RITA - Revista de Informática Teórica e Aplicada, vol. 7, n. 2, p. 7-41,
Dezembro 2000. Instituto de Informática, UFRS, Brasil

Ferramentas de Programação Livres para Computação Gráfica e Animação por Computador

Alberto B. Raposo*, Adailton J. A. da Cruz*,†, Alessandro L. Bicho*,

Alfredo K. Kojima‡, Carlos A. M. dos Santos§, Isla C. F. da Silva*,

Léo P. Magalhães*, Paulo C. P. de Andrade‡

Resumo

Este artigo aborda uma série de tecnologias de domínio público nas áreas de Computação Gráfica, Realidade Virtual e Animação por Computador, enfocando ferramentas de programação para apoio ao desenvolvimento de sistemas gráficos interativos. A primeira parte apresenta duas tecnologias de menor nível de abstração para a geração de interfaces gráficas, formas geométricas e imagens: X Window System e OpenGL. A segunda parte apresenta tecnologias de mais alto nível, voltadas para a Internet, que podem ser usadas para a criação de aplicações sofisticadas, tais como mundos virtuais e animações interativas. Dentre estas tecnologias destacam-se Java 3D e VRML. Java 3D é a biblioteca padrão da linguagem Java para a criação de programas com gráficos tridimensionais. VRML é um padrão muito usado para a modelagem e transmissão de informação 3D e mundos virtuais pela Web.

Palavras-chave: Computação Gráfica, Software Livre, Animação, Realidade Virtual.

*Departamento de Engenharia de Computação e Automação Industrial - DCA
Faculdade de Engenharia Elétrica e de Computação – FEEC
Universidade Estadual de Campinas – UNICAMP
e-mails: {alberto, ajcruz, bicho, isla, leopini}@dca.fee.unicamp.br
†Centro Universitário de Dourados
Universidade Federal do Mato Grosso do Sul – UFMS
‡Conectiva Informática S.A.
e-mails: {kojima, pcpa}@conectiva.com.br
§Centro de Pesquisas Meteorológicas, Universidade Federal de Pelotas - UFPel
Instituto de Informática, Universidade Federal do Rio Grande do Sul – UFRGS
e-mail: casantos@cpmet.ufpel.tche.br

Abstract

This paper discusses some publicly-available technologies in the fields of Computer Graphics, Computer Animation and Virtual Reality, focusing on programming tools to support the development of interactive graphic systems. The first part presents two technologies of lower abstraction level, used for the generation of graphic interfaces, geometric drawings and images: the X Window System and OpenGL. The second part presents two higher abstraction level technologies: Java 3D, the Java standard library for 3D graphics programming, and VRML, a language for virtual reality modelling. Both Java 3D and VRML can be used to create sophisticated WWW-oriented applications, such as virtual worlds and interactive animations.

Keywords: Computer Graphics, Free Software, Computer Animation, Virtual Reality.

1 Introdução

Software livre pode ser definido como software desenvolvido por um grupo amplo e heterogêneo de pessoas, empresas e grupos acadêmicos, cuja diversidade de interesses impede que se estabeleça o monopólio de um único fornecedor ou fabricante, como normalmente ocorre com software não-livre.

Recentemente, surgiu um forte movimento por parte de empresas que desenvolvem software comercial no sentido de abrir o código de seus produtos, visando aumentar sua aceitação no mercado, facilitar sua distribuição e fazer uso dos serviços da comunidade para testar o produto e torná-lo compatível com um maior número de plataformas. Este tipo de software é chamado de *open source* (código aberto).

Portanto, o que se pode chamar de genericamente de "software de uso público" ou "disponível publicamente", está dividido em três classes: software livre (não-proprietário, código disponível e gratuito), software de código aberto (proprietário, código disponível e gratuito) e software gratuito (apenas o executável está disponível gratuitamente).

O crescimento de software livre e de código aberto em Computação Gráfica também é uma realidade, motivado principalmente pela preocupação com a utilização de padrões gráficos para trabalhar de forma eficiente em diversas plataformas. Este artigo aborda algumas ferramentas de programação livres (ou de código aberto) para o desenvolvimento de sistemas gráficos interativos.

Inicialmente, será estudado o X Window System, um sistema de janelas com suporte à construção de aplicações gráficas distribuídas e arquitetura cilente-servidor. Depois será apresentado o OpenGL, uma API (*Application Programmer's Interface*) aberta para o desenvolvimento de aplicações gráficas tridimensionais que pode ser

incorporada a qualquer sistema de janelas, inclusive o X. Java 3D, abordada a seguir, é a biblioteca da linguagem Java para a criação de aplicações tridimensionais. Como toda a plataforma Java, Java 3D é, segundo o conceito apresentado anteriormente, *open source*, embora a Sun, proprietária do software, imponha restrições adicionais à distribuição do código-fonte. Finalmente, será apresentada a VRML, uma linguagem para a modelagem e transmissão de informação 3D pela Internet.

É importante ressaltar que estas ferramentas se destinam ao desenvolvimento de diferentes classes de aplicações gráficas. O X Window System e suas interfaces de programação (*toolikts*) são usados para o desenvolvimento de componentes gráficos de interfaces interativas (janelas, botões, etc.) e primitivas básicas de desenho. OpenGL é uma biblioteca gráfica para a renderização de imagens estáticas. Java 3D e VRML, por sua vez, são voltados à Web e possuem recursos sofisticados de animação, interação e navegação por mundos tridimensionais.

2 X Window System

O X Window System foi criado em 1984, no Massachusetts Institute of Technology (MIT) [18]. Devido à sua aceitação no mercado, em 1988 o MIT formou junto com fabricantes de software o MIT X Consortium, destinado a prover suporte técnico e administrativo ao desenvolvimento do sistema. O consórcio tornou-se uma entidade independente em 1993 e, após várias fusões e reorganizações, deu origem a uma organização chamada The Open Group, que hoje detém os direitos sobre o X.

O papel do X Consortium sempre foi o de produzir padrões, ou seja, especificações de software, junto com as quais distribuía o código-fonte de uma implementação de amostra (SI — *sample implementation*). Pode-se criar versões modificadas, a partir da SI (a maioria dos fabricantes de sistemas UNIX comerciais faz isso), mas o X Window System continua sendo um só, definido por suas especificações formais. Qualquer implementação que não siga à risca as especificações deve ser considerada, no mínimo, defeituosa.

2.1 Arquitetura do X Window System

X foi concebido para que uma aplicação pudesse funcionar com quaisquer tipos de monitores e dispositivos de entrada. Esses dispositivos são tratados, em conjunto, como um *display* (ver Seção 2.3). As aplicações deveriam poder usar qualquer *display*, colorido ou monocromático, remoto ou local, sem que fosse necessário alterar o seu código ou recompilá-las, e sem perda sensível de desempenho. Mais de uma aplicação deveria poder acessar o *display* simultaneamente, cada uma apresentando informações em uma janela.

O sistema possui arquitetura cliente-servidor. O servidor localiza-se na estação de trabalho do usuário e provê acesso transparente ao hardware gráfico, mecanismos de

comunicação entre clientes e entrada de dados por dispositivos como mouse e teclado. Os clientes podem ser executados na própria estação ou remotamente, conectados ao servidor através de uma rede.

Qualquer coisa que se queira apresentar na tela deve ser desenhada dentro de uma janela. Uma aplicação pode usar muitas janelas de cada vez. Menus e outros elementos da interface podem ser construídos com janelas que se sobrepõem às demais. As janelas são organizadas em uma hierarquia, o que facilita o gerenciamento. Qualquer janela, exceto uma, chamada de raiz, é subjanela de outra.

Ao invés de um modelo de objetos de alto nível, X oferece um substrato de operações básicas, deixando a abstração para camadas superiores. Um mecanismo de extensão permite às comunidades de usuários estender o sistema e mesclar essas extensões harmoniosamente.

Suporte a gráficos 3D não era um dos requisitos originais do sistema. Posteriormente o X Consortium padronizou uma extensão baseada em PHIGS (*Programmer's Hierarchical Interactive Graphics System*), chamada *PHIGS extension for X* (PEX), que não teve sucesso. A extensão GLX, desenvolvida pela Silicon Graphics (SGI) para dar suporte ao padrão OpenGL (Seção 3), acabou se tornando a mais aceita.

2.2 Interfaces que constituem o sistema de janelas

Um sistema de janelas é constituído por várias interfaces:

Interface de programação. Biblioteca de rotinas e tipos para interação com o sistema de janelas. Há dois tipos de interface de programação. A **de baixo nível** provê funções gráficas primitivas. No X, a biblioteca Xlib [2] provê essas funções por meio de troca de mensagens com o servidor. A **de alto nível** provê funções para tratar os elementos constituintes da interface. Existem diversas bibliotecas (*toolkits*) de interface. X não define uma interface de alto nível padrão, mas provê os mecanismos necessários à construção de uma por meio de uma biblioteca chamada X Toolkit (Xt) [11].

Interface de aplicação. Define a interação mecânica entre a aplicação e o usuário, que é específica da aplicação. X também não define regras para interação usuário/aplicação.

Interface de gerenciamento. Define a política de controle do posicionamento, tamanho e sobreposição das janelas sobre a área de trabalho. Este papel cabe a uma aplicação chamada gerenciador de janelas (*window manager*). X não provê um gerenciador padrão, mas define um conjunto de atributos para cada janela chamado *window manager hints* (dicas). Gerenciadores de janelas devem fazer uso dessas dicas para determinar as alterações que o usuário pode fazer na geometria das janelas.

Essa classificação define uma arquitetura em camadas e sugere independência entre **mecanismo**, provido pelas camadas abaixo da interface de programação de baixo nível, inclusive, e **política**, provida pelas camadas acima desta.

A **interface com o usuário** é a soma das interfaces de aplicação e de gerenciamento [18]. Baseado no princípio de que o conceito de melhor interface é cultural, X foi projetado de forma a prover mecanismos para que diferentes políticas de interface pudessem ser implementadas, propiciando que as aplicações convivam com diversas culturas. A função de prover tais interfaces é delegada às aplicações clientes e bibliotecas adicionais (Seção 2.9).

2.3 Conceitos importantes e terminologia

No X Window System, ***display*** é uma instância de um servidor X rodando em uma máquina, ou seja, o conjunto de hardware de vídeo, teclado e mouse mais o software que gerencia esse hardware. Uma máquina dedicada que serve apenas como *display*, é chamada de terminal X. Cada monitor ligado à máquina é denominado **tela** (*screen*). Um *display* sempre contém pelo menos uma tela, mas múltiplos monitores ligados à mesma máquina podem conviver formando um *display multiscreen*.

Janela (*window*) pode ter dois sentidos, dependendo do contexto. No nível mais baixo, é uma área, normalmente retangular, em uma das telas do *display*, na qual os clientes podem realizar operações de entrada e saída (neste texto, o termo "janela" será usado com este significado). Em um nível mais alto, janela é a área ocupada na tela por uma aplicação, também chamada de janela de aplicação, ou *shell*. **Área de trabalho** é o conjunto formado pela janela-raiz de uma tela e as janelas de aplicação criadas sobre ela.

