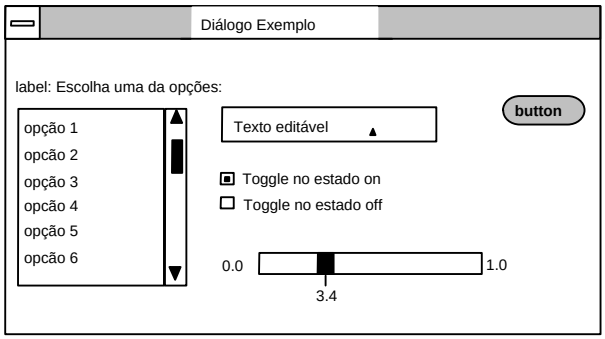


Figura 3.3: elementos de interfaces: *button, toggle, label, text, list, valuator.*

3.1.6 *Toggle*

Este elemento é um *button* de dois estados: *on* e *off*. O *toggle*, ao lado da sua apresentação visual, reserva uma área onde duas imagens (uma representando o estado *on* e outra o estado *off*) aparecem alternadamente, à medida em que o usuário o seleciona. Da mesma forma que o *button*, o *toggle* executa uma ação da aplicação sempre que o seu estado é modificado (ver Figura 3.3).

3.1.7 *Text*

Este elemento captura uma ou mais linhas de texto digitado pelo usuário. *Text* pode ter uma barra de rolamento vertical e/ou horizontal para permitir que o usuário entre com um texto maior que a área reservada pelo elemento (ver Figura 3.3). Uma ação da aplicação é executada sempre que algo for digitado pelo usuário.

3.1.8 *List*

O elemento *list* exibe uma lista de opções, geralmente do tipo *string*, associada a uma barra de rolamento vertical (ver Figura 3.3). O usuário pode selecionar uma ou mais opções, dependendo de como o elemento *list* tiver sido criado. Não há diferença na apresentação visual entre as listas que permitem uma única seleção e as que permitem mais de uma. Uma ação da aplicação é executada sempre que uma opção trocar de estado (selecionada/não selecionada).

3.1.9 *Valuator*

Este elemento captura um valor de ponto flutuante. Sua apresentação visual varia entre os sistemas e dentro de cada sistema. Geralmente, sua apresentação é parecida com uma barra de rolamento (ver Figura 3.3). Para cada ação do usuário sobre o *valuator*, uma ação da aplicação é executada.

3.1.10 *File Selection Box*

Este elemento captura um nome de arquivo. Este elemento é uma composição de elementos do tipo *list*, *label* e *text*, que permite ao usuário navegar nos discos e diretórios do sistema e escolher um arquivo, selecionando o seu respectivo nome em uma lista, ou então digitando o nome do arquivo no campo apropriado.

Uma aplicação pode não usar este elemento e construir o seu próprio, mas isso pode distanciar o *look-and-feel* da aplicação do sistema nativo (ver Figura 3.4 e 3.5).

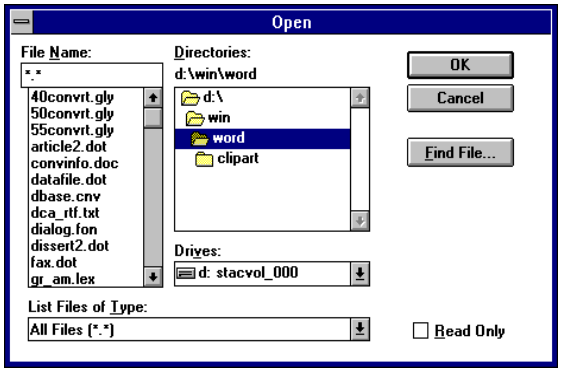


Figura 3.4: *file selection box* do Windows.

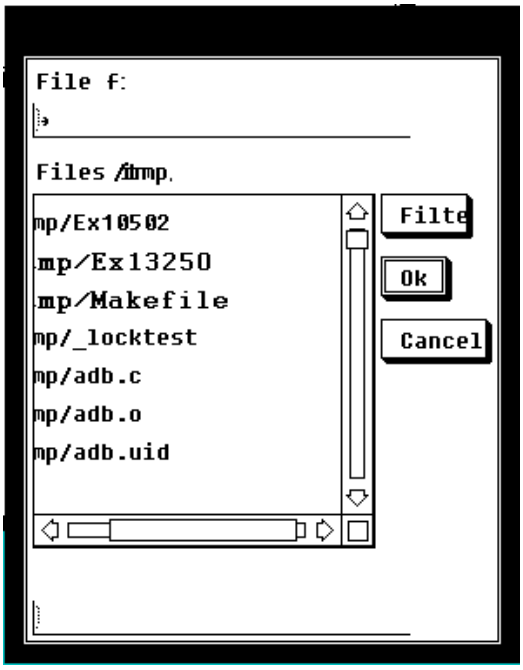


Figura 3.5: *file selection box* do Motif.

3.1.11 *Message*

Este elemento é um *dialog* pré-definido para informar ao usuário avisos e erros. Da mesma forma que o *file selection box*, este elemento pode ser substituído por uma composição de elementos especificado pela aplicação. Contudo, a aplicação pode se distanciar do *look-and-feel* sistema de interface nativo (ver Figura 3.6 e 3.7).

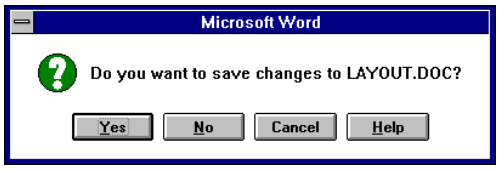


Figura 3.6: *message* do Windows

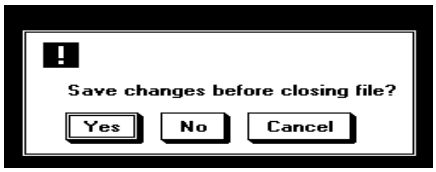


Figura 3.7: *message* do Motif

3.2 *Layout* de Diálogos

Embora os elementos básicos de interação sejam praticamente os mesmos em todos os sistemas e *toolkits*, não existe uma maneira uniforme para especificação do *layout* de diálogos. Nesta seção, examinaremos os vários métodos de composição de diálogos disponíveis nos sistemas de interface: Motif, XView, Macintosh e MS-Windows.

Como exemplo, apresentamos em cada um dos sistemas estudados a especificação do diálogo ilustrado na Figura 3.8. Cabe ressaltar que o texto e os dois botões são centralizados horizontalmente, e que verticalmente, os botões estão próximos da base do diálogo e o texto centralizado entre os botões e o topo do diálogo. Isto é o que chamamos de *layout* abstrato do diálogo (ver capítulo 4).

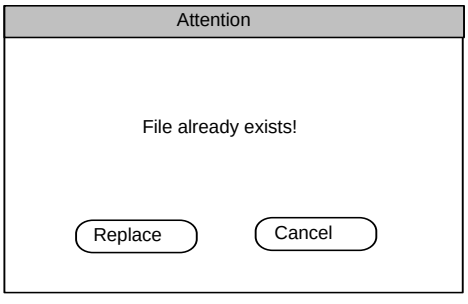


Figura 3.8: um diálogo simples.

3.2.1 Motif

O Motif fornece várias formas de se compor um diálogo, sem que seja necessário especificar posições exatas de cada *widget* (nome dado aos elementos de interfaces no ambiente do Xt *Intrinsics toolkit*). Cada forma é específica para um determinado tipo de *layout*. No Motif, existem vários *widgets* que pertencem à classe *Composite*, isto é, aqueles que permitem agrupar e compor *widgets*. Não vamos descrever todos, mas apenas três deles que oferecem formas diferentes para especificação de *layout*: *RowColumn, BulletinBoard* e *Form.* Não obstante, o Motif oferece um mecanismo para a aplicação criar e exibir os *widgets* com o mesmo tamanho, independente da resolução do dispositivo. Este mecanismo permite à aplicação escolher um tipo de unidade para definir a posição, o tamanho e o espaçamento do *widgets*, minimizando o esforço em especificar *layouts*, quando é preciso definir explicitamente posições e tamanhos para os *widgets.* Os possíveis tipos de unidades são:

- *raster* (unidade do dispositivo de saída);
- milímetro;
- polegada;
- pontos (unidade típica usada em processamento de texto, equivalente a 1/72 de polegada);
- proporcional ao fonte de caracteres utilizado.

O *widget RowColumn* utiliza a metáfora de uma matriz bidimensional para definir *layouts*. O *RowColumn* permite que a aplicação controle alguns parâmetros do *layout* definindo valores para alguns atributos, como: `XmNorientation` e `XmNpacking`. `XmNorientation` define a orientação de preenchimento das células da matriz. Quando este atributo vale `XmVERTICAL`, a matriz é preenchida de cima para baixo, isto é, os *widgets* são posicionados um abaixo do outro até chegar na última linha da matriz; neste momento, os próximos *widgets* são posicionados nas células da próxima coluna. Um algoritmo equivalente é executado quando o atributo `XmNorientation` vale `XmHORIZONTAL`. Já o `XmNpacking` permite construir matrizes uniformes, onde todas as células possuem o mesmo tamanho. A combinação desses e outros atributos permite uma boa flexibilidade na especificação de *layouts* com forma de matriz.

Não apresentamos a especificação do nosso diálogo exemplo, ilustrado na Figura 3.8, pois não é possível fazê-lo utilizando somente os recursos de posicionamento relativo oferecidos pelo *RowColumn.* Teríamos que utilizar posicionamento explícito, indo contra a razão da finalidade do *RowColumn.*

O *BulletinBoard* é um *widget* de composição que possibilita a criação de *layout* simples. Ele não força posicionamento nos seu *widgets*, mas pode ser definido para rejeitar *layouts* com *widgets* sobrepostos. O *BulletinBoard* é a base de vários outros *widgets*, como: o *FileSelectionBox*, o *MessageBox* e o *Form*, que é descrito a seguir.

O *widget Form* é uma derivação do *BulletinBoard* que permite definir um *layout* a partir de relações espaciais entre seus *widgets.* Este é o mais interessante de todos, pois mantém o *layout* inalterado quando o *Form* tem seu tamanho alterado, ou quando há qualquer modificação nos *widgets* nele contido. As relações espaciais são estabelecidas pela aplicação, que define atributos, reconhecidos pelo *Form*, nos *widgets* que irão se compor. São vinte atributos que combinados definem relações espaciais de um *widget* com o *Form*, com um outro *widget*, com posições relativas dentro do *Form*, ou com a posição inicial de outro *widget*. A desvantagem deste *widget* é que ele é difícil de ser entendido, pois, além de possuir grande quantidade de atributos, existem atributos cujo significado pode depender do valor de um outro atributo.

Apesar da grande flexibilidade oferecida pelo *Form*, não é possível criar *layouts* com *widgets* centralizados. Isto ocorre porque o *Form* só permite que a aplicação controle a relação espacial entre dois *widget*s, e para centralizar um conjunto de elementos de interface é necessário relacionar espacialmente vários *widget* simultaneamente. Portanto, não é possível reproduzir exatamente o *layout* do nosso diálogo exemplo, ilustrado na Figura 3.8, pois há necessidade de centralizar horizontalmente os botões do diálogo. Porém, podemos definir um *layout* bem próximo do *layout* original.

Abaixo, apresentamos uma descrição textual de uma especificação para o nosso diálogo exemplo, utilizando o *Form*:

- o elemento de interface *label* representando a mensagem centralizada, tem seu topo grudado com o topo do *Form* e a sua base também grudada ao topo de um dos botões;
- o elemento de interface *button* representando o botão “Replace” tem sua base grudada na base do *Form* e o seu lado direito distante do lado direito do *Form* de 40% da largura do *Form;*
- o elemento de interface *button* representando o botão “Cancel” tem sua base grudada na base do *Form* e o seu lado esquerdo distante do lado esquerdo do *Form* de 60 % da largura do *Form* .

A descrição acima não é a única forma de relacionarmos os *widgets* para conseguirmos um *layout* próximo do *layout* do nosso diálogo exemplo. Poderíamos, ao

invés de relacionar o segundo botão com *Form*, relacioná-lo com o primeiro botão da seguinte forma:

- O elemento de interface *button* representando o botão “Cancel” tem seu lado esquerdo distante de alguns pixels do lado direito do botão “Replace”.

