UAI — Um Framework para Suporte a Objetos Visuais

Renato Borges Carlos Cassino Renato Cerqueira Roberto Ierusalimschy
ICAD — Departamento de Informática — PUC/Rio
rborges, cassino, rcerq, roberto@icad.puc-rio.br

Resumo

UAI é um framework de classes C++ para desenvolvimento de programas gráfico-interativos com suporte a objetos visuais. Entre as principais características do sistema, pode-se citar a alta integração do sistema de interfaces com o sistema gráfico e a portabilidade. Este artigo descreve as facilidades de UAI para modelagem de objetos ativos. UAI roda atualmente em plataformas X-Windows, MS-DOS e MS-Windows.

Palavras-Chave: Interfaces Gráficas Interativas, Objetos Visuais, Visões Abstratas de Dados (ADV)

Abstract

UAI is a framework of C++ classes for developing interactive applications, with support for visual objects. The main features of UAI are the high integration between the interface and the graphic systems and its portability. This work describes the facilities of UAI for modeling active objects. The software is currenly running on X-Windows, MS-DOS, and MS-Windows.

Keywords: Graphical User Interfaces, Visual Objects, Abstract Data Views (ADV)

1 Introdução

O TeCGraf é um projeto interdisciplinar da PUC-Rio, que mantém convênios com diversos parceiros industriais na área de computação gráfica. Dentro das aplicações desenvolvidas pelo TeCGraf nos últimos anos, cada vez mais se torna necessário um suporte apropriado para interfaces gráficas interativas. Entretanto, os serviços oferecidos por *toolkits* tradicionais, como Motif e MS-Windows, se restringem a suportar um conjunto reduzido e padronizado de *widgets*, oferecendo apenas o que já foi chamado de "um pouco mais que um sistema de menus glorificado" [MR93]. Estes *toolkits* oferecem pouca ajuda quando se trata de construir sistemas de interface mais complexos, com facilidades de manipulação direta e alta integração com os recursos gráficos.

Visando estudar e validar alternativas para uma biblioteca adequada a estas necessidades, desenvolvemos o sistema UAI. UAI é um *framework* de classes C++, que pode ser dividido em:

- Um pequeno conjunto de classes básicas, que fazem a interface com o ambiente. O sistema é todo construído em cima desta base, garantindo uma boa portabilidade.
- Um conjunto de classes suportando operações gráficas diversas. Cada classe implementa uma operação específica, como *clipping*, transformações geométricas (rotação, escala etc), padrões etc. Esta parte não será tratada aqui; para maiores informações veja [CCCI94].
- Um conjunto de classes para suporte a "objetos visuais", isto é, objetos que têm uma representação visual. Estas classes facilitam a implementação de operações como manipulação direta destes objetos, agrupamento, *zoom* etc. O mecanismo de objetos visuais é uma implementação do conceito de *Abstract Data Views*, apresentado em [CILS93].

Na próxima seção, discutimos o modelo de orientação a eventos adotado em UAI, bem como o conceito de objetos visuais. A seção 3 descreve a arquitetura geral do *framework*. Em seguida, mostramos uma aplicação especial desenvolvida com este *framework*, que permite a especificação e criação de objetos visuais através da linguagem de configuração Lua [FIF94]. Finalmente, a última seção é dedicada a algumas conclusões.

2 O Modelo de UAI

UAI adota o modelo de orientação a eventos, difundido a partir de Smalltalk [GR83]. Segundo este modelo, as ações executadas por um programa são ativadas a partir de eventos gerados pelo usuário. Com a popularização dos sistemas de interface gráfica com o usuário, este modelo tende a se tornar padrão para o desenvolvimento de aplicações interativas.

No modelo de eventos a aplicação tem definidas as ações para os eventos que o usuário pode gerar. Como consequência deste paradigma, tem-se que o controle de fluxo sai da aplicação e passa para o usuário.

As linguagens tradicionais, como C ou Pascal, não oferecem mecanismos de suporte para este novo enfoque. Desta forma, torna-se necessária uma simulação desta inversão de controle. Isto geralmente é implementado através do uso de um loop principal que lê os eventos e os processa. Neste caso, as próprias aplicações implementam este loop ou passam explicitamente o controle a uma rotina do sistema que o faz.