Servidor é o programa que compartilha um *display* com os clientes. O servidor é construído em duas camadas, uma dependente do hardware e outra independente. Portar o servidor para outro hardware implica em modificar apenas a primeira. **Cliente** é uma aplicação que usa os serviços de um ou mais servidores de *display*, podendo ter mais de uma conexão aberta simultaneamente com o mesmo servidor.

Sessão (*session*) é o período de tempo durante o qual um usuário tem acesso ao *display*. Normalmente apenas um usuário pode usar o *display* de cada vez, mas ele pode dar autorização a outros para que o usem.

2.4 Protocolo X

A comunicação cliente-servidor se dá por troca de mensagens, usando o *protocolo X* [19], que é independente do protocolo de transporte usado (TCP/IP, DECnet, etc.). O protocolo provê representação de dados independente da organização dos computadores em que são executados o cliente e o servidor. Isto é negociado no estabelecimento da conexão e o servidor trata de fazer no seu lado as conversões necessárias.

As mensagens podem ser de três tipos: *Request*, para solicitação de serviços do cliente ao servidor, *Reply*, para respostas a requisições; ou *Event*, para eventos, tais como alteração da geometria de uma janela, pressionamento de uma tecla ou movimentação do mouse. *Requests* podem ser de dois tipos, de acordo com o modo como o servidor notifica o atendimento: unidirecionais (*one way trip*) ou de ida-e-volta (*round trip*).

Requisições unidirecionais não necessitam esperar por uma confirmação imediata de recebimento, que será enviada "de carona" em uma mensagem posterior do servidor. Isto melhora o desempenho do sistema, pois várias requisições podem ser armazenadas no *buffer* de transmissão e descarregadas em bloco, otimizando o tráfego na rede. Requisições de ida-e-volta precisam esperar a resposta do servidor e, por serem bloqueantes, seu uso deve ser evitado tanto quanto possível, para não degradar o desempenho do sistema.

2.5 Recursos do servidor

Recursos (*resources*) são os dados que residem na memória do servidor e são compartilháveis entre clientes de um mesmo *display*, mas não entre *displays* diferentes. Um recurso sobrevive enquanto durar a conexão do servidor com o cliente que o criou.

Janelas são áreas para desenho na tela, que podem conter outras janelas. A destruição de uma janela implica na destruição de todas as que nela estiverem contidas, mas a janela-raiz de um *screen* nunca pode ser destruída. **Mapas de pixels** (*pixmaps*) são espécies de janelas ocultas sobre as quais o cliente pode realizar qualquer operação de desenho que seria feita em uma janela. O conteúdo de um *pixmap* pode ser copiado para uma janela e vice-versa. Aplicações que fazem desenhos complexos, como animações ou imagens de alto realismo, podem acelerar a exibição mandando o servidor copiar partes de um *pixmap* para a área exposta da janela, ao invés de redesenhá-la.

Contextos gráficos (GCs — *graphic contexts*) guardam conjuntos de atributos gráficos como cores de frente e fundo, padrões para desenho de linhas, preenchimento de polígonos, fontes de caracteres, etc. GCs são globais para todo um *display* e podem ser usados por qualquer cliente em operações sobre qualquer janela.

Fontes são descrições da aparência de conjuntos de caracteres, incluindo a família, inclinação, tamanho, espaçamento e tipo de codificação, de acordo com os idiomas aos quais eles se destinam.

Mapas de cores (*colormaps*) são tabelas nas quais cada cor é referenciada por um índice. O servidor provê um mapa global para ser usado por todos os clientes, mas aplicações que fazem uso intenso de cores podem criar novos mapas de cores no servidor, para seu uso privado.

Cursores descrevem a aparência do cursor associado ao dispositivo apontador. O cliente não precisa desenhar o cursor, apenas requisitar ao servidor que use um certo padrão (ou nenhum) sempre que o apontador estiver sobre uma certa janela.

2.6 Comunicação entre clientes

X provê mecanismos de comunicação não só entre cliente e servidor, mas também entre clientes. A rigor, um cliente não pode realmente enviar eventos para outro cliente, apenas para as janelas que este possui no servidor. Clientes que desejam receber eventos enviados por outros deverão fazer uma requisição ao servidor, que a partir de então os enviará. Um cliente não precisa solicitar eventos para uma janela criada por ele, mas pode informar que não os deseja. As convenções para interação entre aplicações são descritas detalhadamente no *Inter-Client Conventions Manual* (ICCCM) [17].

2.6.1 Propriedades, átomos, áreas de recorte e seleções

Eventos permitem envio de mensagens, mas não são adequados ao compartilhamento de dados entre clientes. Para fazer isto existem as **propriedades** (*properties*), blocos de dados que um cliente "pendura" em uma janela. Propriedades podem ser criadas, destruídas, lidas e gravadas. Aplicações que compartilham dados via propriedades devem solicitar ao servidor a notificação de alterações nas mesmas.

Átomos (*atoms*) são identificadores únicos que os clientes podem usar para comunicar informação uns aos outros. Um átomo é simplesmente uma cadeia de bytes de tamanho arbitrário. Ao invés de enviar essas cadeias pela rede o cliente registra-as no servidor, obtendo um rótulo único. Átomos também são usados para especificar tipos de dados e nomes de propriedades e seleções.

A janela-raiz de cada tela possui um grupo de **áreas de recorte** (*cut buffers*), propriedades identificadas pelos átomos pré-definidos `XA_CUTBUFFER0` a `XA_CUTBUFFER7`. A Xlib possui funções para armazenar bytes nestas áreas de recorte, permitindo às aplicações implementar um sistema simples de recortar-e-colar.

Seleções (*selections*) são mecanismos mais sofisticados do que os *cut buffers* para compartilhamento de dados [14, cap. 11]. Uma seleção é tratada como um bastão em um sistema de revezamento. Apenas um cliente pode deter o bastão de cada vez. O detentor do bastão deve então atender às requisições dos outros clientes, convertendo os dados selecionados para um formato solicitado, armazenando o resultado em uma propriedade e notificando o solicitante da disponibilidade. Seleções permitem que clientes armazenem os dados em seu próprio espaço de dados e não na memória do servidor X.

2.7 Arrastar e soltar

Arrastar e soltar, ou *drag-and-drop* (DnD), é a técnica interativa baseada na metáfora de selecionar um objeto em uma janela e transportá-lo para outra, resultando em uma transferência dos dados entre as duas aplicações. X não provê explicitamente tal recurso, pois ele está associado a uma política. As aplicações devem implementá-lo

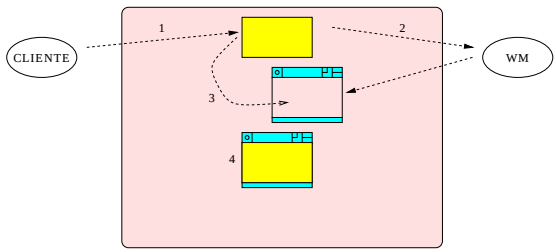

Figura 1: Criação de uma janela *top-level*. O cliente cria a janela dentro da janela-raiz e requisita seu mapeamento numa tela ao servidor X (1). O gerenciador de janelas recebe uma notificação (2), verifica as propriedades colocadas na janela pelo cliente e cria um container cuja aparência depende dos *hints* acoplados à janela. Finalmente, o gerenciador de janelas requisita o *reparenting* da janela para dentro da borda (3). O usuário sempre verá a janela da aplicação dentro do container (4).

usando os mecanismos disponíveis.

Em decorrência disso, há muitos protocolos DnD para X, dentre os quais destacam-se três. O *toolkit* Motif [16] define um protocolo DnD baseado em mensagens entre clientes, propridades e seleções. Como exemplo de aplicação que usa esse protocolo, podemos citar o Netscape Navigator (versão para UNIX). O pacote de aplicativos Offix [15], que posteriormente evoluiu para um *toolkit* gráfico completo, incluía um protocolo DnD que foi usado por um grande número de aplicações. Xdnd [9] é um protocolo recente, criado para o *toolkit* JX [10], mas que está se tornando padrão de fato entre os pacotes de software livre.

Esses "padrões" são incompatíveis entre si e a falta de um padrão real é, provavelmente, o motivo principal da relativa escassez de aplicações para X que suportem DnD. Uma comparação entre os diversos protocolos é feita em [9].

2.8 Gerenciamento de janelas

O gerenciador de janelas é um cliente especial que implementa as políticas de atribuição do foco (a janela em foco recebe os eventos de teclado), redimensionamento e sobreposição de janelas. O programa faz isso com base nas convenções definidas no ICCCM, que define um conjunto padrão de propriedades acopladas a cada janela que funcionam como dicas (*hints*) para o gerenciador de janelas, contendo informações relativas à forma como a janela deve ser tratada. A Figura 1 mostra como o gerenciador de janelas trata as aplicações.

2.9 Bibliotecas e *toolkits*

As bibliotecas que implementam interfaces de programação para X são chamadas *toolkits*. O X Consortium nunca impôs um *toolkit* padrão para interfaces gráficas, o que resultou na grande proliferação desse tipo de software. Há dezenas de *toolkits* diferentes, que vão de simples conjuntos de *widgets* (elementos da interface gráfica, tais como botões, caixas de texto, etc.) usados em uma única aplicação até pacotes bastante sofisticados. Abordaremos aqui dois deles.

Xlib. É a API de mais baixo nível disponível para X [2]. É um padrão estabelecido pelo X Consortium, implementado como uma biblioteca em C, que oferece funções com correspondência quase direta com o protocolo X. Todos os *toolkits* conhecidos são implementados sobre Xlib, mas oferecem níveis mais altos de abstração.

X Toolkit (Xt). Foi criado com a intenção de prover recursos básicos para o desenvolvimento de outros *toolkits* [11]. Xt não tem *widgets* completos, mas fornece os mecanismos necessários para se criar uma biblioteca que os possua. A idéia é que Xt ofereça o suporte básico e que um *toolkit* complementar forneça os *widgets*.

2.10 Técnicas de programação

2.10.1 Estrutura de um programa

Um programa para X divide-se em três partes, em linhas gerais:

Inicialização. Durante a inicialização, o programa faz o tratamento de opções de linha de comando, conexão com o servidor X (seja ela local ou remota) e alocação de estruturas internas.

Montagem da interface gráfica. Após aberta a conexão com o servidor, o programa pode criar e montar sua interface utilizando os *widgets* oferecidos pelo *toolkit*, ou manualmente, usando as primitivas do Xlib.

Laço de tratamento de eventos. Finalmente, o programa entra em um laço, onde fica esperando por eventos vindos do servidor e reagindo a eles. Em *toolkits*, estes eventos são primeiro tratados internamente e dependendo do caso (quando o mouse é clicado em um *widget* botão, por exemplo) são repassados ao programa através de *callbacks* ou mecanismos similares.

2.10.2 Tratamento de eventos

Ao usar a aplicação, o usuário move o mouse, clica seus botões e digita no teclado. Essas ações são comunicadas pelo servidor à aplicação por meio de eventos. Outras

origens de eventos são a modificação da hierarquia de janelas, obscurecimento e exposição de uma janela ou de partes dela. Um cliente X consiste então de um programa que envia requisições ao servidor e reage a eventos.