Vale observar que a relação espacial definida para centralizar a mensagem “File already exists !” só foi possível porque o elemento de interface em questão é um *Label*. O *Label* não responde a ações do usuário e a sua apresentação não demonstra o seu real tamanho. Por exemplo, ao relacionarmos um botão desta maneira, a sua apresentação visual ocupará todo o seu tamanho, além do que, o usuário poderá acionar o botão, selecionando qualquer ponto entre o topo do *Form* e os botões de baixo.

O Motif também permite que os diálogos sejam definidos fora do código computacional da aplicação através da sua *user interface language* (UIL). A vantagem desta separação, como definiu Hartson–Hix (1989), é “a separação do problema da ciência da computação do problema de fatores humanos”. Desta forma, cada problema pode ser estudado e solucionado, pelos seus respectivos especialistas, conseguindo-se assim, um aprimoramento no projeto da aplicação.

UIL é uma linguagem de especificação para descrever o estado inicial da interface com usuários em aplicações Motif, permitindo a especificação:

- de *widget*s e de seus atributos*;*
- das *callbacks* de cada *widget*;
- da estrutura hierárquica dos *widgets*;
- de valores que podem ser consultados pela aplicação.

Um código UIL pode ser compilado e ligado com a aplicação, mas também pode ser compilado em tempo de execução da aplicação. Isto permite algo fundamental em projetos de interface com usuário: prototipação rápida. O usuário não precisar alterar o código computacional da aplicação para testar um novo projeto de interface. Na verdade, ele nem precisa da aplicação para projetar a interface: basta que ele crie um programa que apenas chame os serviços do Motif para compilar um programa UIL e mostrar os seus diálogos.

Outra vantagem é a customização da interface para diferentes usuários. Para usuários novos, podemos oferecer uma interface mais simples onde não esteja disponível toda a funcionalidade da aplicação, apenas a funcionalidade básica. À medida

em que o usuário vai se tornando experiente, podemos oferecer interfaces com funções mais avançadas.

3.2.2 XView

No XView, os elementos de interface podem ser posicionados explicitamente, indicando o seu par de coordenadas (x,y), em unidades *raster*, em relação ao canto superior esquerdo do diálogo. Mas também podem ser posicionados de forma relativa, onde o diálogo é visto como uma matriz bidimensional, de linhas e colunas. Quando um novo elemento de interface é inserido em um diálogo, ele é disposto à direita ou abaixo do último elemento inserido, dependendo do valor do atributo `PANEL_LAYOUT`: `PANEL_HORIZONTAL` e `PANEL_VERTICAL` respectivamente. Quando um elemento alcança a largura do diálogo, previamente definido, uma nova linha é iniciada e o elemento é posicionado no canto esquerdo desta nova linha. Efeito similar ocorre quando um novo elemento de interface alcança a altura do diálogo.

É importante ressaltar que o posicionamento relativo não mantém o *layout* inalterado quando o diálogo sofre alterações de tamanho. Isto porque, ao posicionarmos relativamente, estamos relacionando muito fracamente os elementos de interface, não sendo suficiente para o XView reposicionar os elementos de forma que o *layout* se mantenha o mesmo.

O nosso diálogo exemplo, ilustrado na Figura 3.8, não pode ser definido somente com os recursos de posicionamento relativo; é preciso explicitar as coordenadas de alguns elementos de interface. Textualmente, a especificação do diálogo pode ser:

- definir um tamanho para o diálogo;
- criar a mensagem indicando a sua posição através das suas coordenadas;
- criar o botão mais à esquerda, também indicando a sua posição através das suas coordenadas;
- criar o segundo botão através de posicionamento relativo.

Codificar a descrição acima não é trivial, pois primeiramente temos que definir um valor para o tamanho do diálogo e, a partir daí, calcular as posições da mensagem e do primeiro botão. Para o cálculo da posição da mensagem e do botão, é necessário saber o tamanho da mensagem e o tamanho dos dois botões. Então, a melhor forma de codificar esta descrição é primeiro criar os elementos sem se preocupar com suas

posições, e não torná-los visíveis. Com eles criados, consultar o XView para conhecer os tamanhos de cada um, definir um tamanho para o diálogo e então posicioná-los e torná-los visíveis.

Para diálogos mais complexos, este processo pode se repetir algumas vezes; infelizmente, o XView não permite que os elementos de interface sejam especificados em separado do código da aplicação. Portanto, o programador não vai escapar dos tediosos processos de compilação e ligação do programa. Isto aumenta o tempo de prototipagem, indo contra a idéia de que sistemas de interface devem possibilitar prototipagem rápida.

3.2.3 Macintosh Toolbox

O Macintosh não oferece nenhum mecanismo de posicionamento relativo para especificação de *layouts*. O programador tem que especificar a área retangular que cada elemento de interface ocupa, fornecendo a posição do canto superior esquerdo e do canto inferior direito. As coordenadas são em unidades *raster*, em relação ao canto superior esquerdo do diálogo. O par ordenado que identifica uma posição não é descrito na ordem convencional do sistema cartesiano. A primeira coordenada se refere ao eixo vertical `y` e a segunda ao eixo horizontal `x`.

O Macintosh permite que os elementos de interface sejam definidos em separado do código computacional da aplicação, através de uma linguagem de descrição. Entretanto, esta linguagem é bastante rudimentar, chegando a ser parecida com linguagem de máquina. Para evitar a utilização desta linguagem, o Macintosh oferece um editor gráfico interativo para criar e manipular os elementos de interface.

O editor gráfico facilita muito a construção dos diálogos, mas não retira da aplicação a responsabilidade de recalcular posições e tamanhos dos elementos de interface, para manter inalterado o *layout* do diálogo quando o usuário da aplicação promove uma alteração no seu tamanho.

3.2.4 MS-Windows

O Microsoft Windows exige que os diálogos sejam compostos especificando as dimensões e as coordenadas exatas de cada elemento em um painel de controle, não oferecendo nenhuma forma de posicionamento relativo para definir os *layouts* dos diálogos. Para compensar esta dificuldade, o MS-Windows permite a utilização de diferentes unidades, semelhante às do Motif, para o sistema de coordenadas:

- *raster;*
- milímetro;
- polegada;
- proporcional ao fonte de caracteres utilizado.

O MS-Windows também permite que os elementos de interfaces sejam especificados em separado da aplicação, através de uma linguagem de descrição. Porém, eles ainda precisam ser compilados e ligados com a aplicação, retardando o processo de prototipagem.

Abaixo, apresentamos a especificação do nosso diálogo exemplo na linguagem de descrição do MS-Windows. As posições estão definidas em unidades *raster*, ou em relação ao canto superior esquerdo da tela ou em relação ao diálogo a que pertencem. E os tamanhos são definidos utilizando a unidade proporcional ao fonte de caracteres utilizado.

```
ATTENTION_DIALOG DIALOG 16, 18, 140, 72
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Attention" BEGIN
  PUSHBUTTON
"Replace",101,30,50,34,14,WS_CHILD|WS_VISIBLE|WS_TABSTOP
  PUSHBUTTON                                  "Cancel",102,80,50,30,14,
WS_CHILD|WS_VISIBLE|WS_TABSTOP
  CTEXT "File already exists!,-1,35,20,70,10,WS_CHILD|WS_VISIBLE
END
```

3.2.5 Comparação

Fazendo uma comparação dos sistemas estudados, vemos que o mais completo ambiente para especificação de diálogos é o Motif. Ele possibilita, através da UIL, a separação da programação dos elementos de interface do código computacional do programa, permitindo uma prototipação rápida. Por outro lado, o Motif é tão grande, a UIL uma linguagem tão poderosa e existe tanta variedade de formas para compor diálogos que o programador da aplicação pode se confundir, deixando-o em dúvida sobre qual o método adequado a cada situação. Já o XView é um *toolkit* bem compacto, que oferece um modelo de posicionamento relativo para compor diálogos bastante simples e de fácil aprendizado, o qual o torna agradável de ser programado. Porém, a falta de uma linguagem de especificação de diálogos dificulta a separação da interface com usuário da aplicação e, conseqüentemente, aumenta o tempo de prototipação. O Macintosh e o Microsoft Windows não oferecem nenhum modelo abstrato para a definição de *layouts*, exigindo que o programador desenhe o diálogo em escala num papel milimetrado, para que ele possa conhecer as posições e tamanhos de cada elemento de interface. Como conseqüência, a alteração de um diálogo — seja inserindo um novo elemento, apagando um já existente, ou simplesmente modificando o tamanho de um elemento — pode modificar muitas ou até todas as coordenadas, obrigando o programador a recalcular o *layout* do diálogo. Este processo poderia ser menos cansativo se a linguagem de descrição de diálogos pudesse ser compilada em tempo de execução da aplicação, o que não ocorre, obrigando o programador ao tedioso processo de compilação e ligação.

Para evitar que os diálogos sejam projetados num papel milimetrado, o Macintosh oferece um editor gráfico que permite a construção de diálogos por manipulação direta dos elementos de interface. No MS-Windows, este tipo de editor é fornecido pelos programas que permitem a especificação de algum tipo de diálogo (e.g., compilador Borland C++, compilador Microsof C, Visual Basic, Microsoft Access). Estas ferramentas facilitam a especificação de um *layout* específico, mas geralmente não usam um modelo abstrato de *layout*. Como conseqüência, os diálogos criados visualmente são incapazes de reagir automaticamente a modificações no tamanho do diálogo promovidas pelo usuário final, ou pela inserção, modificação ou eliminação de um elemento de interface.

Existem editores interativos de diálogos para os outros sistemas de interface, mas quase todos especificam o *layout* dos diálogos utilizando um modelo concreto ao invés de um modelo abstrato (veja Capítulo 4). A exceção aqui são os editores de diálogos `ibuild` do InterViews [Linton (1989)] e o do FormsVBT [Avrahami (1989)]: ambos utilizam o paradigma de *boxes-and-glue* do processador de

texto TEX [Knuth (1984)] como modelo de *layout* abstrato. Além disso, algumas destas ferramentas (como o Guide do Open Look para XView) geram descrições que precisam ser traduzidas, compiladas e acopladas à aplicação antes da execução, o que limita a utilidade para prototipagem rápida.

O editor de diálogos do FormsVBT utiliza o paradigma de duas vistas: uma contendo a descrição textual dos diálogos em uma linguagem específica e uma outra vista contendo um editor interativo para construção de diálogos por manipulação direta. O usuário pode interagir nas duas vistas, e a alteração em uma das vistas reflete na outra. Esta combinação une as vantagens da descrição textual com as vantagens da manipulação direta do tipo WYSIWYG, sem sofrer as limitações de ambas.

O IUP/LED propõe como solução para descrição de *layout* um modelo de *layout* abstrato, semelhante ao do InterViews e do FormsVBT, também baseado no paradigma de *boxes-and-glue* do editor de texto TEX [Knuth (1984)]. Como veremos no próximo capítulo, este modelo permite que um diálogo seja especificado sem definir explicitamente as posições dos elementos de interface, possibilitando um reposicionamento automático dos elementos de interface após uma alteração no tamanho do diálogo.

3.3 Diálogos vs. Aplicação

Em programas interativos, a comunicação entre a aplicação e o usuário é bidirecional: a aplicação transmite e recebe informações para o usuário. É através dos diálogos que a comunicação se concretiza. A aplicação monta um diálogo e o torna visível; a partir daí, a aplicação fica em um estado de espera, reagindo às ações do usuário.

Basicamente, existem dois tipos de modelos de integração da aplicação com as ações dos usuários gerenciadas pelos diálogos: *callbacks* e eventos. O modelo de *callback* associa uma rotina da aplicação a cada ação possível sobre os elementos de interface. O *toolkit* captura o evento gerado pelo usuário e executa a rotina da aplicação correspondente. Este modelo é utilizado pelo XView, Motif e pelos *toolkits* baseados no X Window Intrinsic Toolkit.

Outros sistemas, como o MS-Windows e o Macintosh, utilizam o modelo de eventos. Este modelo coloca em uma fila todos os eventos gerados pelo usuário. É responsabilidade da aplicação retirá-los da fila, interpretá-los, e disparar as rotinas apropriadas.

Os dois modelos são equivalentes. O modelo de *callback* pode ser convertido para o modelo de eventos, associando todas ações a uma única rotina da aplicação que seria a rotina de tratamento de eventos. Já o modelo de eventos pode ser convertido para o modelo de *callback* construindo uma camada de *software* que contenha a rotina de tratamento de eventos e as rotinas responsáveis em associar rotinas da aplicação aos eventos do usuários.

