Programação Gráfica 3D com OpenGL, Open Inventor e Java 3D

ALESSANDRO L. BICHO [1]
LUIZ GONZAGA DA SILVEIRA JR [1]
ADAILTON J. A. DA CRUZ [1]
ALBERTO B. RAPOSO[2]

[1]DCA - FEEC - Unicamp
Rua Albert Einstein, 400
Caixa Postal : 6101
13083-970 - Campinas - SP
e-mails: {bicho,gonzaga,ajcruz}@dca.fee.unicamp.br

[2]TECGRAF - Depto. de Informática - PUC-Rio
R. Marquês de São Vicente, 225
22453-900 - Rio de Janeiro - RJ
e-mail: abraposo@tecgraf.puc-rio.br

Resumo: Este tutorial aborda três tecnologias de domínio público nas áreas de computação gráfica e animação por computador, enfocando ferramentas de programação para apoio ao desenvolvimento de sistemas gráficos interativos. A primeira parte aborda a OpenGL, uma tecnologia de menor nível de abstração para a geração de formas geométricas e imagens. Na segunda parte é estudado a Open Inventor, uma toolkit orientada a objetos construída sobre a OpenGL, contemplando funções mais sofisticadas, como gerenciamento de janelas e manipulação de eventos. Finalmente, será apresentada a Java 3D, que é fortemente inspirada na Open Inventor. Ela é a biblioteca padrão da linguagem Java para a criação de programas com imagens tridimensionais, animação e interação com o usuário. Cabe salientar que este trabalho é de caráter introdutório, com referências a estudos avançados.

Palavras-chaves: computação gráfica, animação por computador, realidade virtual, interação homem-computador, OpenGL, Open Inventor, Java 3D.

1 Introdução

Este tutorial é uma introdução à programação gráfica 3D, apresentando três tecnologias que, embora com níveis de abstração e funcionalidades diferentes, representam uma visão geral do variado espectro de ferramentas existentes.

A primeira biblioteca apresentada é a OpenGL, que é uma API (*Application Programmer's Interface*) aberta para o desenvolvimento de aplicações gráficas tridimensionais que pode ser incorporada a qualquer sistema de janelas (*MS Windows*, *X Window*, etc.). A independência de sistema de janelas é, ao mesmo tempo, uma vantagem e um fator limitante da OpenGL, já que, por causa disso, ele não oferece as funcionalidades de gerenciamento de janelas e manipulação de eventos.

A Open Inventor surgiu para preencher essas e outras lacunas da OpenGL. A Open Inventor é uma *toolkit* orientada a objetos, concentrando suas atenções na construção e estruturação de cenas 3D, utilizando a OpenGL para a renderização [1] das cenas. A Open Inventor é implementada em C/C++ e oferece uma interface de programação de mais alto nível de abstração que a OpenGL.

Java 3D, por sua vez, foi desenvolvida para ser a API da linguagem Java para aplicações envolvendo gráficos e imagens tridimensionais. Da mesma forma que a Open Inventor, Java 3D se concentra na estruturação das cenas, deixando a renderização a cargo de bibliotecas gráficas de mais baixo nível, como a OpenGL e a Direct3D. O

[1]Renderização é a obtenção da imagem a partir do modelo. É neste processo que se adiciona, por exemplo, sombreamento, cores e iluminação à cena.

formato de estruturação da cena em Java 3D (grafo de cena) é, por sinal, inspirado no formato definido para a Open Inventor. A grande diferença da Java 3D em relação a Open Inventor é o fato da primeira ser programada em uma linguagem Java e a segunda em C/C++, mas ambas utilizam OpenGL para renderização da cena.

Além de serem tecnologias para a programação de aplicações gráficas 3D e terem os elos apontados acima, essas três tecnologias também têm em comum o fato de estarem disponíveis gratuitamente, facilitando seu uso e manutenção por uma vasta gama de usuários.

2 OpenGL[2]

Padrões gráficos, como o GKS (*Graphical Kernel System* [ANSI, 1985a]) e o PHIGS (*Programmer's Hierarchical Interactive Graphics System* [ANSI, 1985b]), 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 [Segal, 1994].

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 o desempenho gráfico do hardware e sem sacrificar o controle sobre as operações de hardware [Segal, 1996].

Atualmente, a OpenGL ("GL" significa *Graphics Library*) é uma API de grande utilização no desenvolvimento de aplicações em computação gráfica [Neider, 1993]. Este padrão é o sucessor da biblioteca gráfica conhecida como IRIS GL, desenvolvida pela *Silicon Graphics* como uma interface gráfica independente de hardware [Kilgard, 1994]. A maioria das funcionalidades da IRIS GL foi removida ou reescrita na 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 da 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. Para tanto, a OpenGL funciona como uma máquina de estados, onde o controle de vários atributos é realizado através de um conjunto de variáveis de estado que inicialmente possuem valores *default*, podendo ser alterados caso seja necessário. Por exemplo, todo objeto será traçado com a mesma cor até que seja definido um novo valor para esta variável.