Nem todos os eventos podem ser interessantes para o cliente. Pode-se informar ao servidor que tipos de eventos devem ser reportados para cada um dos objetos criados pelo cliente, o que ajuda a reduzir o tráfego de mensagens desnecessárias, além de simplificar o código da aplicação e consumir menos processamento.

Uma conseqüência indesejável do sistema de eventos é que, mesmo selecionando os tipos que se quer receber, muitos deles ainda podem ser desnecessários. A maneira correta de tratar esse problema é descartar todos os eventos de uma cadeia de eventos do mesmo tipo e reagir apenas ao último [5]. O trecho de programa a seguir demonstra essa técnica, exemplificando o uso de funções existentes na Xlib para verificar se há eventos na fila (`XEventsQueued`), consultar o próximo evento sem retirá-lo da fila (`XPeekEvent`) e retirar um evento da fila (`XNextEvent`). A verificação tem que ser feita antes de tentar ler o evento porque a chamada `XNextEvent` é bloqueante.

```
/* laço de tratamento de eventos */
done = 0;
while(!done) {
   /* lê o próximo evento */
   XNextEvent(mydisplay, &myev);
   switch(myev.type) {
      /* redesenha a janela em eventos Expose */
      case Expose:
        /* Comprime uma seqüência de Exposes, para evitar repetidas
           repinturas, já que para um redimensionamento da janela o
           servidor X envia vários Exposes consecutivos.   */
         while(
            (XEventsQueued(myev.xexpose.display,QueuedAfterReading) > 0)
            && (XPeekEvent(myev.xexpose.display, &nextev),
                (nextev.type == Expose))
            ) {
            XNextEvent(mydisplay, &myev);
         }
         desenhar(mydisplay, mywindow, mygc);
         break;
```

2.10.3 Primitivas de desenho

X possui uma série de mecanismos para a geração de formas geométricas simples. Pontos individuais são desenhados usando a função `XDrawPoint`:

```
XDrawPoint(display, d, gc, x, y)
    Display *display;
    Drawable d;
    GC gc;
    int x, y;
```

No código acima, `Drawable` pode ser uma janela ou um *pixmap* e `GC` é um contexto gráfico que determina os atributos usados para desenho. Múltiplos pontos podem ser desenhados de uma só vez com a função `XDrawPoints`.

À semelhança dos pontos, linhas são desenhadas com a função `XDrawLine` e múltiplas linhas podem ser desenhadas de uma única vez usando a função `XDrawLines`.

`XDrawArc` e `XDrawArcs` desenham um ou diversos arcos (círculos e elipses). Ao usar essas funções, é importante observar que no X os ângulos são sempre expressos em $\frac{1}{64}$ de grau. Isto pode parecer estranho, mas tem a vantagem de permitir que se usem valores inteiros para os ângulos.

`XDrawRectangle` e `XDrawRectangles` desenham um ou diversos retângulos de cada vez. Os retângulos não podem ser rotacionados em relação aos eixos x e y da tela. Caso seja necessário fazer isto, será preciso calcular as coordenadas de cada vértice do retângulo e desenhá-lo usando `XDrawLines`. Retângulos são os únicos polígonos suportados diretamente. Caso se queira desenhar outros tipos, será necessário calcular as coordenadas dos vértices e desenhá-los usando `XDrawLines`.

Para o preenchimento de áreas, as funções `XFillArc`, `XFillArcs`, `XFillPolygon`, `XFillRectangle` e `XFillRectangles` permitem desenhar as mesmas formas geométricas mencionadas anteriormente, mas preenchidas com uma cor ou padrão de pontos.

2.11 Usando X Window System

Graças à sua filosofia de oferecer mecanismo e não política, associada à extensibilidade do protocolo, X tem evoluído ao longo de sua existência, sempre se conservando como uma plataforma de desenvolvimento estável, confiável e versátil para a construção de aplicações gráficas interativas. A existência de uma especificação formal rígida para o protocolo garante também a interoperabilidade em ambientes heterogêneos de hardware/software.

Por ser baseado em um protocolo de comunicações e ter uma arquitetura cliente-servidor, que garante independência entre aplicação e hardware gráfico, X provê suporte transparente à operação em ambiente distribuído. A maioria dos sistemas de janelas opera em modo "kernel", o que obriga a aplicação a ser executada na mesma máquina em que está o *display*. Além disso, X tem escalabilidade: uma máquina poderosa pode servir simultaneamente a dezenas de usuários conectados a ela por terminais X remotos.

As noções sobre programação apresentadas nas seções anteriores representam muito pouco daquilo que se pode fazer com X, principalmente quando combinado com bibliotecas de interface e ferramentas de mais alto nível, como OpenGL que será visto na próxima seção. Para obter informações mais profundas, a documentação de referência se encontra na forma de páginas de manual, normalmente instaladas em sistemas com o X. Manuais de referência em PostScript(r) podem ser obtidos via FTP anônimo em `ftp://ftp.x.org/pub/R6.4/xc/docs/hardcopy/`. Referências sobre sobre X Toolkit e Motif podem ser encontradas a partir das páginas do projeto

LessTif: `http://www.lesstif.org/` e da *Motif Zone* `http://www.motifzone.net/`.

3 OpenGL

Padrões gráficos, como GKS (*Graphics Kernel System*) e PHIGS, tiveram importante papel na década de 80, inclusive ajudando a estabelecer o conceito de uso de padrões mesmo fora da área gráfica, tendo sido implementados em diversas plataformas. Nenhuma destas APIs, no entanto, conseguiu ter grande aceitação [20].

Atualmente, o OpenGL ("GL" significa *Graphics Library*) é uma API de grande utilização no desenvolvimento de aplicações em Computação Gráfica [13]. Este padrão é o sucessor da biblioteca gráfica conhecida como IRIS GL, desenvolvida pela Silicon Graphics como uma interface gráfica independente de hardware [6]. A maioria das funcionalidades da IRIS GL foi removida ou rescrita no OpenGL e as rotinas e os símbolos foram renomeados para evitar conflitos (todos os nomes começam com `gl` ou `GL_`). Na mesma época foi formado o OpenGL Architecture Review Board, um consórcio independente que administra o uso do OpenGL, formado por diversas empresas da área.

OpenGL é uma interface que disponibiliza um controle simples e direto sobre um conjunto de rotinas, permitindo ao programador especificar os objetos e as operações necessárias para a produção de imagens gráficas de alta qualidade. OpenGL é uma máquina de estados, onde o controle de vários atributos é realizado através de um conjunto de variáveis de estado que inicialmente possuem valores default, podendo ser alterados caso seja necessário. Assim, por exemplo, todo objeto será traçado com a mesma cor até que seja definido um novo valor para esta variável.

Por ser um padrão destinado somente à renderização [21], OpenGL pode ser utilizado em qualquer sistema de janelas (por exemplo, X ou Windows), aproveitando-se dos recursos disponibilizados pelos diversos hardwares gráficos existentes. No X Window System ele é integrado através do GLX (*OpenGL Extension for X*), um conjunto de rotinas para criar e gerenciar um contexto de renderização do OpenGL no X [6], [21]. Além do GLX, existem outras bibliotecas alternativas para interfaceamento no X, tais como GLUT (*OpenGL Utility Toolkit*) [7] e GTK [3]. Estas bibliotecas possuem um conjunto de ferramentas que facilita a construção de programas utilizando o OpenGL.

3.1 Objetos geométricos

OpenGL é uma interface que não possui rotinas de alto nível de abstração. Desta forma, as primitivas geométricas são construídas a partir de seus vértices. Um vértice é representado em coordenadas homogêneas (x, y, z, w). Se w for diferente de zero, estas coordenadas correspondem a um ponto tridimensional euclidiano (x/w, y/w, z/w). Além disso, todos os cálculos internos são realizados com pontos definidos no espaço

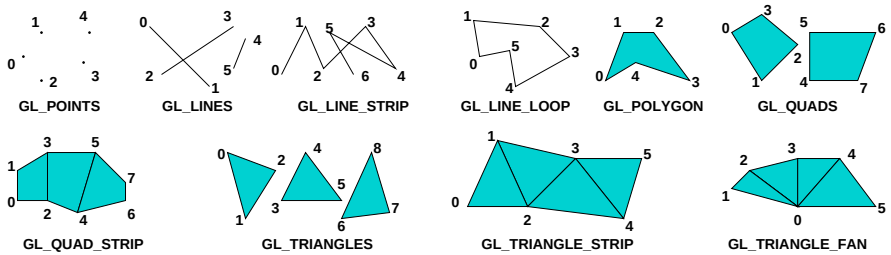

Figura 2: Primitivas geométricas do OpenGL [6].

tridimensional. Assim sendo, os pontos bidimensionais especificados pelo usuário são trabalhados como pontos tridimensionais, onde a coordenada z é igual a zero. Os segmentos de reta são representados por seus pontos extremos e os polígonos são áreas definidas por um conjunto de segmentos. No OpenGL, alguns cuidados quanto à definição de um polígono devem ser tomados: um polígono deverá ser sempre convexo e não poderá ter interseção das suas arestas (conhecido como polígono simples). A especificação de um vértice é feita através das funções `glVertex*` (O * será utilizado neste texto para representar variantes no nome da função; as variações restringem-se ao número e/ou ao tipo dos argumentos — por exemplo, `glVertex3i` definirá um vértice com três coordenadas inteiras).

Em muitas aplicações gráficas há a necessidade de definir polígonos não simples, côncavos ou com furos. Como qualquer polígono pode ser formado a partir da união de polígonos convexos, algumas rotinas mais complexas, derivadas das primitivas básicas, são fornecidas na GLU (*OpenGL Utility Library*) [13]. Esta biblioteca utiliza somente funções padrões do OpenGL e está disponível em todas as implementações do OpenGL.

Para traçar um conjunto de pontos, um segmento ou um polígono, os vértices necessários para a definição destas primitivas são agrupados entre as chamadas das funções `glBegin` e `glEnd`. Pode-se adicionar também informações a um vértice, como uma cor, um vetor normal ou uma coordenada para a textura. O argumento da função `glBegin` indicará a ordem como serão associados os vértices, conforme ilustrado na Figura 2.

No OpenGL, uma primitiva pode ser traçada de diferentes maneiras, conforme a ordem selecionada e o conjunto de vértices definido. O trecho de código a seguir apresenta um exemplo do traçado de uma circunferência.

```
#define PI 3.1415926535
int circle_points = 100;

glBegin(GL_LINE_LOOP);
for (i = 0; i < circle_points; i++) {
   angle = 2*PI*i/circle_points;
```

```
    glVertex2f(cos(angle), sin(angle));
}
glEnd();
```

O exemplo acima não é o modo mais eficiente para traçar uma circunferência, especialmente se esta primitiva for utilizada várias vezes. Neste caso, o desempenho ficará comprometido porque há o cálculo do ângulo e a execução das funções `sin` e `cos` para cada vértice, além do *overhead* do *loop*. Poderíamos solucionar este problema calculando as coordenadas dos vértices uma única vez e armazenando-os em uma tabela, ou utilizando uma rotina GLU/GLUT, ou ainda criando uma *display list*, que é uma maneira de definir um grupo de rotinas do OpenGL que serão executadas posteriormente, respeitando a seqüência em que foram definidas.