Cabe observar que o modelo de eventos pode ser ineficiente, pois mesmo os eventos não tratados pela aplicação são inseridos na fila, acarretando um processamento desnecessário. Para as aplicações altamente interativas, que precisam de respostas rápidas, isto pode ser um fator limitante, se for freqüente a ocorrência de eventos sem significado. Os exemplos mais comuns desses eventos são aqueles gerados pelo usuário ao movimentar o *mouse* sem que nenhum botão esteja pressionado. Neste contexto, quando o usuário desloca o *mouse*, a sua intenção, a princípio, não é interagir com os diálogos por onde o *mouse* passa, mas sim posicioná-lo sobre um elemento de interface para iniciar uma interação. Porém, ao atravessar um diálogo, vários eventos são gerados; entre eles: um *enter window*; vários *mouse movement*; e um *leave window*. Sendo a arquitetura do X-Window cliente-servidor, a aplicação não precisa estar necessariamente sendo executada na mesma máquina na qual o usuário está interagindo. Isto aumenta a gravidade dos eventos desnecessários, pois, além de sobrecarregar o processamento da aplicação, aumenta o tráfego na rede, afetando o desempenho de todas as aplicações da rede. A solução adotada pelo *toolkit* Xlib é permitir que os eventos a serem colocados na fila de eventos sejam especificados pela aplicação, através de máscaras. Já o MS-Windows não oferece nenhum mecanismo que evite a geração destes eventos, talvez porque ele use o sistema operacional PC-DOS que não oferece recursos de rede. Sendo o Windows-NT um sistema operacional direcionado para uso em rede, este deverá suportar algum tipo de mecanismo que contorne esse problema.

O IUP/LED utiliza o modelo de *callbacks*. Este modelo foi escolhido por permitir um método mais natural de programação e por evitar o problema descrito acima.

4

IUP/LED

IUP/LED é um sistema portátil de interface com usuário composto por uma Linguagem de Especificação de Diálogos (LED) e um pequeno conjunto de funções (IUP), agrupados em uma biblioteca, para criação e manipulação de diálogos. O sistema IUP/LED foi projetado para ser:

- portátil, podendo ser utilizado tanto no PC-DOS no modo texto quanto no ambiente Unix/Motif, entre outros;
- simples e de rápido aprendizado;
- extensível;
- aberto, no sentido de que o programador pode combinar chamadas ao IUP com chamadas ao *toolkit* nativo.

O sistema IUP/LED foi projetado para ser portátil no sentido de Cowan–Wilkinson (1984), isto é, de forma que a instalação em um novo ambiente computacional exiga um esforço muito menor que o necessário para reescrevê-lo no novo ambiente. Este objetivo foi alcançado através de uma combinação de estratégias de portabilidade já descritas no capítulo anterior. Desta forma, o IUP/LED pode ser implementado em ambientes tão diferentes quanto o PC-DOS no modo texto e o MS-Windows.

Uma preocupação no projeto do IUP/LED foi a questão da simplicidade. Os elementos de interface são criados e manipulados consistentemente pela aplicação, através de um pequeno conjunto de funções. A dinâmica da interação é feita, principalmente, através de duas funções IUP que estabelecem e consultam atributos associados aos elementos de interface. A descrição estática dos diálogos pode ser feita através da linguagem LED, que é uma linguagem de expressão cuja sintaxe é muito simples.

O IUP/LED permite facilmente aumentar o conjunto de atributos reconhecidos pelos elementos de interface. Este aumento não influencia de modo algum as funções IUP/LED. A inclusão de um novo elemento de interface requer um pouco mais de trabalho, mas, para o programador IUP/LED representa apenas uma nova função LED e uma nova função IUP que cria o novo elemento e um conjunto de atributos específicos para o novo elemento.

O IUP/LED associa a cada elemento de interface um atributo (`WID`) onde seu valor é o identificador do elemento de interface no sistema nativo. Desta forma, a aplicação pode consultar este valor e utilizá-lo como parâmetro na execução de funções do sistema de interface nativo.

Para o IUP, uma aplicação é formada exclusivamente por um conjunto de diálogos potencialmente concorrentes. Um diálogo é formado por elementos de interface que interagem com o usuário, capturando e exibindo dados manipulados pelo programa. Desta forma, escrever uma aplicação se resume na especificação dos seus diálogos (que pode ser feita ulizando-se LED) e na implementação das rotinas da aplicação.

A filosofia principal de LED é a distinção entre *layout* abstrato e *layout* concreto. Descrever o *layout* concreto de um diálogo é descrever a posição geométrica de cada objeto de interface que compõe o diálogo. Por outro lado, descrever o *layout* abstrato de um diálogo é descrever as posições *relativas* destes objetos. Freqüentemente, o programador tem um idéia bem clara do *layout* abstrato, enquanto o cálculo do *layout* concreto é complicado e tedioso. Além disso, se o *layout* de um diálogo é descrito abstratamente, então é simples recalcular o *layout* concreto quando o tamanho do diálogo é modificado pelo usuário, ou quando elementos são adicionados ou removidos do diálogo, seja na prototipagem, ou na execução da aplicação.

4.1 LED - Uma linguagem para especificação de diálogos

LED é uma linguagem de expressões para especificação de diálogos. LED é compilada em tempo de execução da aplicação por funções do *toolkit* IUP, garantindo, assim, três qualidades fundamentais em um sistema de interface:

- independência de diálogos;
- prototipação rápida; e
- customização para diferentes tipos de usuários e plataformas.

A independência de diálogos é alcançada no próprio uso da linguagem LED, que foi projetada com esse objetivo: separar a especificação dos diálogos com usuários do código computacional da aplicação .

Prototipação rápida é possível porque um programa LED é compilado em tempo de execução e não exige nenhuma função específica da aplicação; apenas exige que o programa principal chame a função IUP que compila um programa LED. Este único programa pode ser utilizado na prototipação rápida da interface de qualquer aplicação.

A customização da aplicação pode ser feita pelo próprio usuário, pois, junto com o programa executável, podem ir as definições dos diálogos da aplicação. Sendo LED uma linguagem simples e de fácil compreensão, o próprio usuário pode modificar a especificação dos diálogos para criar versões simplificadas do programa ou traduzir a interface para uma outra linguagem (e.g. para o inglês).

4.1.1 Modelo

A linguagem LED define um modelo abstrato de interface com usuário no qual os diálogos são definidos pelo seu *layout* abstrato, e os elementos que compõem os diálogos são especificados, principalmente, pela sua funcionalidade e não pela sua aparência final. Em LED, o programador só precisa fornecer alguns poucos parâmetros associados à funcionalidade de cada elemento de interface; atributos de aparência podem ser especificados, mas não são obrigatórios.

Utilizando um modelo abstrato de interface, o programador da aplicação pode criar os seus diálogos sem se preocupar com o sistema de interface sob o qual o programa irá executar. Além disso, a implementação da aplicação em um novo ambiente deverá ser imediata, pelo menos no que diz respeito à interface com usuário, pois basta escrever um *driver* para o novo sistema de interface "nativo". (Note a analogia intencional com a estratégia de portabilidade para aplicações gráficas descrita anteriormente.) Desta forma, um programa pode rodar *sem modificações* em sistemas tão diferentes quanto Microsoft Windows, OS/2 Presentation Manager, Motif, OpenLook, Macintosh. O *toolkit* IUP fornece uma API que implementa o modelo abstrato de LED (veja Figura 4.1).

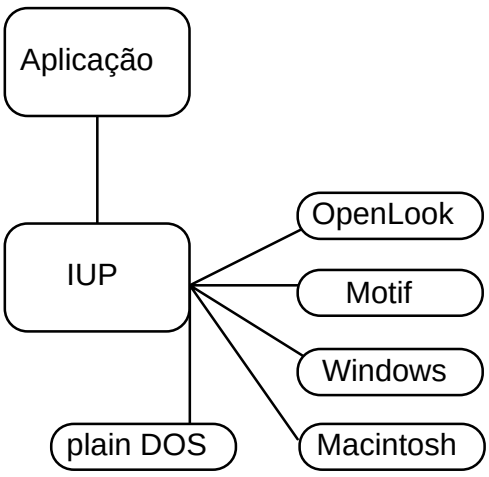


Figura 4.1: arquitetura de uma aplicação IUP.

O modelo de *layout* de LED é baseado no paradigma *boxes-and-glue* do processador de texto TEX [Knuth (1984)]. Além de ser um modelo bastante simples, de rápido aprendizado, este modelo é capaz de manter o *layout* abstrato, seja qual for o tamanho do diálogo. Assim, a disposição relativa dos elementos de interface que compõem um diálogo ficará inalterada após uma alteração de tamanho promovida pelo usuário da aplicação, ou adição e remoção de elementos. Portanto, o programador fica liberado de calcular tamanhos e posições para os elementos de interface em cada diálogo. O paradigma *boxes-and-glue* também é utilizado em InterViews [Linton (1989)] e FormsVBT [Avrahami et al. (1989)].

4.1.2 Sintaxe

LED é uma linguagem de expressões projetada para permitir que um diálogo seja definido, basicamente, a partir da especificação do seu *layout* abstrato e da funcionalidade dos objetos de interface que compõem o diálogo. Os atributos de aparência, tal como cor e fonte de caracteres, são especificados opcionalmente em forma de variáveis de ambiente, similares às existentes no Unix e no DOS. Esta distinção entre informações obrigatórias (associadas à funcionalidade) e informações opcionais (que definem aparência) está explícita na sintaxe de LED.

A sintaxe das expressões em LED é simplesmente:

v=f[a](p),

onde:

- **v** é o nome que deverá ser utilizado pela aplicação para acessar o elemento de interface que está sendo definido pela expressão **f[a](p)**;
- **f** é o tipo do elemento de interface que está sendo descrito;
- **a** é a lista de atributos para esse elemento de interface;
- **p** é a lista de parâmetros que definem a funcionalidade de elementos do tipo **f**.

A atribuição de um nome a uma expressão é opcional. Entretanto, uma aplicação só pode se comunicar diretamente com elementos que tenham nome. Assim, uma aplicação não pode alterar nem consultar atributos de elementos anônimos, embora estes elementos estejam plenamente ativos.

A lista de atributos **a** tem a seguinte forma:

$$\mathbf{v_1 = a_1\ ,\ v_2 = a_2\ ,\ \ldots\ ,}$$

onde $\mathbf{v_i}$ é um nome de um atributo e $\mathbf{a_i}$ é o seu valor (um *string*).

4.1.3 Exemplo

Como exemplo do uso de LED na especificação de um diálogo típico, considere o diálogo da Figura 4.2. Este diálogo é composto por um texto (**"File already exists!"**) e dois botões (**"Replace"** e **"Cancel"**). O *layout* abstrato deste diálogo pode ser descrito da seguinte forma: os botões estão centrados na parte inferior da área do diálogo e o texto está centrado na área restante, acima dos botões. A especificação em LED é imediata a partir desta descrição:

```
confirm=dialog[TITLE="Attention"](body)
body=vbox(fill(),prompt,fill(),buttons)
prompt=hbox(fill(),warning,fill())
buttons=hbox(fill(),replace,fill(),cancel,fill())
warning=label("File already exists!")
replace=button("Replace",do_replace)
cancel=button("Cancel",do_cancel)
```

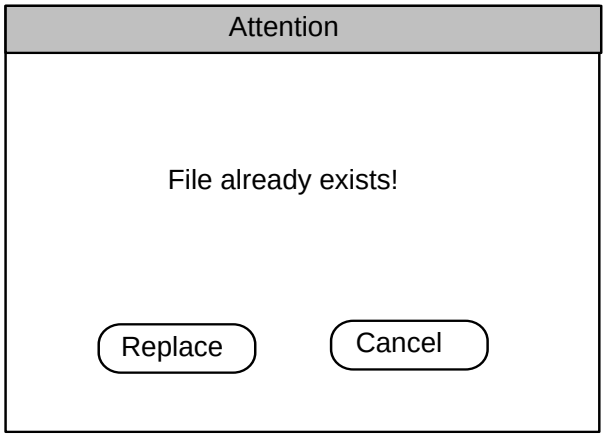


Figura 4.2 um diálogo simples.

