Software Livre 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 curso 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: o X Window System e o 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 por Computador, Realidade Virtual.

Abstract

This course discusses some public domain tecnologies in the fields of Computer Graphics, Virtual Reality and Computer Animation, focusing on programming tools to support the development of interactive graphic systems. The first part presents two tecnologies of lower abstraction level used for the generation of graphic interfaces, geometric drawings and images: X Window System and OpenGL. The second part presents higher abstraction level Internet technologies that can be used for the creation of sophisticated applications, such as virtual worlds and interactive animations. Among these technologies, Java 3D and VRML will be studied. Java 3D is the Java standard library for the creation of programs using tridimensional graphics. VRML is a standard normally used for the modeling and transmission of 3D information and virtual worlds in the Web.

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, tem ocorrido 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

[*] Departamento de Engenharia de Computação e Automação Industrial – Faculdade de Engenharia Elétrica e de Computação – UNICAMP. {alberto, ajcruz, bicho, isla, leopini}@dca.fee.unicamp.br
[†] Centro Universitário de Dourados – UFMS
[‡] Instituto de Informática – UFRGS. kojima@inf.ufrgs.br
[§] Conectiva Informática S.A. – Curitiba, PR. pcpa@conectiva.com.br
[¶] Centro de Pesquisas Meteorológicas – Universidade Federal de Pelotas. casantos@cpmet.ufpel.tche.br

serviços da comunidade para testar o produto e portá-lo para um maior número de plataformas. Este tipo de software é chamado de *open source* (código aberto), embora muitos também os considerem como "livres".

A Computação Gráfica tem apresentado um crescimento significativo nos últimos anos, devido principalmente ao avanço tecnológico, disponibilizando recursos que propiciam o desenvolvimento de aplicações cada vez mais complexas. Com o crescente interesse evidenciado nesta área, a utilização de um padrão gráfico para trabalhar de forma eficiente em diversas plataformas tornou-se uma preocupação. Desta forma, o crescimento de software livre e *open source* nessa área também é uma realidade. Este artigo aborda algumas ferramentas livres (ou *open source*) de programação 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, um padrão aberto para desenvolvimento de aplicações gráficas tridimensionais que pode ser incorporado 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 *open source*. Finalmente, será apresentada a VRML, uma linguagem para a modelagem e transmissão de informação 3D pela Internet.

É importante ressaltar que estas ferramentas se aplicam em contextos diferentes, sendo utilizadas para o 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. O OpenGL é uma biblioteca gráfica para a renderização de imagens estáticas. Java 3D e VRML, por sua vez, estão relacionadas 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) como parte das pesquisas do Projeto Athena e do Sistema Argus, voltados à construção de aplicações multiprocessadas distribuídas [1]. O sistema teve grande aceitação no mercado e por isso, em 1988, o MIT formou junto com fabricantes de software o MIT X Consortium, destinado a prover suporte técnico e administrativo ao desenvolvimento subseqüente do sistema. Em 1993 o consórcio tornou-se uma organização independente chamada X Consortium, Inc., à qual o MIT cedeu os direitos sobre o X Window System. No final de 1996 o X Consortium fechou suas portas e transferiu os direitos para a Open Software Foundation (OSF). Pouco depois, em 1997, a OSF se fundiu à X/Open (detentora dos direitos sobre o UNIX) formando uma organização chamada The Open Group.

O papel do X Consortium sempre foi o de produzir *padrões*, ou seja, especificações de software. O consórcio também distribuía o código-fonte de uma implementação de amostra (SI — *sample implementation*). É possível criar tantas variações quantas se queira a partir da SI, 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. A maioria dos fabricantes de sistemas UNIX comerciais distribui implementações do X baseadas na SI. O Projeto XFree86, por sua vez, desenvolve e mantém a implementação usada em sistemas operacionais livres, como Linux e FreeBSD, e em alguns sistemas comerciais, como OS/2 e QNX.

Ao anunciar a revisão 6.4, em fevereiro de 1998, o Open Group quis adotar uma política de licenciamento que permitia o uso do código da SI para fins não comerciais e de pesquisa, mas exigia para sua comercialização o pagamento de uma taxa, proporcional ao número de cópias a distribuir. A justificativa para tal atitude era a necessidade de custear o desenvolvimento do software. Devido aos protestos da comunidade de usuários de software livre, em especial do Projeto XFree86, em setembro do mesmo ano o Open Group voltou atrás em sua decisão retomando a antiga política de licenciamento, sem restrições a cópia, modificação e distribuição do código.

2.1 Arquitetura do X Window System

X possui arquitetura cliente-servidor na qual o servidor localiza-se na estação de trabalho do usuário e os clientes podem ser executados na própria estação ou remotamente, conectados ao servidor através de uma rede. Os serviços oferecidos pelo servidor incluem acesso transparente ao hardware gráfico (primitivas de desenho 2D e criação/manipulação de janelas), mecanismos de comunicação entre clientes e entrada de dados por dispositivos como mouse e teclado.

O sistema foi concebido para funcionar com quaisquer tipos de display e dispositivos de entrada, variando desde *framebuffers* mapeados por bit com monitor monocromático até coprocessadores gráficos com display colorido. As aplicações

deveriam ser executadas em 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.

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, conforme mostra a Figura 1.

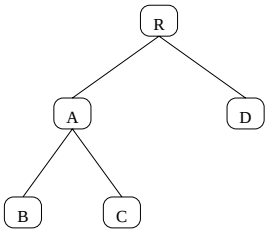

Figura 1: Relacionamento hierárquico entre janelas: A é *ascendente* de B se B está contido na hierarquia de A; B é *irmã* de C se ambas possuem o mesmo ascendente imediato; se A é ascendente de B, obviamente B é *descendente* de A; R é a janela *raiz*, ascendente de todas as demais (há somente uma janela raiz em cada tela); C é uma janela *folha*, pois sua hierarquia é vazia.

Baseado no princípio de que o conceito de melhor interface é cultural, X permite o uso mais de uma interface gráfica com o usuário, propiciando que as aplicações convivam com diversas culturas. Ao invés de um modelo de objetos de alto nível, X oferece um substrato de operações básicas, deixando 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 quando da concepção do sistema. Embora o X Consortium tenha posteriormente padronizado uma extensão para modelagem 3D baseada em PHIGS (*Programmer's Hierarchical Interactive Graphics System*), chamada *PHIGS extension for X* (PEX), ela nunca se tornou popular. A extensão GLX, desenvolvida pela Silicon Graphics (SGI) para dar suporte ao padrão OpenGL (ver Seção 3), acabou se tornando a mais aceita no mercado.

2.2 Interfaces que constituem o sistema de janelas

Um sistema de janelas é constituído por várias interfaces:

Interface de programação. Uma biblioteca de rotinas e tipos para interação com o sistema de janelas. Há dois tipos de interface de programação:

- **De baixo nível.** Provê funções gráficas primitivas. No X, a biblioteca Xlib [2] provê essas funções primitivas, por meio de troca de mensagens com o servidor de display.
- **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) [3]. A maioria dos *toolkits*, entretanto, é construída diretamente sobre a Xlib, sem usar o Xt.

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, embora o uso de um *toolkit* possa levar a aplicação a ter aparência e comportamento semelhantes às outras desenvolvidas com ele.

Interface de gerenciamento. Define a política de controle do posicionamento, tamanho e sobreposição das janelas sobre a área de trabalho. Este papel deve ser desempenhado por 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. Uma conseqüência desta abordagem é que cada usuário pode usar o gerenciador de janela que desejar. Se isso é bom ou não, é uma questão de gosto, e motivo de intermináveis controvérsias sobre qual o melhor gerenciador de 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 a soma das interfaces de aplicação e de gerenciamento [1]. X foi projetado de forma a prover mecanismos para que diferentes políticas de interface pudessem ser implementadas, ou seja, X não é nem possui uma interface com o usuário. A função de prover tal interface é delegada a clientes e bibliotecas externos ao servidor X e às bibliotecas de funções do sistema.

2.3 Conceitos importantes e terminologia

Na terminologia do X Window System alguns termos são usados com significados diferentes daqueles a que estamos acostumados, conforme descrito a seguir:

Display. Instância de um servidor X sendo executado em uma máquina. Geralmente um 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*.

Tela (*screen*). X possui suporte a múltiplos monitores ligados a uma mesma máquina ao mesmo tempo. Cada monitor ligado é denominado *screen*. Múltiplas telas podem conviver formando um display. A revisão 6.4 introduziu a extensão *Xinerama* [4] que permite combinar várias telas formando uma única tela virtual, de modo totalmente transparente às aplicações.

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 um ou mais clientes podem realizar operações de entrada e saída (desenhos, etc.). 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. Esse tipo de janela é conhecido como *janela de aplicação*, ou *shell*.

A hierarquia de janelas permite ao cliente estabelecer políticas de tratamento de eventos. O servidor pode ser instruído a remover uma janela de dentro de outra, colocando-a dentro de uma terceira. Este recurso, de adoção (*reparent*) é utilizado por vários aplicativos, especialmente pelo gerenciador de janelas.

Área de trabalho. O conjunto formado pela janela raiz de uma tela e as janelas criadas sobre ela. O gerenciamento da área de trabalho cabe ao gerenciador de janelas. Somente um gerente pode controlar um display de cada vez, mas não existe a obrigatoriedade de se usar um determinado gerente.

***Widget*.** Um elemento de interface gráfica. Exemplos são botões, caixas de texto, etc. Janelas *top-level* também são consideradas *widgets*.

Servidor. 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. *Drivers* de vídeo fazem parte do servidor X e somente ele tem acesso ao hardware.

Cliente. Uma aplicação que usa um ou mais displays, podendo ter mais de uma conexão aberta simultaneamente com o mesmo display, caso seja necessário. O cliente não precisa ser executado no mesmo computador que o display.

Gerente de Display (*display manager*). Aplicação responsável por fornecer autenticação de acesso a um ou mais displays. O gerente de display padrão do X é o programa XDM.

Sessão (*session*). 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. Uma aplicação especial, chamada *session manager* (SM) pode ser usada para salvar o estado de uma sessão e restaurá-lo posteriormente, mas isso exige que as demais aplicações reconheçam algumas mensagens enviadas pelo SM.

2.4 Protocolo X

A comunicação cliente-servidor se dá por troca de mensagens, usando o *protocolo X* [5], que por sua vez é independente do protocolo de transporte usado (TCP/IP, DECnet, etc.). Para estabelecer a conexão, o cliente deve saber o nome da máquina onde o servidor é executado, mas as funções da biblioteca Xlib permitem que se configure um servidor a ser usado por omissão. Em UNIX, o servidor padrão é informado pela variável de ambiente DISPLAY, que tem seu valor devidamente atribuído pelo XDM durante o processo de login (ver Seção 2.5). 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.

Para o controle de seqüência das mensagens, ao invés de enviar uma confirmação imediatamente após o recebimento de cada requisição, a confirmação vai de "carona" na mensagem seguinte. Este recurso, chamado *piggybacking* [6, p.202], permite o envio de confirmações, embora com retardo, e economiza tráfego de rede. Os pacotes transmitidos entre servidor possuem um cabeçalho de tamanho também variável, com uma identificação única de tipo, seguidos de outros dados, dependentes do tipo de mensagem:

Request. Solicitação de serviços do cliente ao servidor. Têm um cabeçalho de tamanho fixo, seguido de uma estrutura dependente do tipo. Existem dois tipos de requisição:

Unidirecionais (*one way trip*). São aquelas que não necessitam esperar por uma confirmação imediata de recebimento. A confirmação é enviada usando o mecanismo *piggybacking*, já mencionado. Esta forma de tratar as requisições contribui para o desempenho do sistema, pois várias delas podem ser armazenadas no *buffer* de transmissão e descarregadas em bloco, otimizando o tráfego na rede.

De ida e volta (*round trip*). A maioria das requisições é implementada da forma anterior, mas algumas delas exigem uma resposta por parte do servidor. É o caso típico da criação de uma janela, em que deve ser retornado o seu número ao cliente, pois ele será usado nas operações posteriores. Chamadas de ida e volta descarregam o *buffer* de saída e são enviadas imediatamente. Como são bloqueantes, seu uso deve ser evitado tanto quanto possível para não degradar o desempenho do sistema.

Reply. Respostas a requisições. Têm um cabeçalho de tamanho fixo, seguido de uma estrutura dependente da requisição. Teoricamente uma resposta poderia ter tamanho máximo de 16GB, mas na prática isso não acontece.

Event. Eventos, tais como alteração da geometria de uma janela, pressionamento de uma tecla ou movimentação do mouse. Têm tamanho fixo (32 octetos) e possuem no seu início um identificador de tipo e de número de seqüência, que tem o mesmo uso que nas respostas.

O protocolo é assíncrono, mas garante que as mensagens serão entregues ao servidor na mesma seqüência em que foram geradas pelo cliente, e vice-versa. Como mostra a Figura 2, o cliente pode receber eventos de muitos tipos diferentes antes da resposta a uma requisição. As funções de tratamento de eventos da Xlib permitem consultar a fila de entrada e retirar eventos segundo vários critérios, deixando os demais enfileirados.

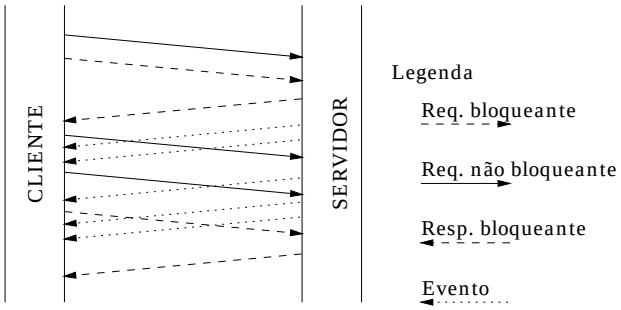

Figura 2: Fluxo de mensagens entre servidor e cliente. Como a geração de eventos é assíncrona, eventos provenientes do servidor podem chegar ao cliente antes da resposta à última requisição enviada.

2.5 Autenticação de usuários

Como X é um ambiente de trabalho distribuído, os métodos normais de autenticação de usuários existentes nos sistemas operacionais nem sempre podem ser aplicados. O controle de acesso ao display pode ser feito por diversos métodos descritos na página de manual `Xsecurity`, que faz parte do pacote X (alguns métodos não são disponíveis em todas as implementações).

O programa XDM usa o X Display Manager Control Protocol (XDMCP) para prover serviços similares àqueles oferecidos pelo login do UNIX, aguardando conexões na porta 177 com transporte UDP, ou via *broadcast*. A Figura 3 ilustra o processo de autenticação.

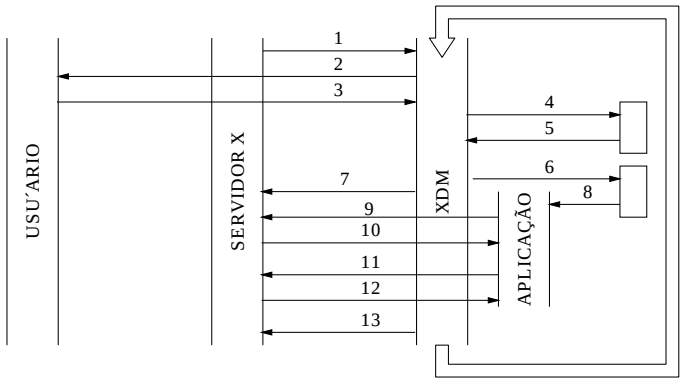

Figura 3: Autenticação via XDMCP. O servidor X é configurado para, ao ser carregado, fazer uma chamada ao XDM via XDMCP (1). XDM, por sua vez apresenta uma tela de login ao usuário (2). Após receber o nome do usuário e a senha (3), XDM gera uma chave de autenticação, grava-a no arquivo `.Xauthority` no diretório do usuário (6) envia uma cópia ao servidor (7). Depois disso XDM executa um programa correspondente à sessão de trabalho do usuário, que normalmente ativa um gerenciador de janelas. Enquanto durar a sessão, cada cliente que quiser usar o servidor deverá ler a chave (8) e conectar-se à porta de comunicação do servidor (9), recebendo um *desafio* (10). O cliente usará a chave para formar a resposta ao desafio (11), à qual o servidor responderá, se estiver correta, com uma autorização de acesso (12). Quando a sessão termina, XDM reinicializa o servidor (13) e recomeça todo o processo.

Durante o período da sessão do usuário, novas conexões ao display deverão ser autenticadas usando a chave gerada pelo XDM, cujo valor depende do protocolo escolhido. A função `XOpenDisplay` da biblioteca Xlib usará o conteúdo do arquivo .`Xauthority`, no diretório *home* do usuário, para construir esta chave.

Os métodos de autenticação suportados são descritos na página de manual `Xsecurity`[1]. Alguns deles têm a desvantagem de necessitar de um servidor para autenticação, mas permitem centralizar o cadastramento de usuários e aumentam a segurança em ambiente de rede.