A solução adotada em UAI segue um modelo mais puro de orientação a eventos onde, do ponto de vista da aplicação, inexistem loop e programa principal. Isto traz como consequências para a aplicação a perda do conceito de "modo" — uma vez que ela não tem controle sobre qual ação será executada a cada momento — e a necessidade do uso de temporizadores. Este mecanismo possibilita que um determinado evento seja gerado em intervalos regulares de tempo, permitindo que uma aplicação possa realizar tarefas assincronamente dos eventos de interface.

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

De um modo geral, uma vez inicializada uma aplicação UAI, esta passa a criar objetos e inserí-los na interface. A partir daí, estes objetos são capazes de reagir a eventos. Introduz-se assim, os conceitos de *Visual Object* (VO) e *Visual Space* (VS). O conceito de VO, inspirado no de *Abstract Data Views* [CILS93], modela objetos gráficos ativos, oferecendo métodos de desenho e de resposta a ações do usuário, o que dá forte suporte à manipulação direta destes objetos. Por outro lado, o conceito de VS modela o espaço onde VOs são posicionados.

Estendendo esses conceitos que dão apoio a manipulação direta de objetos, pode-se também construir elementos de interface, desde botões até elementos mais sofisticados como "painéis de controle". Por construção, estes elementos de interface têm, além de um alto grau de versatilidade, um elevado índice de integração com o sistema gráfico.

Para oferecer um elevado grau de conectividade aos VOs, UAI define um padrão de interface para estes, seguindo a linha dos *custom controls* de Visual Basic (VBX) e do *application framework* do NextStep. Essa característica é apontada como uma das principais virtudes desses dois sistemas [Ude94]. Em C++, este padrão de interface se traduz em classes virtuais puras, que contêm apenas funções virtuais puras (isto é, sem implementação). Várias classes de UAI são definidas desta forma, sendo herdadas não pelo seu código, inexistente, mas para tornar a nova classe compatível com determinadas interfaces. Vale observar que, por causa deste uso de herança, a hierarquia de classes de UAI inclui vários casos de herança múltipla. Conforme observado em [Wal93], este uso de herança múltipla não traz embutido algumas das desvantagens que herança múltipla apresenta quando usada para misturar código de diferentes classes.

Além disso, UAI também se vale do mecanismo de herança de C++ para que o desenvolvedor possa "modificar um objeto que faz 90% do que você quer, adicionando apenas os últimos 10%". Esta característica também está presente no *framework* do NextStep, já que este tem objective-C como linguagem básica de desenvolvimento. Sistemas não apoiados em linguagens orientadas a objetos, como o Visual Basic, não podem se valer deste recurso.

Com essas características, UAI oferece um modelo com bom suporte para o desenvolvimento de objetos reutilizáveis, na linha dos *component-softwares*. Entretanto, o sistema não dispõe, até o momento, de recursos de ligação dinâmica, que também é uma das características fundamentais para o sucesso de uma política de reutilização de componentes de software.

Nas próximas seções poderá ser observada a preocupação constante no projeto da UAI em desenvolver um *framework* que dê um bom suporte para definir objetos facilmente reutilizáveis, principalmente objetos ativos. Como já foi dito, são previlegiadas as aplicações gráfico-interativas com necessidades de recursos para manipulação direta de objetos.

3 A Arquitetura de UAI

Para atender seus objetivos, UAI define um *framework* que se concentra em dar suporte a operações gráficas diversas e a objetos visuais. Para garantir a portabilidade do sistema, também é definido um pequeno conjunto de classes básicas que faz a interface com a plataforma hospedeira, encapsulando as peculiaridades destas. Desta forma, basta portar este conjunto de classes básicas para uma determinada plataforma para que o restante do *framework* seja diretamente portado.

Para se definir o conjunto de classes básicas, procurou-se "um nível adequado de abstração". Nesta linha, foram definidas três classes básicas, dependentes da plataforma:

- A classe *YApplication*, que é responsável por manter o protocolo necessário para registrar uma aplicação no sistema hospedeiro. Por exemplo, no Xlib este objeto se encarrega de criar a aplicação junto ao Xprotocol definindo seu *display*, *screen* etc. Esta classe, ou qualquer derivada, só pode ser instanciada uma única vez.