Nesta especificação, usamos os seguintes elementos de interface: **dialog**, **vbox**, **hbox**, **label**, **button** e **fill**. Descreveremos estes e os outros elementos de interface na próxima seção. No exemplo acima, usamos nomes para todos os elementos criados. Isto não é necessário; o mesmo diálogo pode ser especificado sem dar nomes aos elementos intermediários:

```
confirm=dialog[TITLE="Attention"](
 vbox(
     fill(),
     hbox(
         fill(),
         label("File already exists!"),
         fill()),
     fill(),
     hbox(
         fill(),
         button("Replace",do_replace),
         fill(),
         button("Cancel",do_cancel),
         fill())))
```

4.1.4 Elementos

Os vários elementos de interface disponíveis em LED podem ser divididos nas seguintes categorias:

- **agrupamento**: definem uma funcionalidade comum para um grupo de elementos;
- **composição**: definem uma forma de exibir elementos;
- **preenchimento**: ocupam dinamicamente espaços vazios;
- **primitivos**: efetivamente interagem com o usuário.

Como a lista de parâmetros **p** pode conter outras expressões, os elementos que compõem um diálogo estão organizados em uma estrutura hierárquica do tipo árvore, como apresentado na Figura 4.3 para o diálogo da Figura 4.2. Nota-se que os nós internos da árvore ou são elementos de composição ou são elementos de agrupamento, e os nós folhas ou são elementos primitivos ou são elementos de preenchimento.

Esta estrutura hierárquica facilita a programação, pois permite o uso de metodologias de programação estruturada como a *bottom-up*, onde os diálogos podem ser especificados gradativamente, combinando diálogos simples, previamente testados, na formação de um diálogo mais complexo.

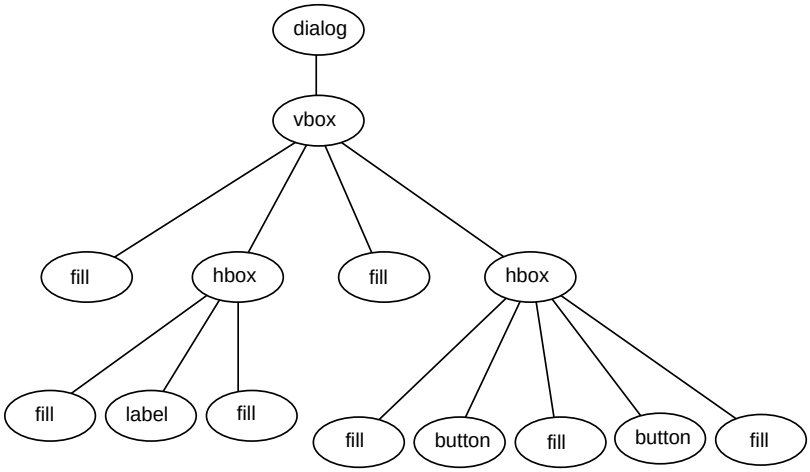


Figura 4.3: estrutura do diálogo da Figura 4.2.

4.1.4.1 Elementos de agrupamento

Os elementos de agrupamento definem uma funcionalidade comum para um grupo de elementos. Os elementos de agrupamento disponíveis em LED são:

- **dialog:** compõe o diálogo de interação com o usuário;
- **radio:** agrupa **toggle**'s restringindo o estado *on* a apenas um deles;
- **menu:** agrupa **item**'s e **submenu**'s.

4.1.4.2 Elementos de composição

Os elementos de composição definem a forma de exibição dos seus elementos descendentes. Seguindo o paradigma de TEX, temos dois elementos de composição:

- **hbox:** exibe horizontalmente os seus elementos;
- **vbox:** exibe verticalmente os seus elementos.

Através destes dois elementos de composição, é possível construir uma variedade enorme de diálogos sem que seja necessário definir explicitamente as coordenadas de cada elemento que compõe o diálogo. A especificação em LED para o exemplo ilustra o uso de **vbox** para dispor os dois itens principais (**prompt** e **buttons**) verticalmente um sobre o outro, e o uso de **hbox** para dispor os botões horizontalmente um ao lado do outro.

4.1.4.3 Elementos de preenchimento

Existe apenas um elemento de preenchimento: **fill**, que ocupa proporcional e dinamicamente os espaços vazios em um diálogo. O **fill** é responsável tanto pela invariância do *layout* abstrato sob alterações de tamanho do diálogo quanto pelo posicionamento relativo dos elementos de interface dentro dos elementos de composição (**hbox** e **vbox**). A especificação em LED para o exemplo ilustra o uso de **fill** para centralizar horizontalmente um **label** e verticalmente este mesmo elemento centralizado (**prompt**) na maior área do diálogo. Se a especificação de **body** fosse:

```
body=vbox(prompt, fill(), buttons)
```

então, o texto ficaria no topo da área do diálogo.

4.1.4.2 Elementos primitivos

Os elementos primitivos disponíveis em LED são uma abstração dos elementos primitivos descritos, anteriormente, no Capítulo 3.

- **button**: botão;
- **canvas**: área de trabalho;
- **frame**: coloca uma borda em volta de um elemento de interface;
- **hotkeys**: teclas de funções;
- **image**: imagem estática;
- **item**: item de menu;
- **label**: texto estático;
- **list**: lista de *strings* com barra de rolamento;
- **submenu**: submenu de menu;
- **text**: captura um texto de uma ou mais linhas;
- **toggle**: botão de dois estados (*on/off*);
- **valuator**: captura um valor numérico.

Com exceção do **canvas**, todos os outros elementos primitivos estão bem definidos e têm o mesmo comportamento em todos os sistemas de interface existentes.

O elemento de interface **canvas** é um elemento bastante particular, pois é o principal elo de ligação entre a parte gráfica da aplicação e o sistema de interface. É através do **canvas** que os objetos da aplicação são exibidos e manipulados pelo usuário

final. Esta ligação mais próxima com a aplicação torna difícil a definição abstrata do seu comportamento. O sistema de interface InterViews define formalmente o comportamento do **canvas**, retirando da aplicação o tratamento de alguns eventos, como *repaint* e *resize*. (Uma abstração alternativa foi proposta por Neelamkavil–Mullarney (1990).) No sistema IUP/LED, o comportamento do **canvas** é simples: os eventos ocorridos no **canvas** são passados para a aplicação, que deve tratá-los convenientemente.

Vale ressaltar que, dos elementos primitivos, o **canvas** é o único que compete com o **fill** por espaços vazios. Esta característica se deve ao fato dele ser a área de exibição dos objetos da aplicação, geralmente onde o usuário mais se concentra.

4.1.5 Atributos

Os atributos de elementos de interface são implementados como variáveis de ambiente, isto é, são representados por pares ordenados **(v,a)**, onde **v** é o nome de uma variável e **a** é o seu conteúdo (um *string*).

As variáveis de ambiente implementam um mecanismo de herança para atributos: as variáveis definidas em um elemento são automaticamente exportadas para os seus filhos. Por exemplo, uma variável definida em um **hbox** também está definida, com o mesmo valor, em todos os elementos agrupados neste **hbox**. Se um destes elementos definir uma variável com o mesmo nome, o valor associado neste elemento tem prioridade sobre o valor associado no **hbox**. Isto permite a alteração de atributos em bloco, com mudanças locais. Por exemplo, para mudar o fonte de caracteres usado globalmente no diálogo **confirm** e localmente no botão **replace**, poderíamos escrever:

```
confirm=dialog[FONT="Helvetica"](...)
replace=button[FONT="HelveticaBold"](...)
```

Alguns nomes de variáveis são reconhecidos pelo sistema e representam atributos dos elementos de interface. Estes atributos alteram, principalmente, aspectos de aparência, tais como cor, fonte de caracteres, cursor. Alguns poucos atributos definem funcionalidade. Este é o caso das teclas de função que são associadas aos diálogos através do atributo `HOTKEYS`.

Os nomes não reconhecidos pelo sistema podem ser usados pela aplicação para qualquer fim. Assim, temos uma tabela de atributos extensível e de uso geral, à disposição da aplicação. Isto permite que os objetos de interface mantenham “estado” próprio. Além disso, isto permite que atributos específicos para uma plataforma sejam

especificados e interpretados pelo *driver* correspondente, sem qualquer conseqüência para as outras plataformas. Assim, é possível fazer *fine-tuning* para cada plataforma usando um mesma especificação LED.

4.2 IUP: um toolkit baseado em LED

A partir da linguagem LED, especificamos um *toolkit*, com aproximadamente 30 funções, para construção e manipulação de diálogos em programas. Este *toolkit* é chamado IUP e é basicamente uma API para implementar LED, contendo funções para:

- converter as especificações em LED para objetos do sistema de interface "nativo" (isto é, o IUP intermedia de forma portátil a comunicação entre a aplicação e os *drivers)*;
- criar elementos de interface sem utilizar LED;
- registrar funções da aplicação correspondentes às ações usadas em LED (isto corresponde às *callbacks* existentes em outros sistemas);
- associar nomes aos elementos de interface;
- exibir e esconder diálogos;
- consultar e estabelecer atributos para os elementos de interface.

Todo o código foi escrito em ANSI C e é portátil para diversos ambientes, como Microsoft Windows, OpenLook via XView, Motif e DOS (veja Figura 4.1). Para o DOS, escrevemos um sistema de interface completo e portátil, formado por um conjunto de *drivers* para diversos dispositivos gráficos, um sistema de janelas, um *window manager* e um *toolkit* de apoio, estes com aparência Motif. Como mencionamos anteriormente, este sistema de interface é uma aplicação puramente gráfica, à qual aplicamos a estratégia de portabilidade já descrita no Capítulo 3.

O fluxo de controle em uma aplicação que usa IUP com LED é análogo ao de aplicações que usam outros *toolkits* e pode ser resumido da seguinte forma:

- inicializar IUP, chamando a função `int IupOpen (void);`
- criar os diálogos através da compilação de um arquivo LED, chamando a função `int IupLoad (char *)`, ou fazendo chamadas IUP para criar cada elemento de interface;
- registrar funções correspondentes às ações usadas em LED (no exemplo, `do_replace` e `do_cancel` são ações e não funções da aplicação), chamando a função `Icallback IupSetFunction(char *, Icallback);`
- exibir os diálogos iniciais, chamando a função `int IupShow(Ihandle *);`

- passar o controle para o IUP, chamando a função `int IupmainLoop (void),` que então fica esperando ações do usuário e chamando a função da aplicação correspondente.

4.2.1 Integrando IUP com LED

Em LED, podemos associar nomes aos elemento de interface, porém estes nomes não podem ser usados *diretamente* nas funções IUP que criam e manipulam elementos de interface. As funções IUP esperam receber um identificador do tipo `Ihandle*;` portanto, a aplicação, ao referenciar um elemento de interface criado em LED, tem que chamar a função `Ihandle* IupGetHandle(char *)`, para consultar o seu identificador. Como exemplo, apresentamos um código IUP que exibe o diálogo da Figura 4.2 especificado em LED. Apesar de não estarmos testando erros, todas as funções IUP retornam um código indicando o sucesso ou falha na execução da função.

```
/* carrega e compila a especificação em LED */
IupLoad ("attention.led");

/* registra as funções da aplicação */
IupSetFunction ("do_ok", (Icallback)funcao_ok);
IupSetFunction ("do_cancel", (Icallback)funcao_cancel);

/* exibe o diálogo */
IupShow(IupGetHandle("attention"));/*repare o aninhamento de funções*/
```

4.2.2 Criando elementos de interface em IUP

A criação de elementos de interfaces em IUP difere em dois pontos da criação em LED. O primeira diferença é que em IUP a associação de nomes ao elementos de interface não ocorre no mesmo momento da criação dos elementos, como acontece em LED. Ao criarmos um elemento, a função IUP retorna um identificador que deve ser passado para a função `Ihandle *IupSetHandle(char*,Ihandle*)`, para associarmos um nome ao elemento de interface.

A segunda diferença é que em LED a definição dos atributos ocorre no mesmo instante da criação dos elementos, enquanto que em IUP é preciso que o elemento já esteja criado, para se poder definir os seus atributos.