3.2 Visualização

Em Computação Gráfica, organizar as operações necessárias para converter objetos definidos em um espaço tridimensional para um espaço bidimensional (tela do computador) é uma das principais dificuldades no desenvolvimento de aplicações. Para isso, aspectos como transformações, *clipping* e definição das dimensões de *viewport* (porção da janela onde a imagem é desenhada) devem ser considerados.

Transformações são operações descritas pela multiplicação de matrizes. Estas matrizes podem descrever uma modelagem, uma visualização ou uma projeção, dependendo do contexto. ***Clipping*** é a eliminação de objetos (ou partes de objetos) que estão situados fora do volume de visualização. O enquadramento das imagens no ***viewport*** é a operação de correspondência entre as coordenadas transformadas e os pixels da tela.

As matrizes de modelagem posicionam e orientam os objetos na cena, as matrizes de visualização determinam o posicionamento da câmera e as matrizes de projeção determinam o volume de visualização (análogo à escolha da lente para uma máquina fotográfica). No OpenGL, as operações com estas matrizes são realizadas através de duas pilhas: a pilha que manipula as matrizes de modelagem e de visualização (*modelview*) e a pilha que manipula as matrizes de projeção (*projection*). As operações de modelagem e de visualização são trabalhadas na mesma pilha, pois pode-se posicionar a câmera em relação à cena ou vice-versa, onde o resultado de ambas operações será o mesmo. A definição da pilha na qual se deseja trabalhar é feita através da rotina `glMatrixMode`, indicada pelo argumento `GL_MODELVIEW` ou `GL_PROJECTION`.

Para o posicionamento da câmera ou de um objeto são utilizadas as rotinas `glRotate*` e/ou `glTranslate*`, que definem respectivamente matrizes de rotação e de translação. Por default, a câmera e os objetos na cena são originalmente situados na origem. Há também a rotina `glScale*`, que define uma matriz de escalonamento.

Quanto à definição da projeção, OpenGL provê duas transformações: a perspectiva e a ortogonal. Na perspectiva, a projeção do objeto é reduzida à medida que ele é

afastado da câmera. Na ortogonal, a projeção não é afetada pela sua distância em relação à câmera.

Para estabelecer a área na tela onde a imagem será renderizada é utilizada a rotina `glViewport`. Esta rotina poderá distorcer a imagem, caso a relação entre a altura e a largura da janela na tela não corresponder a mesma relação utilizada para definir a janela no volume de visualização.

3.3 Cor

OpenGL possui dois modos diferentes para tratar cor: o modo RGBA e o modo indexado de cor [6]. O modo RGBA possui as componentes vermelho, verde, azul e alfa, respectivamente. Os três primeiros representam as cores primárias e são lineares (variando de 0.0 a 1.0). A componente alfa é utilizada, por exemplo, em operações de *blending* (mistura) e transparência. Esta componente representa a opacidade da cor, variando de 0.0 (totalmente transparente) até 1.0 (totalmente opaca).

O modo indexado de cor utiliza um mapa de cores. Este mapa armazena em cada índice os valores para cada componente primária (RGB). A cor então é trabalhada pelo índice, e não por suas componentes. OpenGL não tem rotinas específicas para alocação de cores, sendo o sistema de janelas responsável por esta função.

3.4 Aparência dos Objetos

OpenGL utiliza o modelo de Gouraud para a tonalização, provendo uma cena com realismo. Uma cena é renderizada levando-se em consideração alguns aspectos como, por exemplo, o tipo de fonte de iluminação que está sendo usada na cena e as propriedades do material para cada superfície. Alguns efeitos complexos como a reflexão da luz e sombra não são diretamente suportados, embora estejam disponíveis códigos-fonte na rede para simular tais efeitos.

Para implementar o modelo de iluminação, OpenGL decompõe o raio luminoso nas componentes primárias RGB para cada componente de luz do modelo de iluminação, que são:

Componente ambiente. Componente proveniente de uma fonte que não é possível determinar. Por exemplo, a "luz dispersa" em uma sala tem uma grande quantidade da componente ambiente, pois esta luz é resultante de multireflexões nas superfícies contidas na cena.

Componente difusa. Componente refletida em todas as direções quando esta incide sobre uma superfície, proveniente de uma direção específica. A intensidade de luz refletida será a mesma para o observador, não importando onde ele esteja situado (superfície Lambertiana).

Componente especular. Componente refletida em uma determinada direção quando esta incide sobre uma superfície, proveniente de uma direção específica. Su-

perfícies como metais brilhantes produzem grande quantidade de reflexão especular. O modelo de Phong é utilizado para o cálculo da reflexão especular.

Quanto aos tipos de fonte de iluminação, OpenGL possui **fontes pontuais**, que irradiam energia luminosa em todas as direções e **spots**, que são fontes pontuais direcionais, i.e., têm uma direção principal na qual ocorre a máxima concentração de energia luminosa; fora desta direção ocorre uma atenuação desta energia.

Da mesma maneira que a luz, diferentes materiais de superfícies podem apresentar comportamentos distintos em relação às componentes especular, difusa e ambiente de uma ou mais fontes luminosas, determinando assim a cor da superfície. Uma reflexão da componente ambiente do material é combinada com a componente ambiente da luz, da mesma forma a reflexão da componente difusa do material com a componente difusa da luz e similarmente para a reflexão especular. As reflexões difusa e ambiente definem a cor do material, enquanto a reflexão especular geralmente produz uma cor branca ou cinza. As rotinas `glMaterial*` são utilizadas para determinar as propriedades dos materiais.

OpenGL também provê suporte para o mapeamento de texturas, embora esse ainda seja um recurso bastante limitado, pois não existem facilidades para mapear imagens de outras fontes (é necessário um programa auxiliar para converter uma imagem em uma representação aceita pelo OpenGL).

3.5 *Framebuffer*

Um conjunto de *buffers* de uma determinada janela ou de um determinado contexto é denominado *framebuffer*. No OpenGL, há um conjunto de *buffers* que podem ser manipulados conforme a necessidade [13]:

***Buffers* de cor.** São normalmente utilizados para traçar a imagem. Podem conter cores indexadas ou RGBA. Dependendo da aplicação, pode-se trabalhar com imagens estéreo ou *double buffering* (dois *buffers* de imagem, um visível e outro não), desde que o sistema de janelas e o hardware suportem. Como no OpenGL não há rotinas específicas para produção de animações, a utilização de um *double buffer* é uma opção. No X, por exemplo, há um comando denominado `glXSwapBuffers`, que disponibiliza este recurso. Dessa forma, enquanto um quadro é exibido na tela, o próximo quadro está sendo renderizado no *buffer* não visível.

***Buffer* de profundidade.** É utilizado para armazenar, durante a renderização, o pixel cuja profundidade (coordenada z) é dada pela função de comparação `glDepthFunc`. Por exemplo, para realizar o algoritmo *z-buffer*, a função pode escolher o pixel de menor coordenada z dentre os que tiverem as mesmas coordenadas x e y.

***Buffer Stencil* (seleção).** Serve para eliminar ou manter certos pixels na tela, dependendo de alguns testes disponibilizados para este *buffer*. É muito utilizado em simuladores onde é necessário manter certas áreas e alterar outras.

***Buffer* de acumulação.** Contém cores especificadas em RGBA. É utilizado para acumular um conjunto de imagens através de uma operação pré-especificada. A imagem resultante destas acumulações é exibida através da transferência para um *buffer* de cor. Pode ser utilizado para trabalhar com diversas técnicas como, por exemplo, *antialiasing*, *motion blur* (borrão) ou profundidade de campo.

3.6 Usando OpenGL

A página do consórcio que administra o OpenGL é `http://www.opengl.org`, e a página da SGI é `http://www.sgi.com/software/opengl`. Informações de âmbito geral e bibliotecas relacionadas podem ser obtidas em `http://reality.sgi.com/opengl/`. Existe também um clone não-proprietário do OpenGL chamado Mesa (`http://www.mesa3d.org`).

OpenGL foi projetado para fornecer o máximo acesso às capacidades de diferentes hardwares gráficos, sendo implementável e executável em uma variedade de sistemas. O Microsoft DirectX SDK (`http://www.microsoft.com/directx/`) disponibiliza, através das APIs DirectDraw e Direct3D, recursos semelhantes aos encontrados no OpenGL, mas este é um sistema comercial, disponível apenas para o Windows. Além disso, segundo [8], Direct3D é uma API muito complexa e difícil de utilizar.

Pela sua flexibilidade em modelar e renderizar objetos geométricos, OpenGL serve como uma excelente base para construir bibliotecas para determinados contextos de aplicação, como Java 3D que será vista na próxima seção.

4 Java 3D

Java 3D é uma interface criada para o desenvolvimento de aplicações gráficas trimidensionais em Java, executada no topo de bibliotecas gráficas de mais baixo nível, tais como OpenGL e Direct3D, conforme ilustra a Figura 3. Com isto, os programadores de aplicações passam a explorar, agora no âmbito das aplicações gráficas tridimensionais, o conjunto de facilidades e vantagens da plataforma Java, como orientação a objetos, segurança e independência de plataforma. Em particular, a orientação a objetos oferece uma abordagem de alto nível à programação e possibilita que o desenvolvedor se dedique mais à criação do que aos problemas de mais baixo nível pertinentes à programação 3D, os quais exigem um esforço considerável. Esta tecnologia gráfica vem ainda ao encontro de uma crescente demanda por operações 3D requisitada hoje pela Web.

Java 3D utiliza alguns conceitos que são comuns a outras tecnologias, tais como a VRML (Seção 5), considerada por alguns autores como sendo sua "prima" mais

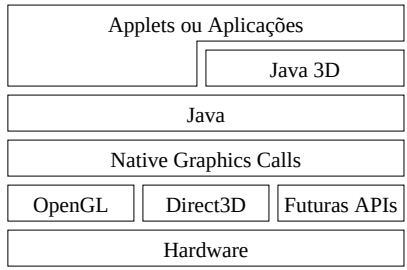

Figura 3: Relação entre as várias camadas de software no contexto de Java 3D.

próxima [1]. Uma aplicação Java 3D é projetada a partir de um grafo de cena contendo objetos gráficos, luz, som, objetos de interação, entre outros, que possibilitam ao programador criar mundos virtuais com personagens que interagem entre si e/ou com o usuário [22]. Descrever uma cena usando um grafo é tarefa mais simples do que construir a mesma usando linhas de comando que especificam primitivas gráficas, tais como as do OpenGL.