Por ser um padrão destinado somente à renderização [Segal, 1996], a OpenGL pode ser utilizada em qualquer sistema de janelas (por exemplo, *X Window System* ou *MS Windows*), aproveitando-se dos recursos disponibilizados pelos diversos hardwares gráficos existentes. No *X Window System*, ela é integrada através da GLX (*OpenGL Extension for X*), um conjunto de rotinas para criar e gerenciar um contexto de renderização da OpenGL no X [Kilgard, 1994]. Além da GLX, existem outras bibliotecas alternativas para interfaceamento no X, tais como GLUT (*OpenGL Utility Toolkit* [Kilgard, 1996]) e GTK [GTK+, 2002]. Estas bibliotecas possuem um conjunto de ferramentas que facilita a construção de programas utilizando a 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 da OpenGL e, quando necessário, apresenta algumas rotinas disponíveis na GLX e na GLUT.

2.1 Objetos geométricos

OpenGL é uma interface para aplicações gráficas que não possui rotinas de alto nível de abstração. Sendo assim, 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 um 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. Portanto, os pontos bidimensionais especificados pelo usuário são trabalhados como pontos tridimensionais, onde a coordenada z é igual a zero. Os segmentos de reta são representados por seus pontos extremos e os polígonos são áreas definidas por um conjunto

[2] Biblioteca desenvolvida pela *Silicon Graphics Inc.*

de segmentos. Na OpenGL, alguns cuidados quanto à definição de um polígono devem ser tomados: um polígono deverá ser sempre convexo e simples (não poderá haver interseção das suas arestas). A especificação de um vértice é feita através das funções `glVertex*()` [3].

Em muitas aplicações gráficas há a necessidade de definir polígonos não simples, côncavos ou com furos. Como qualquer polígono pode ser formado a partir da união de polígonos convexos, algumas rotinas mais complexas, derivadas das primitivas básicas, são fornecidas na GLU (*OpenGL Utility Library* [Neider, 1993]). Esta biblioteca utiliza somente funções padrões e está disponível em todas as implementações da OpenGL.

Para traçar um conjunto de pontos, um segmento ou um polígono, os vértices necessários para a definição destas primitivas são agrupados entre as chamadas das funções `glBegin()` e `glEnd()`. Pode-se adicionar também informações a um vértice, como uma cor, um vetor normal ou uma coordenada para a textura, utilizando um número restrito de funções da OpenGL válidas entre o par `glBegin()` e `glEnd()`. É importante salientar que a restrição quanto à utilização é apenas para as rotinas da OpenGL; outras estruturas de programação poderão ser utilizadas normalmente entre o par `glBegin()` e `glEnd()`. O argumento da função `glBegin()` indicará a ordem como serão associados os vértices, conforme ilustrado na Figura 1.

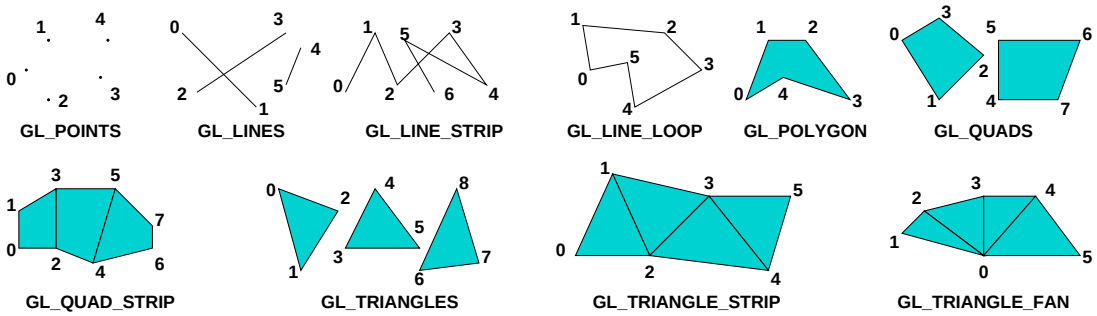

Figura 1: Primitivas geométricas da OpenGL.

Na OpenGL, uma primitiva pode ser traçada de diferentes maneiras, conforme a ordem selecionada e o conjunto de vértices definido. O trecho de código a seguir apresenta um exemplo do traçado de uma circunferência.

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

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

O exemplo acima não é o modo mais eficiente para traçar uma circunferência, especialmente se esta primitiva for utilizada várias vezes. Neste caso, o desempenho ficará comprometido porque há o cálculo do ângulo e a execução das funções `sin()` e `cos()` para cada vértice, além do *overhead* do *loop*. Poderíamos solucionar este problema calculando as coordenadas dos vértices uma única vez e armazenando-os em uma tabela, ou utilizando uma rotina da GLU/GLUT, ou ainda criando uma *display list* (lista de instruções).