- A classe *YWindow*, que modela a área sobre a qual agem as primitivas gráficas. Uma *YWindow* é responsável pelo mapeamento entre uma janela do sistema hospedeiro e a abstração de área de trabalho da biblioteca UAI. Isto tem como finalidade permitir que a aplicação não seja obrigada a trabalhar diretamente com pixels. Nesse sentido, esta classe fornece uma transformação de coordenadas básica de pixel para milímetro e vice-versa. Além disso, objetos do tipo *YWindow*, ou derivados, têm métodos para o tratamento dos eventos gerados pelo sistema. Desta forma, para criar uma janela, uma aplicação deve implementar uma classe própria para o seu tipo desejado herdando de *YWindow* e definindo o tratamento para os eventos enviados à janela.

- A classe *YWindowPen*, derivada da classe abstrata *YRasterPen*. Ela é responsável pela implementação real dos métodos de desenho nas janelas.

Uma classe importante derivada de *YWindow*, mas totalmente independente de plataforma, é a *YCanvas*. Ela modela o conceito de um *canvas*, onde a aplicação pode trabalhar com o sistema de coordenadas que mais lhe convenha.

O diagrama da hierarquia estática das classes de UAI pode ser visto na figura 1. No restante desta seção, é dada uma maior ênfase à arquitetura das classes que dão suporte ao modelo de objetos visuais. Conforme comentado anteriormente, este artigo não discutirá as classes relativas ao suporte gráfico da biblioteca (*YPen* e suas subclasses).

No que diz respeito aos serviços de sistema de interface, UAI não dispõe de elementos primitivos como botões, menus etc. Ao invés disto, oferece meios para agrupar primitivas gráficas e associá-las a determinados estímulos. Neste contexto, são introduzidos os conceitos de *Visual Object* e *Visual Space*.

Como já foi descrito, VOs modelam objetos gráficos ativos para dar suporte à manipulação direta. Para isto, eles oferecem métodos para auto-desenho e de resposta às ações do usuário. Por outro lado, um VS modela o espaço onde VOs são posicionados.

Para facilitar a especialização e a composição de objetos visuais, foram definidas as seguintes classes para a implementação de VOs e VSs:

- *YBasicVS* — classe abstrata que define a interface básica de um VS, ou seja, define quais são os serviços elementares que um VS deve prestar. Entre os serviços prestados temos: o fornecimento de *pens* que permitem operações gráficas sobre o espaço representado pelo VS; operações para remover, inserir ou trocar um VO associado a ele; redesenho, total ou parcial, de seu domínio.

- *YVS* — classe abstrata derivada de *YBasicVS* que, ao contrário desta que não se compromete com o número de VOs associados, assume a existência de um único VO.

- *YCanvasVS* — classe derivada de *YCanvas* e *YVS*, que permite que um *canvas* possa ser visto como um VS (podendo ter um único VO associado). Além disto, esta classe