4.1 O grafo de cena em Java 3D

O primeiro procedimento na elaboração de uma aplicação Java 3D é definir o universo virtual, que é composto por um ou mais grafos de cena. O grafo de cena é uma estrutura do tipo árvore cujos nós são objetos instanciados das classes Java 3D e os arcos representam o tipo de relação estabelecida entre dois nós. Os objetos definem a geometria, luz, aparência, orientação, localização, entre outros aspectos da cena. A Figura 4 ilustra um possível exemplo de grafo de cenas.

Os grafos de cenas são conectados ao universo virtual (representado na Figura 4 através do nó `VirtualUniverse`) por meio de um nó `Locale`. Um nó `VirtualUniverse` pode ter um ou mais nós `Locale`, cuja finalidade é fornecer um sistema de coordenadas ao mundo virtual. O nó-raiz de um grafo de cena (*branch graph*) é sempre um objeto `BranchGroup`.

Os *branch graphs* são classificados em duas categorias: de conteúdo (*content branch graph*) e de vista (*view branch graph*). Os *content branch graphs* descrevem os objetos que serão renderizados, i.e., especificam a geometria, textura, som, objetos de interação, luz, como estes objetos serão localizados no espaço, etc. (na Figura 4 é o ramo à esquerda do nó `Locale`). Os *view branch graphs* especificam as atividades e parâmetros relacionados com o controle da vista da cena, tais como orientação e localização do usuário (na figura é o ramo à direita do nó `Locale`).

Um caminho entre o nó-raiz do grafo de cenas até um de seus nós folhas determina de forma única todas as informações necessárias para se processar este nó. O modelo de renderização de Java 3D explora este fato renderizando os nós folhas em uma ordem

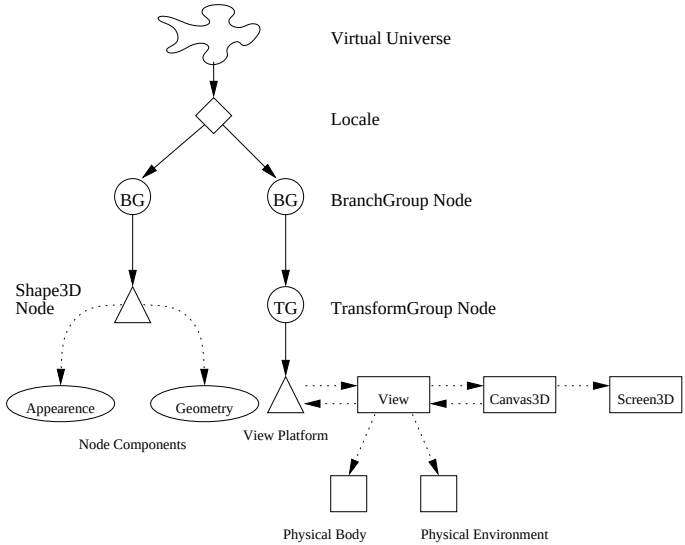

Figura 4: Grafo de uma cena em Java 3D.

que ele determina ser a mais eficiente. Em geral o programador não se preocupa em determinar uma ordem de renderização, uma vez que Java 3D fará isto da forma mais eficiente. No entanto, um programador poderá exercer, de forma limitada, algum controle usando um nó `OrderedGroup`, que assegura que seus filhos serão renderizados em uma ordem pré-definida, ou um nó `Switch`, que seleciona um ou mais filhos a serem renderizados.

Java 3D organiza o universo virtual usando o conceito de agrupamento, i.e., um nó mantém uma combinação de outros nós de modo a formar um componente único. Estes nós são denominados `Group`. Um nó `Group` pode ter uma quantidade arbitrária de filhos que são inseridos ou removidos dependendo do que se pretende realizar. Discutimos anteriormente os nós `BranchGroup`, `OrderedGroup` e `Switch`. Inserem-se ainda nesta categoria os nós `TransformGroup`, que são usados para alterar a localização, orientação e/ou escala do grupo de nós descendentes.

Um nó que não possui filhos pertence a uma segunda categoria e é denominado nó `Leaf`. Estes nós são usados para especificar luz, som, procedimentos de interação, forma dos objetos geométricos, orientação e localização do observador no mundo virtual, entre outros. Estas informações estão armazenadas no próprio nó `Leaf` ou então é feita uma referência a um objeto `NodeComponent` que mantém os dados necessários para o processo de renderização. Os objetos `NodeComponent` não fazem parte do grafo de cenas, i.e., a relação entre um nó `Leaf` e um `NodeComponent` não é do tipo pai-filho, mas de referência. Este fato possibilita que diferentes nós `Leaf` referenciem um

mesmo `NodeComponent` sem violar as propriedades do grafo de cenas, que é um grafo direcionado acíclico.

Como exemplos de nós `Leaf` podem ser citados: `Light`, `Sound`, `Behavior`, `Shape3D` e `ViewPlatform`. Nós `Shape3D` são usados para construir formas 3D a partir de informações geométricas e atributos que estão armazenados em um objeto `NodeComponent` (na Figura 4, são os elementos referenciados por arcos tracejados). Os nós `Behavior` são usados na manipulação de eventos disparados pelo usuário e na animação de objetos do universo virtual. Um nó `ViewPlatform` é usado para definir a localização e orientação do observador (ou do usuário) no mundo virtual. Um programa Java 3D pode fazer o observador navegar pelo mundo virtual aplicando transformações de translações, rotações e escalonamentos neste nó.

4.1.1 Escrevendo um programa em Java 3D

A estrutura típica de um programa Java 3D em geral tem dois ramos: um *view branch* e um *content branch*. Assim, escrever um programa em Java 3D requer basicamente criar os objetos necessários para construir os ramos *view branch* e *content branch*. Um bom ponto de partida é a seqüência de passos sugerida por [23]:

1. Criar um objeto `Canvas3D`
2. Criar um objeto `VirtualUniverse`
3. Criar um objeto `Locale` e anexá-lo ao `VirtualUniverse`
4. Construir um grafo *view branch*
   (a) Criar um objeto `View`
   (b) Criar um objeto `ViewPlatform`
   (c) Criar um objeto `PhysicalBody`
   (d) Criar um objeto `PhysicalEnvironment`
   (e) Anexar os objetos criados em (b), (c) e (d) ao objeto `View`
5. Construir um ou mais grafos *content branch*
6. Compilar o(s) *branch graph(s)*
7. Inserir os subgrafos ao nó `Locale`

A construção do grafo *view branch* (item 4 acima) mantém a mesma estrutura para a maioria dos programas Java3D. Uma forma de ganhar tempo é usar a classe `SimpleUniverse`, que retorna um universo virtual com os nós `VirtualUniverse` (item 2 acima), `Locale` (item 3), `ViewPlatform` e os objetos relativos ao item 4.

A seguir é apresentado o código de uma aplicação que cria um grafo de cenas que desenha um cubo rotacionado de $\pi/4$ radianos no eixo x e $\pi/5$ radianos no eixo y.

```
public class Exemplo01 extends Applet {
    public Exemplo01 () {
        setLayout(new BorderLayout());
        Canvas3D canvas3D = new Canvas3D(null);          // passo 1
        add("Center", canvas3D);
        BranchGroup s = ConstroiContentBranch();         // passo 5
        s.compile();                                     // passo 6

        // A instanciação de um objeto SimpleUniverse corresponde
        // aos passos 2, 3, e 4 da "receita"
        SimpleUniverse su = new SimpleUniverse(canvas3D);

        // Desloca o ViewPlatform para trás para que os
        // objetos da cena possam ser vistos.
        su.getViewingPlatform().setNominalViewingTransform();
        // Anexa o content graph ao nó Locale : passo 7
        su.addBranchGraph(s);
    }
public BranchGroup ConstroiContentBranch() {

        // Especificação dos conteúdos gráficos a serem renderizados
        BranchGroup objRoot = new BranchGroup();

        // Especificação das 2 rotações do cubo: uma no eixo x
        // e outra no eixo y. Depois as duas são combinadas.

        Transform3D rotate1 = new Transform3D();
        Transform3D rotate2 = new Transform3D();
        rotate1.rotX(Math.PI/4.0d);
        rotate2.rotY(Math.PI/5.0d);
        rotate1.mul(rotate2);
        TransformGroup objRotate = new TransformGroup(rotate1);

        objRoot.addChild(objRotate);
        objRotate.addChild(new ColorCube(0.4));
        return objRoot;
    }
   }
```

4.1.2 Configurando as capacidades de um objeto Java 3D

O grafo de cenas Java 3D pode ser convertido para uma representação interna que torna o processo de renderização mais eficiente. Esta conversão pode ser efetuada anexando cada *branch graph* a um nó `Locale`, tornando-os vivos (*live*) e, conseqüentemente, cada objeto do *branch graph* é dito estar vivo. A segunda maneira é compilando cada *branch graph*, usando o método `compile`, de forma a torná-los objetos compilados. Uma conseqüência de um objeto ser vivo e/ou compilado é que seus parâmetros não podem ser alterados a menos que tais alterações tenham sido explicitamente codificadas no programa antes da conversão. A lista de parâmetros

que podem ser acessados é denomominada *capabilities* e varia de objeto para objeto. O exemplo a seguir cria um nó `TransformGroup` e configura-o para escrita. Isto significa que o valor da transformada associada ao objeto `TransformGroup` poderá ser alterada mesmo depois dele tornar-se vivo e/ou compilado.

```
TransformGroup alvo = new TransformGroup();
     alvo.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
```

4.2 Modelagem

Um objeto 3D é uma instância da classe `Shape3D`, representado no grafo de cenas por um nó do tipo `Leaf`. O nó `Shape3D` não contém os atributos que definem o objeto 3D, tais como seu conteúdo geométrico e sua aparência. Estes atributos estão armazenados em objetos do tipo `NodeComponent`.

Uma alternativa é definir um objeto 3D como sendo uma extensão da classe `Shape3D`, o que é bastante útil quando se deseja criar múltiplas instâncias do objeto 3D com os mesmos atributos.

4.2.1 Definindo a geometria de um objeto 3D

Para que um objeto 3D seja funcional é necessário especificar pelo menos seu conteúdo geométrico. Java 3D oferece basicamente três maneiras de se criar um conteúdo geométrico. O primeiro método é o mais simples e consiste no emprego de primitivas geométricas, tais como *Box*, *Sphere*, *Cylinder* e *Cone*, cuja composição determina a forma do objeto desejado. Cada primitiva possui um método construtor onde são especificadas as dimensões do objeto (ou então estes são instanciados com dimensões default). Em razão da limitação inerente a este método como, por exemplo, a impossibilidade de alteração da geometria das primitivas, ele não é o mais apropriado para modelar a geometria de um objeto 3D mais complexo.

Um segundo método permite que os dados geométricos que modelam a forma do objeto 3D (primitivas geométricas definidas pelo programador) sejam especificados vértice a vértice em uma estrutura vetorial. Esta estrutura é definida usando as subclasses de `GeometryArray` como por exemplo: `PointArray` — define um vetor de vértices, `LineArray` — define um vetor de segmentos, `TriangleArray` — define um vetor de triângulos individuais e `QuadArray` — define um vetor de vértices onde cada quatro vértices correspondem a um quadrado individual. Estes polígonos devem ser convexos e planares.