A *display list* é uma maneira eficiente de armazenar um grupo de rotinas da OpenGL que serão executadas posteriormente. Quando uma *display list* é invocada, as rotinas são executadas na ordem em que elas foram originalmente armazenadas. A maioria das rotinas da OpenGL pode ser armazenada em uma *display list*, exceptuando aquelas com passagem de parâmetros por referência ou que retornem um valor. A restrição é adotada porque uma

[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.

lista poderá, por exemplo, ser executada fora do escopo de onde os parâmetros foram originalmente definidos. Estas rotinas, aquelas que não pertencerem à OpenGL e as variáveis serão avaliadas no momento da compilação da lista, sendo substituídas por seus respectivos 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 reescrito 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. Para ser executado o conteúdo de uma lista no momento da compilação, é utilizado o atributo `GL_COMPILE_AND_EXECUTE`. Quando for necessário, a lista poderá ser invocada através da rotina `glCallList()`.

Uma outra 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.

2.2 Visualização

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

Transformações são funções que mapeiam um ponto (ou vetor) em um outro ponto (ou vetor). Na OpenGL, elas são implementadas através de matrizes. Estas matrizes podem descrever uma modelagem, uma visualização ou uma projeção, dependendo do contexto. ***Clipping*** é a eliminação de objetos (ou partes de objetos) que estão situados fora do volume de visualização. O enquadramento das imagens no ***viewport*** é a operação de correspondência entre as coordenadas transformadas e os pixels da tela.

As matrizes de modelagem posicionam e orientam os objetos na cena, as matrizes de visualização determinam o posicionamento da câmera e as matrizes de projeção determinam o volume de visualização (análogo à escolha da lente para uma máquina fotográfica). Na 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, sendo que 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 por uma única matriz de transformação.

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

A pilha de matrizes é utilizada na 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 uma seqüência de operações com 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 [Angel, 1997]. 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, esta 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 do sistema de coordenadas da OpenGL. 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_parafusos()
{
   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 em relacao ao corpo do carro
    // ...
}
```

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 na 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 2.

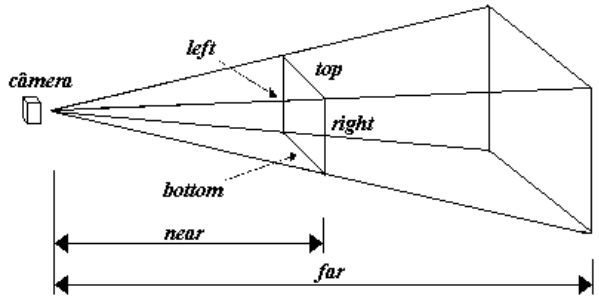

Figura 2: O volume de visualização da projeção perspectiva.

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 na 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, ilustrado na Figura 3.

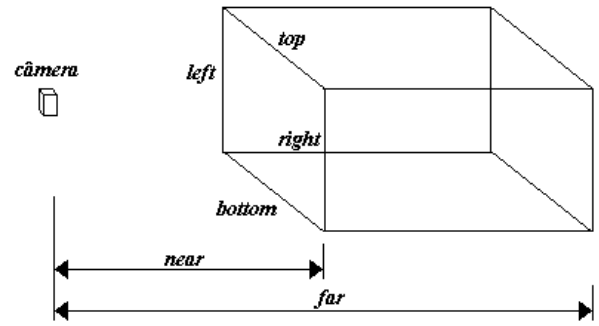

Figura 3: O volume de visualização da projeção ortogonal.

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 corresponda a mesma relação utilizada para definir a janela no volume de visualização.

Além das matrizes de transformação definidas pela 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*()`.

2.3 Cor

OpenGL possui dois modos diferentes para tratar cor: o modo RGBA e o modo indexado de cor [Neider, 1993]. 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 do último vértice definido do polígono será utilizada como padrão para toda a primitiva geométrica, ou `GL_SMOOTH`, sendo as cores para o interior do polígono calculadas a partir da interpolação das cores definidas para os vértices (método *Gouraud shading*) (ver Figura 4), sendo este último modo o default da OpenGL.

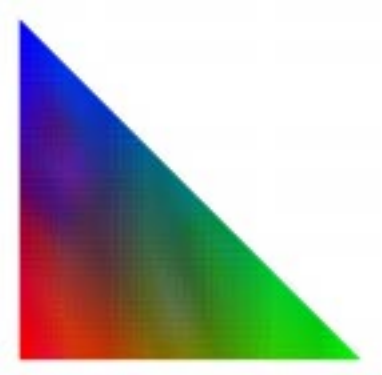

Figura 4: Um triângulo desenhado no modo *smooth*.

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 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()`, responsável pela definição das componentes primárias para um determinado índice no mapa de cores. As rotinas `glIndex*()` são usadas para selecionar o índice da cor atual no mapa de cores.

2.4 Iluminação

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

Para implementar o modelo de iluminação, a OpenGL decompõe o raio luminoso nas componentes primárias RGB. Dessa forma, a cor para uma fonte de luz é caracterizada pela percentagem 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. Para os materiais, os valores correspondem à percentagem refletida de cada componente primária. Se a componente vermelha for 1.0, a componente verde for 0.5 e a componente azul for 0.0 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 intensidades iguais 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, a OpenGL considera ainda que esta divisão será realizada para cada componente de luz do modelo de iluminação, que são:

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

Componente difusa. 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.

Componente especular. Componente de luz refletida em uma determinada direção quando esta incide sobre uma superfície, proveniente de uma direção específica. Uma superfície como um espelho 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. O modelo de Phong é utilizado para o cálculo da reflexão especular [Angel, 1997].

No modelo de iluminação da 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. Quanto aos tipos de fonte de iluminação, a OpenGL possui:

Fontes pontuais. Fontes que irradiam energia luminosa em todas as direções.

Fontes *spots*. Fontes pontuais direcionais, isto é, têm 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, a 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.

As fontes de luz somente têm efeito nas superfícies que definiram as suas propriedades do material. Da mesma maneira que a luz, os materiais têm diferentes valores para as componentes especular, difusa e ambiente, determinando assim suas reflexões. Além destas componentes, um material pode também emitir luz própria, definida através da componente emitida. 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 concentrada em um ponto do material. 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 a OpenGL para os cálculos do modelo de iluminação.

O trecho de código a seguir ilustra a definição de um modelo de iluminação na 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 5.