2.6 Recursos do servidor

Recursos (*resources*) são os objetos (na falta de um termo mais adequado) que residem na memória do servidor mas que são usados pelos clientes. Eles 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, depois disso o servidor destrói todos os recursos a ela associados. Recursos podem ser dos seguintes tipos:

Janelas. Áreas para desenho na tela, normalmente de formato retangular. A destruição de uma janela implica na destruição de todas as que nela estiverem contidas. A janela raiz contém todas as demais e não pode ser destruída. O gerenciador de janelas assume o controle da janela raiz e é informado pelo servidor de cada criação/remoção de novas janelas.

Mapas de pixels (*pixmaps*). São espécies de janelas ocultas. O cliente pode realizar sobre um *pixmap* 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. *Pixmaps* e janelas são chamados de *drawables*, porque se pode desenhar neles.

[1] Em sistemas UNIX, essa página de manual pode ser lida usando o comando `man Xsecurity`.

Contextos gráficos (GCs — *graphic contexts*). Um GC guarda um conjunto de atributos gráficos como cores de frente e fundo, padrões para desenho de linhas, preenchimento de polígonos, fontes de caracteres, etc. Uma requisição de desenho em uma janela tem como um dos parâmetros o identificador do GC que deve ser usado. GCs são globais para todo um display e podem ser usados por qualquer cliente em operações sobre qualquer janela. GCs são um dos mecanismos mais importantes do X para melhorar o desempenho das operações.

Fontes. Descrevem o tamanho e aparência de caracteres. X suporta o uso de alfabetos e símbolos internacionais baseados nas especificações ISO-8859 e ISO-10646 (Unicode). O servidor carrega para sua memória os padrões de desenho dos caracteres de uma fonte à medida que os clientes solicitam. Servidores executados em máquinas sem disco (terminais X) podem carregá-las de um servidor de fontes.

Mapas de cores (*colormaps*). O servidor provê um mapa de cores (paleta) global para ser usado por todos os clientes, no qual cada cor é referenciada por um índice. Cada janela tem associado a ela um mapa de cores, que normalmente é o mesmo da janela raiz. Caso o número total de cores suportadas pelo servidor simultaneamente seja pequeno, é possível que todos os elementos do mapa de cores acabem sendo usados. Aplicações que fazem uso intenso de cores podem criar um novo mapa de cores no servidor para seu uso privado, associando-o às janelas criadas.

Cursores. Descrevem a aparência do cursor associado ao dispositivo apontador (normalmente o mouse). 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.

A criação de recursos é feita por requisições *one way trip*. Para que isso seja possível o servidor passa ao cliente, na abertura da conexão, um bloco de identificadores não usados, que vão sendo consumidos à medida que é necessário. As funções tratamento de recursos da Xlib gerenciam automaticamente esse bloco de identificadores, de modo que o programador não precisa se preocupar com eles.

2.7 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, usando a função `XSendEvent` da Xlib. Clientes que desejam receber eventos enviados por outros deverão fazer uma requisição ao servidor, via `XSelectInput`, e a partir de então o servidor os enviará. Um cliente não precisa solicitar eventos para uma janela criada por ele, mas pode informar que não os deseja, conforme descrito na Seção 2.10.1.

As convenções para interação entre aplicações são descritas detalhadamente no *Inter-Client Conventions Manual* (ICCCM) [7]. Faremos aqui apenas uma breve descrição de conceitos úteis ao entendimento do resto do texto.

2.7.1 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 *átomos*, *propriedades* e *seleções*.

Propriedades (*properties*). São blocos de dados que um cliente "pendura" em uma janela, contendo quantidades arbitrárias de informação de um tipo escolhido. Propriedades são mantidas na memória do servidor, junto com as janelas, e podem ser criadas, destruídas, lidas e gravadas. A gravação/leitura do conteúdo de uma propriedade é semelhante ao que se faria em um arquivo.

Aplicações que compartilham dados via propriedades devem solicitar ao servidor, via `XSelectInput`, a notificação de alterações nas mesmas. A partir de então, sempre que houver alteração em uma propriedade o cliente será notificado por um evento do tipo `PropertyNotify`.

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 de 32 bits. Átomos também são usados para especificar tipos de dados e nomes de propriedades e seleções. Existe um conjunto de átomos padrão, pré-registrados no servidor, cujos rótulos têm valores conhecidos por antecipação (declarados no arquivo `Xatom.h`).

Áreas de recorte (*cut buffers*). A janela raiz de cada tela possui um grupo de 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. Um exemplo de aplicação das áreas de recorte é o emulador de terminal `xterm`, que permite marcar uma parte do texto em uma janela e colá-la em outra.

Seleções (*selections*). São mecanismos mais sofisticados do que os *cut buffers* para compartilhamento de dados. 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. Seu uso exige que a aplicação a quem os dados foram solicitados saiba como convertê-los, se necessário, para o formato que o solicitante pede. Descrições detalhadas deste mecanismo são dadas por Packard [8] e por Nye e O'Reilly [9, cap. 11].

2.8 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 usando os mecanismos disponíveis. Em decorrência disso há muitos protocolos DnD para X, dentre os quais destacam-se os três mais populares:

Motif. O *toolkit* Motif [10] define um protocolo DnD baseado em mensagens entre clientes, properties e selections. Como exemplos de aplicações que usam esse protocolo, podemos citar o Netscape Navigator (versão para UNIX) e os programas integrantes do CDE (*Common Desktop Environment*).

Offix. Offix [11] foi um pacote de aplicativos desenvolvido por César Crusius, que mais tarde evoluiu para um *toolkit* gráfico completo. O pacote incluía um protocolo DnD, implementado em uma biblioteca de livre distribuição que é usada por um grande número de aplicações.

Xdnd. É um protocolo recente, criado por John Lindall para o *toolkit* JX [12] 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 Window que suportem *drag-and-drop*. Uma comparação entre os diversos protocolos foi feita por Lindall [13].

2.9 Gerenciamento de janelas

O uso de um gerenciador de janelas não é obrigatório, mas é muito útil porque ele implementa as políticas de atribuição do foco (a janela em foco recebe os eventos de teclado) e redimensionamento e sobreposição de janelas. O gerenciador de janelas faz isso com base em um conjunto convenções, usando mecanismos como troca de mensagens e eventos.

Há um conjunto padrão de propriedades acopladas a cada janela, definido no ICCCM, que funcionam como dicas (*hints*) para o gerenciador de janelas, contendo informações relativas à forma como a janela deve ser tratada. Exemplos dessas propriedades são `WM_NORMAL_HINTS`, que fornece informações sobre a geometria da janela, `WM_NAME`, que contém um texto a ser mostrado ao usuário como sendo o título da janela, e `WM_HINTS`, que fornece informações como um *pixmap* a ser usado em representações iconizadas da janela. O utilitário `xprop` pode ser usado para visualizar o conjunto de propriedades definidas para uma janela. A Figura 4 mostra como isso ocorre.

2.10 Técnicas de programação

2.10.1 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

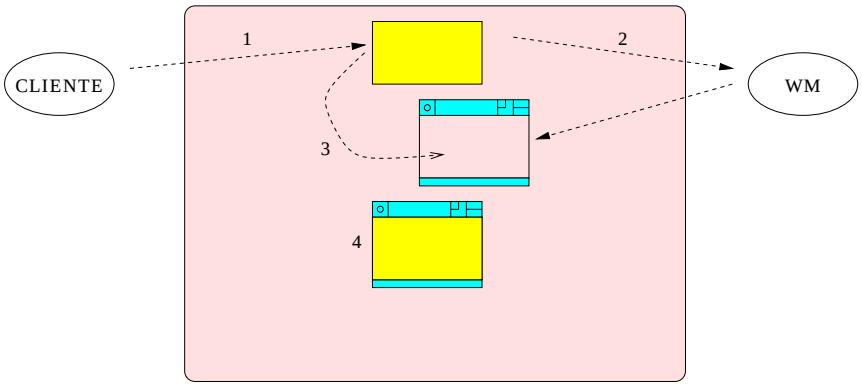

Figura 4: Criação de uma janela *top-level*. O cliente cria a janela dentro da janela raiz e requisita o *mapping* (apresentação na tela de uma janela e todas as suas subjanelas, o que as torna visíveis) ao servidor X (1). O gerenciador de janelas recebe uma notificação (2), verifica as propriedades 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).

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. Existem trinta tipos de eventos declarados na estrutura `XEvent` no arquivo `Xlib.h`.

Nem todos os eventos podem ser interessantes para o cliente. O pressionamento de um botão do mouse sobre uma janela usada apenas para exibição de um texto, por exemplo, não precisa ser informado se o cliente não precisa reagir a isso. 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. Um exemplo é o evento *Expose*: quando uma janela é redimensionada o servidor envia um evento desse tipo para cada modificação do tamanho ao cliente a quem ela pertence; clientes capazes de desenhar apenas a parte exposta terão uso para a informação, mas aplicações simples que reagem a *exposes* redesenhando toda a janela provocarão um efeito de cintilação. A maneira correta de tratar o problema é descartar todos os eventos de uma cadeia de *exposes* e reagir apenas ao último [14]. O trecho de programa a seguir demonstra o uso dessa 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, ja que para um redimensionamento da janela o
         ** servidor X envia varios 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;
```

2.10.2 Tratamento de erros

A notificação de recebimento de requisições por *piggybacking* mencionada anteriormente tem um grave inconveniente: o que fazer quando uma determinada requisição não pode ser atendida por algum motivo? Para resolver o problema existem os eventos do tipo `XErrorEvent`. O servidor sempre envia os eventos de erro, numa política "depois não diga que eu não avisei", mas cabe ao cliente decidir que atitude tomar ao recebê-los. Tratar o erro geralmente consiste em identificar o seu código e reagir condizentemente, o que pode ser bastante difícil. A maioria dos códigos de erro diz respeito ao envio de requisições com parâmetros inválidos, o que significa que há um erro na aplicação.

A Xlib possui um tratador de erros simples, que normalmente mostra uma mensagem na saída de erro padrão e encerra a aplicação abruptamente, mas o cliente pode instalar o seu próprio tratador de erros. O X Toolkit tem um tratador mais sofisticado que permite inclusive que mensagens de erro sejam internacionalizadas (geradas de acordo com o idioma da aplicação).

Um dos fatores que dificulta a detecção de erros é a geração assíncrona de eventos, que pode levar à situação mostrada na Figura 2. É trabalhoso depurar uma cadeia de eventos desse tipo. Para facilitar o trabalho, o cliente pode estabelecer a comunicação em modo síncrono usando a função `XSynchronize` provida pela Xlib. O *buffer* de saída será então descarregado e todas as requisições aguardarão a resposta do servidor. O modo síncrono só deve ser usado para depuração, pois ele implica em queda de desempenho pelo aumento do número de pacotes que trafegam na rede e pela espera do atendimento de cada requisição.

2.10.3 Aplicações *multithread*

Desde a versão 11 revisão 6.1 do X Window System (X11R6.1) a Xlib e o X Toolkit possuem suporte à reentrância, permitindo que múltiplas *threads* façam chamadas concorrentemente às suas funções. Três funções são providas com esta finalidade pela Xlib:

`XInitThreads` inicializa o suporte a *threads*. Esta função deve ser a primeira chamada à Xlib em programas *multithread* e deve ser completada antes que se faça qualquer outra. É necessário chamá-la somente se múltiplas *threads* devem usar a Xlib concorrentemente. Se todas chamadas são protegidas por outro mecanismo de exclusão mútua a inicialização é desnecessária. É importante também observar que a função existe mesmo naquelas plataformas em que a Xlib não suporta *threads*, mas neste caso ela sempre retorna `False`.

`XLockDisplay` bloqueia todas as outras *threads* no uso do display especificado. Outras *threads* que tentem fazê-lo ficarão bloqueadas até que ele seja liberado. Chamadas aninhadas a `XLockDisplay` podem ser feitas, mas o display não será liberado enquanto `XUnlockDisplay` não tiver sido chamada tantas vezes quanto `XLockDisplay` foi.

`XUnlockDisplay` permite às outras *threads* o acesso ao display especificado. *Threads* que tiverem sido bloqueadas à espera do display serão liberadas para continuar.

Chamadas à Xlib que retornam ponteiros para estruturas sempre alocam dados dinamicamente, que devem ser descartados posteriormente com a função `XFree`, e nunca apontam para uma estrutura alocada estaticamente. Deste modo, diminui-se o perigo de haver dependência de dados entre *threads*. O X Toolkit também tem suporte a *threads*, por meio das funções `XtToolkitThreadInitialize`, `XtAppLock`, `XtAppUnlock`, `XtProcessLock` e `XtProcessUnlock`.

Um cuidado adicional que se deve ter em aplicações *multithread* é com relação ao término: se uma *thread* responder a um evento qualquer chamando a função `exit` todas as outras *threads* terminarão também. Um exemplo de como tratar de modo adequado o problema pode ser visto no programa `ico`, incluído no pacote do XFree86.

2.10.4 Construção de aplicações por composição

A operação de *reparent* e o compartilhamento de recursos entre clientes podem ser usados para construir aplicações que se apresentam ao usuário como um só programa, mas que na verdade são vários programas sendo executados dentro de uma mesma janela. Apresentaremos a seguir dois exemplos desse tipo de aplicação.

Ghostview. Ghostscript [15] é um interpretador da linguagem de definição de página PostScript®, capaz apenas de apresentar um documento PS em uma janela X, sem nenhum tipo de interface visual, apenas de comandos. Tim Theisen desenvolveu Ghostview, um *front-end* na qual a linguagem de comandos foi substituída por uma linguagem visual.

FvwmButtons. O gerenciador de janelas Fvwm [16] possui um módulo FvwmButtons (barra de botões) capaz de fazer o que o manual do programa chama de *swallow* (engolir). Ele pode ser configurado para executar uma aplicação e mover a janela desta para dentro da barra de botões. FvwmButtons pode opcionalmente engolir uma aplicação cuja janela já esteja na área de trabalho quando do seu carregamento. A Figura 5 mostra um exemplo.

Figura 5: Um FvwmButtons dentro do qual estão, da esquerda para a direita, as janelas dos programas `Xclock` e `Xload`, um botão contendo um *pixmap* e a janela do módulo `FvwmPager`.

O recurso de *reparent* é muito poderoso, mas não é suficiente como mecanismo genérico de composição de aplicações. Para tanto seria necessário definir um protocolo de comunicação entre componentes, o que não existe como padrão no X. Embora o X Toolkit defina um modelo de componentes baseado em *widgets*, ele é omisso no que tange a múltiplas aplicações operando em conjunto. Essa é uma deficiência do X [17].

As soluções encontradas para o problema variam de aplicação para aplicação. Ghostview e Ghostscript definem um protocolo baseado em propriedades e FvwmButtons "engana" as aplicações passando mensagens que seriam enviadas pelo gerenciador de janelas. O software GNOME [18], por outro lado, adota um modelo de componentes baseado em CORBA.

2.11 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[2]. Há dezenas de *toolkits* diferentes, que vão de simples conjuntos de *widgets* usadas em uma única aplicação até pacotes sofisticados e cheios de recursos.

2.11.1 Xlib

Xlib [2] é a API (*Application Programmer's Interface*) de mais baixo nível disponível para se programar aplicações para X. Ela é um padrão pelo X Consortium, implementado como uma biblioteca em linguagem C, que oferece funções com correspondência direta, ou quase, com o protocolo X.

Todos os *toolkits* conhecidos usam a Xlib na sua implementação, mas oferecem diversos níveis de abstração e funções de mais alto nível que permitem, além da construção de interfaces com o usuário, tratar de outras necessidades da aplicação, tais como configuração, tratamento de erros e acesso a serviços do sistema operacional.

Apesar da Xlib poder ser utilizada no desenvolvimento de aplicações, isso não é aconselhável para programas complexos ou que precisem de interface com o usuário elaboradas, pois além de ser muito trabalhoso pode resultar em aplicações com interfaces inconsistentes umas com as das outras. Por outro lado, programas que realizam operações "de baixo nível", tais como gerenciamento de janelas, desenho e processamento de imagens, terão obrigatoriamente que usar as funções da Xlib. Uma abordagem bastante comum em aplicações complexas é usar um *toolkit* para construir a interface e funções da Xlib para operações de baixo nível. Neste caso deve-se tomar cuidado para que as chamadas feitas à Xlib não interfiram com a política implementada pelo *toolkit*.

[2]Em meados da década de 80 houve uma "guerra de interfaces", com grandes fabricantes tentando estabelecer um padrão para a indústria. A Open Software Foundation venceu a guerra e Motif tornou-se a interface padrão nos sistemas UNIX, mas, por ter uma implementação proprietária, não atraiu a simpatia de muitos desenvolvedores de software livre. Em maio de 2000 o Open Group liberou o código-fonte do Motif, autorizando sua utilização gratuitamente em sistemas operacionais *open source*.

2.11.2 X Toolkit (Xt)

Xt, ou X Toolkit Intrinsics [3] foi criado com a intenção de prover recursos básicos para o desenvolvimento de outros *toolkits*. 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 (os *intrinsics*) e que um *toolkit* complementar forneça os *widgets*. Xt define classes básicas de objetos e *widgets* que por si só não implementam nenhuma política de interface. Elas existem para que se criem subclasses, estas sim, determinantes de uma política. O restante do Xt consiste de funções para:

- inicialização do *toolkit* e da aplicação;
- criação, manipulação e destruição de *widgets*;
- adição e remoção de *callbacks* e *actions* (resposta a eventos) a *widgets*;
- gerenciamento de *widget resources*;
- tratamento de eventos, *timeouts* e sinais;
- controle da execução do programa.

Xt implementa um modelo de Programação Orientada a Objetos (POO), mas como é implementado em C, uma linguagem exclusivamente procedural, isso é feito por meio de uma série de convenções e técnicas de programação. Por esse motivo a programação para Xt é difícil a princípio: é necessário que o programador tenha um bom domínio da linguagem de programação C e desenvolva uma rigorosa disciplina de trabalho.

Como vantagem do modelo adotado, Xt e os *toolkits* nele baseados são extremamente flexíveis e possuem recursos únicos dentre os demais. Uma aplicação Xt pode ter não só sua aparência mas também seu comportamento (até certo ponto, pelo menos) modificados pelo usuário sem que seja necessário alterar uma única linha do código-fonte do programa, e isso pode ser feito com um programa que esteja sendo executado. O mecanismo por meio do qual isso é feito chama-se *widget resources* e é uma das características mais poderosas do Xt.

2.11.3 Athena (Xaw)

O "X Athena Widget Set" (Xaw) [19] foi criado durante as pesquisas do Projeto Athena (daí o nome) como um exemplo de uso do Xt. Xaw nunca foi um *toolkit* completo, para uso profissional, mas acabou sendo muito utilizado porque era distribuído junto com a SI do X, apesar de nunca ter sido adotado como um padrão pelo X Consortium. Como uma conseqüência de sua origem como exercício acadêmico, Xaw carece de muitos recursos necessários a um *toolkit* completo, tais como documentação detalhada e uma grande coleção de *widgets*, mas suas duas principais deficiências são:

1. Implementa uma política muito simplista de atribuição do foco de entrada (*input focus*) para os eventos de teclado. Não há como identificar qual dos *widgets* receberá os eventos correspondentes à digitação (normalmente é o *widget* que está sob o cursor do mouse).
2. Como conseqüência do problema anterior, não há como percorrer a hierarquia de *widgets* usando exclusivamente o teclado (recurso conhecido como *keyboard traversal*).
3. Os *widgets* não se diferenciam muito visualmente. Não há como distinguir uma caixa de entrada de texto de um botão, por exemplo, a não ser tentando digitar nela.

Como alternativa para resolver esses problemas, ou pelo menos o problema da aparência, surgiram diversas variantes de Xaw: Xaw3D, Xaw95, neXtaw, XawM, etc. Algumas aplicações também tentam resolver os problemas por meio de truques de programação, mas como regra geral podemos dizer que Xaw não se presta para o desenvolvimento de aplicações com boa usabilidade; para isso existem outros *toolkits*, sendo Motif considerado o mais "profissional". Para os fins deste curso, porém, Xaw será o suficiente para evidenciar a diferença entre usar puramente Xlib e usar um *toolkit* (ver Seção 2.12.2).

2.12 Prática de programação para X

Devido às restrições existentes, apenas alguns conceitos básicos serão tratados neste texto. Ao invés de descrever longamente cada uma das funções aqui mencionadas, optamos por apresentar apenas uma sinopse do seu uso. Para cada uma delas existe uma página de manual disponível, cuja leitura é recomendada para uma explicação detalhada de cada argumento da função e do seu resultado. Em sistemas UNIX, essas páginas de manual podem ser lidas com o programa `man`. Para ler o manual da função `XCreateWindow`, por exemplo, usa-se o comando

