Uma estratégia de portabilidade para aplicações gráficas interativas

Luiz Henrique de Figueiredo[1,2]
Marcelo Gattass[1]
Carlos Henrique Levy[1]

[1]TeCGraf–Grupo de Tecnologia em Computação Gráfica
Departamento de Informática, PUC–Rio
Rua Marquês de São Vicente, 225
22451–041 Rio de Janeiro, RJ, Brasil
{lhf,gattass,levy}@icad.puc-rio.br

[2]IMPA–Instituto de Matemática Pura e Aplicada
Estrada Dona Castorina, 110
22460–320 Rio de Janeiro, RJ, Brasil
lhf@visgraf.impa.br

Abstract. We discuss strategies for writing portable interactive graphics applications, with emphasis in portable graphics, portable user interfaces, and their integration. We introduce an expression language for describing the layout of user interface dialogs. This language is the basis of a portable user interface toolkit, which supports native look-and-feel across currently available platforms.

Introdução

"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)]:

- sistemas operacionais;
- linguagens de programação;
- dispositivos gráficos;
- interfaces com usuário.

O impacto na mudança de sistemas operacionais e linguagens de programação pode ser minimizado usando padrões de fato, como Posix e ANSI C.

O controle portátil de dispositivos gráficos vem sendo discutido há muitos anos. Como consequência, vários sistemas gráficos foram propostos, embora nenhum tenha emergido como "o melhor" e portanto se tornado um padrão de fato. Apesar disso, há um consenso sobre os tipos de serviços que um bom sistema gráfico bidimensional deve prover. Este consenso é suficiente para escrever aplicações que são facilmente portadas para várias plataformas. Por outro lado, sistemas gráficos tridimensionais ainda estão sendo discutidos, e não há um consenso, embora OpenGL possa se tornar um padrão de fato.

Analogamente, existe um consenso de fato sobre o tipo de objetos de interface que devem estar presentes em interfaces gráficas do tipo *WIMP* (*windows, icons, menus and pointer device)*. Entretanto, não há um consenso sobre como estes objetos são criados ou compostos em diálogos. Além disso, a aparência destes objetos depende da plataforma final, pois é desejável que a interface tenha um *look-and-feel* "nativo". Esta é uma diferença importante na portabilidade entre interfaces e sistemas gráficos, pois a aparência de objetos gráficos só depende de atributos, que são selecionados pela aplicação através do sistema gráfico. Enquanto é possível—e mesmo desejável—que uma *scrollbar* tenha uma aparência "nativa", diferente em cada ambiente, não faz sentido falar de um polígono preenchido com *look-and-feel* "nativo": a aparência final de um polígono preenchido deve ser a mais próxima possível daquela requisitada pelos atributos.

Assim, a construção de programas interativos portáteis é uma tarefa difícil porque interfaces com o usuário não são facilmente portadas. Não é suficiente isolar detalhes em *drivers*, como pode ser feito para sistemas gráficos (veja a próxima seção).

Neste artigo, discutimos estratégias de portabilidade para aplicações gráficas interativas, tanto no aspecto de sistemas gráficos, quanto no aspecto de interfaces com usuários. Inicialmente, discutimos algumas arquiteturas para aplicações gráficas portáteis. Em seguida, discutimos os problemas de portabilidade para interfaces gráficas. Como estratégia de portabilidade, propomos uma linguagem de expressões para a especificação de *layouts* de diálogos. Esta linguagem é a base de um *toolkit* portátil, que permite *look-and-feel* "nativo". Consideramos, também, a relação entre sistemas gráficos e interfaces com usuários, e propomos um modelo de integração.

Portabilidade de sistemas gráficos

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. Algumas destas metáforas, embora originalmente intimamente ligadas aos dispositivos, se transformaram em bibliotecas de uso mais geral, como a `PLOT10` da Tektronix, e foram padrões de fato durante um certo tempo. Este tipo de arquitetura para aplicações gráficas está ilustrado na Figura 1.

Para combater a proliferação de metáforas incompatíveis entre si, foram criados grupos de trabalho no âmbito de organizações como ISO, DIN e ANSI para a elaboração de padrões internacionais, como `CORE` [ACM (1979)], `GKS` [ANSI (1985)] e `PHIGS` [ISO (1989), Gaskins (1992)]. 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 ilustrado na Figura 2.