```
//  Inicializa as propriedades do material
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};
//  Inicializa a posição da fonte de luz
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 de iluminação 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 5: Exemplo do modelo de iluminação da OpenGL.

2.5 Textura

Para realizarmos um mapeamento de textura na 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.

Na 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) [Segal, 1994].

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*()`, enquanto as rotinas `glTextParameter*()` determinam como a textura será organizada no fragmento a ser mapeado e como os pixels serão "filtrados" quando não há um exato casamento 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 6. Muito embora a OpenGL provê suporte para o mapeamento de texturas, este ainda é um recurso bastante limitado, pois não existem facilidades para mapear imagens de outras fontes (é necessário um programa auxiliar para converter uma imagem em uma representação aceita pela OpenGL).

```
// 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();
```

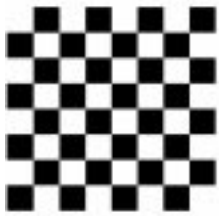

Figura 6: Exemplo do mapeamento de textura em um quadrado.

2.6 *Framebuffer*

Em uma aplicação, a área utilizada para armazenar temporariamente os dados é denominada *buffer*. Um conjunto de *buffers* de uma determinada janela de visualização ou de um determinado contexto é denominado *framebuffer*. Na OpenGL, há um conjunto de *buffers* que podem ser manipulados conforme a necessidade [Neider, 1993]:

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

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

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

***Buffer* de acumulação.** É utilizado para acumular um conjunto de imagens através de uma operação pré-especificada. A imagem resultante destas acumulações é exibida através da transferência para um *buffer* de cor. Pode ser utilizado para trabalhar com diversas técnicas como, por exemplo, *antialiasing*[5], *motion blur* (borrão) ou profundidade de campo.

3 Open Inventor[6]

Como vimos anteriormente, a OpenGL é uma biblioteca que implementa um grande e completo conjunto de algoritmos para a produção de cenas 3D, tais como remoção de linhas escondidas, iluminação, tonalização, desenho no *framebuffer*, etc. No entanto, a OpenGL foi projetada para ser independente dos sistemas de janelas e, portanto, não contempla o gerenciamento de janelas e nem a manipulação de eventos de entrada do usuário. Outras ausências na OpenGL são a possibilidade de salvar e restaurar uma cena completa e de definir objetos gráficos 3D de alto nível. Tudo isso fica a cargo da aplicação, ou seja, o programador terá que implementar. Com isso, surge a necessidade de se desenvolver uma forma de estruturação e representação de objetos 3D, além de um mecanismo para propiciar a junção entre a cena 3D e o sistema de janelas utilizado.

A *Silicon Graphics* desenvolveu originalmente a *toolkit* IRIS Inventor 1.0 para a plataforma *IRIX*[7] [Strauss, 1992] [Strauss, 1993]. Em 1994, após uma série de revisões na IRIS Inventor, a SGI lança a Open Inventor

[5] Processo empregado na redução do efeito de serrilhado em linhas e curvas quando apresentadas, por exemplo, no monitor.
[6] *Toolkit* desenvolvida pela *Silicon Graphics Inc.*
[7] Implementação do Unix para estações de trabalho da *Silicon Graphics*.

2.0 [Wernecke, 1994]. A Open Inventor contempla todas estas lacunas deixadas pela OpenGL. Ela é uma *toolkit* orientada a objetos para o desenvolvimento de aplicações interativas gráficas 3D, dispondo também de um formato de arquivo de dados 3D padrão para troca de informações entre aplicações e plataformas.

A Open Inventor consiste essencialmente de:

- um grafo de cena (nós) para representação da informação 3D,
- uma biblioteca de componentes e utilidades (*Component Library*) para auxiliar o uso da Open Inventor com sistemas de janelas e
- uma API C/C++ para a manipulação dos nós do grafos e componentes da *toolkit*.