```
man XCreateWindow
```

2.12.1 Estrutura de um programa

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

1. 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. Tudo isto é feito internamente pelo *toolkit*, caso seja utilizado um.

2. Montagem da interface gráfica

   Após aberta a conexão com o servidor, o programa pode criar e montar sua interface (janelas, menus, botões etc.) utilizando os *widgets* oferecidos pelo *toolkit*, ou manualmente, usando-se as primitivas do Xlib.

3. Laço de tratamento de eventos.

   Finalmente, o programa entra em um laço, onde fica esperando por eventos vindos do servidor, como pressionamento de teclas ou botões do mouse, movimento do mouse ou requisição do servidor para redesenhar o conteúdo de suas janelas.

   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.

2.12.2 Versão X do "Hello, world!"

Para ilustrar os conceitos apresentados, usamos a versão X do clássico "Hello, world". Obviamente esta versão faz muito mais do que simplesmente escrever uma mensagem na tela. Para fins de ilustração, apresentaremos duas versões do programa, uma usando exclusivamente Xlib e outra usando Xt/Xaw.

Versão Xlib

O arquivo `hello-Xlib.c`, mostrado a seguir, é um exemplo de programa escrito em C usando exclusivamente as funções da Xlib.

```
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <X11/Xutil.h>
#include <X11/keysym.h>
#include <X11/cursorfont.h>

#include <stdio.h>

/* Esta função é chamada cada vez que um botão do mouse é pressionado na janela */
void button_callback(XEvent ev, GC gc)
{
  XDrawImageString(ev.xany.display, ev.xany.window, gc,
                   ev.xbutton.x, ev.xbutton.y, "Hello!", strlen("Hello!"));
```

```
}

/* Redesenha a janela */
void redraw_callback(XEvent ev, GC gc)
{
  XWindowAttributes attr;

  (void) XGetWindowAttributes(ev.xany.display, ev.xany.window, &attr);
  XDrawLine(ev.xany.display, ev.xany.window, gc, 0, 0, attr.width - 1, attr.height -
1);
  XDrawLine(ev.xany.display, ev.xany.window, gc, 0, attr.height - 1, attr.width - 1, 0);
}

/* Função chamada cada vez que a janela é reconfigurada (tamanho, etc.) */
void configure_callback(XEvent ev, GC gc)
{ }

int main(int argc, char *argv[])
{
  Display *display;             /* identifica o display a usar */
  int screen;                   /* número da tela do display */
  Window root, win;             /* a janela raiz e a da nossa aplicação */
  GC gc;                        /* contexto gráfico (atributos de desenho ) */
  XEvent ev, nextev;            /* eventos */
  KeySym key;                   /* símbolo da tecla, se evento de teclado */
  XSetWindowAttributes xswa;    /* atributos da janela */
  XClassHint classhint;         /* classe e nome da aplicação */
  Atom wm_delete_window;        /* mensagem enviada pelo window manager */
  Atom wm_protocols;            /* tipo da mensagem */
  unsigned long foreground,     /* cor de frente */
                background;     /* cor de fundo */
  char text[10];                /* buffer de texto */
  int i, done;                  /* variáveis auxiliares */


/*****************   Inicialização    ******************************/

  display = XOpenDisplay(NULL);                 /* conecta-se ao servidor */
  if (display == NULL) {
    fprintf(stderr, "XOpenDisplay falhou!\n");
    exit(1);
  }
  screen = DefaultScreen(display);              /* descobre a tela padrão, */
  root = DefaultRootWindow(display);            /* a janela raiz, */
  background = WhitePixel(display, screen);     /* e cores de frente e fundo */
  foreground = BlackPixel(display, screen);     /* da tela */

  xswa.background_pixel = background;           /* define os atributos da */
  xswa.border_pixel = foreground;               /* janela da aplicação */
  xswa.backing_store = WhenMapped;              /* usa backing-store */
  xswa.event_mask = ButtonPressMask | KeyPressMask |    /* tipos de eventos */
                    ExposureMask | StructureNotifyMask; /* que interessam */
```

```
  /* Cria a janela */
  win = XCreateWindow(
    display,                            /* display */
    root,                               /* janela pai */
    200, 300,                           /* posição (x,y) */
    350, 250, 5,                        /* largura, altura, largura da borda */
    CopyFromParent,                     /* profundidade */
    InputOutput,                        /* classe */
    CopyFromParent,                     /* visual */
    CWBackPixel | CWBorderPixel |       /* atributos a estabelecer */
    CWBackingStore | CWEventMask,
    &xswa);

  /* "Hints" para o window manager */
  classhint.res_name = "hello";
  classhint.res_class = "Hello";
  XmbSetWMProperties(
    display, win,                       /* display, janela */
    "Hello, cruel world!",              /* título da janela */
    "Hello",                            /* título do ícone */
    argv, argc, None, None, &classhint);

  /* WM_DELETE_WINDOW é a mensagem do window manager, mandando terminar */
  wm_delete_window = XInternAtom (display, "WM_DELETE_WINDOW", False);
  wm_protocols = XInternAtom (display, "WM_PROTOCOLS", False);
  (void) XSetWMProtocols (display, win, &wm_delete_window, 1);

  /* Define o tormato do cursor do mouse usado na janela */
  XDefineCursor(display, win, XCreateFontCursor(display, XC_man));
  gc = XCreateGC(display, win, 0, 0);
  XSetBackground(display, gc, background);
  XSetForeground(display, gc, foreground);

  XMapRaised(display, win);             /* torna a janela visível no display */

/*****************   Laço de Tratamento de Eventos    ******************************/

  done = False;
  while (!done) {
#ifdef TEST_PENDING
    if (XPending(display) > 0)
      XNextEvent(display, &ev); /* obtém o próximo evento */
    else
      fprintf(stderr, "Nenhum evento na fila\n");
#else
    XNextEvent(display, &ev);   /* obtém o próximo evento */
#endif
    switch (ev.type) {
    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.  Verifica o conteúdo da fila de eventos antes de
       * ler porque XNextEvent é bloqueante */
```

```
      while (
        (XEventsQueued(display, QueuedAfterReading) > 0) &&
        (XPeekEvent(display, &nextev), (nextev.type == Expose))
        ) {
        XNextEvent(display, &ev);
      }
      redraw_callback(ev, gc);         /* redesenha a tela */
      break;
    case MappingNotify:
      XRefreshKeyboardMapping(&ev.xmapping);
      break;
    case ConfigureNotify:
      /* Reconfigurações da janela (alteração de tamanho, etc).  Se
       * ConfigureNotify é seguido de Expose, a janela foi redimensionada,
       * caso contrário foi apenas movida.  No primeiro caso é necessário
       * recalcular a geometria do conteúdo da janela. */
      if (
        (XEventsQueued(display, QueuedAfterReading) > 0) &&
        (XPeekEvent(display, &nextev), (nextev.type == Expose))
        ) {
        configure_callback(ev, gc);     /* na verdade, não faz nada :-) */
        }
      break;
    case ButtonPress:                   /* botão do mouse pressionado */
      button_callback(ev, gc);
      break;
    case KeyPress:                      /* tecla pressionada, qual delas? */
      XLookupString(&ev.xkey, text, 10, &key, NULL);
      switch (key) {
        case XK_Q:                      /* 'Q' ou 'q': fim */
        case XK_q:
          done = True;
          break;
      }                                 /* switch(ev.xkey) */
      break;                            /* case KeyPress */
    case ClientMessage:                 /* mensagem de outro cliente */
      if (ev.xclient.message_type == wm_protocols &&
          ev.xclient.data.l[0] == wm_delete_window)
        done = True;            /* window manager mandou fechar a janela */
      else
        XBell (display, 0);     /* mensagem desconhecida: faz "bip" */
      break;                    /* case ClientMessage */
    default:
      fprintf(stderr, "Tipo de evento desconhecido\n");
    }                           /* switch(ev.type) */
  }                             /* while(!done) */

/*****************   Término    ******************************/

  XFreeGC(display, gc);
  XDestroyWindow(display, win);
  XCloseDisplay(display);
  return 0;
```

```
}
```

Para compilar este programa, em Linux ou FreeBSD, são usados os seguintes comandos:

```
cc -c -I/usr/X11R6/include hello-Xlib.c
cc -L/usr/X11R6/lib hello-Xlib.o -o hello-Xlib -lX11
```

Versão Xaw

O arquivo `hello-Xaw.c`, mostrado a seguir, é um exemplo que usa o X Toolkit e Xaw. Observe-se a simplificação das três etapas (inicialização, criação da interface e laço de eventos). Como Xaw não provê um tratamento padrão para a mensagem `WM_DELETE_WINDOW` gerada pelo gerenciador de janelas, é preciso incluir o código necessário, à semelhança do que foi feito na versão Xlib.

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

#include <X11/Xaw/Command.h>

static Atom wm_delete_window;
static Atom wm_protocols;

static void WMProtocols(Widget w, XEvent *ev, String *params, Cardinal *n)
{
    if (ev->type == ClientMessage &&
        ev->xclient.message_type == wm_protocols &&
        ev->xclient.data.l[0] == wm_delete_window) {
        XtAppSetExitFlag(XtWidgetToApplicationContext(w));
    }
}

static void QuitCallback(Widget w, XtPointer cdata, XtPointer cbs)
{
    XtAppSetExitFlag(XtWidgetToApplicationContext(w));
}

XtActionsRec actions[] = {
    {"WMProtocols", WMProtocols}
};

String fallback[] = {
    "*button.background: rgb:c0/c0/c0",
    "*button.font: -b&h-lucida-bold-r-normal-*-*-240-*-*-p-*-*-*",
    "*button.label: Hello, world!",
    NULL
};

int main(int argc, char *argv[])
{
    XtAppContext app;
    Widget top, button;

    XtSetLanguageProc(NULL, (XtLanguageProc)NULL, NULL);
```

```
    top = XtVaAppInitialize(
            &app,               /* contexto da aplicação */
            "XHello",           /* classe da aplicação */
            NULL, 0,            /* lista de opções da linha de comando */
            &argc, argv,        /* argumentos da linha de comando */
            fallback,           /* para arquivo app-defaults perdido */
            NULL);              /* termina lista varargs */

    XtAppAddActions(app, actions, XtNumber(actions));
    XtOverrideTranslations(top,
           XtParseTranslationTable ("<Message>WM_PROTOCOLS: WMProtocols()"));

    button = XtVaCreateManagedWidget(
            "button",           /* nome arbitrário para o widget */
            commandWidgetClass, /* classe de widget de Label.h */
            top,                /* widget pai */
            NULL);              /* termina lista varargs */
    XtAddCallback(button, XtNcallback, QuitCallback, NULL);

    XtRealizeWidget(top);
    wm_delete_window = XInternAtom(XtDisplay(top), "WM_DELETE_WINDOW", False);
    wm_protocols = XInternAtom(XtDisplay(top), "WM_PROTOCOLS", False);
    (void) XSetWMProtocols(XtDisplay(top), XtWindow(top), &wm_delete_window, 1);

    XtAppMainLoop(app);
    return 0;
}
```

Para compilar o programa anterior:

```
cc -c -I/usr/X11R6/include hello-Xaw.c
cc -L/usr/X11R6/lib hello-Xaw.o -o hello-Xaw -lXaw -lXt -lX11
```

2.12.3 Processamento de eventos

Obtenção de eventos. Como evidenciado no programa `hello-Xlib`, o laço de processamento de eventos começa com `XNextEvent(display, &ev)`. `XNextEvent` é a função que obtém o próximo evento recebido do display `display` e o armazena da estrutura `ev`, do tipo `XEvent`. O tipo `XEvent` é declarado como uma estrutura contendo um campo identificador de tipo, chamado `type`, e uma união (`union`, em C) de diversas estruturas de formatos diferentes. Obtido o evento, é preciso responder a ele de acordo com o conteúdo do campo `type`.

Se não houver nenhum evento na fila, a função `XNextEvent` aguarda até que seja enviado um, o que impede a continuação do programa. Uma forma de evitar que isso aconteça, permitindo que a aplicação faça algo de útil enquanto não tem eventos a processar, é usar a função `XEventsQueued`. Conforme mostrado no programa, essa função retorna o número de eventos não processados na fila (o segundo parâmetro determina o comportamento exato da função). A função `XPeekEvent`, por sua vez, faz quase o mesmo que `XNextEvent`, com uma diferença: o evento não é retirado da fila.

Seleção de eventos. O cliente X pode se selecionar os tipos de eventos que deseja receber. No nosso exemplo, isso é feito por meio do campo `event_mask` da estrutura `xswa`, do tipo `XSetWindowAttributes`, passada como parâmetro para a função `XCreateWindow`. Isso, porém, não evita que seqüências de eventos do mesmo tipo sejam enviadas para o cliente.

O programa demonstra uma técnica bastante comum, chamada *compressão de eventos*, que consiste em descartar seqüências de eventos do mesmo tipo e processar apenas o último deles. No caso, estamos comprimindo o tratamento

do evento do tipo `Expose`, recebido sempre que a janela é exposta. A finalidade é evitar que o conteúdo da janela seja redesenhado múltiplas vezes à medida que ela é redimensionada pelo gerenciador de janelas.