Um objeto `GeometryArray` pode armazenar, além das coordenadas de localização, coordenadas do vetor normal à superfície, coordenadas de cores e de texturas. Uma alternativa ao `GeometryArray`, eliminando possíveis redundâncias, é usar a classe `IndexedGeometryArray` (subclasse de `GeometryArray`). Como o próprio nome sugere,

um objeto `IndexedGeometryArray` precisa, além do vetor de dados, de um vetor de índices para fazer referências aos elementos do vetor de dados.

As alternativas apresentadas no segundo método oferecem ao programador mais flexibilidade na definição da forma dos objetos, mas ainda pesam contra elas algumas desvantagens. A criação de um conteúdo geométrico mais elaborado vai exigir grande quantidade de tempo e de linhas de código para computar matematicamente ou especificar a lista de vértices do objeto de interesse. Esta baixa performance não motivará o programador a desenvolver formas mais sofisticadas.

Ainda com relação a esta abordagem de definir a forma do objeto 3D usando "força bruta", Java 3D disponibiliza um pacote `com.sun.j3d.utils.geometry` que oferece algumas facilidades neste processo. Por exemplo, ao invés de especificar exaustivamente as coordenadas, triângulo a triângulo, especifica-se um polígono arbitrário P (que pode ser não-planar e com "buracos") usando um objeto `GeometryInfo`, e a seguir efetua-se a triangulação de P usando um objeto `Triangulator`. A triangulação de um polígono não-planar, no entanto, pode gerar diferentes superfícies, de modo que o resultado obtido pode não ser o desejado.

Java 3D também oferece um terceiro método, baseado na importação de dados geométricos criados por outras aplicações, que resolve em grande parte os problemas citados anteriormente. Neste tipo de abordagem é comum usar um software específico para modelagem geométrica que ofereça facilidades para criar o modelo desejado. Feito isto, o conteúdo geométrico é então armazenado em um arquivo de formato padrão que posteriormente será importado para um programa Java 3D, processado e adicionado a um grafo de cenas. O trabalho de importação destes dados para um programa Java 3D é realizado pelos *loaders* [23]. Um *loader* sabe como ler um formato de arquivo 3D padrão e então construir uma cena Java 3D a partir dele. Existe uma grande variedade de formatos disponíveis na Web, por exemplo o formato VRML (.wrl), Wavefront (.obj), AutoCAD (.dfx), 3D Studio (.3ds), LightWave (.lws), entre outros. Uma cena Java 3D pode ainda ser construída combinando diferentes formatos de arquivo. Para isso é suficiente usar os *loaders* correspondentes. Por fim, vale observar que estes *loaders* não fazem parte da Java 3D, são implementações da interface definida no pacote `com.sun.j3d.loaders`.

4.2.2 Definindo a aparência de um objeto 3D

Um nó `Shape3D` pode, opcionalmente, referenciar um objeto `Appearance` para especificar suas propriedades, tais como textura, transparência, tipo de material, estilo de linha, entre outros. Estas informações não são mantidas no objeto `Appearance`, que faz referência a outros objetos `NodeComponent` que mantêm tais informações.

Estas propriedades, ou atributos, são definidas usando as subclasses de `NodeComponent`. Alguns exemplos, entre as várias subclasses existentes, são: `LineAttributes`, `PoligonAttributes` e `ColoringAttributes`. Um objeto `LineAttributes` é usado para definir a largura em pixels da linha, seu padrão (linha solida, tracejada,

pontilhada ou tracejada-pontilhada) e tratamento de *antialiasing*. Um objeto `PolygonAttributes` define, por exemplo, como os polígonos serão renderizados (preenchido, *wireframe* ou apenas os vértices). Um objeto `ColoringAttributes` define a cor dos objetos e o modelo de *shading*.

4.3 Interação

Programar um objeto para reagir a determinados eventos é uma capacidade desejável na grande maioria das aplicações 3D. Por exemplo, um evento poderia ser uma tecla pressionada, o movimento do mouse ou a colisão entre objetos, cuja reação seria alterar algum objeto (mudar cor, forma, posição, entre outros) ou o grafo de cenas (adicionar ou excluir um objeto). Quando estas alterações são resultantes diretas da ação do usuário elas são denominadas *interações*. As alterações realizadas independentemente do usuário são denominadas *animações* [23].

Java 3D implementa os conceitos de interação e animação na classe abstrata `Behavior`. Esta classe disponibiliza uma grande variedade de métodos para capacitar seus programas a perceber e tratar diferentes tipos de eventos. Permite ainda que o programador implemente seus próprios métodos. Os *behaviors* são os nós do grafo de cena usados para especificar o início da execução de uma determinada ação baseado na ocorrência de um conjunto de eventos, denominado `WakeupCondition`, ou seja, quando determinada combinação de eventos ocorrer Java 3D deve acionar o *behavior* correspondente para que execute as alterações programadas.

Uma `WakeupCondition`, condição usada para disparar um *behavior*, consiste de uma combinação de objetos `WakeupCriterion`, que são os objetos Java 3D usados para definir um evento ou uma combinação lógica de eventos.

Os *behaviors* são representados no grafo de cena por um nó `Leaf`, sempre fazendo referência a um objeto do grafo. É através deste objeto, denominado objeto-alvo, e em geral representado por um `TranformGroup`, que os *behaviors* promovem as alterações no mundo virtual.

Todos os *behaviors* são subclasses da classe abstrata `Behavior`. Eles compartilham uma estrutura básica composta de um método construtor, um método inicializador e um método `processStimulus`.

O método `initialize` é chamado uma vez quando o *behavior* torna-se vivo. No caso de um nó `Behavior`, o fato de estar vivo significa que ele está pronto para ser invocado. Define-se neste método o valor inicial da `WakeupCondition`.

O método `processStimulus` é chamado quando a condição de disparo especificada para o *behavior* ocorrer e ele estiver ativo. Esta rotina então efetua todo o processamento programado e define a próxima `WakeupCondition`, i.e., informa quando o *behavior* deve ser invocado novamente.

Por uma questão de desempenho, o programador define uma região limitada do espaço denominada *scheduling bound* para o *behavior*. Ele é dito estar ativo quando seu *scheduling bound* intercepta o volume de cena. Apenas os *behaviors* ativos estão

aptos a receber eventos [24].

4.4 Animação

Algumas alterações do mundo virtual podem ser realizadas independentemente da ação do usuário. Elas podem, por exemplo, ser disparadas em função do passar do tempo. Como comentado anteriormente, estas alterações são denominadas animações. Animações em Java 3D também são implementadas usando *behaviors*. Java 3D disponibiliza alguns conjuntos de classes, também baseadas em *behaviors*, que são próprias para implementar animações. Um primeiro conjunto destas classes são os interpoladores.

Os `Interpolators` são versões especializadas de *behaviors* usados para gerar uma animação baseada no tempo. O processo de animação acontece através de dois mapeamentos. No primeiro, um dado intervalo de tempo é mapeado sobre o intervalo fechado *I=[0.0, 1.0]*, a seguir *I* é mapeado em um espaço de valores pertinente ao objeto do grafo de cenas que se deseja animar (por exemplo, atributos de um objeto `Shape3D` ou a transformada associada a um objeto `TransformGroup`).

O primeiro mapeamento é efetuado por um objeto `Alpha`, que determina como o tempo será mapeado de forma a gerar os valores do intervalo *I*, denominados valores alpha. `Alpha` pode ser visto como uma aplicação do tempo que fornece os valores alpha em função dos seus parâmetros e do tempo.

O segundo mapeamento é determinado por um objeto `Interpolator` que, a partir dos valores alpha, realiza a animação do objeto referenciado. O programador pode projetar seus próprios interpoladores usando os valores alpha para animar objetos do mundo virtual. No entanto, Java 3D disponibiliza os `Interpolators`, que atendem a maioria das aplicações (`ColorInterpolator`, `PositionInterpolator`, `RotationInterpolator` entre outros). O procedimento usado para adicionar um objeto `Interpolator` segue um padrão básico, que independe do tipo de interpolador a ser usado: *i)* criar o objeto a ser interpolado, o objeto-alvo; *ii)* criar um objeto `Alpha` que descreve como gerar os valores alpha; *iii)* criar um objeto `Interpolator` usando `Alpha` e o objeto-alvo; *iv)* atribuir um *scheduling bound* ao `Interpolator`; *v)* anexar o `Interpolator` ao grafo de cenas;

Existem ainda as classes `billboard` e `LOD` (*Level of Detail*) que também permitem especificar animações, mas neste conjunto de classes as alterações são guiadas segundo a orientação ou posição da vista [23].

4.5 Usando Java 3D

A API Java 3D e sua documentação são encontradas em `http://java.sun.com/products/java-media/3D/`. Para sua utilização é necessário o ambiente Java 2 (SDK 1.2 para Solaris ou SDK ou JRE 1.2.2 para Windows, ou versões mais recentes). Para executar programas Java 3D na forma de *applets*, pode também ser necessário um

plug-in para atualizar a máquina virtual Java do browser para a versão 2. O *plug-in* pode ser obtido gratuitamente em `http://java.sun.com/products/plugin`.

O conjunto de capacidades incorporado por Java 3D oferece um ambiente de desenvolvimento de alto nível para o desenvolvimento e manipulação de complexas aplicações 3D. A programação de alto nível, no entanto, é obtida escondendo do programador detalhes de mais baixo nível, por exemplo, no processo de renderização, o que implica menos flexibilidade. Em contrapartida programadores com pouca experiência em programação 3D podem criar mais facilmente mundos virtuais em suas aplicações.

Java 3D, por ser uma API da linguagem Java, trabalha no nível desta linguagem (linguagem de programação orientada a objetos de alto nível). No entanto, para modelar ambientes 3D, ainda existem possibilidades de mais alto nível, tais como a VRML, que é essencialmente uma linguagem de *script* mais descritiva e direta que Java 3D.

5 VRML (*Virtual Reality Modeling Language*)

A linguagem VRML surgiu da necessidade de prover um formato gráfico 3D para a Web seguindo um modelo similar à HTML, ou seja, uma linguagem textual independente de plataforma para a descrição de cenas. A linguagem escolhida como referência foi a Open Inventor da SGI. Em 1995 foi lançada a VRML 1.0, que era basicamente um formato para a descrição de cenas estáticas 3D. Em 1997 foi lançada a VRML 2.0 (ou VRML 97) [25], que adicionou à linguagem conceitos de realidade virtual, tais como possibilidade de mover objetos da cena e criação de sensores para detectar e gerar eventos.

O projeto da VRML sempre foi aberto. As especificações são escritas por um consórcio envolvendo várias empresas e pesquisadores acadêmicos e são imediatamente disponibilizadas para realimentação, sugestões e críticas de toda a comunidade interessada. Até 1999, este consórcio se chamava *VRML Consortium*, e depois passou a se chamar *Web 3D Consortium* [29]. Sua principal atividade é a elaboração e manutenção de novos padrões para a transmissão de conteúdo tridimensional através da Web. Dentre outras tarefas, isto inclui uma melhor integração entre VRML, Java 3D, MPEG-4 e outras tecnologias relacionadas.