Implementada em C/C++, a Open Inventor enfoca a criação de objetos 3D e oferece uma interface orientada a objetos com alto nível de abstração para o desenvolvimento de aplicações interativas gráficas 3D, concentrando suas atividades na construção e manipulação dos objetos. A Open Inventor oferece ainda uma biblioteca para a construção de interfaces gráficas convencionais, com um completo modelo de eventos para interação 2D e 3D, eximindo o programador de conhecer a tediosa e difícil programação com bibliotecas gráficas dos vários sistemas de janelas. A Figura 7 exibe os módulos principais da Open Inventor.

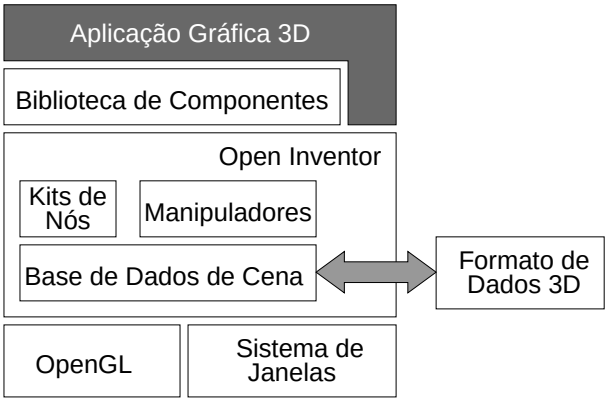

Figura 7: Arquitetura da Open Inventor.

A Open Inventor, como mostra a Figura 7, é construída sobre a OpenGL e, portanto, funciona como uma API C++ para esta biblioteca, já que a utiliza para renderização da cena 3D. O mecanismo de renderização dentro da Open Inventor realiza chamadas eficientes de rotinas da OpenGL para gerar a cena 3D e, além disso, tira proveito da eficiência desta biblioteca na renderização das cenas.

A Open Inventor é uma *toolkit* independente do sistema de janelas. Mas, ao contrário da primeira, já contempla uma biblioteca de componentes que realiza conexão com sistemas de janelas específicos. Vamos abordar aqui uma implementação da biblioteca de componentes, denominada *Xt Component Library*, que facilita a programação com a Open Inventor no *X Window System* [Wernecke, 1994].

3.1 Base de dados e grafo de cena

Uma cena na Open Inventor é uma coleção ordenada de nós (objetos) chamada *grafo de cena*. Este grafo é armazenado em uma base de dados interna da Open Inventor. O grafo e a base de dados de cena reconhecem todos os nós contidos na *toolkit* e novos nós adicionados pelo programador. A Figura 8 exibe um exemplo de grafo de cena.

Um *nó*, na Open Inventor, é um bloco de construção básico usado para criar uma cena 3D. Cada nó especifica um conjunto de campos que define uma instância de um objeto da Open Inventor como, por exemplo, formas geométricas, atributos, câmeras, luzes, transformações, etc. Nós especiais respondem diretamente a eventos do usuário e são utilizados para a manipulação direta da cena. Os nós podem ter nomes e seus campos podem ser conectados a campos de outros nós. Depois de construir o grafo, pode-se realizar algumas ações nele, tais como

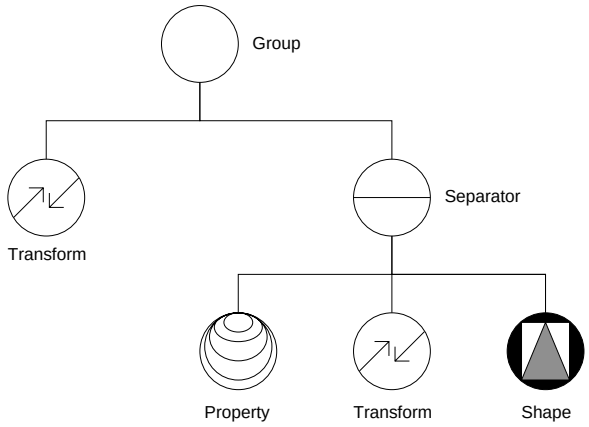

Figura 8: Exemplo de grafo de cena.

renderização, leitura/escrita, *picking*[8], busca, destaque, computação de *bounding box*[9], etc. Dentre as primitivas (nós) da base de dados incluem:

- formas geométricas (*shapes nodes*) - por exemplo, esferas, cubos, cilindros e malhas,
- propriedades (*property nodes*) - por exemplo, material, modelo de iluminação, texturas e
- grupos (*group nodes*) - por exemplo, separador, nível de detalhes e decisão.

Além destes nós básicos, outras primitivas são definidas pela Open Inventor, como *engines* e sensores. Os *engines* são objetos que podem ser conectados a outros objetos e usados em animações ou para a imposição de restrições a partes da cena. Os sensores são objetos que detectam quando algo muda na base de dados e ativam alguma função fornecida pela aplicação.

Os *kits* de nós são mecanismos especiais da Open Inventor que facilitam a criação de agrupamento de nós na base de dados. Cada *kit* é uma coleção de nós com um arranjo específico. Por exemplo, o *kit* de nós **SoShapeKit** pode conter um nó cubo (**SoCube**), permitindo a definição de um material, uma transformação geométrica e outras propriedades, quando necessário. Um outro uso para o *kit* de nós é definir objetos específicos para uma aplicação e suas semânticas dentro da aplicação.