2.12.4 Primitivas de desenho

X possui uma série de mecanismos para a geração de formas geométricas simples.

Pontos. Pontos individuais são desenhados usando a função `XDrawPoint`:

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

Múltiplos pontos podem ser desenhados de uma só vez com a função `XDrawPoints`:

```
typedef struct {
      short x, y;
} XPoint;

XDrawPoints(display, d, gc, points, npoints, mode)
      Display *display;
      Drawable d;
      GC gc;
      XPoint *points;
      int npoints;
      int mode;
```

`Drawable` pode ser uma janela ou um *pixmap*. GC é um contexto gráfico, que determina os atributos usados para desenho.

Linhas. Linhas são desenhadas com a função `XDrawLine`:

```
XDrawLine(display, d, gc, x1, y1, x2, y2)
      Display *display;
      Drawable d;
      GC gc;
      int x1, y1, x2, y2;
```

Assim como no caso dos pontos, múltiplas linhas podem ser desenhadas de uma única vez usando a função `XDrawLines`:

```
XDrawLines(display, d, gc, points, npoints, mode)
      Display *display;
      Drawable d;
      GC gc;
      XPoint *points;
      int npoints;
      int mode;
```

Arcos (círculos e elipses). `XDrawArc` e `XDrawArcs` permitem desenhar um ou diversos arcos de cada vez:

```
XDrawArc(display, d, gc, x, y, width, height, angle1, angle2)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      unsigned int width, height;
      int angle1, angle2;

typedef struct {
        short x, y;
        unsigned short width, height;
        short angle1, angle2;
 } XArc;

XDrawArcs(display, d, gc, arcs, narcs)
      Display *display;
      Drawable d;
      GC gc;
      XArc *arcs;
      intnarcs;
```

É importante observar que os ângulos são sempre expressos em $\frac{1}{64}$ graus. Isto pode parecer estranho, mas tem a vantagem de permitir que se usem valores inteiros para os ângulos.

Retângulos. `XDrawRectangle` e `XDrawRectangle` permitem desenhar um ou diversos retângulos de cada vez:

```
XDrawRectangle(display, d, gc, x, y, width, height)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      unsigned int width, height;

typedef struct {
       short x, y;
       unsigned short width, height;
 } XRectangle;

XDrawRectangles(display, d, gc, rectangles, nrectangles)
      Display *display;
      Drawable d;
      GC gc;
      XRectangle rectangles[];
      int nrectangles;
```

Observe-se que 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`.

Polígonos. 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`.

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.

2.12.5 Manipulação de contextos gráficos

A maioria das funções gráficas usa GCs para informar ao servidor como realizar as operações, isto é, que cores de frente e fundo, estilos de linhas, padrões de preenchimento, etc. usar. O uso de GCs é conveniente porque evita que esses parâmetros tenham de ser enviados ao servidor a cada operação de desenho solicitada.

Criação de GCs. A função `XCreateGC` permite criar um novo GC:

```
GC XCreateGC(display, d, valuemask, values)
      Display *display;
      Drawable d;
      unsigned long valuemask;
      XGCValues *values;
```

O parâmetro `values` é o endereço de uma estrutura contendo os atributos do GC e o parâmetro `valuemask` indica quais desse atributos devem ser copiados para o novo GC em substituição aos valores padrão.

Alteração dos atributos de um GC. `XChangeGC` permite alterar os atributos, de maneira semelhante:

```
XChangeGC(display, gc, valuemask, values)
      Display *display;
      GC gc;
      unsigned long valuemask;
      XGCValues *values;
```

Para facilitar a programação, existe um conjunto de *funções de conveniência* que permitem alterar individualmente os atributos de um GC. Exemplos são as funções `XSetBackground` e `XSetForeground`.

2.13 Documentação/URLs

Documentação sobre a Xlib, Xt e Xaw pode ser encontrada na forma de páginas de manual normalmente instalados em sistemas com o X. Manuais em PostScript® podem ser obtidos via FTP anônimo em:

- `ftp://ftp.x.org/pub/R6.4/xc/docs/hardcopy/X11/`
- `ftp://ftp.x.org/pub/R6.4/xc/docs/hardcopy/Xt/`
- `ftp://ftp.x.org/pub/R6.4/xc/docs/hardcopy/Xaw/`

Muitas referências sobre sobre Motif podem ser encontradas a partir dos seguintes endereços:

- Home-page do projeto LessTif: `http://www.lesstif.org/`
- The Motif Zone: `http://www.motifzone.net/`
- Motif FAQ: `http://www.rahul.net/kenton/mfaq.html`

Documentação sobre o *toolkit* Gtk, assim como as bibliotecas, podem ser encontradas na própria *home page* do Gtk (há ponteiros para tópicos relacionados, como o Glade e GNOME):

- `http://www.gtk.org/`

Documentação sobre o *toolkit* Qt, assim como suas bibliotecas, podem ser encontradas no site da Troll Tech:

- `http://www.trolltech.com/`

Christophe Tronche criou uma versão em HTML de [2], disponível em `http://www.tronche.com/gui/x/xlib/`.

2.14 Conclusão

O X Window System oferece um conjunto poderoso de mecanismos para a construção de aplicações gráficas interativas. Graças à sua arquitetura cliente-servidor e aos princípios de independência entre aplicação e hardware gráfico, torna-se muito mais simples construir programas compatíveis com diferentes arquiteturas.

Graças à sua filosofia de oferecer mecanismo e não política, associada à extensibilidade do protocolo, X tem evoluído ao longo de mais de 15 anos de existência, sempre se conservando estável e confiável. A existência de uma especificação formal rígida para o protocolo garante também a interoperabilidade em ambientes heterogêneos de hardware/software. O X Window System constitui uma plataforma versátil de desenvolvimento combinado com bibliotecas de interface e ferramentas gráficas de mais alto nível como o OpenGL, que será visto na próxima seção.

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].

A interface destinada a aplicações gráficas 2D ou 3D deve satisfazer diversos critérios como, por exemplo, ser implementável em plataformas com capacidades distintas sem comprometer a performance gráfica do hardware e sem sacrificar o controle sobre as operações de hardware [21].

Atualmente, o OpenGL ("GL" significa *Graphics Library*) é uma API de grande utilização no desenvolvimento de aplicações em Computação Gráfica [22]. Este padrão é o sucessor da biblioteca gráfica conhecida como IRIS GL, desenvolvida pela Silicon Graphics como uma interface gráfica independente de hardware [23]. 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. Por ser um padrão destinado somente à renderização [21], o 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 [21], [23]. Além do GLX, há uma biblioteca alternativa para interfaceamento no X denominada GLUT (*OpenGL Utility Toolkit*) [24]. Esta biblioteca possui um conjunto de ferramentas que facilita a construção de programas utilizando o OpenGL. Podemos citar, por exemplo, funções para gerenciamento de janelas, rotinas para geração de vários objetos gráficos 3D ou dispositivos de entrada de dados. Uma vantagem em se utilizar a GLUT é que esta biblioteca é compatível com quase todas as implementações OpenGL em Windows e X. Em aplicações que requerem uma maior utilização dos recursos do X, pode-se utilizar a GLUT juntamente com a GLX.

Esta seção descreve as funcionalidades do OpenGL e, quando necessário, apresenta algumas rotinas disponíveis no GLX e na GLUT.

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). Assim como as demais coordenadas, pode-se também especificar o valor para a coordenada w. Mas isto raramente é feito, sendo assumido o valor 1.0 como default. Além disso, todos os cálculos internos são realizados com pontos definidos no espaço tridimensional. Assim sendo, os pontos bidimensionais especificados pelo usuário (i.e., somente definidos com as coordenadas x e y) são trabalhados no OpenGL 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*()`[3].

[3] 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, por exemplo, polígonos não simples, polígonos côncavos ou polígonos com furos [22]. 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*) [22]. Esta biblioteca utiliza somente funções GL (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 definação 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, utilizando um número restrito de funções que são válidas entre o par `glBegin()` e `glEnd()` [22]. É importante salientar que a restrição quanto à utilização é apenas para as rotinas do OpenGL; outras estruturas de programação poderão ser utilizadas normalmente entre o par de funções. O argumento da função `glBegin()` indicará a ordem como serão associados os vértices, conforme ilustrado na Figura 6.

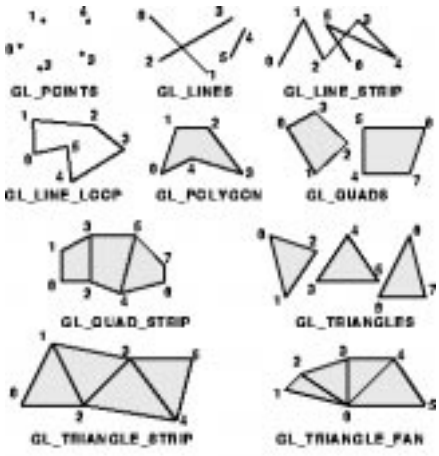

Figura 6: Primitivas geométricas do OpenGL [23].

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 disso, há também o *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* (lista de instruções).

A *display list* é uma maneira de definir um grupo de rotinas do OpenGL que serão executadas posteriormente, respeitando a seqüência em que foram definidas. A maioria das rotinas do OpenGL pode ser armazenada em uma *display list* ou executada na forma imediata (*immediate mode*). Pode-se utilizar ambos os artifícios no desenvolvimento de aplicações gráficas. Entretanto, rotinas do OpenGL com passagem de parâmetros por referência ou que retornem um valor não serão armazenadas. A restrição é adotada porque uma lista poderá, por exemplo, ser executada fora do escopo de onde os parâme-

tros foram originalmente definidos. Estas rotinas, as rotinas que não pertencerem ao OpenGL e as variáveis serão avaliadas no momento da criação da lista, sendo substituídas por seus valores resultantes.

Uma *display list* é definida agrupando as instruções entre as funções `glNewList()` e `glEndList()`, de modo similar à definição de uma primitiva geométrica. O trecho de código para traçar a circunferência pode então ser rescrito utilizando uma lista, como apresentado a seguir.

```
#define PI 3.1415926535
#define CIRC 1
int circle_points = 100;

glNewList(CIRC, GL_COMPILE);
glBegin(GL_LINE_LOOP);
for (i = 0;i < circle_points;i++)
{
   angle = 2*PI*i/circle_points;
   glVertex2f(cos(angle),sin(angle));
}
glEnd();
glEndList();
```

O argumento `CIRC` é um número inteiro que identificará a lista. O atributo `GL_COMPILE` indicará ao sistema que a lista será compilada, porém o seu conteúdo não será executado. Caso seja necessário executar seu conteúdo, é utilizado o atributo `GL_COMPILE_AND_EXECUTE`. Quando for necessário, a lista poderá ser utilizada através da rotina `glCallList()`.

Uma alternativa para modelar objetos que são difíceis de serem definidos através de vértices é utilizar rotinas fornecidas nas bibliotecas GLU e GLUT, como já mencionado anteriormente. Por exemplo, para traçarmos uma esfera simplesmente executamos a rotina `glutWireSphere()`, onde os argumentos definirão o raio, o número de linhas longitudinais e o número de linhas latitudinais.

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, alguns aspectos 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.

***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 matriz atual[4] da *modelview* conterá o produto cumulativo das multiplicações destas matrizes. Ou seja, cada matriz de transformação utilizada será multiplicada pela matriz atual, e o resultado será colocado como a nova matriz atual, representando a transformação composta. A pilha *projection* comporta-se da mesma maneira. Entretanto, na maioria das vezes esta pilha conterá apenas duas matrizes: uma matriz identidade e uma matriz de projeção, pois um volume de visualização pode ser definido apenas por uma matriz de transformação.

A pilha de matrizes é utilizada no OpenGL para facilitar a construção de modelos hierárquicos, onde objetos complexos são construídos a partir de objetos mais simples. Além disso, a pilha é um mecanismo ideal para organizar a seqüência de

[4] A matriz atual é aquela que está no topo da pilha.

operações sobre matrizes. Segundo a metodologia de uma pilha de dados, a transformação especificada mais recentemente (a última a ser "empilhada") será a primeira a ser aplicada [25]. OpenGL possui um conjunto de rotinas que manipulam as pilhas e as matrizes de transformação. Abordaremos, de forma sucinta, as principais rotinas e suas utilizações.

A definição da pilha na qual se deseja trabalhar é feita através da rotina `glMatrixMode()`, indicada pelo argumento `GL_MODELVIEW` ou `GL_PROJECTION`. Após definida a pilha, ela pode ser então inicializada com a matriz identidade, através da rotina `glLoadIdentity()`. Como default, toda pilha conterá apenas a matriz identidade.

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.

O controle sobre a pilha pode ser feito através das rotinas `glPushMatrix()` e `glPopMatrix()`. A rotina `glPushMatrix()` duplica a matriz atual, colocando a cópia no topo da pilha em questão. Este método permite preservar o estado da pilha em um determinado momento para posterior recuperação, realizada por meio da rotina `glPopMatrix()`.

O exemplo a seguir demonstra a utilização destas rotinas. O trecho de código desenha um automóvel, assumindo a existência das rotinas para desenhar o "corpo" do automóvel, a roda e o parafuso.

```
desenha_roda_parafusos()
{
   long i;
   desenha_roda();

   // desenha cinco parafusos na roda
   for (i = 0; i < 5; i++)
   {
      glPushMatrix();
         glRotatef(72.0*i,0.0,0.0,1.0);
         glTranslatef(3.0,0.0,0.0);
         desenha_parafuso();
      glPopMatrix();
   }
}

desenha_corpo_roda_parafuso()
{
   desenha_corpo_carro();

   // posiciona e desenha a primeira roda
   glPushMatrix();
      glTranslatef(40,0,30);
      desenha_roda_parafusos();
   glPopMatrix();

   // posiciona e desenha a segunda roda
   glPushMatrix();
      glTranslatef(40,0,-30);
      desenha_roda_parafusos();
   glPopMatrix();

   // desenha as outras duas rodas de forma similar,
   // alterando apenas a posição
   // ...
}
```

Quanto à definição do volume de visualização, OpenGL provê duas transformações de projeção: a perspectiva e a ortogonal.

A projeção perspectiva define um volume de visualização onde a projeção do objeto é reduzida a medida que ele é afastado da câmera. Esta projeção é fornecida no OpenGL através da rotina `glFrustum()`. O volume de visualização é calculado através de seis planos de corte, sendo os quatro planos que formam a janela (*left*, *right*, *top* e *bottom*), mais os planos *near* e *far*, como ilustrado na Figura 7.

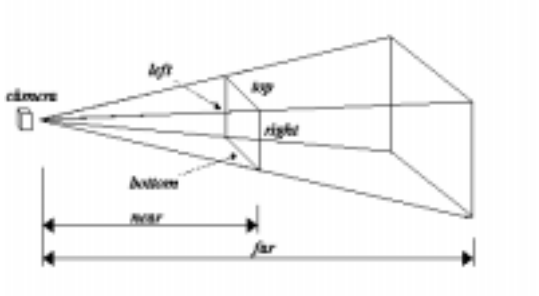

Figura 7: O volume de visualização da projeção perspectiva.

A projeção ortogonal define um volume de visualização onde a projeção do objeto não é afetada pela sua distância em relação à câmera. Esta projeção é fornecida no OpenGL através da rotina `glOrtho()`. O volume de visualização é calculado de modo similar à projeção perspectiva, através dos mesmos seis planos de corte. Os planos formarão um paralelepípedo retangular, ilustrado na Figura 8.

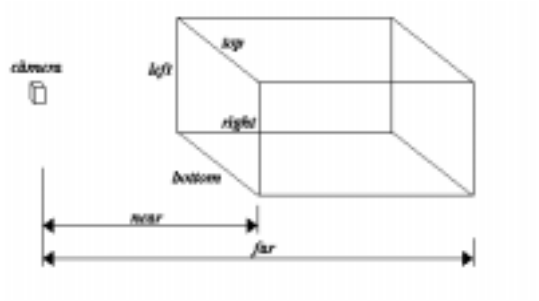

Figura 8: O volume de visualização da projeção ortogonal.

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.

Além das matrizes de transformação definidas pelo OpenGL, pode-se também atribuir ou multiplicar a matriz atual por uma determinada matriz de transformação especificada pelo usuário, respectivamente através das rotinas `glLoadMatrix*()` ou das rotinas `glMultMatrix*()`.

3.3 Cor

OpenGL possui dois modos diferentes para tratar cor: o modo RGBA e o modo indexado de cor [23]. A definição do modo de cor dependerá da biblioteca que o programa está utilizando para interfacear com o sistema de janelas. A GLUT, por exemplo, provê uma rotina denominada `glutInitDisplayMode()`, onde a seleção é feita através dos parâmetros `GLUT_RGBA` ou `GLUT_INDEX`. O default é `GLUT_RGBA`, caso não seja especificado nenhum dos modos.