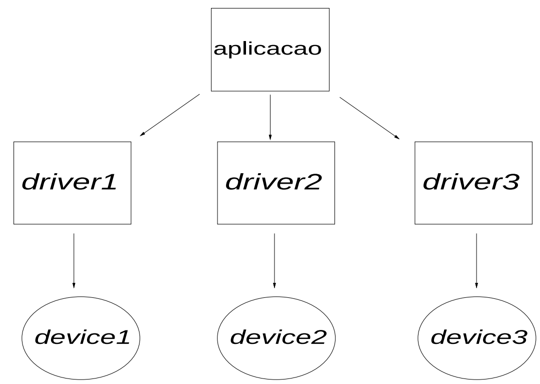

Figura 1: aplicações gráficas com *drivers*.

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 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.

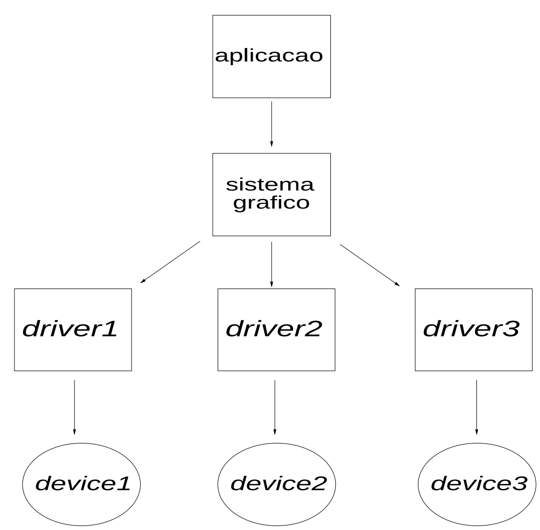

Figura 2: aplicações gráficas com sistemas gráficos.

No início da década de 80, o que mais se aproximou de um padrão 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;
- um 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, indispensáveis a um desempenho adequado.

Como resultado, surgiram o `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 modelagem obrigatória de dados: no `PHIGS`, o programador de aplicação é forçado a modelar cenas tridimensionais usando uma estrutura de dados 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–Davis–Woo (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 estruturas de dados complexas, é mostrada na Figura 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*. (Brittain (1990) descreve uma arquitetura portátil de alto desempenho.)

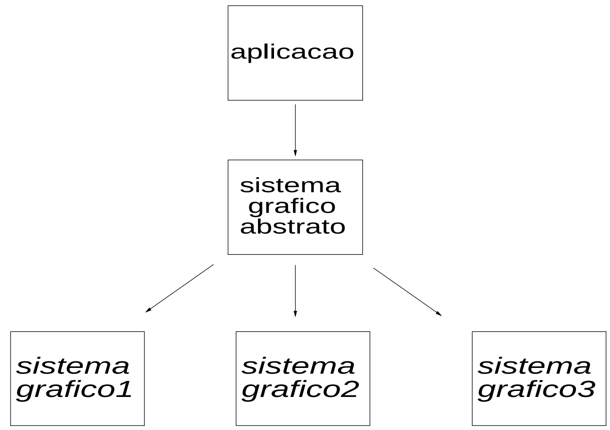

Figura 3: arquitetura com sistemas gráficos abstratos.

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*, *mouse* e *pointer 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 bi-direcional, 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 os mesmos nas várias plataformas. O problema principal é criar uma metáfora portátil, também chamada um *toolkit virtual*. Uma tal metáfora é dada pelo *toolkit* IUP (*Interface com o Usuário Portátil*) que descreveremos abaixo. 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 construção deste sistema portátil. A arquitetura de uma aplicação IUP é ilustrada na Figura 4.

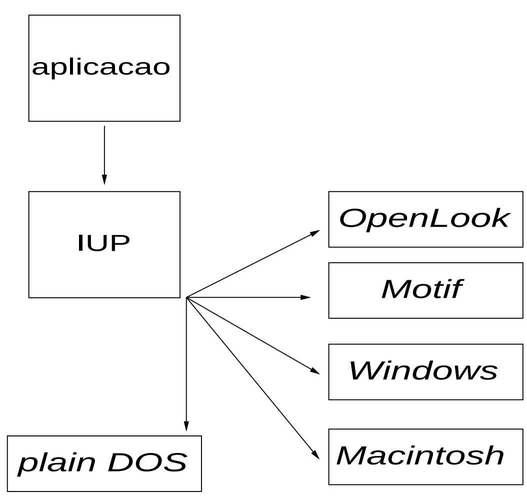

Figura 4: arquitetura de uma aplicação IUP.