5.1 Estrutura hierárquica da cena

O paradigma para a criação de cenas VRML (também chamadas mundos VRML — *VRML worlds*) é baseado em nós, que são conjuntos de abstrações de objetos e de certas entidades do mundo real, tais como formas geométricas, luz e som.

Assim como em Java 3D, um mundo VRML é um grafo hierárquico em forma de árvore. As hierarquias são criadas através de nós de agrupamento, os quais contêm um campo chamado `children` que engloba uma lista de nós filhos. Há vários tipos

de nós em VRML [12]:

Agrupamento. Criam a estrutura hierárquica da cena e permitem que operações sejam aplicadas a um conjunto de nós simultaneamente. Alguns exemplos desse tipo de nó são:

Transform. É um nó de agrupamento que define um sistema de coordenadas para seus filhos que está relacionado com o sistema de coordenadas de seus pais. Sobre este sistema de coordenadas podem ser realizadas operações de translação, rotação e escalonamento.

Anchor. É um nó de agrupamento que recupera o conteúdo de um URL quando o usuário ativa alguma geometria contida em algum de seus nós filhos (pressiona o botão do mouse sobre eles, por exemplo).

Group. Equivalente ao nó `Transform` sem os campos de transformação.

Geométrico. Define a forma e a aparência de um objeto do mundo. O nó `Shape`, em particular, possui dois parâmetros: `geometry`, que define a forma do objeto e `appearance`, que define as propriedades visuais dos objetos (material ou textura). Alguns exemplos de nós geométricos (usados no campo `geometry` do nó `Shape`) são: `Box`, `Cone`, `Cylinder`, `Sphere`, `Text`, `IndexedLineSet` e `IndexedFaceSet`. Estes dois últimos descrevem, respectivamente, objetos 3D a partir de segmentos de reta e polígonos cujos vértices estão localizados em coordenadas dadas.

Appearance. Aparece apenas no campo `appearance` de um nó `Shape` e é responsável pela definição das propriedades visuais das figuras geométricas (material e textura).

Câmera. O nó `Viewpoint` define, entre outros parâmetros, a posição e orientação da câmera (ponto de vista do usuário).

Iluminação. Existem três tipos de nós de iluminação em VRML: `DirectionalLight` (fonte de luz direcional com raios paralelos), `PointLight` (fonte de luz pontual em um local fixo) e `SpotLight` (cone direcional de iluminação).

Além desses tipos básicos, existem ainda os nós sensores, os interpoladores e o nó `Script`, que serão vistos mais adiante.

5.2 Prototipação e reuso

Os mecanismos de prototipação da VRML permitem definir um novo tipo de nó baseado na combinação de nós já existentes. É permitido, inclusive, a criação de cenas distribuídas, pois o subgrafo (protótipo) pode ser definido em um arquivo remoto cujo URL é conhecido.

Atribuindo-se um nome a um nó através da palavra `DEF`, pode-se futuramente referenciá-lo através da palavra `USE`. Sendo assim, caso seja necessária a reutilização de um mesmo nó várias vezes em uma cena, é mais eficiente atribuir-lhe um nome na primeira vez que ele é descrito e posteriormente referenciá-lo por este nome.

Em resumo, VRML permite o encapsulamento (protótipos) e reutilização de subgrafos da cena. O exemplo a seguir mostra uma cena VRML simples (2 cones verdes), utilizando alguns dos conceitos apresentados até aqui (a primeira linha do arquivo deve ser sempre `#VRML V2.0 utf8` e os comentários devem começar com `#`):

```
#VRML V2.0 utf8

# Posição da câmera (ou observador)
Viewpoint { position 0 0 10 }

# Fonte de luz pontual
PointLight{
   color 1 1 1         # luz branca
   location 0 0 2      # posição da fonte
   on TRUE             # está "ligada"
   radius 20 }

Transform {
   # Um cone é definido e posicionado na origem
   children [
       DEF Cone_Verde Shape {
                geometry Cone {}               # cone
                appearance Appearance {
                     material Material {
                        diffuseColor 0 1 0 }   # cor verde
                }
            }

        # reutilização do cone
        Transform {
                translation 2 0 3       # posicionamento em (2 0 3)
                children USE Cone_Verde  }
    ]
}
```

5.3 Tipos de parâmetros e roteamento de eventos

Cada nó VRML define um nome, um tipo e um valor default para seus parâmetros. Há dois tipos de parâmetros possíveis: campos (*fields*) e eventos (*events*). Campos podem ser modificáveis (*exposedFields*) ou não (*fields*).

Os eventos podem ser enviados para outros nós por um parâmetro do tipo *eventOut* e recebidos por um *eventIn*. Eventos sinalizam mudanças causadas por "estímulos externos" e podem ser propagados entre os nós da cena por meio de roteamentos (*Routes*) que conectam um *eventOut* de um nó a um *eventIn* de outro nó, desde que

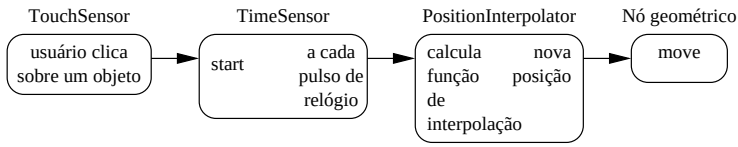

Figura 5: Roteamento de eventos em uma interação típica de VRML.

eles sejam eventos do mesmo tipo.

5.4 Sensores e interpoladores

Os nós sensores e interpoladores são especialmente importantes porque são os responsáveis pela interatividade e dinamismo dos mundos VRML.

Os sensores geram eventos baseados nas ações do usuário. O nó `TouchSensor`, por exemplo, detecta quando o usuário aciona o mouse sobre um determinado objeto (ou grupo de objetos) da cena. O `ProximitySensor`, por sua vez, detecta quando o usuário está navegando em uma região próxima ao objeto de interesse. Os sensores são responsáveis pela interação com o usuário, mas não estão restritos a gerar eventos a partir de ações dos mesmos. O `TimeSensor`, por exemplo, gera automaticamente um evento a cada pulso do relógio. Os *eventOuts* gerados pelos sensores podem ser ligados a outros *eventIns* da cena, iniciando uma animação, por exemplo.

A forma mais comum de se criar animações é usando *keyframes* (quadros-chave), especificando os momentos-chave na seqüência da animação. Os quadros intermediários são obtidos através da interpolação dos quadros-chave. Nós interpoladores servem para definir este tipo de animação, associando valores-chave (de posição, cor, etc.) que serão linearmente interpolados. Alguns exemplos de interpoladores são: `PositionInterpolator`, `OrientationInterpolator`, `ColorInterpolator` e `CoordinateInterpolator`.

Os eventos gerados por sensores e interpoladores, ligados a nós geométricos, de iluminação ou de agrupamento, podem definir comportamentos dinâmicos para os elementos do ambiente. Um exemplo típico (Figura 5) é rotear um `TouchSensor` em um `TimeSensor`, causando o disparo do relógio quando o usuário clicar sobre um objeto. O `TimeSensor`, por sua vez, é roteado em um interpolador, enviando para ele valores de tempo para a função de interpolação. O interpolador então é roteado em um nó geométrico, definindo as alterações deste objeto.

5.5 Nó `Script`

Apesar de ser um recurso poderoso, o roteamento de eventos entre os nós não é suficiente para o tratamento de várias classes de comportamento. Por exemplo, não é possível escolher entre duas trajetórias pré-definidas (lógica de decisão). Para superar

esta limitação, VRML define um nó especial chamado `Script`, que permite conectar o mundo VRML a programas externos, onde os eventos podem ser processados. Este programa externo, teoricamente, pode ser escrito em qualquer linguagem de programação, desde que o browser a suporte. Na prática, apenas Java e JavaScript são usadas.

O nó `Script` é ativado pelo recebimento de um evento. Quando isso ocorre, o browser inicia a execução do programa definido no campo `url` do nó `Script`. Este programa é capaz de receber, processar e gerar eventos que controlam o comportamento do mundo virtual. Por meio do nó `Script` é possível usar técnicas bem mais sofisticadas que a intepolação linear para a geração de animações.

5.6 EAI (*External Authoring Interface*)

A EAI [27] é uma interface para permitir que ambientes externos acessem os nós de uma cena VRML. Embora tenha objetivos similares aos do nó `Script` (processamento de eventos, definindo o comportamento dos objetos do mundo virtual), a EAI funciona de maneira diferente. Ela opera na forma de um applet Java independente, enquanto o nó `Script` opera dentro do browser VRML. Dentre as vantagens da EAI com relação ao nó `Script` estão incluídas uma maior modularidade e simplicidade dos programas, além de maior liberdade para a construção de interfaces complexas para a interação com o mundo VRML.

Para utilizar os recursos da EAI, é necessário criar uma página HTML incluindo a cena VRML e um applet que realiza a interação com a cena. O applet que interage com a cena é um applet Java convencional, que utiliza alguns métodos para realizar a comunicação com a cena.

Recentemente, a EAI e os nós `Script` foram englobados no que está sendo chamado de *Scene Authoring Interface* (SAI) [26], que usa ambos os métodos para a manipulação de objetos de uma cena.

5.7 X3D (*Extensible 3D*)

Apesar de ser um padrão bastante usado, o Web 3D Consortium reconhece a necessidade de aprimoramento da VRML 97. A principal tendência nesse sentido é a X3D. X3D [30] é uma proposta, ainda em fase de elaboração, para uma nova versão de VRML com quatro objetivos principais:

Compatibilidade com VRML 97. A tecnologia X3D continuará permitindo a utilização do conteúdo escrito em VRML 97.

Integração com XML. A idéia é prover as capacidades da VRML 97 usando XML (*Extensible Markup Language*) [31]. XML é chamada "extensível" porque não é uma linguagem de marcação pré-definida, como HTML. Na verdade, ela é uma metalinguagem (linguagem para descrever outras linguagens) que permite

definir novas linguagens de marcação. Na prática, a XML permite que seja definido um novo conjunto de rótulos (*tags)*, adequado à classe de documentos que se deseja representar (no caso da X3D, ambientes tridimensionais). O objetivo da integração entre VRML e XML é, além de ampliar o público usuário da VRML, estar em sintonia com a próxima geração da Web, que deverá ter XML como padrão para transmissão de dados.

Componentização. Pretende-se identificar a funcionalidade crucial da VRML e encapsulá-la em um núcleo simples e extensível, que todas as aplicações devem implementar.

Extensibilidade. O núcleo pode ser expandido para prover novas funcionalidades (adição de componentes). A VRML 97 seria, portanto, apenas uma das possíveis extensões da X3D. Outras extensões propostas são a GeoVRML [28] e a H-Anim (*Humanoid Animation*) [4].

5.8 Usando VRML

A visualização de uma cena VRML se dá por meio de um browser VRML, normalmente um *plug-in* de um browser Web convencional. O browser VRML lê e interpreta o arquivo de descrição do mundo virtual, o "desenha" em uma janela e provê uma interface para que o usuário possa navegar pelo ambiente criado e interagir com objetos dele. Para criar mundos VRML, é necessário um editor de texto (caso se deseja trabalhar diretamente com a linguagem) ou um modelador geométrico que produza o formato VRML.