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), sendo muito úteis para renderizar cenas realísticas. 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, onde a cor é totalmente transparente, até 1.0, onde a cor é totalmente opaca. Desta forma, o valor alfa não é visível na tela, sendo usado apenas para determinar como o pixel será exibido. As rotinas `glColor*()` são utilizadas para definir os valores para cada componente. O trecho de código a seguir define um triângulo no modo RGBA.

```
glBegin(GL_TRIANGLES);
   glColor3f(1.0, 0.0, 0.0);
   glVertex2f(5.0, 5.0);
   glColor3f(0.0, 1.0, 0.0);
   glVertex2f(25.0, 5.0);
   glColor3f(0.0, 0.0, 1.0);
   glVertex2f(5.0, 25.0);
glEnd();
```

Cada vértice do polígono foi definido com uma cor e o seu interior será preenchido conforme o modo indicado através da rotina `glShadeModel()`. Pode-se indicar o modo `GL_FLAT`, onde a cor de um vértice do polígono será utilizada como padrão para toda a primitiva geométrica, ou `GL_SMOOTH`, onde as cores para o interior do polígono são interpoladas entre as cores definidas para os vértices (método *Gouraud shading*) (ver Figura 9). Este último modo é o default.

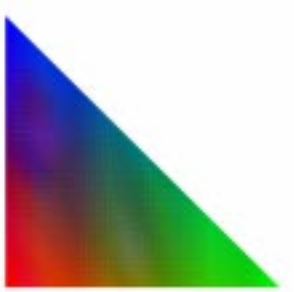

Figura 9: Um triângulo desenhado no modo *smooth*.

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. No X, por exemplo, a rotina `XAllocColor()` é utilizada para alocação de cores. Já a biblioteca GLUT provê a rotina `glutSetColor()`, que define as componentes primárias para um determinado índice no mapa de cores. As rotinas `glIndex*()` são usadas para selecionar o índice, no mapa de cores, da cor atual.

3.4 Iluminação

OpenGL utiliza o modelo de Phong para prover uma cena com realismo [25]. 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 fornecidos pelo modelo, embora técnicas e algoritmos estejam disponíveis para simular tais efeitos.

Para implementar o modelo de iluminação, o OpenGL decompõe o raio luminoso nas componentes primárias RGB. Dessa forma, a cor para uma fonte de luz é caracterizada pela porcentagem da intensidade total de cada componente emitida. Se todas as componentes possuírem o valor 1.0, a luz será a mais branca possível. Se todos os valores forem 0.5, ainda será a cor branca, mas com uma intensidade menor (aparentando a cor cinza). Para os materiais, os valores correspondem à porcentagem refletida de cada componente primária. Se a componente vermelha for 1.0, a componente verde for 0.5 e a componente azul for zero para um determinado material, este refletirá toda a intensidade da luz vermelha, metade da intensidade da luz verde e absorverá a luz azul. Por exemplo, uma bola vermelha que receba a incidência das luzes vermelha, verde e azul refletirá somente a luz vermelha, absorvendo as luzes verde e azul. Caso seja incidida uma luz branca (composta por uma intensidade igual das componentes vermelha, verde e azul), a superfície da bola refletirá apenas a luz vermelha e, por isso, a bola será vista com esta cor. Mas caso seja incidida apenas uma luz verde ou azul, a bola será vista com a cor preta, pois não haverá luz refletida.

Uma vez que um raio luminoso será dividido nas suas componentes primárias RGB, o OpenGL considera ainda que está divisão será realizada para cada componente de luz do modelo de iluminação, que são:

Componente emitida. É aquela componente de luz originada de um objeto, não sendo afetada por qualquer fonte de luz.

Componente ambiente. É a componente de luz 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 várias reflexões nas superfícies contidas na cena.

Componente difusa. É a componente de luz 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. Qualquer luz proveniente de uma determinada posição ou direção provavelmente tem uma componente difusa.

Componente especular É a componente de luz refletida em uma determinada direção quando esta incide sobre uma superfície, proveniente de uma direção específica. Uma superfície como um espelho de alta qualidade produz uma grande quantidade de reflexão especular, assim como os metais brilhantes e os plásticos. Entretanto, materiais como o giz possui uma baixa reflexão especular.

No modelo de iluminação do OpenGL, a luz na cena pode ser proveniente de várias fontes, sendo controladas individualmente. Algumas luzes podem ser provenientes de uma determinada direção ou posição, enquanto outras podem estar dispersas na cena. Entretanto, as fontes de luz somente têm efeito nas superfícies que definiram as suas propriedades do material. Em relação às propriedades do material, elas podem ser definidas de modo a emitir luz própria, a dispersar a luz incidente em todas as direções e a refletir uma porção da luz incidente em uma determinada direção, como uma superfície espelhada ou reluzente.

Quanto aos tipos de fonte de iluminação, o OpenGL possui:

Fontes pontuais. É uma fonte que irradia energia luminosa em todas as direções.

Fontes *spots*. É uma fonte pontual direcional, isto é, tem uma direção principal na qual ocorre a máxima concentração de energia luminosa; fora desta direção ocorre uma atenuação desta energia.

Além das fontes citadas anteriormente, o OpenGL provê uma luz que não possui uma fonte específica, denominada luz ambiente. As rotinas `glLight*()` são utilizadas para especificar as propriedades da fonte de iluminação e as rotinas `glLightModel*()` descrevem os parâmetros do modelo de iluminação como, por exemplo, a luz ambiente.

Da mesma maneira que a luz, os materiais têm diferentes valores para as componentes especular, difusa e ambiente, determinando assim suas reflexões na 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.

Depois de definidas as características de cada fonte de luz e dos materiais, deve-se utilizar a rotina `glEnable()` para habilitar cada fonte de luz previamente definida. Antes, entretanto, esta rotina deve ser utilizada com o parâmetro `GL_LIGHTING`, de modo a preparar o OpenGL para os cálculos do modelo de iluminação.

O trecho de código a seguir ilustra a definição de um modelo de iluminação no OpenGL para traçar uma esfera azul com o efeito do reflexo da luz (brilho) concentrado em um ponto. Quanto maior o valor da variável `mat_shininess`, maior será a concentração da luz e conseqüentemente menor é o ponto e maior é o brilho. O resultado está demonstrado na Figura 10.

```
//  Inicializa as propriedades do material e da fonte de luz
GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
GLfloat mat_diffuse[] = {0.0, 0.0, 1.0, 1.0};
GLfloat mat_shininess[] = {50.0};
GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};

glShadeModel(GL_SMOOTH);

// Define as componentes do material
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
```

```
// Define somente a posição da fonte de luz - os valores das
// componentes serão os valores default
glLightfv(GL_LIGHT0, GL_POSITION, light_position);

// Habilita o modelo de iluminação
glEnable(GL_LIGHTING);
// Habilita a fonte de luz definida
glEnable(GL_LIGHT0);
```

Figura 10: Exemplo do modelo de iluminação do OpenGL.

3.5 Textura

Para realizarmos um mapeamento de textura no OpenGL, o procedimento utilizado segue um padrão básico, conforme descrito a seguir:

1. Especificar a textura.
2. Indicar como a textura será aplicada para cada pixel.
3. Habilitar o mapeamento de textura.
4. Desenhar a cena, fornecendo as coordenadas geométricas e as coordenadas de textura.

No OpenGL, quando um mapeamento de textura é realizado, cada pixel do fragmento a ser mapeado referencia uma imagem, gerando um *texel*. O *texel* é um elemento de textura que representa a cor que será aplicada em um determinado fragmento, tendo entre um (uma intensidade) e quatro componentes (RGBA) [20].

Uma imagem de textura é disponibilizada pelas funções `glTextImage*()` podendo, caso necessário, ser especificada em diferentes resoluções, através de uma técnica denominada *mipmapping*. O uso de uma textura com multiresolução é recomendado em cenas que possuam objetos móveis. A medida que estes objetos se movem para longe do ponto de visão, o mapa de textura deve ser decrementado em seu tamanho na mesma proporção do tamanho da imagem projetada. Desta maneira, o mapeamento sempre utilizará a resolução mais adequada para o fragmento.

Para indicar como a textura será aplicada para cada pixel, é necessário escolher uma das três possíveis funções que combinam a cor do fragmento a ser mapeado com a imagem da textura, de modo a calcular o valor final para cada pixel. Pode-se utilizar os métodos *decal*, *modulate* ou *blend*, de acordo com a necessidade do usuário. O controle do mapeamento da textura na área desejada é especificado através das rotinas `glTextEnv*()` e as rotinas `glTextParameter*()` determinam como a textura será organizada no fragmento a ser mapeado e como os pixels serão "filtrados" quando não há uma exata adaptação entre os pixels da textura e os pixels na tela.

Para desenhar a cena é necessário indicar como a textura estará alinhada em relação ao fragmento desejado. Ou seja, é necessário especificar as coordenadas geométricas e as coordenadas de textura. Para um mapeamento de textura bidimensional, o intervalo válido para as coordenadas de textura será de 0.0 a 1.0 em ambas direções, diferentemente das coordenadas do fragmento a ser mapeado onde não há esta restrição. No caso mais simples, por exemplo, o mapeamento é feito em um fragmento proporcional às dimensões da imagem de textura. Nesta situação, as coordenadas de textura são (0,0), (1,0), (1,1) e (0,1). Entretanto, em situações onde o fragmento a ser mapeado não é proporcional à textura, deve-se ajustar as coordenadas de textura de modo a não distorcer a imagem. Para definir as coordenadas de textura é utilizado as rotinas `glTextCoord*()`.

Para habilitar o mapeamento de textura é necessário utilizar a rotina `glEnable()`, utilizando a constante `GL_TEXTURE_1D` ou `GL_TEXTURE_2D`, respectivamente para um mapeamento unidimensional ou bidimensional.

O trecho de código a seguir demonstra o uso do mapeamento de textura. No exemplo, a textura — que consiste de quadrados brancos e pretos alternados como um tabuleiro de xadrez — é gerada pelo programa, através da rotina `makeCheckImage()`. O programa aplica a textura em um quadrado, como ilustrado na Figura 11.

```
// Trecho do código responsável por toda a inicialização do
// mapeamento de textura
makeCheckImage();
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexImage2D(GL_TEXTURE_2D, 0, 3, checkImageWidth, checkImageHeight, 0, GL_RGB,
             GL_UNSIGNED_BYTE, &checkImage[0][0][0]);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
glEnable(GL_TEXTURE_2D);

// Trecho do código responsável pela definição das coordenadas
// de textura e das coordenadas geométricas
glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(-1.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
glEnd();
```

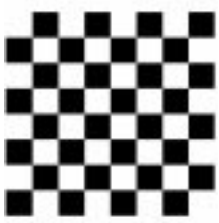

Figura 11: Exemplo do mapeamento de textura em um quadrado.

3.6 *Framebuffer*

Em uma aplicação, a área utilizada para armazenar temporariamente os dados é denominada *buffer*. 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 [22]:

***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.

***Buffer* de profundidade.** É utilizado para armazenar, durante a renderização, o pixel que possuir o menor valor da coordenada z, para as mesmas coordenadas x e y (algoritmo *z-buffer*).

***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.** Pode ser utilizado para trabalhar com diversas técnicas como, por exemplo, *antialiasing*, *motion blur* (borrão) ou profundidade de campo. Em um conjunto de imagens, cada imagem renderizada é acumulada neste *buffer*, manipulando-a caso necessário. A imagem resultante destas acumulações é exibida através da transferência para um *buffer* de cor.

3.7 Animação

No OpenGL não há rotinas específicas para produção de animações. Uma opção é a utilização de um *double buffering*, desde que seja disponível pelo sistema de janelas. No X, por exemplo, há um comando denominado `glXSwapBuffers()`, que disponibiliza este recurso [22]. Dessa forma, enquanto um quadro é exibido na tela, o próximo quadro está sendo renderizado no *buffer* não visível (auxiliar).

3.8 Conclusão

OpenGL é uma API gráfica 3D que permite ao programador descrever uma variedade de tarefas de renderização. Este padrão foi projetado para fornecer o máximo acesso às capacidades de diferentes hardwares gráficos, sendo implementável e executável em uma variedade de sistemas. Pela sua flexibilidade em modelar e renderizar objetos geométricos, 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 12. De forma mais precisa, Java 3D é um componente da Sun Microsystems, junto com as várias tecnologias multimídia e gráficas suportadas pela extensão padrão Java Media Framework, para o desenvolvimento de aplicações (aplicativos e/ou applets) 3D.

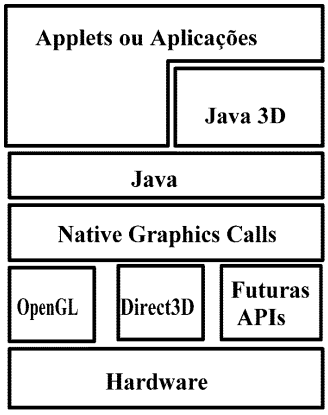

Figura 12: Relação entre as várias camadas de software no contexto de Java 3D.

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. Por essa razão, programadores não familiarizados com tais detalhes podem também explorar o universo 3D em suas aplicações. Esta tecnologia gráfica vem ainda ao encontro de uma crescente demanda por operações 3D requisitada hoje pela Web. Neste sentido, Java 3D se apresenta como uma solução fortemente viável, considerando que a mesma disponibiliza uma interface robusta para Web.

Estas características facilitam e agilizam o desenvolvimento de aplicações 3D mais complexas, uma vez que a reutilização é uma realidade e a compatibilidade com diferentes plataformas é uma das premissas básicas de Java.

Java 3D utiliza alguns conceitos que são comuns a outras tecnologias, tais como a VRML (ver Seção 5), considerada por alguns autores como sendo a "prima" mais próxima da Java 3D [26]. 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 [27]. 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. Esta abordagem de mais alto nível valoriza significativamente a produtividade dos desenvolvedores e facilita em muito a tarefa dos programadores com pouca experiência em programação 3D.

O conjunto de ferramentas oferecidas por Java 3D possibilita, além de construir uma cena 3D a partir de um programa, que esta seja carregada de um arquivo externo, por exemplo uma cena no formato VRML. Este conjunto de propriedades dá a Java 3D grande flexibilidade, fazendo dela uma plataforma viável para diferentes aplicações gráficas. A literatura cita aplicações em visualização molecular, visualização científica, realidade virtual, sistemas de informação geográfica, animação, entre outros [28].

Antes de começarmos a estudar Java 3D, é importante fazer uma breve introdução da linguagem Java.

4.1 Java

Java 3D é um componente da linguagem Java, que é uma linguagem de programação de alto nível desenvolvida pela Sun Microsystems. Java se tornou muito popular para a construção de páginas altamente interativas na Web.

Java pode ser usada tanto para o desenvolvimento de programas independentes quanto para o de applets, que são executados dentro de um "ambiente hospedeiro" (o browser). Os applets são tratados pelo browser como qualquer outro tipo de objeto da página HTML, como uma imagem ou um vídeo: ele é transmitido do servidor para o cliente, onde é executado e visualizado dentro do browser.

Java é uma linguagem orientada a objetos de propósito geral (semelhante a C++) e projetada para ser simples. Todos os recursos considerados desnecessários foram propositalmente deixados de fora da linguagem. Java não possui, por exemplo, apontadores, estruturas, vetores multi-dimensionais e conversão implícita de tipos. Também não há necessidade de gerenciamento de memória em Java, pois ela tem um programa interno (*garbage collector*) que automaticamente libera partes ocupadas da memória que não terão mais uso.

Outra característica essencial de Java é ser independente de plataforma. O código-fonte de um programa Java é pré-compilado em *bytecodes*, que são conjuntos de instruções semelhantes ao código de máquina, mas sem serem específicos de qualquer plataforma. As instruções em *bytecodes* são verificadas na máquina local antes de serem executadas, garantindo a segurança da linguagem. Os *bytecodes* podem ser interpretados por Máquinas Virtuais Java (JVMs — *Java Virtual Machines*) instaladas em qualquer plataforma, sem necessidade de recompilação do programa. Praticamente todos os browsers já incorporam a JVM em sua implementação.

4.2 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, tanto dos personagens quanto do cenário que compõem um dado mundo virtual. A Figura 13 ilustra um possível exemplo de um grafo de cenas. Nos próximos parágrafos discute-se, entre outros aspectos, os tipos de nós que estão representados nesta figura. Foge do escopo deste texto uma abordagem mais próxima dos construtores de cada tipo de nó, para este fim sugere-se [29]. Na seção 4.2.1 mostra-se um procedimento básico que pode ser empregado na construção de programas Java 3D e um primeiro exemplo de código.

Os grafos de cenas (ou subgrafos) são conectados ao universo virtual (representado na Figura 13 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 13 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`). Os *branch graphs* não determinam a ordem em que os objetos serão renderizados, mas sim o que será renderizado.