Como exemplo, apresentamos abaixo a descrição do diálogo da Figura 4.2.

```
/* declaração dos identificadores dos elementos de interface */
Ihandle      *icancel,*ireplace,*iwarning,*ibuttons,
             *iprompt,*ibody,*iconfirm;

/* cria os elementos de interface */
icancel  = IupButton ("Cancel", "do_cancel");
ireplace = IupButton ("Replace", "do_replace");
iwarning = IupLabel  ("File already exists!");
ibuttons = IupHbox   (IupFill(),ireplace,IupFill(),
                      icancel,IupFill(),NULL);
iprompt  = IupHbox   (IupFill(), iwarning, IupFill(), NULL);
ibody    = IupVbox   (IupFill(), iprompt, IupFill(),ibuttons,NULL);
iconfirm = IupDialog (ibody);

/* define um título para o diálogo */
IupSetAttribute (iconfirm, "TITLE", "Attention");

/* registra as funções da aplicação */
IupSetFunction ("do_ok", (Icallback)funcao_ok);
IupSetFunction ("do_cancel", (Icallback)funcao_cancel);

/* exibe o diálogo */
IupShow (iconfirm);
```

No trecho de código acima, não foi preciso definir nomes para os elementos de interface; nos referimos aos elementos através dos seus próprios identificadores que foram retornados pelas funções que criaram os elementos. Para associarmos nomes aos elementos de interface, devemos chamar `Ihandle* IupSetHandle(char*,Ihandle*)` após cada função de criação, como mostra o código abaixo:

```
/* cria os elementos de interface e associa o nome "cancel" */
icancel  = IupButton ("Cancel", "do_cancel");
IupSetHandle ("cancel", icancel);
```

Da mesma forma que em LED, não é necessário guardar os identificadores dos elementos intermediários; o diálogo da Figura 5.2 pode ser especificado assim:

```
iconfirm = IupDialog(
        IupVbox(
             IupFill(),
             IupHbox(
               IupFill(),
               IupLabel("File already exists!"),
               IupFill(),NULL),
             IupFill(),
             IupHbox(
               IupFill(),
               IupButton("Replace", "do_replace",
               IupFill(),
             IupButton("Cancel","do_cancel"),
                IupFill(), NULL),
             NULL));

IupSetAttribute (iconfirm, "TITLE", "Attention");
```

4.3 Implementação

A implementação do IUP/LED tem como principal rotina aquela que converte do modelo de *layout* abstrato para o modelo concreto. O algoritmo implementado ocorre em três fases. A primeira calcula o tamanho natural de cada elemento (ver Tabela 4.1), isto é, o menor tamanho que engloba o elemento de interface. A segunda fase calcula o tamanho corrente, isto é, o tamanho com o qual o elemento realmente vai ser exibido para o usuário. A terceira e última fase calcula a posição de cada elemento de interface. Na prática, a implementação deste algoritmo exigiu a resposta para algumas questões:

- qual a política de distribuição de espaços vazios entre **fill** e **canvas**?
- um **fill/canvas** mais profundo na hierarquia recebe menos espaços do que um **fill/canvas** menos profundo?
- quando o usuário define explicitamente o tamanho de um elemento (especificando o atributo `SIZE`**)**, a que tamanho ele está se referindo: ao natural ou ao corrente?
- em que unidades o usuário define os tamanhos dos elementos de interface?

As respostas a estas questões e suas combinações foram extensivamente estudadas com a ajuda de protótipos. Sobre a primeira questão, a solução mais razoável é dar prioridade ao **canvas** em relação ao **fill** na distribuição dos espaços vazios. A justificativa para esta decisão foi encontrada na funcionalidade dos dois elementos em questão: o **fill** serve para justificar elementos, enquanto o **canvas** é o espaço utilizado pela aplicação e pelo usuário para se comunicarem através de objetos próprios da aplicação. Portanto, parece razoável que, quando o usuário aumenta o tamanho de um diálogo, ele esteja querendo aumentar o tamanho do **canvas**.

Elemento	tamanho natural
dialog	igual ao tamanho do elemento que ele contém
radio	não tem tamanho, pois define apenas funcionalidade
menu	menor tamanho que engloba os elementos que ele contém
hbox	a altura é igual a maior altura dos elementos que ele contém e a largura é a soma das larguras dos elementos que ele contém
vbox	a altura é a soma das alturas dos elementos que ele contém e a largura é igual a maior largura entre os elementos que ele contém
button	um pouco maior (depende do sistema nativo) que o tamanho do seu texto ou da sua imagem
canvas	tamanho de um caracter
frame	tamanho é igual ao tamanho do elemento contido mais um espaço (depende do sistema nativo) para o título e borda
hotkeys	não tem tamanho, pois define apenas funcionalidade
image	tamanho igual ao da sua imagem
item	um pouco maior que o tamanho do seu texto
label	tamanho igual ao do seu texto ou imagem
list	tamanho depende do sistema nativo
submenu	um pouco maior que o tamanho do seu texto
text	tamanho um pouco maior (depende do sistema nativo) que o tamanho do texto inicial
toggle	o suficiente para englobar o texto ou sua imagem e a caixa de *feedback* (depende do sistema nativo)
valuator	depende do sistema nativo.

Tabela 4.1: tamanho natural dos elementos de interface.

A divisão de espaços vazios para elementos no mesmo nível na estrutura hierárquica de um diálogo é proporcional: todos recebem a mesma quantidade de espaços vazios. A dúvida é quanto aos elementos que estão em diferentes níveis. Se todos os elementos recebessem a mesma quantidade de espaços, independentemente das suas posições na estrutura hierárquica, não seria possível dividir um diálogo em regiões como ilustra a Figura 4.4, pois todos os elementos receberiam a mesma quantidade de espaços, e o diálogo ilustrado na Figura 4.4 ficaria como ilustra a Figura 4.5 (a) e a Figura 4.5 (b), dependendo de como foi definido o diálogo: uma caixa vertical contendo duas caixas horizontais; ou uma caixa horizontal contendo duas caixas verticais, respectivamente. O algoritmo implementado contempla a divisão de diálogos em regiões, dividindo os espaços vazios de uma caixa igualmente entre os seus elementos expansíveis, que por sua vez dividem os espaços ganhos igualmente entre os seus elementos, e assim por diante. Desta forma, os elementos mais externos ganham mais espaços que os elementos mais internos, numa proporção exponencial.

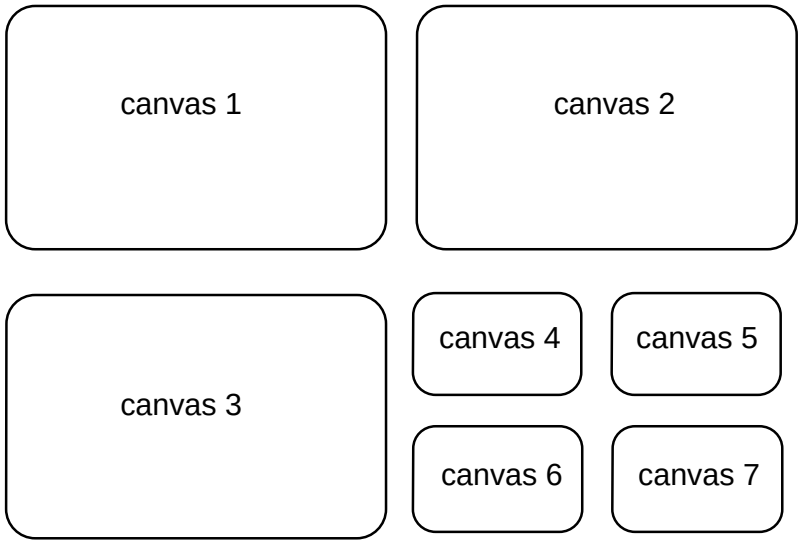


Figura 4.4: sete `canvas` dividido um diálogo em regiões simétricas.

Para solucionar a terceira questão, foi considerado que, ao definir um tamanho, o usuário deseja que o elemento de interface seja exibido com tamanho especificado e não com um outro calculado pelo IUP. Seguindo este raciocínio, fica claro que o tamanho explicitado pelo usuário se refere ao tamanho corrente, assim, o elemento de interface não tem seu tamanho re-calculado após uma alteração no tamanho do diálogo que o contém. A única exceção é para o elemento *dialog*, que tem seu tamanho re-calculado mesmo tendo o seu tamanho definido pelo programador da aplicação.

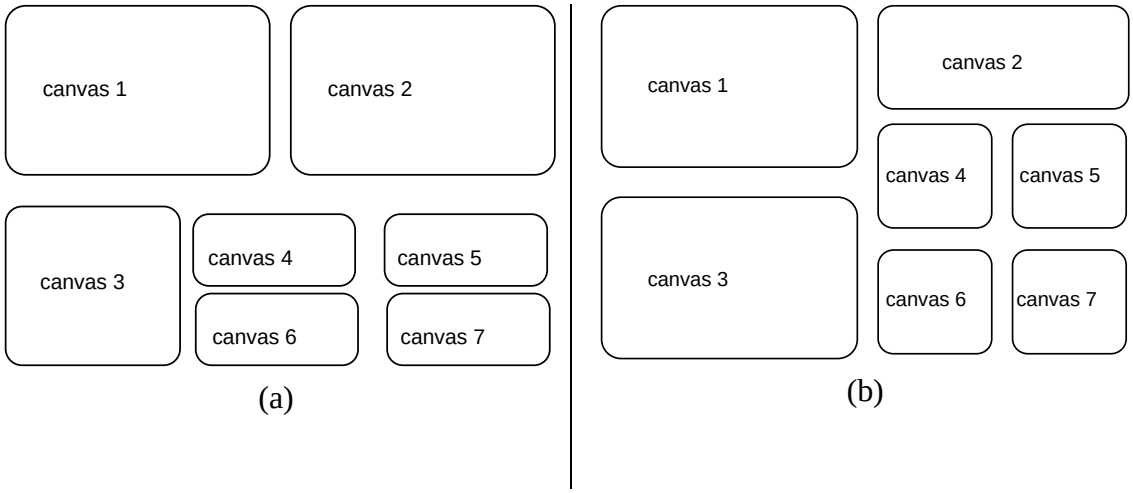


Figura 4.5: sete `canvas` divindido um diálogo em regiões assimétricas

Sobre a última questão levantada, sabia-se apenas que a unidade *raster* (i.e., em *pixels*) não poderia ser utilizada, pois criaria uma óbvia dependência da resolução dos dispositivos na especificação dos elementos de interface. Portanto, adotamos que os valores relacionados com tamanhos devem ser especificados proporcionalmente a uma fração do tamanho médio de um caracter do fonte de caracteres utilizado pelo elemento em questão. A unidade da largura representa 1/4 da largura média de um caracter e da altura representa 1/8 da altura de um caracter (estes valores também são utilizados no MS-Windows). Este tipo de unidade é bastante interessante para descrição de *layout*, pois garante que os elementos de interface mostrem para os usuários as mesmas informações, independente do dispositivo de saída. Como vimos no Capítulo 3, o MS-Windows e o Motif também permitem esta unidade, além de outras, na especificação de tamanhos e posições dos elementos de interface.

A seguir, apresentamos três algoritmos recursivos que foram implementados para executar a conversão de um *layout* abstrato em um *layout* concreto. O primeiro algoritmo calcula o tamanho natural, o segundo o tamanho corrente, e o último a posição de cada elemento de interface.

4.3.1 Algoritmo para o cálculo do *layout* concreto

Abaixo, apresentamos o algoritmo, em pseudocódigo, que calcula o tamanho natural dos elementos de interface. Na verdade, ele calcula apenas os tamanhos dos elementos de composição (*hbox* e *vbox*) e agrupamento (*dialog*); e os tamanhos dos

elementos primitivos (e.g., *label*, *button*, *text*) são conhecidos através de consultas ao *driver* do sistema de interface nativo.