3.2 A biblioteca de componentes e utilidades

A biblioteca de componentes e utilidades está relacionada a interação e visualização na Open Inventor, pois promove a junção entre a parte 3D e o sistema de janela, através do uso de janelas para visualização da cena 3D e componentes GUI (*Graphics User Interface*) para interface da aplicação, bem como o tratamento de eventos de entrada e saída. A biblioteca de componentes é, portanto, dependente do sistema de janela, apesar de conservar o mesmo *look and feel*[10] para as várias plataformas. No X temos a implementação da biblioteca de *Xt/Components*, que utiliza o *Xt/Motif* para geração de janelas e tratamento de eventos. Esta biblioteca inclui uma área de renderização (para visualização da cena renderizada), rotinas para inicialização de janelas e tratamento de eventos (*Main Loop*) e um conjunto de componentes usados para edição e interação com a cena ou com os objetos da cena, além da manipulação dos parâmetros de visualização.

A área de renderização aceita eventos do X, traduz estes eventos para eventos da Inventor e então os repassa para os objetos, tais como os manipuladores (tipo especial de nó que reage a eventos da GUI e podem ser editados pelo usuário). A biblioteca de componentes também contém um conjunto de editores e visualizadores, sendo

[8] *click* sobre um objeto para identificar que este pode ser selecionado.
[9] caixa envoltória sobre o objeto.
[10] aparência.

utilizados para editar os nós da cena e fornecer a visualização da cena de diversas formas. Alguns exemplos de componentes são: editor de material, editor de luz direcional, *fly viewer* [11] e o *examiner viewer* [12].

3.3 Um *tour* pela Open Inventor

A informação retida no grafo de cena, dentro da base de dados, pode ser utilizada para vários fins, no entanto, a maioria das aplicações objetivam visualizar uma imagem dos objetos 3D na tela, bem como permitir a interação do usuário com a cena. Assim, vamos construir um conjunto de exemplos que mostra um cone vermelho em uma janela. Este exemplo será gradualmente incrementado para demonstrar o uso de importantes objetos e funcionalidades da Open Inventor, tais como instanciação de objetos geométricos, transformações geométricas, agrupamento de objetos, definição de propriedades para os objetos *engines*, manipuladores e componentes.

Alô Cone

O código mostrado abaixo constrói um grafo de cena com uma câmera, uma luz, um material e um cone. Vamos segmentar este primeiro exemplo para explicar todas as suas partes (os demais exemplos serão colocados por completo).

Todo programa Open Inventor deve conter alguns *include* que indicam os protótipos das classes e rotinas da Open Inventor e bibliotecas auxiliares. Vejamos que *include* necessitamos neste exemplo:

```
#include <stdlib.h> // da linguagem C
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>
```

A inicialização da Open Inventor, com a criação da janela principal da aplicação, é feita através das seguintes linhas de código:

```
Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);
```

Observe que o `argv[0]` é passado para o método `init`, a fim de nomear a janela. Se o programa não obtiver sucesso na inicialização da Open Inventor e da criação da janela principal, o programa é abortado (retorna `NULL`), senão a janela é construída e o programa segue. Já a construção de fato da cena 3D é realizada pelas seguintes linhas de códigos:

```
   SoSeparator *root = new SoSeparator;
   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   SoMaterial *myMaterial = new SoMaterial;
   root->ref();
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);
   myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);   // RGB
   root->addChild(myMaterial);
   root->addChild(new SoCone);
```

Observe a criação de um grupo raiz, ao qual são adicionados uma câmera, um nó para definição da aparência do objeto (material de cor vermelha), uma luz tipo direcional e um cone. Assim, teremos um cone vermelho!

Agora, resta criar uma área de renderização para mostrar o grafo de cena que irá ser colocada dentro da janela principal criada anteriormente.

[11] Simula um vôo através da cena.

[12] Usa um *trackball* virtual para rotacionar o grafo da cena em torno de um ponto de interesse.

```
SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
```

Ajusta-se a câmera para visualizar a cena inteira do grafo.

```
myCamera->viewAll(root, myRenderArea->getViewportRegion());
```

Coloca-se a cena na área de renderização e altera-se o seu título usando as seguintes linhas:

```
myRenderArea->setSceneGraph(root);
myRenderArea->setTitle("Alo Cone");
myRenderArea->show();
```

Por fim, exibe-se a janela principal e coloca-se o tratador de eventos para processar os eventos de entrada provenientes do usuário da aplicação:

```
SoXt::show(myWindow);
SoXt::mainLoop();
```