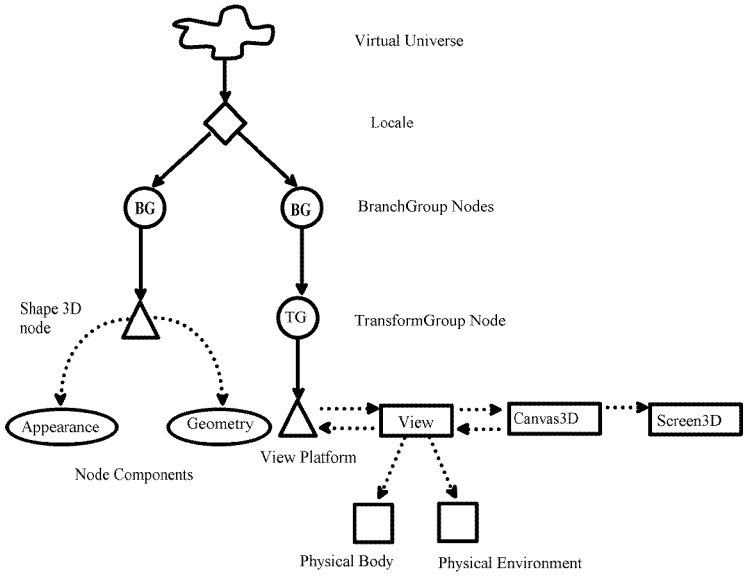

Figura 13: Grafo de uma cena em Java 3D.

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. Assim, uma forma 3D depende somente das informações do seu caminho para ser renderizada. O modelo de renderização de Java 3D explora este fato renderizando os nós folhas em uma ordem 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. O modelo de renderização é mais largamente discutido em [26].

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. A Figura 14 mostra alguns níveis da hierarquia de classes dos componentes que compõem um grafo de cenas.

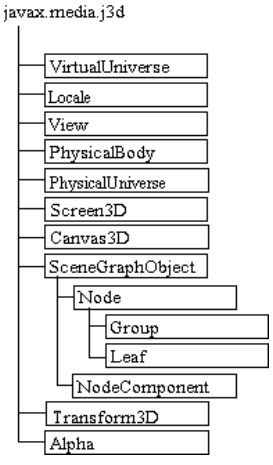

Figura 14: Hierarquia de classes de alguns componentes de uma cena Java 3D.

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 13, são os elementos referenciados por arcos tracejados — a próxima seção aborda detalhadamente este tópico). Os nós `Behavior` são usados na manipulação de eventos disparados pelo usuário e na animação de objetos do universo virtual, as Seções 4.4 e 4.5 fornecem as informações básicas sobre como estes objetos podem ser especificados em um programa Java 3D. 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.

Ao contrário das APIs que possuem apenas o modelo de vista que simulam uma câmera (denominado *camera-based*), Java 3D oferece um sofisticado modelo de vista que diferencia claramente o mundo virtual do mundo físico. O `ViewPlatform` é o único nó presente no grafo de cenas que faz referências aos objetos que definem este mundo físico (`PhysicalBody`, `PhysicalEnvironment`, `View`, `Canvas 3D` e `Screen 3D`). Essa associação entre o mundo virtual e o físico possibilita o desenvolvimento de aplicações que exigem um nível de controle e um sincronismo entre estes ambientes, como por exemplo aplicações de realidade virtual. Para uma descrição detalhada do modelo de vista de Java 3D verificar [29].

4.2.1 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 (as superestruturas para anexar os `BranchGroup`), construir um *view branch* e um *content branch*. Um bom ponto de partida é a seqüência de passos sugerida por [30]:

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` definida no pacote `com.sun.j3d.utils.universe`. Esta classe cria todos os objetos necessários para a composição de um *view branch*. O construtor `SimpleUniverse()` 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 completo de uma aplicação cuja utilidade se limita a ser um exemplo didático. Seu objetivo é criar um grafo de cenas, ilustrado na Figura 15, que desenha um cubo rotacionado de $\pi/4$ radianos no eixo *x* e $\pi/5$ radianos no eixo *y* (ver Figura 16). Procura-se evidenciar os passos da seqüência dada anteriormente e fornecer comentários que contribuam para o entendimento do Exemplo01 (adaptado de [30]):

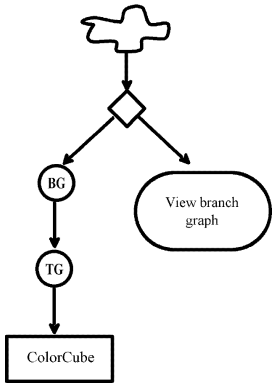

Figura 15: Grafo de cena do Exemplo01.

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.*;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.ColorCube;
import javax.media.j3d.*;
import javax.vecmath.*;

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() {

        // Especifica-se aqui os conteúdos gráficos a serem renderizados
        BranchGroup objRoot = new BranchGroup();

        // O trecho de código a seguir especifica duas transformações
        // 3D, uma para rotacionar o cubo no eixo x e a outra no eixo y
        // e por fim combina as duas transformações

        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;
    }
    //  O método a seguir permite que o applet Exemplo01 seja
    // executado como uma aplicação
    public static void main (String[] args) {
        Frame frame = new MainFrame (new Exemplo01(), 256, 256);
   }
}
```

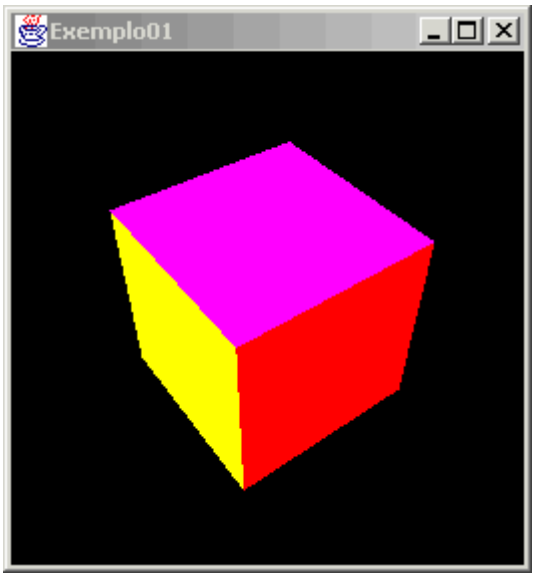

Figura 16: Imagem produzida pelo Exemplo01.

4.2.2 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);
```

4.3 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 (cor, transparência, tipo de material, textura, entre outros). Estes atributos estão armazenados em objetos do tipo `NodeComponent`. Um objeto `Shape3D` pode ser definido pela fórmula *Shape3D = geometria + aparência.* Em Java 3D este objeto pode ser instanciado pelo construtor:

```
Shape3D (Geometry geometry, Appearance appearence);
```

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. Um pseudo-código para esta finalidade pode ter a seguinte organização:

```
public class Modelo3D extends Shape3D
{
  private Geometry g;
  private Appearance a;

  public Modelo3D()
  {
    g = constroiGeometria();
    a = constroiAparencia();
    this.setGeometry(g);
    this.setAppearance(a);
  }

  private Geometry constroiGeometria ()
  {
   // Inserir o código que cria a geometria desejada
  }

  private Appearance constroiAparencia ()
  {
  // Inserir o código que cria a aparência desejada
  }
}
```

4.3.1 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. Por exemplo, um haltere pode ser a "soma" das seguintes primitivas: esfera + cilindro + esfera. Cada primitiva possui um método construtor onde são especificadas as dimensões do objeto (ou então estes são instanciados com dimensões default). Por exemplo, é mostrado a seguir um trecho de código para criar uma caixa, centrada na origem, com aparência `a` e de dimensões `X`, `Y`, `Z`, que especificam comprimento, largura e altura respectivamente:

```
Appearance a = new Appearance();
Primitive caixa = new Box (X, Y, Z, a);
```

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 o programador especifique outras formas geométricas para um objeto 3D em alternativa às formas pré-definidas discutidas acima. Neste método, os dados geométricos que modelam a forma do objeto 3D (primitivas geométricas definidas pelo programador) são 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.

A subclasse `GeometryStripArray` permite definir através de suas subclasses (`LineStripArray`, `TriangleStripArray` e `TriangleFanArray`) estruturas geométricas que "compartilham" o uso dos vértices especificados. Um objeto `LineStripArray` define uma lista de segmentos conectados, um objeto `TriangleStripArray` define uma lista de triângulos, onde cada dois triângulos consecutivos possuem uma aresta em comum e um objeto `TriangleFanArray` define uma lista de triângulos onde cada dois triângulos consecutivos possuem uma aresta em comum e todos compartilham um mesmo vértice (Figura 17).

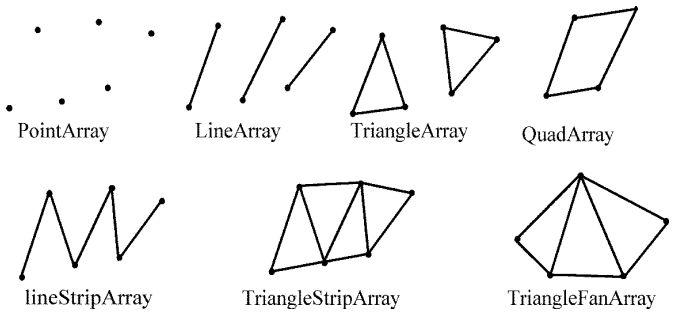

Figura 17: Objetos *array* e *strip*.

Um objeto `GeometryArray` pode armazenar, além das coordenadas de localização, coordenadas do vetor normal à superfície, coordenadas de cores e de texturas.

O código a seguir exemplifica como construir a geometria de um objeto `Shape3D` usando a classe `QuadArray`.

```
private Geometry constroiGeometria (){

      private static final float[] vertice = {
                     1.0f, -1.0f, 1.0f,
                     1.0f,  1.0f, 1.0f,
                    -1.0f,  1.0f, 1.0f,
                    -1.0f, -1.0f, 1.0f
      }
      private static final float[]     cor = {
                     // Azul
                     0.0f,  0.0f, 1.0f,
                     0.0f,  0.0f, 1.0f,
                     0.0f,  0.0f, 1.0f,
                     0.0f,  0.0f, 1.0f,
       }

      // Cria um objeto QuadArray vazio com 4 vértices e o flag de formato
      // de vértices é definido para coordenadas de localização e cores
      QuadArray quadrado = new QuadArray (4, COODINATES| COLOR_3);

      // Atribui o valor dos vertices ao quadrado
```

```
      quadrado.setCoordinates (0, vertice);

     // Atribui as informações de cores ao quadrado
     quadrado.setColor (0, cor);

     return quadrado;
}
```

No exemplo acima foi necessário especificar 4 vértices para definir um quadrado. Se o objeto a ser modelado fosse um cubo seria necessário especificar 4 x 6 = 24 vértices. Certamente existe uma grande redundância, pois um cubo tem apenas 8 vértices. Uma forma alternativa de definir esta geometria, eliminando esta redundância, 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. Voltando ao exemplo do cubo, um possível pseudo-código seria:

```
private Geometry constroiGeometria (){

     // Criar vetor de dados com 8 vértices
     // Criar objeto IndexedGeometryArray tendo como parâmetros
               // - quantidade de vértices
               // - tipo de coordenada
               // - tamanho do vetor de índices: 4 x 6 = 24
}
```

As alternativas apresentadas no segundo método são mais satisfatórias que as apresentadas no primeiro. Elas 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`, como exemplifica o trecho de código a seguir.

```
private Geometry constroiGeometria (){
// . . .
GeometryInfo P = new GeometryInfo (GeometryInfo.POLYGON_ARRAY);
P.setCoordinates (vertices_do_poligono);
Triangulator PT = new Triangulator();
PT.triangulate(P);

return P.getGeometryArray();
// . . .
}
```

A triangulação de um polígono não-planar pode gerar diferentes superfícies, de modo que o resultado obtido pode não ser o desejado. Isto leva o programador a um processo de tentativas até obter a forma desejada. Estes problemas mostram que escrever universos virtuais 3D complexos não é uma tarefa trivial.

Felizmente, 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*. 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`. Maiores detalhes sobre como escrever um *loader* para um dado formato de arquivo são mostrados em [30].

4.3.2 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 (ver Figura 18).

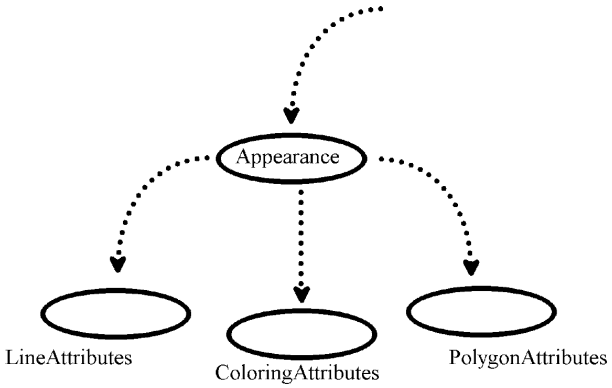

Figura 18: Subgrafo da cena relativa ao Exemplo02.

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* (habilita ou desabilita). 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*.

O exemplo02 a seguir emprega os métodos das subclasses discutidas no parágrafo anterior para especificar os atributos de linha, polígono e cor para um objeto 3D qualquer. O grafo de cena resultante é mostrado na Figura 18.

```
private Appearance constroiAparencia () {
    Appearance ap = new Appearance();

    LineAttributes al = new LineAttributes();
    al.setLineWidth(2.0f);
    al.setLinePattern(PATTERN_SOLID);
    ap.setColoringAttributes(al);

    ColoringAttributes aCor = new ColoringAttributes();
    aCor.setColor(new Color3f(1.0f, 0.0f, 0.0f));
    ap.setColoringAttributes(aCor);

    PolygonAttributes pa = new PolygonAttributes();
    pa.setPolygonMode(PolygonAttributes.Polygon_FILL);
    ap.setPolygonAttributes(pa);

    return ap;
```

```
}
```

4.4 Interação

Programar um objeto para reagir a determinados eventos é uma capacidade desejável na grande maioria das aplicações 3D. A reação resultante de um certo evento tem por objetivo provocar alguma mudança no mundo virtual que dependerá da natureza da aplicação e será determinada pelo programador. Por exemplo, um evento poderia ser pressionar uma tecla, 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* [30].

Java 3D implementa os conceitos de interação e animação na classe abstrata `Behavior`. Esta classe disponibiliza ao desenvolvedor de aplicações 3D 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 (ver seção seguinte). 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.

4.4.1 Estrutura de um *behavior*

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()`. Estes dois útimos são fornecidos pela classe `Behavior` e devem ser implementados por todos *behaviors* definidos pelo usuário.

O método `initialize()` é chamado uma vez quando o *behavior* torna-se vivo (*live*). Um objeto do grafo de cena é dito vivo se ele faz parte de um *brach graph* anexado a um nó `Locale` (ver Seção 4.2.2). A conseqüência deste fato é que estes objetos são passíveis de serem renderizados. 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 (ver seção seguinte). Esta rotina então efetua todo o processamento programado e define a próxima `WakeupCondition`, i.e., informa quando o *behavior* deve ser invocado novamente.

O código apresentado a seguir (adaptado de [30]) exemplifica como escrever um *behavior* para rotacionar um objeto sempre que uma tecla for pressionada.

```
public class  Rotaciona extends Behavior {
    private TransformGroup alvo;
    private Transform3D r = new Transform3D();
    private double  angulo = 0.0;

    // Método construtor
    Rotaciona ( TransformGroup  alvo ){
        this.alvo = alvo;
    }

    // Método initialize
    public void initialize() {
       this.wakeupOn (new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
    }

    // Método processStimulus
    public void  processStimulus( Enumeration criteria ) {
```

```
        angulo += 0.1;
        r.rotY (angulo);
        alvo.setTransform(r);
        this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
    }
}
```

Para adicionar um objeto `Behavior` a um programa Java 3D deve-se basicamente: *(i)* criar o suporte necessário exigido pelo *behavior* (e.g., um *behavior* Rotaciona precisa referenciar um nó `TransformGroup`), *(ii)* criar um objeto `Behavior` e efetuar as referências necessárias, *(iii)* definir um *scheduling bound* para o *behavior* (ver seção seguinte) e *(iv)* configurar as capacidades do objeto alvo. O método `ConstroiContentBranch` visto a seguir ilustra este processo para o *behavior* `Rotaciona`.

```
public BranchGroup ConstroiContentBranch( ) {

   // (i) suporte para o behavior
   BranchGroup objRoot = new BranchGroup();

   // (iv) configurar capacidades do objeto alvo
   TransformGroup objRotate = new TransformGroup();
   objRotate.setCapability (TransformGroup.ALLOW_TRANSFORM_WRITE);

   objRoot.addChild(objRotate);
   objRotate.addChild(new ColorCube(0.4));

   // (ii) criação do behavior referenciando o objeto que será rotacionado
   Rotaciona RotacionaBehavior = new Rotaciona(objRotate);

   // (iii) definição do scheduling bound
   RotacionaBehavior.setSchedulingBounds (new BoundingSphere());
   objRoot.addChild(RotacionaBehavior);
   objRoot.compile();

   return objRoot;
   }
```

4.4.2 Behaviors ativos

Por uma questão de desempenho, o programador define uma região limitada do espaço (por exemplo, uma esfera ou um paralelepípedo), denominada *scheduling bound* para o *behavior* (Figura 19). Ele é dito estar ativo quando seu *scheduling bound* intercepta o volume de cena. Apenas os *behaviors* ativos estão aptos a receber eventos. Dessa forma, o cômputo necessário a manipulação dos behaviors se reduz apenas aos que estão ativos, liberando assim mais tempo de CPU para o processo de renderização. Na verdade o *scheduling bound* é mais do que uma questão de performance, se ele não for definido o *behavior* associado nunca será executado [29].