Como a construção de interfaces com o usuário é um processo no qual o ciclo *implementação – teste – avaliação – correção* deve ser o mais breve possível, optamos por permitir descrever interfaces através de arquivos texto, externos às aplicações. O *toolkit* IUP faz isso colaborando com uma linguagem para especificação de diálogos, chamada LED, que também descreveremos abaixo. As especificações em LED podem ser interpretadas em tempo de execução, o que permite facilmente a criação de versões diferentes, tanto em termos de *layout*, quanto em termos de funcionalidade. Além disso, isto permite a prototipagem rápida de interfaces, isto é, a construção de interfaces sem necessariamente acoplar código da aplicação.

Existem outras ferramentas de interface com usuário similares à proposta IUP/LED. Entre elas, podemos citar *InterViews* [Linton–Vlissides–Calder (1989)], *FormsVBT* [Avrahami–Brooks–Brown (1989)], *XVT* [Rochkind (1989)], e *CIRL/PIWI* [Cowan–Durance–Guigère–Pianosi (1992)]. Destas, *InterViews* e *FormsVBT* têm *look-and-feel* fixo; *CIRL/PIWI* permite *look-and-feel* nativo e descreve diálogos textualmente, mas as descrições não são interpretadas em tempo de execução, requerendo pré-tradução para as várias plataformas; *XVT* permite *look-and-feel* nativo, mas não descreve diálogos textualmente.

Layout de 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?

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. Alguns sistemas, como o *Motif*, fornecem várias formas de composição, sem que seja necessário especificar posições exatas para cada elemento; cada uma destas formas é específica para um determinado tipo de *layout*. Outros sistemas, como o *Microsoft Windows*, exigem que os diálogos sejam compostos especificando as dimensões e as coordenadas exatas de cada elemento em um painel de controle.

A variedade de opções no *Motif* pode confundir o programador, deixando-o em dúvida sobre qual o método adequado a cada situação. No *Microsoft Windows*, como a única forma de compor um diálogo é através de posições explícitas, o programador não tem essa dúvida, mas acaba tendo que desenhar o diálogo em escala num papel milimetrado, para que ele possa conhecer as posições e tamanhos de cada elemento de interface. Como consequência, 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 re-calcular o *layout* do diálogo. Isto torna penosa a prototipagem de interfaces, aumentando o custo de desenvolvimento.

Existem ferramentas que permitem a construção de diálogos visualmente, por manipulação direta de elementos de interface (e.g, *Guide* para *Open Look* e *Visual Basic* para *MicroSoft Windows*). Estas ferramentas facilitam a especificação de um *layout* específico, mas geralmente não usam um modelo abstrato de *layout*. Como consequência, os diálogos criados visualmente são incapazes de reagir automaticamente às alterações descritas acima. (A exceção aqui é o editor de diálogos *ibuild* do *InterViews*.) Além disso, algumas destas ferramentas (como o *Guide*) geram descrições que precisam ser traduzidas, compiladas, e acopladas à aplicação *antes* da execução, o que limita a utilidade para prototipagem rápida.

Descreveremos na próxima seção uma linguagem de expressões, chamada LED, que propomos como solução para o problema de especificação de *layout* de diálogos. Baseados nesta linguagem, descrevemos IUP, um *toolkit* portátil para interface com usuário. A linguagem LED fornece uma forma de descrever diálogos textualmente; como vimos acima, isto permite uma maior agilidade na criação e teste de interfaces com o usuário. Entretanto, é importante frisar que LED é principalmente uma alternativa para a criação de diálogos, pois a interpretação de especificações em LED é feita através do *toolkit* IUP. Neste artigo, descreveremos apenas LED em detalhe, pois IUP é basicamente uma API para LED.

Uma linguagem para especificação de diálogos

A filosofia principal da linguagem para especificação de diálogos 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. Frequentemente, 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.

A linguagem LED define um modelo abstrato de interface com usuário no qual os diálogos são especificados pelo seu *layout* abstrato, e os elementos que compõem os diálogos são especificados, principalmente, pela sua funcionalidade e não pela sua aparência final. (Como vimos, se queremos permitir *look-and-feel* nativo, esta aparência só pode ser decidida em tempo de execução.) Em LED, o programador só precisa fornecer alguns poucos parâmetros que descrevem a 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*, *Open Look*, *Macintosh*. O *toolkit* IUP fornece uma API que implementa o modelo abstrato de LED (veja Figura 4).

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 seu 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 de *boxes-and-glue* também é utilizado em *InterViews* e *FormsVBT*.