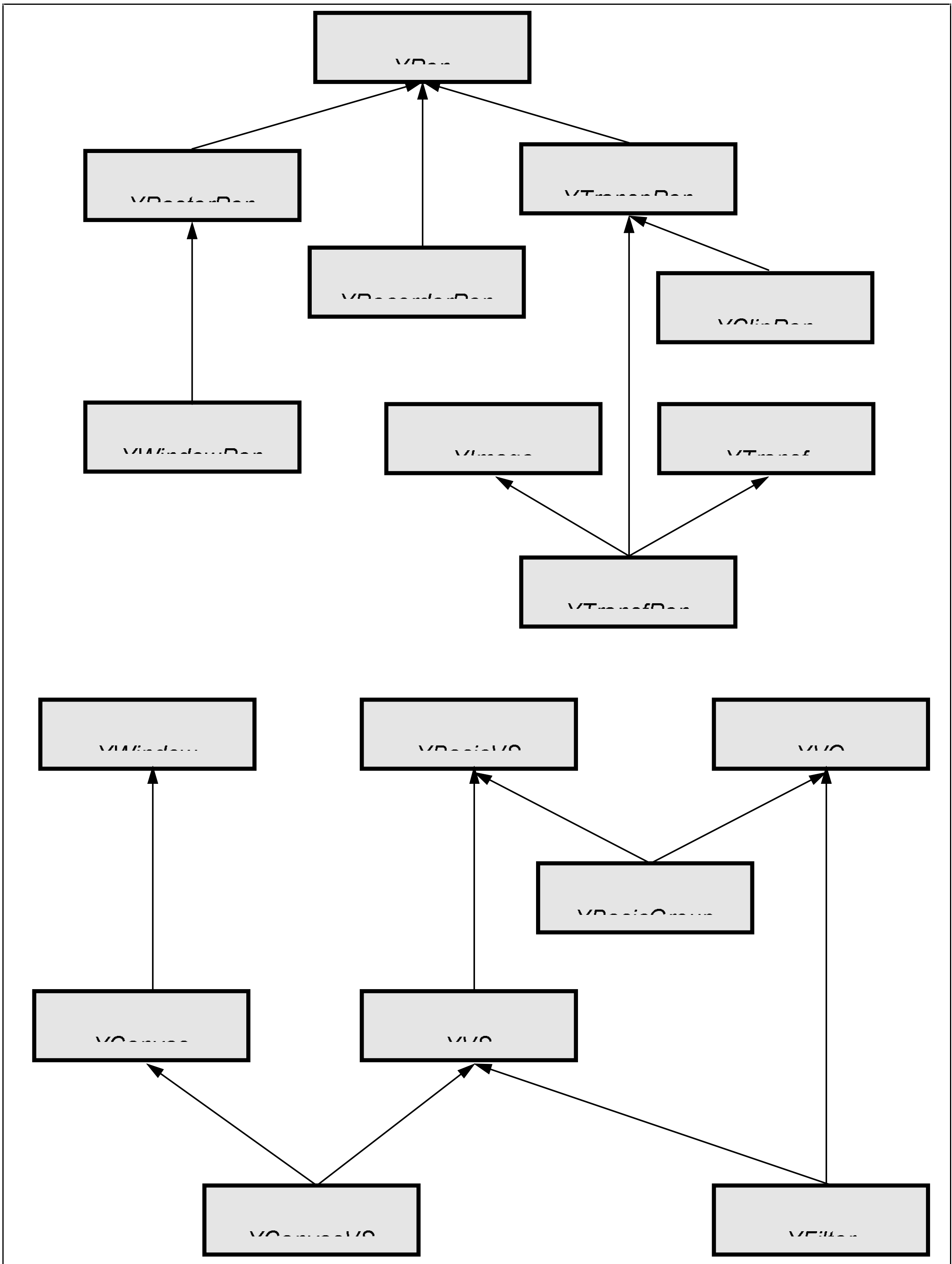

Figure 1: Hieraquia estática de classes

redefine os métodos de tratamento de eventos de *YCanvas* para repassá-los ao seu VO para que este os trate (primeiro passo para a criação de objetos ativos).

- *YVO* — classe abstrata que define a interface básica de um VO. Esta interface define métodos para tratamento de eventos, auto-desenho (inclusive modo "rascunho"), manipulação direta (*pick*, *move* e *track*) e fornecimento da área ocupada pela representação gráfica do VO em seu VS.

Outros conceitos úteis podem ser derivados da composição de VO e VS, tais como grupos e filtros. O conceito de filtro serve para modelar objetos que fazem interface entre um VS e um VO. Desta forma ele repassa para o seu VO associado os eventos gerados em seu VS, podendo dar algum tratamento especial a esses eventos. A comunicação inversa também é válida, isto é, as requisições feitas por um VO a seu VS podem também ser filtradas por esse tipo de objeto, ou passadas adiante. A classe que modela esse conceito é a *YFilter*, derivada de *YVS* e *YVO*. Para facilitar sua especialização essa classe não é abstrata e define um filtro "transparente". Este filtro simplesmente repassa as requisições feitas a ele.

Uma especialização de *YFilter* já incorporada ao *framework* é a classe *YDrag*, que fornece um mecanismo para arrastar VOs em seu espaço de visualização. Estes objetos, quando ativados, interceptam os eventos de movimentação do cursor e geram eventos de redesenho adequados à visualização da operação de arraste.

Já o conceito de grupo modela um tipo de objeto que contém uma coleção de VOs e está associado a um VS. Este VS o trata como se ele fosse um único VO, sinalizando os eventos do usuário. A função básica do grupo é repassar estes eventos para um seus VOs, selecionando qual o VO adequado para receber a mensagem. Da mesma forma que um filtro, um grupo é visto por seus VOs associados como se fosse um VS, atendendo todas as suas requisições. A classe que implementa este conceito na biblioteca é a *YBasicGroup*.