4.5 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 (em milissegundos)

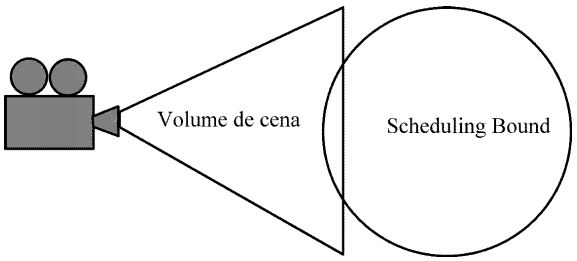

Figura 19: *Scheduling bound.*

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.

4.5.1 Mapeamento do tempo em `Alpha`

As caracteristicas de um objeto `Alpha` são determinadas por uma lista de parâmetros que podem variar até um total de dez. A Figura 20 mostra o gráfico de `Alpha` onde os valores alpha são obtidos em função de todos estes parâmetros. A forma do gráfico varia conforme a quantidade de parâmetros especificados.

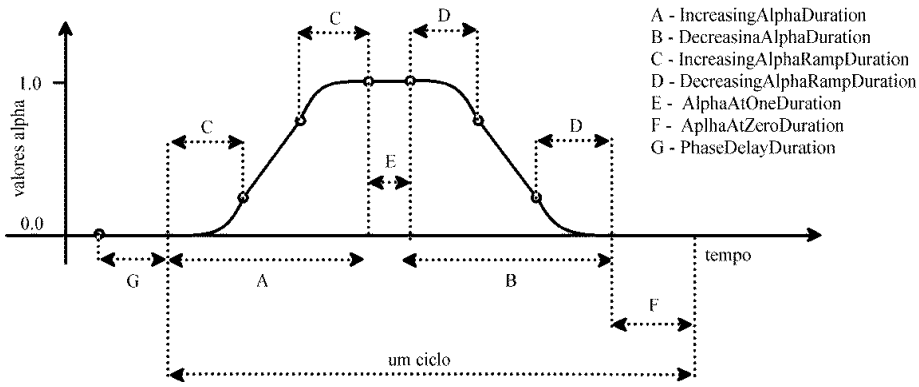

Figura 20: Gráfico Alpha: tempo x valores alpha.

É fácil identificar quatro fases bem distintas em um gráfico `Alpha`: uma fase de crescimento dos valores alpha (`increasingAlphaDuration` e `increasingAlphaRampDuration`), uma fase de decrescimento (`decreasingAlphaDuration` e `decreasingAlphaRampDuration`), uma fase com valores alpha constantes em zero
(`alphaAtOneDuration`) e uma fase com valores alpha constantes em um (`alphaAtZeroDuration`). A definição de um objeto `Alpha` pode ter todas ou apenas algumas destas fases. Os parâmetros correspondentes a cada uma das fases de `Alpha` permitem controlar o número de vezes (`loopCount`) que os valores alpha são gerados, o modo em que os valores alpha são gerados (`INCREASING_ENABLE` e/ou `DECREASING_ENABLE`), a velocidade (`increasingAlphaDuration` e `decreasingAlphaDuration`) e aceleração (`increasingAlphaRampDuration` e `decreasingAlphaRampDuration`) com que uma trajetória do espaço de valores será percorrida. O parâmetro `phaseDelayDuration` permite especificar um tempo de espera, antes que os valores alpha comecem a ser gerados. Este tipo de controle é bastante útil em um ambiente *multithreading* como o Java 3D. Imagine-se, por exemplo, a situação onde Java 3D tenha iniciado o processo de renderização antes que a janela de display seja aberta. Neste caso, o usuário poderá ver a animação a

partir de um instante que não corresponda ao inicial, o que pode ser evitado estabelecendo um atraso.

O exemplo a seguir define um objeto `Alpha` que gera os valores alpha um número *infinito* de vezes, gastando 4000 milissegundos (ou 4 segundos) em cada período. Os parâmetros que não aparecem na lista recebem o valor zero, a menos do parâmetro que define o modo em que os valores alpha são gerados que é configurado com a constante `INCREASING_ENABLE`. A Figura 21 mostra o gráfico relativo a este exemplo.

```
Alpha valoresalpha = new Alpha(-1, 4000);
```

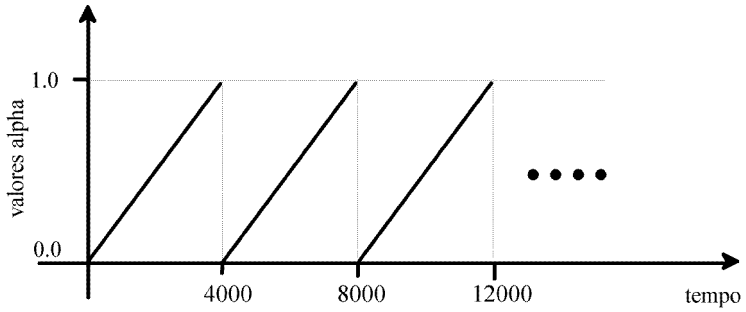

Figura 21: Gráfico Alpha com infinitos ciclos.

O grande número de parâmetros que podem ser especificados na construção de um objeto `Alpha` oferece um maior controle do processo de interpolação. No entanto, isto tem o custo do programador necessitar calibrar corretamente os valores de tais parâmetros para obter um resultado mais próximo possível do desejado, o que faz este método ser um tanto quanto trabalhoso. Por outro lado, um objeto `Alpha` pode ser compartilhado, permitindo economizar trabalho e memória. Uma apresentação detalhada da lista dos parâmetros de `Alpha` pode ser encontrada em [26] e [29].

4.5.2 Mapeamento de `Alpha` para o espaço de valores

O programador pode projetar seus próprios interpoladores usando os valores alpha para animar objetos do mundo virtual. No entanto, Java 3D disponibiliza classes de *behaviors* pré-definidos, denominados `Interpolators`, que atende a maioria das aplicações (`ColorInterpolator`, `PositionInterpolator`, `RotationInterpolator` entre outros). O procedimento usado para adicionar um objeto `Interpolator` segue um padrão básico, parecido com o apresentado na Seção 4.4.1, que independe do tipo de interpolador a ser usado:

1. Criar o objeto a ser interpolado, o objeto alvo, e configurar os bits de capacidades necessários;
2. Criar um objeto `Alpha` que descreve como gerar os valores alpha;
3. Criar um objeto `Interpolator` usando `Alpha` e o objeto alvo;
4. Atribuir um `scheduling bound` ao `Interpolator`;
5. Anexar o `Interpolator` ao grafo de cenas;

O exemplo apresentado a seguir (adaptado de [30]) usa a classe `RotationInterpolator` para criar uma animação. Esta animação consiste em rotacionar um cubo um número *infinito* de vezes, onde cada rotação gasta 4 segundos.

```
public BranchGroup ConstroiContentBranch () {
// Cria a raiz do grafo content branch
BranchGroup objRoot = new BranchGroup();

// Passo 1
```

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

// Passo 2
Alpha vAlpha = new Alpha(-1, 4000);

// Passo 3
RotationInterpolator r = new RotationInterpolator(vAlpha, alvo);

// Passo 4
BoundingSphere bounds = new BoundingSphere();
r.setSchedulingBounds(bounds);

// Passo 5
objRoot.addChild(alvo);
alvo.addChild(new ColorCube(0.4));
objRoot.addChild(r);

return objRoot;
}
```

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. Para maiores detalhes consultar [30].

4.6 Conclusão

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

As dificuldades encontradas na construção da geometria de modelos mais complexos são resolvidas usando programas de modelagem. A VRML, que é essencialmente um formato de arquivo para cenas 3D, é uma excelente escolha para esta finalidade. Aguarda-se para um futuro próximo uma maior compatibilidade entre estas duas tecnologias.

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) [31], 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* [32]. 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.

Apesar de ser geralmente vista como uma espécie de HTML tridimensional, a VRML é muito mais que isso [33], sendo capaz de criar ricos ambientes 3D interativos e se conectar a programas externos que podem realizar processamentos sofisticados, úteis em áreas como a visualização científica.

A visualização de uma cena VRML (também chamada mundo VRML — *VRML world*) 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 (extensão .wrl), o "desenha" em uma janela e provê uma interface para que o usuário possa andar pelo ambiente criado e interagir com objetos dele.

5.1 Estrutura hierárquica da cena

O paradigma para a criação de mundos VRML 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. Nós são os componentes fundamentais de uma cena VRML porque esta é constituída a partir da disposição, combinação e interação entre os nós.

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 [34].

Agrupamento. Como já comentado, nós de 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:

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 (clica com o mouse sobre eles, por exemplo). É o nó que cria *links* com outros mundos VRML, páginas HTML, ou qualquer outro tipo de documento presente no referido URL. Um `Anchor` é definido com os seguintes valores default para seus parâmetros:

```
Anchor {
   # objetos da cena que contém hyperlinks para outros arquivos.
   children     [ ]
   description  " "
   parameter    [ ]
   url          [ ]  # url do arquivo a ser carregado
   bboxCenter   0 0 0
   bboxSize    -1 -1 -1  }
```

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. Os seguintes valores default são definidos:

```
Transform  {
   bboxCenter           0 0 0
   bboxSize            -1 -1 -1
   translation          0 0 0
   rotation             0 0 1
   scale                1 1 1
   scaleOrientation     0 0 1
   center               0 0 0
   # nós filhos, que são afetados pelas transformações especificadas neste nó.
   children             [ ]
}
```

Group. É um nó de agrupamento que contém um número qualquer de filhos. Ele é equivalente ao nó `Transform` sem os campos de transformação. A diferença básica entre o `Group` e o `Transform` é que o primeiro é usado quando se deseja criar um novo objeto constituído da combinação de outros. Quando se deseja agrupar espacialmente os objetos, ou seja, posicioná-los em certa região da cena, usa-se o nó `Transform`.

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 são:

Box. Este nó geométrico representa um sólido retangular (uma "caixa") centrado na origem (0, 0, 0) do sistema de coordenadas local e alinhado com os eixos cartesianos. O campo `size` representa as dimensões do sólido nos eixos *x*, *y* e *z*.

```
Box { size  2 2 2 }
```

Cone. Representa um cone centrado na origem do sistema local de coordenadas cujo eixo central está alinhado com o eixo y.

Cylinder. Define um cilindro centrado na origem do sistema de coordenadas e com eixo central ao longo do eixo y.

Sphere. Especifica uma esfera centrada na origem.

Text. Desenha uma ou mais *strings* de texto em um estilo específico.

IndexedLineSet e IndexedFaceSet. Estes nós descrevem, respectivamente, objetos 3D a partir de segmentos de reta e polígonos cujos vértices estão localizados em coordenadas dadas.

Appearance. Este nó 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). Este tipo de nó é chamado *bindable* porque apenas um pode estar ativo na cena.

Iluminação. Luzes em VRML não são como luzes no mundo real. No mundo real, luzes são objetos físicos que emitem luz e que podem ser vistos, assim como a luz por ele emitida. Em VRML, um nó de iluminação é descrito como parte do mundo mas não cria automaticamente uma geometria para representá-lo. Para que uma fonte de luz em uma cena seja um objeto visível é necessário criar uma geometria e colocá-la em um determinado local na cena. Existem três tipos de nós de iluminação em VRML:

DirectionalLight. Este nó define uma fonte de luz direcional com raios paralelos.

PointLight. Define uma fonte de luz pontual em um local 3D fixo. Este tipo de fonte ilumina igualmente em todas as direções.

SpotLight. Define um 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. Essa técnica torna o arquivo menor e mais fácil de ler, diminuindo o tempo necessário para carregar a cena.

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{
   ambientIntensity 0
   attenuation 1 0 0
   color 1 1 1        # luz branca
```

```
   intensity .2
   location 0 0 2     # posição da fonte
   on TRUE            # está "ligada"
   radius 20 }

Transform {
   # Um cone é definido e posicionado na origem (valor default = 0 0 0)
   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. Estes parâmetros diferenciam um nó de outros de mesmo tipo (por exemplo, o raio de uma esfera a diferencia de outra). Há dois tipos de parâmetros possíveis: campos (*fields*) e eventos (*events*). Campos podem ser modificáveis (*exposedFields*) ou não (*fields*). Um *exposedField* é, na verdade, uma composição de um evento de entrada, um evento de saída e o valor do campo (*exposedField = eventIn + field + eventOut*).

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 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 seguir alguns sensores são descritos:

TimeSensor. O nó `TimeSensor` gera eventos como passos de tempo e em conjunto com interpoladores pode produzir animações.

```
TimeSensor {
      cycleInterval     1
      enabled           TRUE
```

```
      loop              FALSE
      startTime         0
      stopTime          0 }
```

O campo `cycleInterval` determina a duração, em segundos, de cada ciclo, devendo ser maior que zero. Se o campo `enabled` for `TRUE` os eventos relacionados ao tempo são gerados caso outras condições sejam encontradas e se for `FALSE` os eventos relacionados ao tempo não são gerados. O campo `loop` indica que o sensor de tempo deve repetir indefinidamente ou deve parar depois de um ciclo. O campo `startTime` indica quando a geração de eventos se inicia e o campo `stopTime` quando a geração de eventos pára.

TouchSensor. O nó `TouchSensor` detecta quando um objeto do grupo do seu pai é ativado (clique do mouse, por exemplo). Esse sensor gera um evento de saída chamado `touchTime` que pode disparar um `TimeSensor`, iniciando uma animação.

```
TouchSensor {
      enabled TRUE }
```

O campo `enabled` indica se o sensor está ou não habilitado para enviar e receber eventos.

ProximitySensor. O nó `ProximitySensor` gera eventos quando o usuário entra, sai e/ou se move em uma região do espaço. O sensor de proximidade é habilitado ou desabilitado pelo envio de um evento `enabled` com o valor `TRUE` ou `FALSE`.

VisibilitySensor. O nó `VisibilitySensor` detecta quando certa parte do mundo (área ou objeto específico) torna-se visível ao usuário. Quando a área está visível, o sensor pode ativar um procedimento ou animação.

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. O nó `PositionInterpolator` permite realizar uma animação *keyframe* em uma localização no espaço 3D. Exemplo:

```
PositionInterpolator {
      key      [0, .5, 1]                  # instantes de tempo
      keyValue [ 0 0 0, 0 10 0, 0 0 0 ] }  # valores-chave associados aos instantes
```

CoordinateInterpolator, ColorInterpolator, OrientationInterpolator. De forma similar interpolam, respectivamente, uma lista de coordenadas, valores de cor e valores de rotação.

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 22) 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.

A seguir é mostrado um exemplo de utilização de sensores e interpoladores [33].

```
#VRML V2.0 utf8

# Uso de sensores e interpoladores.
# Quando a bola for tocada (clique do mouse sobre ela) o texto irá mover-se na
# horizontal e quando a caixa for tocada o texto irá mover-se na vertical.

Viewpoint { position 0 0 50 } # observador
```

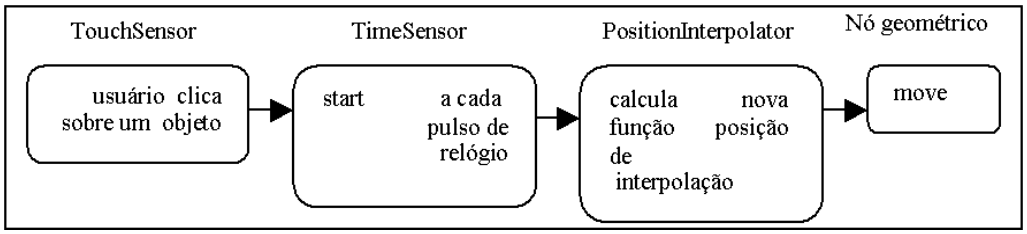

Figura 22: Roteamento de eventos em uma interação típica de VRML.