O algoritmo recebe como entrada um nó da estrutura hierárquica de um diálogo, representando um elemento de interface, e calcula o seu tamanho natural. O valor de retorno indica as direções (vertical ou horizontal) em que o elemento pode se expandir e com quais prioridades (alta ou baixa). Por exemplo, um `hbox` que contém um `fill` pode se expandir na horizontal com prioridade baixa, e um `hbox` que contém um `canvas` pode se expandir em ambas as direções com prioridade alta. Esta informação será útil no cálculo do tamanho corrente que promoverá a distribuição dos espaços vazios.

```
FUNÇÃO tamanho_natural (n): inteiro
     SE tipo(n)
          DIALOG:
              calcula tamanho_natural(filho(n))
              expansão(n) ← ambas as direções nas duas prioridades
              largura_natural(n) ← largura_natural(filho(c))
              altura_natural(n) ← altura_natural(filho(c))
          HBOX:
              expansão(n) ← nenhuma direção
              largura_natural(n) ← 0
              altura_natural(n) ← 0
              PARA CADA filho c de n FAÇA
                  expansão(n) ← expansão(n) | tamanho_natural(c)
                  largura_natural(n)←largura_natural+largura_natural(c)
                  altura_natural(n) ← MAX( altura_natural(n),
                                           altura_natural(c))
          VBOX:
              expansão(n) ← nenhuma direção
              largura_natural(n) ← 0
              altura_natural(n) ← 0
              PARA CADA filho c de n FAÇA
                  expansão(n) ← expansão(n) | tamanho_natural(c)
                  largura_natural(n) ← MAX(largura_natural(n),
                                           largura_natural(c))
                  altura_natural(n) ← altura_natural+altura_natural(c);
          BUTTON,LABEL,TEXT:
              pergunta tamanho natural ao driver
              expansão(n) ← nenhuma direção
          CANVAS:
              consulta o driver para o pegar o tamanho natural de n
              expansão(n) ← nas duas direções com prioridade alta
          FILL:
              largura_natural(n) ← 0
              altura_natural(n) ← 0
              SE n está contido em um hbox ENTÃO
                  expansão(n) ← horizontal com prioridade baixa
              SENAO
                  expansão(n) ← vertical com prioridade baixa
RETORNA expansão(n)
```

O algoritmo anterior prepara os elementos de interface para o cálculo dos tamanhos correntes. Abaixo, descrevemos, em pseudo-código, o algoritmo que distribui os espaços vazios e calcula os tamanhos correntes dos elementos de interface. Este algoritmo não tem valor de retorno e recebe como parâmetros: um nó da estrutura hierárquica de um diálogo, representando um elemento de interface; e a largura e a altura de um retângulo onde os elementos de interface irão se compor.

```
FUNÇÃO tamanho_corrente (n,larg,alt)
     SE (n pode se expandir horizontalmente) ENTÃO
          largura_corrente(n) ← MAX(largura_natural(n),larg)
     SENAO
          largura_corrente(n) ← largura_natural(n)
     SE (n pode se expandir varticalmente) ENTÃO
          altura_corrente(n) ← MAX(altura_natural(n),alt)
     SENAO
          altura_corrente(n) ← altura_natural(n)
     SE tipo(n)
          DIALOG:
              tamanho_corrente (filho(n),largura_corrente(n),
                                          altura_corrente(n))
          HBOX:
              SE n expande na horizontal com prioridade alta ENTÃO
                  espacos ← largura_corrente(n)-largura_natural(n) /
                                número de filhos de n que podem se
                                expandir na horizontal com prior. alta
                  prioridade ← alta
              SENAO
                  espacos ← largura_corrente(n)-largura_natural(n) /
                                número de filhos de n que podem se
                                expandir na horizontal
                  prioridade ← baixa

              PARA CADA filho c de n FAÇA
                  SE c expande na horiz com prioridade alta e
                          prioridade for alta ENTÃO
                       larg ← espacos
                  SENAO SE c expande na horiz. com prioridade baixa e
                             prioridade for baixa ENTÃO
                       larg ← espacos
                  SENAO
                       larg ← 0
                  tamanho_corrente(c,largura_corrente(c)+larg,
                                     altura_corrente(n))
          VBOX:
              SE n expande na vertical com prioridade alta ENTÃO
                  espacos ← altura_corrente(n)-altura_natural(n) /
                                número de filhos de n que podem se
                                expandir na vertical com prior. alta
                  prioridade ← alta
              SENAO
                  espacos ← altura_corrente(n)-altura_natural(n) /
                                número de filhos de n que podem se
                                expandir na vertical
                  prioridade ← baixa
```

```
PARA CADA filho c de n FAÇA
    SE c expande na vert. com prioridade alta e
            prioridade for alta ENTÃO
        alt ← espacos
    SENAO SE c expande na vert. com prioridade baixa
            prioridade for baixa ENTÃO
        alt ← espacos
    SENAO
        alt ← 0
    tamanho_corrente(c,largura_corrente(n),
                      altura_corrente(c)+alt)
```

Para completar a conversão do *layout* abstrato para o *layout* concreto, descrevemos, em pseudo-código, o algoritmo de posicionamento dos elementos de interface. O algoritmo não possui valor de retorno e tem como parâmetros: um nó da estrutura hierárquica; e um par de coordenadas (x,y) indicando a posição do canto superior esquerdo onde o elemento deve ser posicionado.

```
FUNÇÃO posiciona (n, x, y)
    pos_x(n) ← x
    pos_y(n) ← y
    SE tipo(n)
        DIALOG:
            posiciona(filho(n),x,y)
        HBOX:
            PARA CADA filho c de n FAÇA
                posiciona(c,x,y)
                x ← x + largura_corrente(c)
        VBOX:
            PARA CADA filho c de n FAÇA
                posiciona(c,x,y)
                y ← y + altura_corrente(c)
```

A conversão do modelo de *layout* abstrato para o concreto tem a sua complexidade estabelecida pela soma das complexidades dos três algoritmos apresentados. Os três algoritmos percorrem todos os nós da árvore hierárquica apenas uma vez, sendo portanto lineares no número de elementos de interface que o diálogo contém. Podemos concluir que a conversão para o *layout* concreto também é linear com uma constante de proporcionalidade igual a 3. Esta constante não impõe nenhum problema para o cálculo em tempo real do *layout* concreto. Entretanto, o recálculo só ocorre no final da alteração do tamanho do diálogo promovida pelo usuário, para evitar o efeito de “pisca-pisca” que seria gerado em conseqüência do remanejamento dos elementos de interface, que obriga apagar cada elemento da posição antiga e redesenhá-lo na posição nova a cada movimento de *mouse*.

5

Conclusão

5.1 Algumas experiências com o IUP/LED

O IUP/LED já está sendo utilizado em vários projetos do TeCGraf e foi extensivamente utilizado por alunos do curso de Computação Gráfica e por alguns alunos do curso de Interface com Usuário ministrados no primeiro semestre de 1993 pelo Departamento de Informática da PUC-Rio. Nesta seção, examinamos a utilização do IUP/LED no projeto PETROX do TeCGraf; este projeto requer a definição de quase cinqüenta diálogos, sendo portanto um bom programa para avaliar o modelo de *layout* abstrato. Examinaremos também a utilização do IUP/LED no curso de Computação Gráfica, onde os alunos tiveram que desenvolver um editor gráfico interativo que é um programa que requer uma interação muito grande entre o IUP/LED e o sistema gráfico.

5.1.1 Projeto PETROX

O projeto PETROX envolve o desenvolvimento de um programa gráfico interativo multi-plataforma para a edição de diagramas de processos químicos, que servirão como dados de entrada para um simulador. Os cinqüenta diálogos que capturam os dados numéricos associados aos elementos do diagrama estão sendo especificados com a LED por uma engenheira química do CENPES. É importante ressaltar que esta profissional, ao iniciar este projeto, não possuía nenhum conhecimento sobre interface com o usuário e, após um mês de treinamento no TeCGraf, conseguiu absorver os conceitos de interface com o usuário e aprender o *toolkit* IUP e a linguagem LED. Até o presente momento, ela já especificou trinta diálogos, contendo aproximadamente quatro mil elementos de interface. Como os diálogos possuem grupos de elementos de interface comuns, ela isolou estes grupos em arquivos, formando, assim, uma biblioteca. Desta forma, sempre que um diálogo precisava de um destes grupos, ele era retirado da biblioteca e inserido na especificação do diálogo. Esta estratégia foi decisiva na elaboração dos diálogos. Entretanto, a modificação de um grupo exige que todos os diálogos sejam alterados, apagando-se o grupo antigo e inserindo-se o novo. A solução deste problema é a utilização de um pré-processador similar ao que existe na linguagem C. Desta forma, a mudança em um grupo exige apenas executar o pré-processador para cada um dos diálogos que contenha o grupo modificado. Esta execução pode ser automatizada com a utilização de utilitários do tipo `make`.

À medida em que os diálogos foram sendo construídos, surgiram necessidades de utilização de elementos de interface não oferecidos pelo IUP/LED. Basicamente, foram dois os elementos requisitados:

- o *drop down list*, que tem a semântica do elemento primitivo `list`, mas com apresentação diferente. Este elemento lista, ao invés de exibir, permanentemente, toda a lista para o usuário, exibindo apenas a opção correntemente selecionada ao lado de um botão com a imagem de uma seta para baixo. Este botão, ao ser selecionado, apresenta a lista de opções, permitindo a escolha de uma nova opção;
- o *vector*, que possibilita entrar com valores de um vetor, sem que seja necessário definir um elemento do tipo `text` para cada posição do vetor.

O número de *fills* utilizados no alinhamento dos elementos de interface nos diálogos do projeto PETROX é bastante grande. Com o objetivo de diminuir esse número, foi criado o atributo `ALIGNMENT` para os elementos de composição (`hbox` e `vbox`) que alinham os seus elementos de acordo com o valor deste atributo. No `hbox` os valores possíveis são: `IUP_TOP`, `IUP_CENTER` e `IUP_BOTTOM;` no `vbox` são: `IUP_LEFT`, `IUP_CENTER` e `IUP_RIGHT`.

5.1.2 Editor Gráfico

Os alunos do curso de Computação Gráfica tiveram que desenvolver um editor gráfico interativo utilizando como sistema de interface o IUP/LED e como sistema gráfico o GKS/puc. Eles começaram a utilizar o IUP/LED após uma única aula, onde foi exposto o modelo abstrato de *layout* adotado e as funções implementadas e foi entregue, como documentação, um guia rápido de referência das funções IUP/LED e dos atributos de cada elemento de interface. Os alunos conseguiram especificar os diálogos e utilizar o *toolkit* sem maiores problemas. Entretanto, alguns alunos encontraram dificuldades no modelo de programação, que deixou de ser seqüencial e passou a ser assíncrono. Isto não é um problema do IUP/LED e sim do modelo de interação com usuário, que retira o controle da aplicação e passa para o usuário. Este problema foi comum – e ainda é – na migração de programas com interfaces de menus numéricos para interfaces *event-driven*.

O sistema gráfico foi utilizado passivamente e todo o *input* foi feito com o IUP. O único problema detectado na utilização deste modelo é que as posições retornadas pelo IUP são em unidades *raster* relativas ao canto superior esquerdo do *canvas* e as posições que o GKS espera são em coordenadas do mundo (relativas ao canto inferior

esquerdo da *workstation)* definidas pelo programador. A solução foi desenvolver uma rotina de transformação entre os dois sistemas e incorporá-la à interface IUP para o GKS/puc.

Uma questão importante é que o padrão GKS, como foi elaborado sem a preocupação com interfaces WIMP, não oferece nenhum mecanismo para tratar alterações no tamanho da *workstation (canvas*). A solução adota foi fehar e abrir a *workstation* sempre que o *canvas* mudar de tamanho. Desta forma, o GKS/puc consegue atualizar os parâmetros de transformações de coordenadas em função do novo tamanho do *canvas,* sem que seja necessário introduzir uma rotina fora do padrão GKS no GKS/puc.

5.2 Contribuições do IUP/LED

Abaixo, listamos as contribuições fornecidas pelo IUP/LED na construção de programas interativos portáteis :

- IUP/LED define um modelo abstrato para *layouts*, possibilitando a especificação de diálogos de uma forma natural, sem precisar explicitar a posição dos elementos de interface;
- o modelo adotado pelo IUP/LED permite que ele seja implementado sob os mais variados tipos de sistemas de interface, indo desde os sistemas baseados em caracteres até aqueles que utilizam massivamente recursos gráficos;
- LED descreve um diálogo a partir da sua funcionalidade; os atributos de aparência são opcionais;
- LED permite que aplicações sejam customizadas para diferentes usuários e plataformas, pelos próprios usuários;
- LED oferece um mecanismo de atributos que possibilita fazer ajustes finos específicos para um sistema de interface. Através deste mecanismo, também é possível agregar dados da aplicação aos elementos de interface, evitando a criação de variáveis globais;
- LED é uma linguagem de expressões, cuja sintaxe é muito simples, permitindo rápido aprendizado;
- IUP é um *toolkit* virtual que permite a construção de programas interativos portáteis, não exigindo do programador conhecimentos do sistema de interface nativo;
- IUP permite que os programas possuam tanto o *look-and-feel* do sistema nativo, beneficiando o usuário de um único ambiente, quanto um *look-and-feel*

fixo, beneficiando o usuário de uma única aplicação, que precisa rodar o programa em diferentes máquinas;

- Semelhante ao XView, o IUP tem aproximadamente trinta funções, um conjunto bastante pequeno se comparado com as centenas de funções do MS-Windows, do Macintosh Toolbox e do Motif. Este número reduzido permite um rápido aprendizado das funções IUP.