Devido a natureza híbrida das classes *YFilter* e *YBasicGroup* (e suas especializações), suas instâncias podem ser encadeadas entre VOs e VSs de forma transparente. Esse encadeamento é ilustrado na figura 2.

4 Exemplo de uma Aplicação

Nesta seção descrevemos uma aplicação particular do sistema UAI. O propósito desta descrição é duplo: por um lado, ilustra como a biblioteca pode ser usada para criação de programas específicos. Por outro lado, a idéia desenvolvida neste exemplo pode servir de base para outros tipos de aplicação, onde as propriedades da interface não estão amarradas dentro do programa, mas são descritas por uma linguagem de configuração auxiliar.

A idéia desta aplicação é fornecer ao usuário meios de especificar as características de objetos visuais via arquivos de configuração. A linguagem utilizada nestes arquivos foi a linguagem de configuração Lua [FIF94].

Lua é uma linguagem procedural com facilidades para descrição de objetos estruturados. Por ser uma linguagem embutida, não tem a noção de programa principal. Além disto, Lua provê um mecanismo de construtores, permitindo que funções sejam chamadas quando da criação de uma estrutura. Tais características a tornam perfeitamente compatível com a filosofia utilizada em UAI: aplicações não modais, fortemente baseadas em construtores para inicialização e composição de objetos.

Chegamos então a um ponto onde pode ser visto como as aplicações podem deixar grande parte das características de suas interfaces para serem especificadas via arquivos de configuração. Nestes arquivos, temos código Lua que, lido durante a execução do programa cria

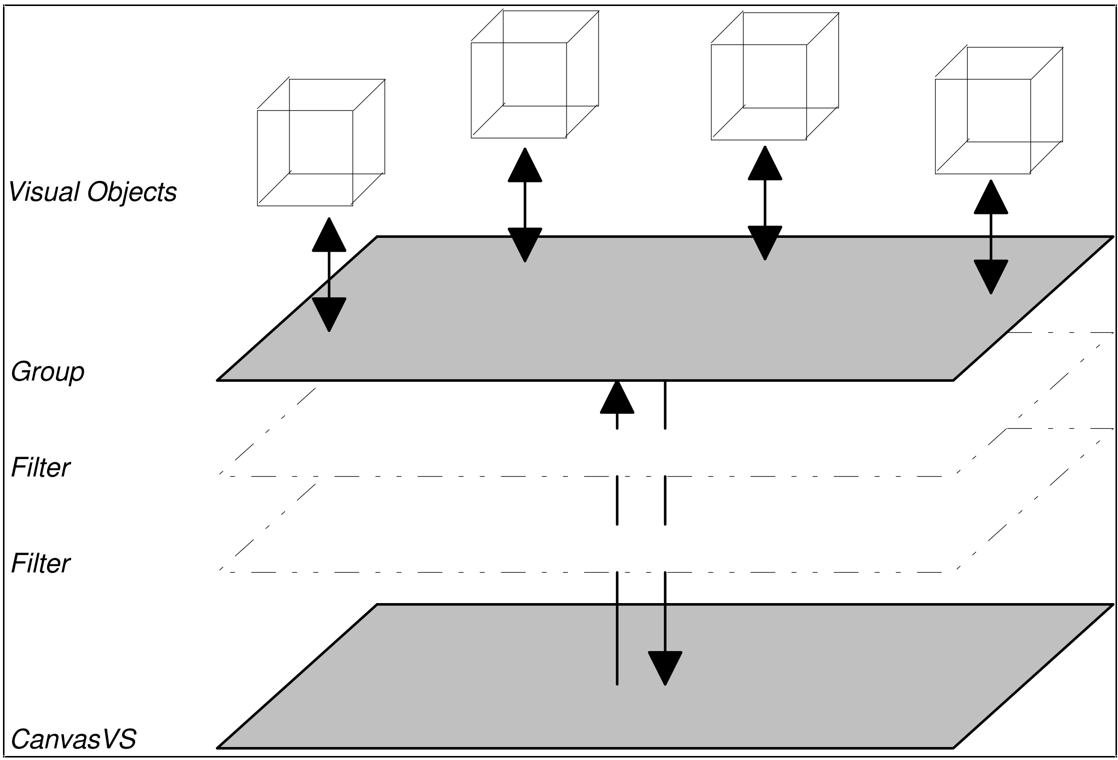

Figure 2: Esquema de troca de mensagens entre Visual Objects e Visual Spaces

objetos visuais completos, definindo formas e comportamentos. Tais objetos compõem a interface, determinando suas características. Visto que os comportamentos destes objetos podem ser ações pré-definidas na aplicação ou novas ações criadas via funções Lua, a interface ganha uma grande maleabilidade.