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. Desta forma, na definição de um elemento de interface, o programador necessita especificar, apenas, dois ou três parâmetros relacionados com a funcionalidade. 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 (que definem 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 a 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:

$$v_1 = a_1, v_2 = a_2, \ldots,$$

onde v_i é um nome de um atributo e a_i é o seu valor (um *string*).

Como exemplo do uso de LED na especificação de um diálogo típico, considere o diálogo da Figura 5. 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)
```

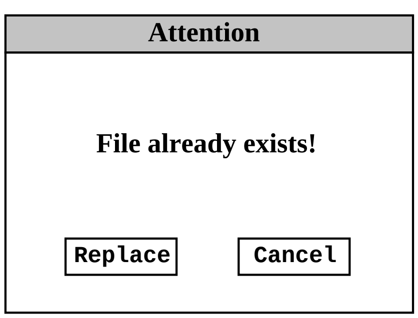

Figura 5: um diálogo simples.

Nesta especificação, usamos os seguintes elementos de interface: `dialog`, `vbox`, `hbox`, `label`, `button` e `fill`. Descreveremos estes e os outros elementos de interface a seguir. 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)))
```

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 os espaços vazios entre os elementos primitivos;
- *primitivos*: efetivamente interagem com 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 6 para o diálogo da Figura 5. 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.

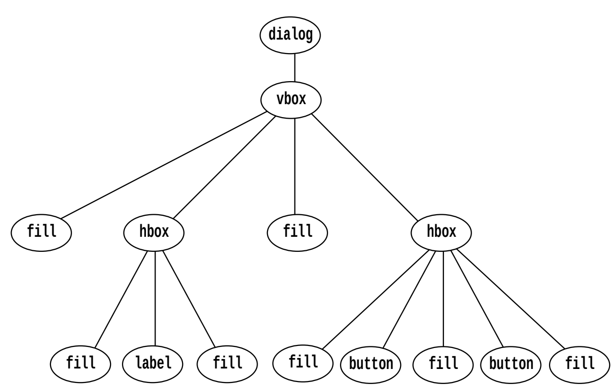

Figura 6: estrutura do diálogo da Figura 5.

Esta estrutura hierárquica facilita a programação, pois permite que diálogos sejam especificados gradativamente, combinando diálogos simples, previamente testados, na formação de um diálogo mais complexo.

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

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

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.

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.

Os elementos primitivos disponíveis em LED são uma abstração dos elementos primitivos existentes nas várias plataformas; eles representam o consenso de fato descrito na Introdução:

- `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 com barra de rolamento;
- `submenu`: submenu de `menu`;
- `text`: captura um texto de uma ou mais linhas;
- `toggle`: botão de dois estados (ligado/desligado);
- `valuator`: captura um valor numérico.

Com exceção do `canvas`, que discutiremos na próxima seção, todos os outros elementos primitivos estão bem definidos e têm o mesmo comportamento em todos os sistemas de interface existentes.

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 dos elementos de interface, 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 da variável de ambiente `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 consequência para as outras plataformas. Assim, é possível fazer *fine-tuning* para cada plataforma, usando um mesmo arquivo LED.

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 a 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*, *Open Look* via *XView*, *Motif* e *DOS* (veja Figura 4). Para o *DOS*, escrevemos um sistema de interface completo e portátil, formado por um sistema de janelas, um *window manager* e um *toolkit* de apoio. Como mencionamos anteriormente, este sistema de interface é uma aplicação puramente gráfica, à qual aplicamos a estratégia de portabilidade já descrita.

O fluxo de controle de 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;
- registrar funções correspondentes às ações usadas em LED (no exemplo, `do_replace` e `do_cancel` são ações e não funções da aplicação);
- criar os diálogos através da compilação de um arquivo LED, ou fazendo chamadas IUP para criar cada elemento de interface;
- exibir os diálogos iniciais;
- passar o controle para o IUP que então fica esperando ações do usuário e chamando a função da aplicação correspondente.

Portabilidade de aplicações gráficas interativas

O elemento `canvas` de LED 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. Dentre os sistemas considerados acima, apenas o *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).) Em LED, o comportamento do `canvas` é simples: os eventos ocorridos no `canvas` são passados para a aplicação, que deve tratá-los convenientemente.

A principal implicação desta abstração é que os sistemas gráficos podem (e devem) ser passivos. Isto evita a discussão sobre modelos de entrada de dados em sistemas gráficos, que já se mostravam obsoletos com o advento de sistemas de janelas e interfaces gráficas. A principal consequência 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.