5.3 Análise comparativa com outros Sistemas Portáteis de Interface

Existem outros sistemas de interface que, como o IUP/LED, propõem soluções para resolver o problemas de portabilidade, tal como: IntGraf [TeCGraf (1991)], CIRL/PIWI [Cowan et al. (1992)], XVT [Rochkind (1989)] e OI Toolkit [Neuron Data (1991)]. Nesta seção, fazemos uma análise comparativa do IUP/LED com o IntGraf e o com o CIRL/PIWI.

5.3.1 IntGraf

IntGraf é um sistema portátil de interface gráfica com usuário desenvolvido sobre o GKS/puc. Ele é composto de um gerador de menu (`criamenu`) e de uma biblioteca de diálogos pré-definidos. Apesar de utilizar recursos gráficos e *mouse*, o IntGraf não é um sistema baseado na metáfora de uma mesa de trabalho, não possuindo nem estando integrado em um sistema de janelas. Ele assume que toda aplicação é composta de menu de barra e de uma única região de trabalho; e que a interação com o usuário se desenvolve seqüencialmente através de diálogos pré-definidos controlados pela aplicação. Na época do seu desenvolvimento, os diálogos das aplicações eram predominantemente seqüenciais, e isto permitiu a utilização deste modelo.

O IntGraf obteve uma boa aceitação tanto pelos programadores das aplicações quanto pelos usuários finais, principalmente por três motivos:

- facilidade de uso;
- aparência 3-D dos elementos de interface;
- portabilidade.

A facilidade de uso foi alcançada pela simplicidade do modelo adotado para as aplicações. Os diálogos pré-definidos oferecidos pelo IntGraf são simples, bem definidos e atendem a grande parte das necessidades das aplicações. Cada diálogo

implementa uma única técnica de interação, não havendo duplicidade. Desta forma, a programação da captura de dados de entrada para uma aplicação fica trivial, pois o programador precisa apenas definir a ordem dos diálogos pré-definidos que irão interagir com o usuário. Por outro lado, o usuário final consegue aprender facilmente uma aplicação IntGraf, pois além de, simplesmente ter que responder diálogos controlados pela aplicação, os diálogos tem sempre a mesma aparência independente da aplicação e do ambiente computacional.

A aparência 3-D dos elementos de interface proporciona um *design* bonito e agradável para as aplicações, além da interação com usuário ocorrer de uma forma clara e harmoniosa. Os benefícios para o usuário final e para o programador da aplicação são óbvios [Marcus (1992)].

A portabilidade foi o que realmente garantiu o sucesso do IntGraf. As aplicações IntGraf podem ser implementadas em ambientes computacionais tão diferentes quanto PC-DOS, Unix/X11, VAX/VMS, sem precisar alterar uma linha de código da aplicação, pelo menos no que diz respeito à interface com usuário. Isto é possível graças ao GKS/puc que foi utilizado como base no desenvolvimento do IntGraf.

A principal diferença entre o IntGraf e o IUP/LED se concentra no modelo de aplicação adotado por cada um dos sistemas. Enquanto no IntGraf uma aplicação é composta de um menu de barra e de uma única região de trabalho, e a interação com usuário se desenvolve seqüencialmente através de diálogos pré-definidos, no IUP/LED o modelo para aplicações é baseado na metáfora de uma mesa de trabalho, permitindo que a aplicação crie os seus próprios diálogos com os quais o usuário interage assincronamente.

O modelo adotado pelo IUP/LED oferece uma liberdade muito grande na especificação dos diálogos, permitindo a construção de diálogos bem mais poderosos que os diálogos pré-definidos do IntGraf, através de composições de elementos de interface que implementam diferentes técnicas de interação. Esta composição minimiza o número de diálogos com os quais o usuário tem que interagir, pois o usuário programador, ao invés de utilizar um diálogo para cada tipo de interação, pode combiná-los em um único diálogo. Entretanto, esta combinação exige do programador cuidados na elaboração dos diálogos para que estes sejam harmoniosos e facilitem o seu entendimento pelo usuário final. Estes cuidados são mínimos, se comparados com o resultado final proporcionado pelos diálogos compostos.

Uma diferença marcante entre os dois sistemas, também decorrente da diferença entre os modelos da aplicação adotados, é sobre o controle da interação com o usuário

final. No IntGraf, a interação com o usuário ocorre seqüencialmente: a aplicação exibe um diálogo e espera que o usuário responda, para então exibir um outro diálogo. Este tipo de interação conversacional está ultrapassado [Hartson—Hix (1989)]. O IUP/LED permite interação conversacional e assíncrona. Na interação assíncrona, a aplicação exibe vários diálogos simultâneos para o usuário, e ele decide quando e com qual diálogo vai interagir. Este modelo de interação é uma tendência atual nas interfaces do programa e está sendo muito utilizado nas interfaces por manipulação direta.

Uma outra diferença significante entre os dois sistemas é no tratamento da região de trabalho, isto é, a região que a aplicação utiliza para interagir com usuários através de objetos próprios. Primeiramente, a região de trabalho no IntGraf responde a apenas uma única ação do usuário: quando o usuário aponta (pressionando e soltando o botão da esquerda do *mouse)* uma posição dentro da região de trabalho. Esta ação não é suficiente para realizar a principal técnica de interação das interfaces por manipulação direta: *drag-and-drop*, que consiste em pressionar um botão do *mouse* e movimentá-lo sem soltar o botão pressionado. Esta deficiência é imposta pelo GKS/puc, que não permite este tipo de interação. No IUP/LED, a região de trabalho é representada pelo elemento de interface **canvas**. O **canvas** responde a todas as ações do usuário, possibilitando a implementação, pela aplicação, da técnica de interação *drag-and-drop*.

Além disso, toda a aplicação IntGraf possui uma única região de trabalho. Isto faz com que as aplicações que precisam de várias áreas de trabalho tenham que controlar a divisão na única região de trabalho permitida pelo IntGraf. O IUP/LED assume este controle, pois permite que uma aplicação tenha um número ilimitado de áreas de trabalho, podendo até não ter nenhuma.

Por último, comparamos a utilização do sistema gráfico nas aplicações IntGraf e IUP/LED. Como o IntGraf foi desenvolvido sobre o GKS/puc, eles estão intimamente ligados. Desta forma, uma aplicação IntGraf é obrigada a utilizar como sistema gráfico o GKS/puc. O IUP/LED não foi desenvolvido sobre nenhum sistema gráfico; pelo contrário, seu projeto inicial prevê mecanismos de integração dele com qualquer sistema gráfico. Na versão atual do IUP/LED, existem mecanismos de integração com o GKS/puc e com a biblioteca Xlib [Nye (1990)]. Assim, a aplicação fica livre para escolher o sistema gráfico a ser utilizado. Ao contrário das aplicações IntGraf, as aplicações IUP/LED utilizam os sistemas gráficos passivamente, isto é, não utilizam os recursos de *input* oferecidos pelos sistemas gráficos; todos os dados necessários são capturados através do sistema de interface. A principal conseqüência é que os *feedbacks* padrões para entrada interativa no **canvas** (e.g., *rubberbanding*) têm agora que ser produzidos pela aplicação. Este não é um grande problema, pois a aplicação

provavelmente usará uma biblioteca padrão, facilmente construída a partir da própria tecnologia de sistemas gráficos interativos.

5.3.2 CIRL/PIWI

CIRL/PIWI [Cowan et al. (1992)] é uma ferramenta portátil de interface com usuário desenvolvida na Universidade de Waterloo. Esta ferramenta utiliza como estratégia para portabilidade a estratégia de portar uma abstração de interface com usuário. Esta abstração tem dois componentes: CIRL, uma linguagem de especificação de elementos de interface; e PIWI, um *toolkit* virtual semelhante ao IUP.

O CIRL é uma *tagging language* para descrever elementos de interface. A descrição dos elementos de interface ocorre em dois arquivos distintos. O primeiro arquivo, obrigatório, descreve a funcionalidade; o segundo arquivo, opcional, descreve as aparências dos elementos de interface definidos no primeiro arquivo. Esta separação da funcionalidade e da aparência permitem a especificação dos diálogos apenas por sua funcionalidade, semelhantemente ao que ocorre na LED. Estes dois arquivos são compilados juntos e resultam na descrição dos elementos de interface na linguagem de especificação de elementos de interface do sistema nativo. O compilador é acoplado a uma base de conhecimento que permite especificar a aparência dos elementos que o usuário não definiu.

O PIWI é um *toolkit* virtual semelhante ao IUP e está implementado em Macintosh, X11/Motif, Microsoft Windows 3.0, e Presentation Manager OS/2. Além das funções de interface com usuário, também estão disponíveis funções gráficas para desenho. Assim, PIWI contém um sistema gráfico abstrato.

Os elementos de interface se comunicam com a aplicação através do modelo de eventos. Cada diálogo tem uma rotina tratadora de eventos, que se resume em um conjunto de `if`'s aninhados que tratam eventos gerados pelos usuários, pelo próprio sistema nativo, ou por outras aplicações.

Comparando esse sistema com o IUP/LED, detectamos algumas semelhanças, entre elas:

- implementam uma linguagem de especificação;
- os elementos de interface são definidos a partir da sua funcionalidade;
- os *layouts* são definidos sem precisar definir as suas coordenadas;
- implementam um *toolkit* virtual;
- suas aplicações herdam o *look-and-feel* do sistema nativo.

Porém, existem várias diferenças que distanciam uma solução da outra:

- o compilador de CIRL contém uma base de conhecimento sobre o *look-and-feel* dos vários sistemas de interface. Isto permite que o compilador decida algumas questões, relacionadas com *layout*, que não foram definidas na especificação. Por outro lado, a LED define um modelo bastante simples de definição de *layouts*, baseado no paradigma *boxes-and-glue* do processador de texto TEX;
- o compilador CIRL gera uma especificação na linguagem do sistema de interface nativo que, posteriormente, precisa ser ligada à aplicação. A especificação em LED pode ser compilada em tempo de execução da aplicação, diminuindo, assim, o tempo de prototipação;
- os elementos de interface só podem ser criados através da CIRL. Já no IUP/LED, os elementos de interface pode ser definidos utilizando a LED ou utilizando serviços do IUP;
- o IUP/LED oferece o mecanismo de atributos, que possibilita, de uma forma consistente, definir aparências dos elementos de interface, fazer *fine-tuning* para sistemas de interfaces específicos e associar dados aos elementos de interface, evitando a criação de variáveis globais. O sistema CIRL/PIWI não oferece o mecanismo de atributos. As aparências dos elementos de interface são definidas em um arquivo opcional, o *fine-tuning* é feito sobre a especificação gerada pelo compilador CIRL, e não é possível associar dados aos elementos de interface;
- no CIRL/PIWI, a comunicação dos elementos de interface com a aplicação é baseada no modelo de eventos, enquanto no IUP/LED é baseada no modelo de *callbacks;*
- o IUP/LED permite um *look-and-feel* fixo, além do *look-and-feel* nativo;
- o CIRL/PIWI permite um melhor *fine-tuning* da interface com usuário que o IUP/LED, pois ele gera uma descrição dos elementos de interface na própria linguagem de especificação do sistema de interface nativo, onde então é feito o ajuste fino; enquanto o IUP/LED só permite fazer *fine-tuning* nas características implementadas no *driver* IUP/LED para o sistema de interface nativo. Entretanto, o IUP permite consultar o identificador nativo de um elemento de interface (via o atributo `WID`), permitindo que o *fine-tuning* seja feito através das próprias funções do sistemas de interface nativo.

5.4 Conclusão

Neste trabalho descrevemos o projeto e a implementação da ferramenta portátil de interface com usuário – IUP/LED. O IUP é um *toolkit* virtual e a LED é uma linguagem de especificação de diálogos. Esta ferramenta permite construir aplicações interativas que sejam portáveis para diferentes ambientes computacionais com um mínimo de esforço. Para as aplicações gráficas interativas, o IUP/LED não exige a utilização de um sistema gráfico específico, mas sugere que ele seja usado apenas passivamente, de modo a poder intermediar toda a interação do usuário com a aplicação.

Discutimos várias estratégias de portabilidade e mostramos que a estratégia adotada no IUP/LED a combinação de portar uma ferramenta de interface com portar uma abstração de interface foi bem escolhida, pois une as vantagens de ambas sem ganhar as suas desvantagens.