A implementação deste exemplo, em UAI, é razoavelmente simples. O passo principal é a definição de uma nova classe de VOs, denominada *YLuaVO*, que ao invés de ter um código fixo, em C++, descrevendo seu comportamento diante de eventos, busca a descrição deste comportamento em um arquivo de configuração. Com isto, objetos visuais com diferentes comportamentos podem ser criados e modificados sem necessidade de modificações ou recompilações do programa fonte C++.

No exemplo a seguir, o arquivo de configuração cria dois objetos visuais. No momento desta criação, o construtor dos VOs é chamado, recebendo como parâmetro o objeto em questão. Este construtor se encarrega de registrar o novo objeto, colhendo de sua estrutura dados básicos. Neste ponto, o construtor associa um filtro de *dragging* ao VO e o insere no VS corrente. A partir daí o VO está apto a receber seus eventos. Desta forma, todos os VOs criados em Lua podem sofrer operações de *dragging* e são visíveis a partir do momento de sua criação.

Aos VOs especificados em Lua estão disponíveis a funcionalidade básica de UAI para desenhos (linhas, polígonos, cor, etc), assim como todas as demais funções que a aplicação desejar cadastrar. No exemplo, para cada VO é especificado um método para desenho, um para "rascunho" — a ser usado durante as operações de *dragging* — sua cor e suas dimensões. Estes dois VOs são definidos usando a sintaxe de construtor de objetos de Lua:

```
@VO {
   left = -1.5,
```

```
    bottom = -1.5,
    right = -0.5,
    top = -0.5,
    draw = dashes,
    draft = outline,
    color = @{red = 0, green = 1, blue = 0}
}

@VO {
    left = -1.5,
    bottom = 0.5,
    right = -0.5,
    top = 1.5,
    draw = circle,
    draft = outline,
    color = @{red = 0, green = 0, blue = 1}
}
```

As duas definições acima funcionam da mesma forma. Em ambos os casos, um objeto Lua é criado contendo os campos *left*, *bottom*, *right*, *top*, *draw*, *draft* e *color*. Estes campos são inicializados com os valores especificados e o construtor *VO* é então chamado, recebendo como parâmetro o objeto com todos os seus campos já preenchidos.

Todos os campos especificados nos objetos são campos básicos previstos para um VO que o construtor, ou outras funções, podem consultar. Os campos *left*, *bottom*, *right* e *top* determinam a área ocupada pelo objeto em seu VS correspondente. *Draw* e *draft* assinalam os métodos a serem chamados quando o objeto precisar, respectivamente, desenhar-se por completo e desenhar um rascunho de sua forma. O campo *color* é uma estrutura com as componentes *rgb* de sua cor.

Neste exemplo particular, os métodos de desenho foram definidos como:

```
function dashes(obj)
    YSetColor(obj)
    local i = obj.left
    local delta = (obj.right-obj.left)/10;
    while i<obj.right do
        YLineXY(obj.left, obj.top, i, obj.bottom)
        i = i+delta
    end
    YLineXY(obj.left, obj.top, obj.right, obj.bottom)
    i = obj.top
    while i>obj.bottom do
        YLineXY(obj.left, obj.top, obj.right, i)
        i = i-delta
    end
end

function circle(obj)
    YSetColor(obj)
    local pi_2 = 2*3.1415
    local r = 0.5
```

```
    local cx = (obj.left+obj.right)/2
    local cy = (obj.bottom+obj.top)/2
    local num_pts = 20
    local t = 0
    YBeginPolyline()
    while t < pi_2 do
       YAddPointXY(cx+r*cos(t), cy+r*sin(t))
       t = t+pi_2/num_pts
    end
    YAddPointXY(cx+r*cos(0), cy+r*sin(0))
    YEnd()
 end

 function outline(obj)
    YSetColor(obj)
    YLineXY(obj.left, obj.bottom, obj.left, obj.top)
    YLineXY(obj.left, obj.top, obj.right, obj.top)
    YLineXY(obj.right, obj.top, obj.right, obj.bottom)
    YLineXY(obj.right, obj.bottom, obj.left, obj.bottom)
 end
```