```
Group {
      children [
            Transform {
                  translation -4 8 0
                  children [
                        Shape {
                              geometry Sphere { radius 1.5 } # bola
                              appearance Appearance {
                                    material Material {
                                          diffuseColor .73 .56 .56
                                    }
                              }
                        }
                        DEF STOUCH TouchSensor{} # sensor da bola
                  ]
            }
            Transform {
                  translation 4 8 0
                  children [
                        Shape {
                              geometry Box { size 2 2 2 } # caixa
                              appearance Appearance {
                                    material Material {
                                          diffuseColor 0 1 0
                                    }
                              }
                        }
                        DEF BTOUCH TouchSensor{} # sensor da caixa
                  ]
            }
            # Sensores de tempo
            DEF XTIMERH TimeSensor { cycleInterval 2 }
            DEF XTIMERV TimeSensor { cycleInterval 2 }

            # Interpoladores
            # Horizontal
            DEF ANIMAH PositionInterpolator {
                  key [ 0, .25, .5, .75, 1 ]
                  keyValue [ 0 0 0, 8 0 0, 16 0 0, -8 0 0, 0 0 0 ]
            }
            # Vertical
```

```
            DEF ANIMAV PositionInterpolator {
                  key [ 0, .25, .5, .75, 1 ]
                  keyValue [ 0 0 0, 0 -8 0, 0 -16 0, 0 -8 0, 0 0 0 ]
            }

            # Texto
            DEF SFORM Transform {
                  children Shape {
                        geometry Text {
                              string ["Virtual" ]
                              fontStyle FontStyle {
                                    style "BOLD"
                                    justify "MIDDLE"
                              }
                              length[7]
                              maxExtent 20
                        }
                  }
            }
      ]
}

# Toque na esfera inicia relógio
ROUTE STOUCH.touchTime TO XTIMERH.set_startTime
# Valores do relógio entram no interpolador horizontal
ROUTE XTIMERH.fraction_changed TO ANIMAH.set_fraction
# Valores gerados pelo interpolador transladam texto horizontalmente
ROUTE ANIMAH.value_changed TO SFORM.set_translation
# Procedimento similar para a caixa
ROUTE BTOUCH.touchTime TO XTIMERV.set_startTime
ROUTE XTIMERV.fraction_changed TO ANIMAV.set_fraction
ROUTE ANIMAV.value_changed TO SFORM.set_translation
```

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.

```
Script {
      ExposedField MFString url     [name.class ] # url do programa
      Field SFBool mustEvaluate     FALSE # browser atrasa o envio dos eventos ao
                                          # script até serem necessários ao browser
      Field SFBool DirectOutput     FALSE # script tem acesso a informação via seus eventos
}
```

A seguir é dado um exemplo utilizando o nó `Script`, cujo roteamento é representado graficamente na Figura 23.

```
#VRML V2.0 utf8
```

```
# Este exemplo define um cubo e uma esfera. O cubo muda de cor com um toque. A esfera
# está continuamente rotacionando a menos que o cubo tenha mais de 50% de cor vermelha

Group {
      children [
            # Esfera
            DEF Sph Transform {
                  children Shape {
                        geometry Sphere {}
                        appearance Appearance {
                              material Material { diffuseColor 1 0 0 }
                        }
                  }
            }
      Transform {
                  translation -2.4 .2 1
                  rotation 0 1 1 .9
                  children [
                        # Cubo
                        Shape {
                                    geometry Box {}
                                    appearance Appearance {
                                          material DEF MATERIAL Material {}
                                    }
                        }
                        DEF TS TouchSensor {} # sensor associado ao cubo
                  ]
            }
            # Arquivo script associado
            DEF SC Script {
                  url "extouchcube.class"
                  field SFColor currentColor 0 0 0
                  eventIn SFColor colorIn
                  eventOut SFBool isRed
            }
      ]
}

DEF myColor ColorInterpolator {
      keys [0.0, 0.3, 0.6, 1.0 ]
      # red, green, red, blue
      values [1 0 0 , 0 1 0, 1 0 0, 0 0 1]
}

DEF myClock TimeSensor {
      cycleInterval 10  # 10 segundos de animação red -> green -> red -> blue
}

DEF XTIMER TimeSensor {
      loop TRUE
      cycleInterval  5
}
```

```
DEF ENGINE OrientationInterpolator {
      keys [ 0, .5, 1]
      values [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28]
}

# Toque no cubo inicia alteração de cor no mesmo
# sensor -> relógio -> interpolador -> cor do cubo
ROUTE TS.touchTime TO myClock.set_startTime
ROUTE myClock.fraction TO myColor.set_fraction
ROUTE myColor.value_changed TO MATERIAL.diffuseColor
# Alteração de cor do cubo enviada para Script
ROUTE myColor.value_changed TO SC.colorIn
# Evento gerado pelo nó Script habilita ou desabilita relógio
ROUTE SC.isRed TO XTIMER.enabled
# Relógio determina rotação da esfera
# relógio -> interpolador -> movimento da esfera
ROUTE XTIMER.fraction TO ENGINE.set_fraction
ROUTE ENGINE.value_changed TO Sph.set_rotation
```

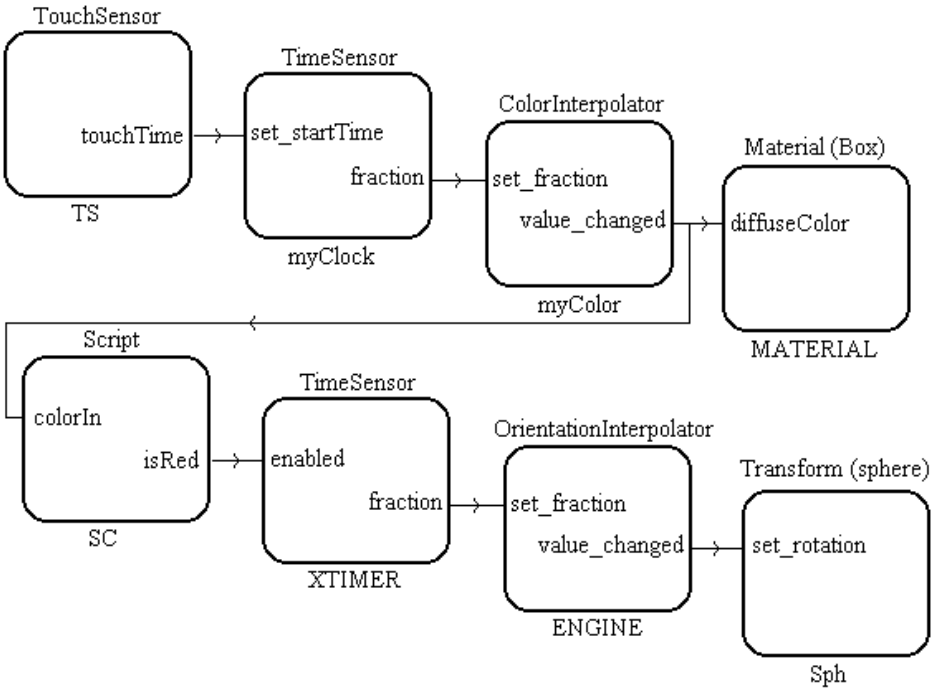

Figura 23: Roteamento de eventos para o exemplo usando o nó `Script`.

O programa Java que implementa a funcionalidade do nó `Script` do exemplo acima (`extouchcube.java`) é mostrado a seguir.

```
import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class extouchcube extends Script
```

```
{
   // declarações dos campos e eventOuts usados
   private SFColor currentColor;
   private SFBool isRed;

   public void initialize() {
       // alocação dos campos e eventos
       currentColor = (SFColor) getField("currentColor");
       isRed = (SFBool) getEventOut("isRed");
       }

  // método chamado no recebimento do evento colorIn
  public void processEvent(Event e) {
       currentColor.setValue((ConstSFColor)e.getValue());
       }

  public void eventsProcessed() {
       // componente vermelho da cor maior que 50% -> envio de isRed = FALSE,
       // o que desabilita o relógio que controla a rotação da esfera da cena
       if (currentColor.getRed() >= 0.5)
            isRed.setValue(false);
       else isRed.setValue(true);
       }
}
```

5.6 EAI (*External Authoring Interface*)

A EAI [35] 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. A EAI 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.

De uma maneira geral, pode-se dizer que o nó `Script` é adequado quando se deseja adicionar um comportamento individual a objetos da cena. A EAI, por sua vez, é mais adequada para a criação de sistemas multimídia complexos, onde VRML é apenas um meio de apresentação. O blaxxun Contact [36] é um exemplo de sistema que usa a EAI. Ele é um cliente para comunicação multimídia que oferece recursos como suporte a VRML, *chats*, *message boards*, interação com agentes, etc. Vários usuários "imersos" no mesmo mundo virtual VRML podem se comunicar e interagir por meio de recursos implementados com a EAI.

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.

Para receber eventos da cena, a seguinte seqüência de passos deve ser realizada [37]:

1. Criar uma classe observadora dos eventos, que implementa a interface `EventOutObserver`. Deve-se implementar o método `callback` desta classe, que realiza o processamento dos eventos.

2. Criar uma instância dessa classe no código do applet.

3. Obter a referência do nó gerador do evento e do `EventOut` desejado. O método `advise` é usado para indicar ao browser que há o interesse em detectar este evento.

A seqüência acima é traduzida no seguinte código (adaptado de [37]).

```
// 1. Cria classe observadora
class MeuObservador implements EventOutObserver {
    public void callback (EventOut ev, double timeStamp, Object o)
       {
       // Processamento sobre o evento...
       }
}

class MinhaClasseVRML {
     public void MinhaClasseVRML ( )
        {
        // Conectando ao browser VRML
        Browser b = Browser.getBrowser();

        // 2. Criando uma instância da classe observadora
        MeuObservador obs = new MeuObservador();

        // 3. Obtendo a referência do nó gerador do evento e
        // registrando o evento de interesse.
        // "NodeName" deve ser o nome do nó na cena VRML (DEF NodeName)
        b.getNode( "NodeName" ).getEventOut( "fraction_changed" ).advise(obs, null);

        // ...
        }
}
```

Para enviar eventos para a cena, o procedimento é similar, consistindo nos seguintes passos:

1. Obter a referência do nó para o qual se deseja enviar o evento.
2. Obter a referência do `EventIn` deste nó.
3. Enviar os valores desejados por meio do método `setValue`.

Traduzindo isto em código, considere a esfera a seguir, que será movimentada através do campo `translation` do nó `Transform` (adaptado de [37]).

```
#VRML v2.0 utf8
DEF move_esfera Transform {
    translation 2 0 -1
    children Shape {
             geometry Sphere { radius 1.5 }
    }
}
```

O seguinte applet colocará a esfera em uma nova posição.

```
// Importando algumas classes da EAI
import vrml.external.*;
import vrml.external.field.*;
import java.applet.*;

public class MeuApplet extends Applet {
   public void start ( ) {
       Browser b = b.getBrowser();
```

```
        // 1. Obtendo referência do nó Transform
        Node aTransform = b.getNode( "move_esfera" );

        // 2. Obtendo referência do EventIn "set_translation"
        EventInSFVec3f tx =
            (EventInSFVec3f) aTransform.getEventIn( "set_translation" );

        // 3. Enviando nova posição para o EventIn obtido acima
        float nova_posicao[] = {0, 1, 0.5f};
        tx.setValue(nova_posicao);
    }
}
```

Recentemente, a EAI e os nós `Script` foram englobados no que está sendo chamado de *Scene Authoring Interface* [38], usada para a manipulação de objetos de uma cena. Esta interface é acessível por meio dos nós `Script` da cena VRML ou por meio de aplicações externas ao browser (EAI).

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 [39] 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*) [40]. XML é um padrão que tem sido bastante usado para a definição de estruturas de dados transmitidos pela Internet. 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. Uma linguagem de marcação define a maneira de descrever a informação em uma certa classe de documentos. A HTML, por exemplo, é uma linguagem de marcação que define a estrutura de documentos hipertexto. 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 [41] e a H-Anim (*Humanoid Animation*) [42]. A GeoVRML é uma extensão para a representação, usando VRML, de dados geográficos, tais como mapas e modelos 3D de terrenos. A H-Anim é uma extensão para modelagem e animação de humanóides.

Como já comentado, a X3D ainda está em fase de elaboração, mas o exemplo a seguir ilustra como descrever uma esfera em X3D baseado no seu estado atual de implementação (é muito parecido com o arquivo em VRML 97, apenas trocando os nós e parâmetros por *tags*). O arquivo DTD (*Document Type Definition*) definido no início é a declaração em XML da sintaxe do X3D, onde estão definidos formalmente os significados das *tags* uilizadas.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE X3D PUBLIC
"http://www.web3D.org/TaskGroups/x3d/translation/x3d-compromise.dtd"
"file://localhost/C:/www.web3D.org/TaskGroups/x3d/translation/x3d-compromise.dtd"
```

```
[
 <!ENTITY % Vrml97Profile  "INCLUDE">
 <!ENTITY % CoreProfile    "IGNORE">
 <!ENTITY % X3dExtensions  "IGNORE">
 <!ENTITY % GeoVrmlProfile "INCLUDE">
 <!ENTITY % HAnimProfile   "INCLUDE">
]>
<X3D>
  <Scene>
    <Transform>
      <children>
        <NavigationInfo headlight="false" avatarSize=" 0.25 1.6 0.75"
                                          type="&#34;EXAMINE&#34;"/>
        <DirectionalLight/>
        <Transform translation="3.0 0.0 1.0">
          <children>
            <Shape>
              <geometry>
                <Sphere radius="2.3"/>
              </geometry>
              <appearance>
                <Appearance>
                  <material>
                    <Material diffuseColor="1.0 0.0 0.0"/>
                  </material>
                </Appearance>
              </appearance>
            </Shape>
          </children>
        </Transform>
      </children>
    </Transform>
  </Scene>
</X3D>
```

6 Conclusão

Este curso mostrou uma visão geral de algumas tecnologias livres (ou abertas) relacionadas à Computação Gráfica, adotando uma abordagem *bottom-up*, partindo da programação de mais baixo nível 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 de Computação Gráfica.

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 conectados via nó `Script` ou EAI. 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.

O que há de comum entre todas as tecnologias apresentadas (além da relação com a Computação Gráfica) e motivou a elaboração deste curso é a filosofia de desenvolvimento das mesmas. Todas estas tecnologias são baseadas nas idéias de

software livre, de forma mais ampla (como X ou VRML, desenvolvidos por consórcios de empresas e grupos acadêmicos) ou menos rigorosa (software proprietário, mas gratuito e de código aberto, como OpenGL e Java 3D). A adoção destas filosofias, além de aumentar a aceitação dos produtos, garantem maior confiabilidade e "sobrevivência" dos mesmos, pois a comunidade de usuários contribui ativamente em seu desenvolvimento, identificando erros e promovendo a manutenção dos mesmos. Além disso, a diversidade de interesses dos grupos desenvolvedores impede a criação de monopólios. Por todos estes motivos, acreditamos em software livre e incentivamos seu uso e desenvolvimento.

Referências

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

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

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

[4] The Open Group. *X Window System, Version 11, Release 6.4: Release Notes.* Cambridge, Open Group, 1998.

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

[6] Tanembaum, A. S. *Computer Networks.* Upper Saddle River, Prentice Hall, 1996.

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

[8] Packard, Keith. *The X Selection Mechanism*.

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

[10] *Open Group Desktop Technologies — Motif*. `http://www.camb.opengroup.org/tech/desktop/motif/`.

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

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

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

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

[15] Theisen, T. *Ghostview and GSview*. `http://www.cs.wisc.edu/~ghost/`

[16] *The F? Virtual Window Manager home page*. `http://www.fvwm.org/`

[17] Gettys, J. *The Two Edged Sword*. Disponível em `http://freshmeat.net/editorials/`

[18] *GNOME — the GNU Network Object Model Environment.* `http://www.gnome.org/`

[19] Peterson, C. D. *Athena Widget Set — C Language Interface*. Maynard, Digital Equipment Corporation/X Consortium, 1994.

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

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

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

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

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

[25] Angel, E. *Interactive computer graphics: a top-down approach with OpenGL*. Addison Wesley Longman, Inc., 1997.

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

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

[28] Sowizral, H. A. and Deering, M. The Java 3D API and Virtual Reality. *IEEE Computer Graphics and Applications*, 19 (3): 12-15, May/June 1999.

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

[30] *Getting Started with Java 3D^{TM} API*. Tutorial preparado para Sun Microsystems por K Computing, 1999. Disponível em `http://sun.com/desktop/java3d/collateral`

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

[32] *Web 3D Consortium*. `http://www.web3d.org`

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

[34] Hartman, J. and Wernecke, J. *The VRML 2.0 Handbook — Building Moving Worlds on the Web*. Addison-Wesley, 1996.

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

[36] blaxxun interactive. *blaxxun Contact 4.4*. 2000. Disponível em `http://www.blaxxun.com/products/contact`

[37] Roehl, B. et al. *Late Night VRML 2.0 with Java*. Ziff-Davis Press, 1997.

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

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

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

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

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

Agradecimentos. Agradecemos às instituições a que estamos vinculados pelo apoio e aos órgãos que nos financiam: FAPESP, CAPES e CNPq.