O arquivo completo deste primeiro exemplo é o seguinte:

```
#include <stdlib.h> // da linguagem C
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

int main(int argc, char **argv)
{

   Widget myWindow = SoXt::init(argv[0]); // passa o nome do programa
   if (myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   SoMaterial *myMaterial = new SoMaterial;
   root->ref();
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);
   myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);   // Rgb
   root->addChild(myMaterial);
   root->addChild(new SoCone);

   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);

   myCamera->viewAll(root, myRenderArea->getViewportRegion());

   myRenderArea->setSceneGraph(root);
   myRenderArea->setTitle("Alo Cone");
   myRenderArea->show();

   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

Este exemplo inicial ilustra os passos básicos para se construir um programa usando a Open Inventor. O resultado do programa é mostrado na Figura 9.

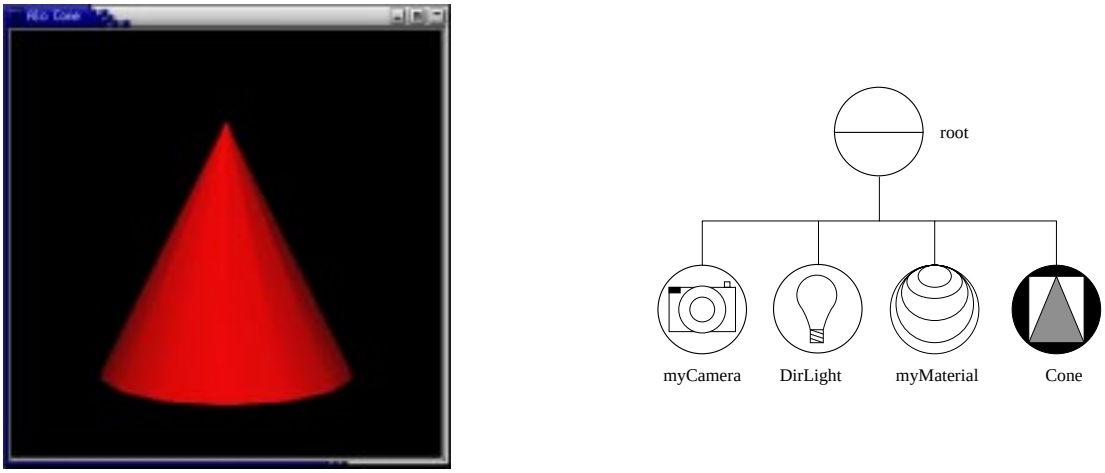

Figura 9: Cone vermelho e o grafo da cena.

Um pouco de animação

Fazendo-se o cone rotacionar continuamente na cena, ilustramos o uso de *engines* para a realização de animação de cenas 3D. Este processo é realizado através de uma ligação entre um *engine* e um campo *angle* de um nó **SoRotationXYZ** no grafo de cena. O *engine* muda o campo *angle* em resposta a mudança do *clock*, que por sua vez causa uma rotação contínua no cone.

```
#include <stdlib.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/engines/SoElapsedTime.h>     // novidade
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoRotationXYZ.h>       // novidade
#include <Inventor/nodes/SoSeparator.h>

int
main(int argc, char **argv)
{
   // Inicializa o Inventor e o Xt
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();
   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);

   // Esta tranformacao e' modificada para rotacionar o cone
```

```
    SoRotationXYZ *myRotXYZ = new SoRotationXYZ;
    root->addChild(myRotXYZ);

    SoMaterial *myMaterial = new SoMaterial;
    myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);   // rgb
    root->addChild(myMaterial);
    root->addChild(new SoCone);

    // Um engine rotaciona o objeto. A saida do myCounter
    // e' o tempo em segundos desde o inicio do programa.
    // Conecta-se esta saida ao campo angle do myRotXYZ
    myRotXYZ->axis = SoRotationXYZ::X;     // rotacao em torno de X
    SoElapsedTime *myCounter = new SoElapsedTime;
    myRotXYZ->angle.connectFrom(&myCounter->timeOut);

    SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
    myCamera->viewAll(root, myRenderArea->getViewportRegion());
    myRenderArea->setSceneGraph(root);
    myRenderArea->setTitle(" Exemplo de Engine");
    myRenderArea->show();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

O resultado do programa é mostrado na Figura 10. Observe que ao executar este programa obtém-se um resultado muito melhor, pois a Figura 10 infelizmente não consegue mostrar a animação resultante!