Todos métodos recebem os objetos a serem desenhados e dispõem das primitivas básicas para atuarem. O método *circle*, por exemplo, recebe o objeto e seleciona a cor corrente de desenho apenas passando este objeto para a função *YSetColor*. Isto é conseguido graças à característica de Lua de referenciar os campos de uma estrutura — seja que estrutura for — diretamente pelo seu nome. Desta forma, as estruturas Lua podem ser trocadas indistintamente pois passam a ser compatíveis entre si de forma automática, pela simples definição de campos comuns. Com esta característica, a passagem do objeto entre as rotinas se assemelha à passagem implícita do *this* entre os métodos de um objeto de uma linguagem OO. Este uso de compatibilidade estrutural entre os diferentes objetos é empregado nos demais métodos.

5 Conclusões

UAI conta atualmente com dezenas de classes para suporte modular a diversas tarefas típicas em sistemas interativos, com ênfase em manipulação direta e objetos gráficos ativos.

Conforme mencionado na introdução, o objetivo principal do desenvolvimento de UAI tem sido estudar alternativas para a arquitetura de bibliotecas e *frameworks* de suporte gráfico-interativo para o projeto TeCGraf. Dentro deste contexto, várias soluções desenvolvidas em UAI têm sido adotadas em produtos desenvolvidos pelo projeto, como por exemplo o conceito de manipuladores, incorporado no projeto PETROX, um editor gráfico estruturado, desenvolvido para o CENPES.

Um produto colateral de UAI é a experiência com C++. O projeto TeCGraf usa C como principal linguagem de desenvolvimento. Existe atualmente uma tendência no projeto para se migrar para C++, principalmente por forças externas: vários clientes começam a solicitar projetos nesta linguagem, e cada vez mais bibliotecas e ferramentas migram para C++. Neste contexto, a experiência com o uso de C++ no desenvolvimento de UAI tem sido de grande importância. Por sua arquitetura orientada a eventos, UAI usa fortemente os recursos de orientação a objetos de C++, como herança múltipla, classes abstratas, etc. Além disto, visando manter alta portabilidade, UAI tem sido compilado nos compiladores gcc 2.5.8, CC-Sun

2.1, Watcom C++ 9.5 e Borland C++ 3.1. Nestas situações percebe-se a falta de maturidade de todos estes compiladores, bem como a falta de padronização da linguagem.

O próximo passo a ser alcançado é viabilizar a especificação de objetos visuais em tempo real. Para tal, o interpretador Lua, já existente, funcionaria como um *shell*, onde o usuário poderia criar os objetos e especificar seu comportamento. Os objetos criados apareceriam imediatamente na interface, podendo ser alterados durante sua existência. Poderiam, também, ter suas ações testadas e modificadas ainda durante a execução.

Uma ferramenta com estas características pode ter grande importância para um projeto como o TeCGraf. Seus projetos implementam interfaces gráficas altamente interativas e que, cada vez mais, exigem facilidades de configuração. Assim, tal ferramenta pode servir a dois propósitos: minimizar os esforços na criação de protótipos dos sistemas a serem implementados e facilitar o desenvolvimento de aplicações. Estas aplicações, além de poderem ser configuradas estaticamente via arquivos de configuração, podem também ter suas características adaptadas em tempo de execução.

Agradecimentos

Os autores gostariam de agradecer a André Costa e André Clinio, que desenvolveram a biblioteca gráfica do sistema UAI, a mais um André, Carregal, que portou UAI para MS-Windows, e a Luiz H. Figueiredo, por numerosas sugestões. Este trabalho foi realizado com incentivo do projeto TeCGraf.

Referências Bibliográficas

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

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

[FIF94] L. H. Figueiredo, R. Ierusalimschy, e W. Celes Filho. The design and implementation of a language for extending applications. Em *XXI Semish*, 1994. (a ser publicado).

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

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

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

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