Atualmente, os browsers VRML mais utilizados são gratuitos, mas proprietários: o Cosmo Player (`http://www.cai.com/cosmo`) e o blaxxun Contact (`http://www.blaxxun.com`). No entanto, existem browsers de código aberto, como o FreeWRL (`http://www-ext.crc.ca/FreeWRL/`), o VRWave (`http://www.iicm.edu/vrwave`) e a iniciativa OVAL (*Open VRML Advancement League* — `http://www.openvrml.org`), um site onde se encontra uma série de projetos de browsers de código aberto.

Com relação aos editores, boa parte dos programas de modelagem mais usados permitem exportação para o formato VRML. Entre os editores de código aberto, merecem destaque o Sced (`http://www.cs.wisc.edu/~schenney/sced/sced.html`) e o Dune (`http://dune.sourceforge.net`).

6 Conclusão

Este artigo mostrou uma visão geral de algumas tecnologias relacionadas à Computação Gráfica, baseadas em software livre e *open source* e em especificações abertas, adotando uma abordagem *bottom-up*, partindo da programação de mais baixo nível

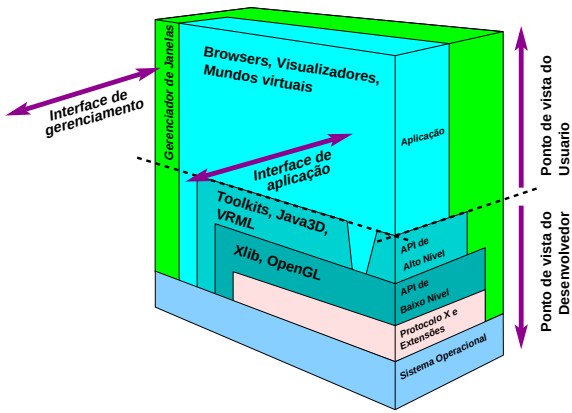

Figura 6: Relacionamento entre as diversas tecnologias discutidas. O usuário final tem acesso às interfaces de aplicação e de gerenciamento (esta pertencente ao gerenciador de janelas). Já o programador pode optar por fazer uso na aplicação de uma ou mais das diversas interfaces de programação disponíveis.

de abstração para a de mais alto nível. À medida que o nível de abstração aumenta, a programação torna-se mais simples, o que leva à possibilidade de implementar aplicações mais sofisticadas. Em contrapartida, só a programação de mais baixo nível provê a flexibilidade necessária em certas situações. Por esta razão, a abordagem passando por todos estes níveis mostra uma visão abrangente das tecnologias e permite mostrar que, apesar de diferentes, elas não são antagônicas, pelo contrário, são complementares, conforme mostra a Figura 6.

O X Window System é um sistema de janelas, que pode ser combinado com o OpenGL para a programação gráfica. Java 3D é construída sobre a linguagem Java e sobre APIs gráficas, como o OpenGL. Além disso, ela oferece recursos mais sofisticados, como estruturação da cena, animação e interação com os usuários. VRML tem funcionalidade similar a Java 3D, mas não é uma linguagem de programação, é apenas um formato para a transmissão de dados 3D pela Internet. Os recursos mais sofisticados são realizados em VRML por programas externos. Tanto Java 3D quanto VRML são esforços no sentido de prover um padrão para a representação e transmissão de conteúdo interativo tridimensional através da Web. A Tabela 1 apresenta uma comparação entre as diversas tecnologias.

O que há de comum entre todas as tecnologias apresentadas (além da relação com a Computação Gráfica), e que motivou a elaboração deste artigo, é a filosofia de desenvolvimento das mesmas. Todas elas são baseadas nas idéias de software livre, de forma mais ampla (como X, OpenGL e VRML, desenvolvidos por consórcios de empresas e grupos acadêmicos) ou menos rigorosa (software proprietário, mas gratuito e de códi-

	X Window	OpenGL	Java3D	VRML
Finalidade	Sistema de janelas com suporte à operação em ambiente distribuído.	Biblioteca para renderização de gráficos 3D.	Biblioteca para construção de aplicações 3D interativas.	Linguagem para modelagem de mundos virtuais.
Modelo de objetos geométricos	Vértices e arestas. Primitivas 2D em coordenadas de tela.	Vértices, arestas e polígonos em coordenadas 3D.	Primitivas básicas, especificação vértice-a-vértice ou importação via *loaders*.	Nós representando primitivas básicas e conjuntos de vértices, arestas e superfícies.
Estruturação da cena	Não há.	Não há	Grafos de cena descrevendo mundos virtuais e propriedades dos objetos.	Estrutura hierárquica de nós descrevendo uma cena.
Fontes de luz	Não possui.	Pontual e *spot*.	Fontes de luz são objetos. Luz ambiente, pontual, direcional e *spot*.	Nós de iluminação direcional, pontual e *spot*.
Textura	Não há suporte direto.	Recursos limitados para mapeamento de imagens sobre objetos.	Texturas 2D e 3D podem ser aplicadas sobre objetos.	Imagens ou filmes podem ser mapeados nas superfícies dos objetos.
Interação	Por programação do tratamento de eventos.	Pobre. Há apenas recursos para selecionar primitivas via mouse.	Nós *behavior* do grafo de cena definem o comportamento dos objetos do mundo virtual.	Sensores e roteamento de eventos, nós *script* e EAI.
Animação	Não há suporte direto, mas podem ser usados *pixmaps*.	Controle em baixo nível, usando *double buffering*.	Classes Alpha e Interpoladores.	Interpoladores e roteamento de eventos, nós *script* e EAI.

Tabela 1: Quadro comparativo entre as tecnologias apresentadas

go aberto, como Java 3D). A adoção destas filosofias, além de aumentar a aceitação dos produtos, garante maior confiabilidade e "sobrevivência" dos mesmos, pois a comunidade de usuários contribui ativamente em seu desenvolvimento, identificando e corrigindo erros e fazendo melhorias. Além disso, a diversidade de interesses dos grupos desenvolvedores impede a criação de monopólios. Como vantagem adicional, é garantido o acesso a uma gigantesca base de conhecimentos utilizados na construção do software. Por todos estes motivos, acreditamos que o uso de software livre seja benéfico, em especial no meio acadêmico, e incentivamos seu uso e desenvolvimento.

Agradecimentos

Este trabalho é apoiado pela FAPESP, CAPES e CNPq.

Referências

[1] K. Brown, and P. Daniel, *Ready-to-Run Java 3D*, Wiley Computer Publishing, 1999.

[2] J. Gettys, and R. Scheifler, *Xlib — C language X interface*, Digital Equipment Corporation/X Consortium, 1994.

[3] *The GIMP Toolkit*, 2000, `http://www.gtk.org/`.

[4] Humanoid Animation Working Group — Web 3D Consortium, *Specification for a Standard Humanoid 1.1*, 1999, `http://ece.uwaterloo.ca:80/~h-anim/spec1.1/`.

[5] O. Jones, *Introduction to the X Window System*, Prentice Hall, 1990.

[6] M. J. Kilgard, *OpenGL and X, Part 1: An Introduction*, 1994, `http://www.sgi.com/software/opengl/glandx/intro/intro.html`.

[7] M. J. Kilgard, *The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3*, 1996, `http://reality.sgi.com/opengl/spec3/spec3.html`.

[8] A. LaMothe, *DirectX-tasy*, `http://www.gamedev.net/reference/articles/article589.asp`.

[9] J. Lindall, *Drag-And-Drop Protocol for the X Window System*, `http://www.newplanetsoftware.com/xdnd/`.

[10] J. Lindall, *JX — C++ Application Framework for the X Window System*, `http://www.newplanetsoftware.com/jx/`.

[11] J. McCormack, P. Asente, R. Swick, and D. Converse, *X Toolkit Intrinsics — C Language Interface*, Digital Equipment Corporation/X Consortium, 1994.

[12] L. P. Magalhães, A. B. Raposo, and F. S. Tamiosso, *VRML 2.0 — An Introductory View by Examples*. `http://www.dca.fee.unicamp.br/~leopini/tut-vrml/vrml-tut.html`.

[13] J. Neither, T. Davis, and W. Mason, OpenGL Architecture Review Board, *OpenGL Programming Guide*, Addison Wesley, 1993.

[14] A. Nye, and T. O'Reilly, *X Toolkit Intrinsics Programming Manual*, O'Reilly & Associates, 1990.

[15] *The OffiX Project*, 1997, `http://leb.net/OffiX/`.

[16] *Open Group Desktop Technologies — Motif*, http://www.opengroup.org/tech/desktop/motif/.

[17] D. Rosenthal, and S. W. Marks, *Inter-Client Communication Conventions Manual*, Sun Microsystems/X Consortium, 1994.

[18] R. Scheifler, and J. Gettys, The X Window System. *ACM Transactions on Graphics*, 5 (2): 79-109, April 1986.

[19] R. Scheifler, *X Window System Protocol*, X Consortium, 1994.

[20] M. Segal, and K. Akeley, *The OpenGL Graphics Interface*, Silicon Graphics Computer Systems, 1994, http://trant.sgi.com/opengl/docs/white_papers/segal.ps.

[21] M. Segal, and K. Akeley, *The Design of the OpenGL Graphics Interface*, Silicon Graphics Computer Systems, 1996, http://trant.sgi.com/opengl/docs/white_papers/design.ps.

[22] H. A. Sowirzal, D. R. Nadeau, and M. Bailey, *Introduction to Programming with Java 3D*, 1998, http://www.sdsc.edu/~nadeau/Courses/SDSCjava3d/.

[23] Sun Microsystems (tutorial preparado por K Computing), *Getting Started with Java 3D^{TM} API*, 1999, http://sun.com/desktop/java3d/collateral.

[24] *Sun's Java 3D API Specification Document*, http://java.sun.com/products/java-media/3D/forDevelopers/j3dguide/j3dTOC.doc.html.

[25] Web 3D Consortium, *The Virtual Reality Modeling Language*, International Standard ISO/IEC DIS 14772-1, 1997, http://www.web3d.org/technicalinfo/specifications/vrml97/index.htm.

[26] Web 3D Consortium, *VRML 200x — Draft Specification*, April 2000, http://www.web3d.org/TaskGroups/x3d/specification.

[27] Web 3D Consortium, *The Virtual Reality Modeling Language — Part 2: External Authoring interface and bindings*, ISO/IEC 14772-2:xxxx, 1999, http://www.web3d.org/WorkingGroups/vrml-eai/.

[28] Web 3D Consortium, *GeoVRML.org*, 2000, http://www.geovrml.org.

[29] *Web 3D Consortium*, http://www.web3d.org.

[30] Web 3D Consortium, *X3D Task Group*, 2000, http://www.web3d.org/TaskGroups/x3d.

[31] The World Wide Web Consortium (W3C), *Extensible Markup Language (XML)*, 2000, http://www.w3.org/XML.