Em resumo, o modelo de integração que propomos para aplicações gráficas interativas portáteis é a combinação de um sistema gráfico passivo, implementando um sistema gráfico abstrato que manipula as estruturas de dados da aplicação, com um *toolkit* virtual contendo `canvas`'s como receptores de eventos para o sistema gráfico. A principal vantagem deste modelo de integração é a separação entre a interface como receptor de entrada de dados e o sistema gráfico como produtor de desenhos. Isto simplifica as duas partes, tornando-as mais portáteis. É claro que este modelo só é viável se o *toolkit* virtual colaborar com o sistema gráfico para, por exemplo, permitir a conversão de coordenadas de dispositivo para coordenadas de modelo. Felizmente, esta colaboração é simples pois o sistema gráfico não precisa conhecer todos os detalhes da interface gráfica mas somente alguns detalhes do `canvas`.

Conclusão

Discutimos a portabilidade de aplicações gráficas e interfaces com o usuário, e propusemos estratégias para a construção de aplicações portáteis.

A estratégia que propusemos para portabilidade de interfaces com o usuário é a construção de um *toolkit* virtual chamado IUP. A especificação de diálogos é feita usando este *toolkit* ou uma linguagem de expressões, chamada LED, cuja sintaxe é simples. Os diálogos são especificados a partir do seu *layout* abstrato e da funcionalidade dos elementos que os compõem. A principal vantagem de especificar *layouts* abstratos é liberar o programador de cálculos geométricos tediosos, permitindo reação automática a alterações no tamanho do diálogo. Além disso, o uso de uma descrição textual para *layouts* permite prototipagem rápida de interfaces.

O uso de um *toolkit* virtual permite que as interfaces geradas tenham *look-and-feel* "nativo". Entretanto, para que este *look-and-feel* seja estritamente nativo, é necessário modificar detalhes na especificação da interface para cada plataforma. O uso de LED facilita este controle fino. Além disso, como LED é uma linguagem extensível, este controle fino pode ser minimizado usando primitivas de mais alto nível. Um exemplo típico é um objeto para seleção de arquivos, que é um objeto complexo e cuja aparência é bastante diferente em cada sistema de interface.

A estratégia que propusemos para portabilidade de aplicações gráficas interativas combina um sistema gráfico abstrato implementado sobre um sistema gráfico nativo usado passivamente, que colabora com um *toolkit* virtual na interação com o usuário por meio de *callbacks*. A tecnologia descrita aqui é a base da estratégia de portabilidade do TeCGraf/Instituto de Tecnologia de Software.

As pesquisas futuras nesta área deverão incluir o estudo de abstrações mais poderosas, como objetos gráficos ativos, que generalizam os objetos de interface tradicionais.

Agradecimentos

A motivação para este estudo teve origem no âmbito do convênio CENPES/Petrobrás. A linguagem LED e o *toolkit* IUP são produtos deste convênio, e estão descritos completamente em [Levy (1993)].

Referências

ACM, Status report of the Graphics Standards Committee, *Computer Graphics* **13**(3) (1979).

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

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

G. Blackham, Building software for portability, *Dr. Dobb's Journal* **146** (1988) 18–26.

D. L. Brittain, Portability of interactive graphics software, *IEEE Computer Graphics & Appplications* **10** (1990) 70–75.

D. D. Cowan, C. M. Durance, E. Giguère, G. M. Pianosi, *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*.

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

T. Gaskins, *PHIGS Programming Manual*, O'Reilly & Associates, 1992.

D. E. Knuth, *The TEXbook*, Addison-Wesley, 1984.

C. H. Levy, *IUP/LED: uma ferramenta portátil de interface com usuário*, dissertação de mestrado, Departamento de Informática, PUC-Rio, 1993.

M. A. Linton, J. M. Vlissides, P. R. Calder, Composing user interfaces with InterViews, *IEEE Computer* **22** (1989) 8–22.

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

ISO–International Organization for Standardization, *Information processing systems – Computer graphics – Programmer's Hierarchical Interactive Graphics System (PHIGS)*, **ISO 9592-1**, 1989.

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

F. Neelamkavil, O. Mullarney, Separating graphics from applications in the design of user interfaces, *The Computer Journal* **33** (1990) 437–443.

J. Neider, T. Davis, M. Woo (OpenGL Architecture Review Board), *OpenGL Programming Guide*, Addison-Wesley, 1993.

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