Estudamos o problema de descrição de *layout*, vimos a dificuldade de definir *layouts* usando posições geométricas explícitas (modelo de *layout* concreto). Como solução para este problema, o IUP/LED define um modelo de *layout* abstrato baseado no paradigma *boxes-and-glue* do processador de texto TEX. As principais vantagens da utilização deste modelo são:

- liberar o programador de calcular tamanhos e posições dos elementos de interface;
- manter o *layout* abstrato após uma alteração de tamanho promovida pelo usuário da aplicação, ou pela adição ou remoção de elementos.

A desvantagem deste modelo é que a especificação de um diálogo pode ficar muito grande, como aconteceu no projeto PETROX, que teve aproximadamente cem elementos de interface por diálogo. Daí a necessidade de ferramentas auxiliares semelhantes ao pré-processador da linguagem C.

Outra desvantagem deste modelo é que diálogos com *layouts* geometricamente irregulares não podem ser definidos. Entretanto, a ocorrência deste tipo de diálogo é bastante rara. Além disso, sempre existe um *layout* geometricamente regular que consegue expor as informações tão bem, ou até mesmo melhor que o *layout* irregular; tudo depende da criatividade e bom senso do projetista da interface.

A linguagem de especificação de diálogos, LED, implementa o modelo de *layout* abstrato através de uma linguagem de expressões, cuja sintaxe é muito simples, permitindo um rápido aprendizado até mesmo para profissionais de áreas diferentes da ciência da computação. LED é uma ferramenta bastante poderosa, que permite:

- a especificação de elementos de interface sem definir atributos de aparência;
- a separação da definição dos diálogos do código computacional da aplicação;
- a customização para diferentes usuários e plataformas, pelo próprio usuário.

A linguagem LED serviu como base para o desenvolvimento do *toolkit* virtual IUP, que permite a aplicação herdar ou não o *look-and-feel* do sistema de interface nativo. É através do *toolkit* IUP que a aplicação controla a dinâmica dos elementos de interface definidos na LED. Os serviços básicos oferecidos no IUP são:

- converter as especificações na LED para objetos do sistema de interface "nativo" ;
- criar elementos de interface sem utilizar a LED;
- registrar funções da aplicação correspondentes às ações usadas na LED ;
- associar nomes aos elementos de interface;
- exibir e esconder diálogos;
- consultar e estabelecer atributos para os elementos de interface.

O IUP é um *toolkit* muito pequeno, com aproximadamente trinta funções, que, comparado às centenas de funções do MS-Windows, do Macintosh Toolbox e do Motif, se torna muito fácil de ser aprendido. Este pequeno número de funções foi alcançado adotando-se um modelo também utilizado pelo XView. Vários controles de elementos de interface deixaram de ser funções e passaram a ser atributos dos elementos. Desta forma, toda a manipulação dos elementos é feita através de duas funções: uma para consultar e outra para atribuir um valor para um atributo qualquer. A vantagem disto é a facilidade em estender o IUP; a desvantagem é que o programador, ao aprender apenas a API do *toolkit* IUP, sabe muito pouco sobre os elementos de interface, pois é necessário conhecer também os atributos de cada elemento.

A principal dificuldade no desenvolvimento do IUP/LED foi a construção do algoritmo para transformar o *layout* abstrato no *layout* concreto. O projeto deste algoritmo foi apresentado e discutido em detalhes.

5.5 Futuros Trabalhos

Primeiramente, podemos identificar alguns temas para trabalhos futuros que são melhorias no IUP/LED:

- o desenvolvimento de um editor interativo de diálogos para o IUP/LED;
- aprimorar a LED, com o objetivo de permitir fazer referência a elementos de interface antes da sua criação;
- implementar outros elementos de interface, como o `vector` que foi sugerido pelos programadores do projeto PETROX;
- implementar mecanismos de *help* e de *clipboard*.

Um tema bastante interessante e importante diz respeito ao conceito *Multiple Document Interface* (MDI) introduzido no MS-Windows. Este conceito padroniza o uso e a programação das aplicações que permitem aos usuários trabalharem com vários documentos (arquivos) simultaneamente. No OS/2 Presentation Manager e no Macintosh, existem conceitos similares. Uma análise dos benefícios reais do MDI e como este conceito pode ser implementado em outros sistemas de interface é uma questão que deve ser levada em consideração.

Um outro conceito muito utilizado nos sistemas de interface é a troca dinâmica de dados entre aplicações, por exemplo o Dynamic Data Exchange (DDE) do MS-Windows. Apesar deste conceito não tratar diretamente com interface com usuário, ele possibilita a integração de dados entre aplicações, oferecendo grandes benefícios para os usuários.

Por último, propomos um tema que pode ser visto como uma extensão do IUP/LED. O IUP/LED trata de elementos de interface previamente definidos; porém as aplicações gráficas possuem objetos próprios, que respondem a ações de usuário de uma maneira bastante específica da aplicação. Hoje, todo o processo de interação do usuário com os objetos gráficos é de responsabilidade da aplicação. O tema sugerido aqui é propor uma solução que retire da aplicação todo ou parte do controle desta interação.

Referências Bibliográficas

Apple Computer, Inc, *Inside Macintosh-Volume I*, Addison-Wesley, Reading, Massachusetts, USA, 1985.

ANSI, *Computer Graphics - Graphical Kernel System (GKS) Functional Description*, **ANSI X3.124-1985**, June 1985.

Avrahami, G., Brooks, K. P., and Brown, M. H., *A two-view approach to constructing user interfaces*, **Computer Graphics (23)**, pp. 137-146, 1989.

Blackham, G, Building Software for Portability, **Dr. Dobb's Journal** (**146)**, pp. 18-27, December 1988.

Cowan, D. D., and Wilkinson, T. A., *Portable software: an overview*, Proceedings of the 1984 Canadian Conference on Industrial Computer Systems 68-1–68-7, Ottawa, May 1984.

Cowan, D. D., Durance, C. M., Giguère E., and Pianosi, G. M., *CIRL/PIWI: A GUI Toolkit Supporting Retargetability*, Technical Report CS-92-28, University of Waterloo, Dept. of Computer Science, Waterloo, Ontario, 1992; a ser publicado em **Software: Practice and Experience**.

Deininger, A. O., and Fernandez, C. V., *The HP OSF/Motif Graphical User Interface*, **HP Journal (41)**, No. 3, pp. 6-12, 1990.

Derraik, A. L. B., *ITF: Sistema de Gerenciamento de Interface Gráfica para Workstations tipo Cell-Units*, Trabalho Final do curso de Engenharia de Computação, Pontifícia Universidade Católica, Rio de Janeiro, 1992.

Durance, C. M., *An approach to application software mobility across user interface toolkits*. Master's thesis, Faculty of Mathematics, University of Waterloo, Ontario, Canada, 1990.

Figueiredo, L. H., Souza, C. S., Gattass, M., and Coelho, L. C. G., *Geração de interfaces para captura de dados sobre desenhos*, Anais do V SIBGRAPI, pp. 169-175, 1992.

Figueiredo, L. H., Gattass, M., and Levy, C. H., Uma Estratégia de portabilidade para aplicações gráficas interativas, Anais do VI SIBGRAPI, pp. 203-211, 1993.

Frey, Donnalyn, Unix vs. Unix, **Dr. Dobb's Journal** (**146)**, pp. 28-35, December 1988.

Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F., *Computer Graphics - Principles and Practice*. Second Edition, Addison-Wesley, Reading, Massachusetts, 1990.

Gattass, M., and Levy., C. H., *New Techniques to Aid Interactive Graphical Finite Element Analysis*, Anais do Workshop Internacional de Aplicação de Mecânica Computacional em Geotecnia, Rio de Janeiro, 29-31 de junho de 1991.

Green, M., *The University of Alberta user interface management system*, Proceedings of SIGGRAPH'85,12th Annual Conference (San Francisco, Calif., July 22-26). ACM, New York, pp. 205-213, 1985.

Hartson, H. R. and Hix, D., *Human-Computer Interface Development: Concepts and Systems for its Management*, **ACM Computing Surveys (21)**, No. 1, pp. 9-93, March 1989.

Hartson, H. R., Hix, D., and Ehrich, R. W., *A human-computer dialogue management system*, Proceedings of INTERACT'84, First IFIP Conference on Human-Computer Interaction (London, Sept.), International Federation for Information Processing, pp. 57-61, 1984

Heller, D., *Volume Seven: XView Programming Manual*, Second Edition, O'Reilly & Associates, Inc., Sebastopol, California, July 1990.

Hix, D., *Generation of User Interface Management Systems*, **IEEE Software**, pp. 77-87, September 1990.

IEEE, *Standard 1003.1-1988 (Posix)*, 1988.

Jacob, R. J., K., *A Specification Language for Direct Manipulation User Interfaces*, **ACM Transactions on Graphics (5)**, No. 4, pp. 283-317, October 1986.

Kernighan, B., and Ritchie, D., *The C programming language*, Second Edition, Addison-Wesley, Reading, Massachusetts, USA, 1988

Krisam, Brock C., and Taylor, Keith M., *The HP OSF/Motif Window Manager*, **HP Journal (41)**, No. 3, pp. 12-26, 1990

Knuth, D. E., *The T_EX book*, Addison-Wesley,1984.

Linton, M. A., Vlissides, J. M., and Calder, P. R., *Composing user interfaces with InterViews*, **IEEE Computer**, No. 22, pp. 8-22, 1989.

Lucena, C. J. P., Cowan, D. D., Campos, I. M., and Cabral, R. H. B., *Interface as Specifications in the MIDAS User Interface Development System*, **ACM SIGSOFT (15)**, No. 2, pp. 55-72, April 1990.

Marcus, A., *Graphic Design for Electronic Documents and User Interfaces*, ACM Press Tutorial Series, Addison-Wesley, 1992.

Marcus, A., van Dam, A., *User Interface Developments for the Nineties*, **IEEE Computer (24)**, No. 9, pp. 49-57, September 1991.

Microsoft, *Visual Basic Programming System for Windows version 2.0, Programmer's Guide*, Microsoft Corporation, 1992.

Neelamkavil, F., Mullarney, O., *Separating graphics from applications in the design of user interfaces*, **The Computer Journal**, No, 33, pp. 437-443, 1990.

Neider, J., Davis, T. and Woo, M., (Open GL Architecture Review Board), *Open GL Programming Guide*, Addison-Wesley, Reading, Massachusetts, USA, 1993.

Ney, A., *Xlib Reference Manual*, O'Reilly & Associates, 1990.

Open Software Foundation, *OSF/MOTIF Programmer's Guide*, Revision 1.1, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Petzold, C., *Programming Windows: the Microsoft guide to writing applications for Windows 3*, Microsoft Press, Redmond, Washington, 1990.

Rochkind, M. J., *XVT: a virtual toolkit for portability between window systems*, USENIX, pp. 151-163, Winter 1989.

Russo, E. E. R., *Um Sistema de Interpretação de Diálogos Gráficos*. Dissertação de Mestrado, Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, 1988.

Scheifler, R. W., and Gettys, J., The X Window System, **ACM Transactions on Graphics (5)**, No. 2, pp. 79-109, April 1986.

Shneiderman, B., *Direct Manipulation: a step beyond programming languages*, **IEEE Computer (16)**, pp. 57-69, August 1983.

Sun Microsystems Inc., *Open Look Intrinsics Toolkit Widget Set Programmers's Guide*, Revision A, 11 June 1990.

Sun Microsystems Inc., *Open Look: Graphical User Interface Application Style Guidelines*, Addison-Wesley, Reading, Massachusetts, USA, 1991

TeCGraf, *Manual de Referência do GKS/puc*, Pontifícia Universidade Católica, Rio de Janeiro, 1989.

TeCGraf, *Manual de Referência do IntGraf: Sistema de Interface Gráfica com Usuário*, Pontifícia Universidade Católica, Rio de Janeiro, 1991.

IUP/LED: Uma Ferramenta Portátil de Interface com Usuário

Carlos Henrique Levy

Dissertação apresentada ao Departamento de Informática da PUC-Rio, no dia 26 de Agosto de 1993, como parte dos requisitos para a obtenção do título de Mestre em Ciências em Informática, tendo sido aprovada pela Banca Examinadora, da qual participaram os seguintes professores:

Prof. Carlos J. P. de Lucena — Orientador
PUC-Rio

Prof. Luiz Henrique de Figueiredo — Co-orientador
Pesquisador do CNPq associado ao IMPA

Prof. Marcelo Gattass
PUC-Rio

Engenheiro Enio E. R. Russo
CENPES-PETROBRÁS

Visto e permitida a impressão
Rio de Janeiro, de de 1993.

Coordenador dos Programas de Pós-Graduação do Centro Técnico Científico