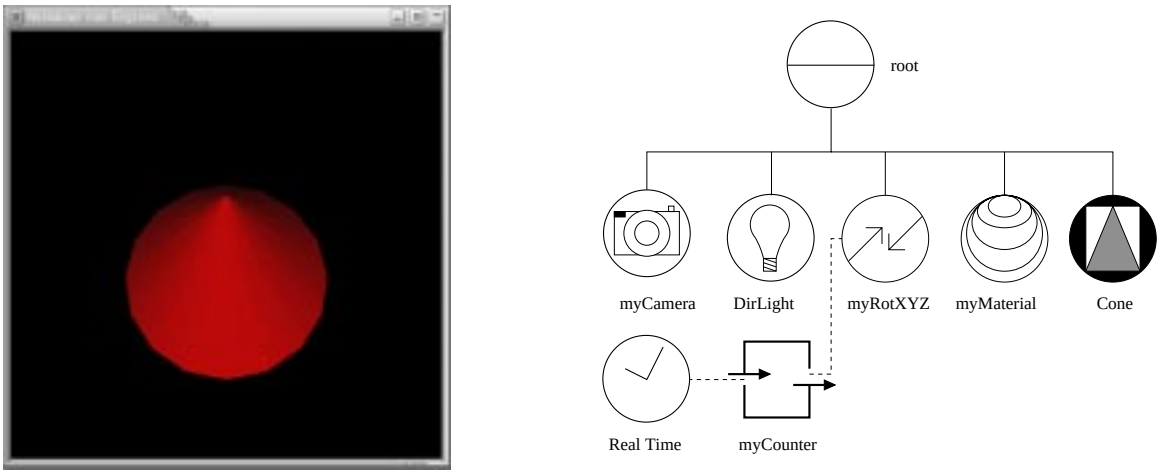

Figura 10: Exemplo de *engines* e o grafo da cena.

Manipulação direta 3D

Em aplicações gráficas interativas 3D é imprescindível a existência de mecanismos/ferramentas pelos quais o usuário possa manipular a cena diretamente, seja alterando os objetos da cena ou os parâmetros de visualização (navegação). Assim, vamos começar ilustrando um nó especial denominado manipulador (*trackball* virtual), através do qual conseguimos manipular os objetos na cena com um mouse ou outro dispositivo de entrada. O *trackball* aparece na cena em forma de três anéis ao redor do cone. Quando o primeiro botão do *mouse* é pressionado sobre

o manipulador, este muda seus atributos, aparecendo em destaque (ativo). Enquanto o manipulador estiver ativo, o *mouse* é usado para rotacionar o cone. Vejamos o programa:

```
#include <stdlib.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/manips/SoTrackballManip.h>    // novidade
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

int
main(int argc, char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();

   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);

   // o trackball e' instanciado e adicionado ao grafo
   root->addChild(new SoTrackballManip);

   SoMaterial *myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);
   root->addChild(myMaterial);
   root->addChild(new SoCone);

   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
   myCamera->viewAll(root, myRenderArea->getViewportRegion());
   myRenderArea->setSceneGraph(root);
   myRenderArea->setTitle("Trackball");
   myRenderArea->show();

   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

Devido à área de renderização ter um sensor associado ao grafo da cena, ela é automaticamente renderizada todas as vezes que a cena é alterada. Este mecanismo faz com que o cone pareça se mover, como de fato ocorre, quando uma manipulação é feita com o *trackball*. A Figura 11 exibe o resultado do programa, com o manipulador tipo *trackball*.

Navegando através da cena

Voltando ao primeiro exemplo, trocamos o nó *SoXtRenderArea* por um componente denominado *SoXtExaminerViewer*. Este componente fornece mecanismos para o usuário da aplicação alterar a posição e a orientação da câmera

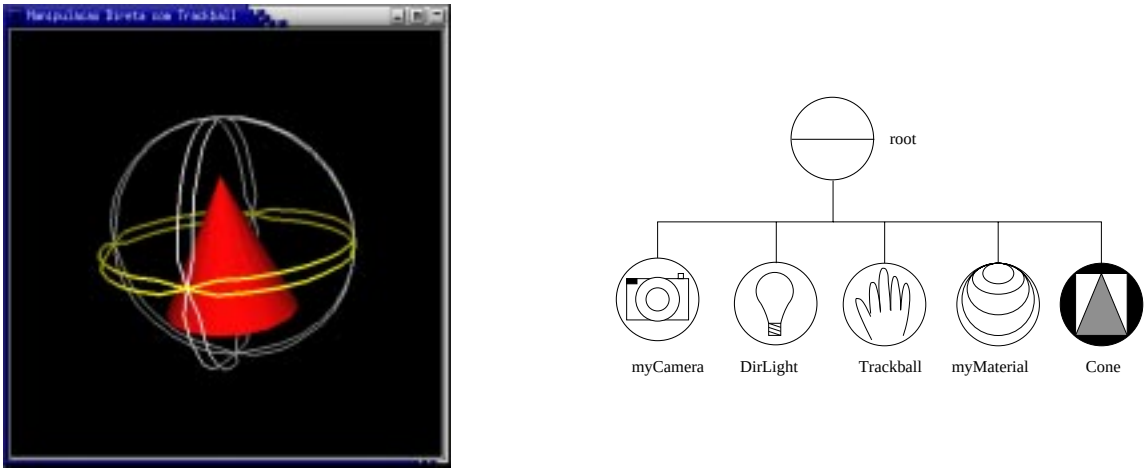

Figura 11: Manipulador tipo *trackball*, juntamente com o grafo da cena.

na cena (parâmetros de visualização) diretamente com o *mouse*, possibilitando a visualização do cone de diversos ângulos e posições. A diferença deste exemplo para o anterior é que neste a câmera está se movendo, ao contrário do *trackball*, onde o objeto é que se move.

```
#include <stdlib.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h> // novidade
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

int
main(int argc, char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();
   SoMaterial *myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);
   root->addChild(myMaterial);
   root->addChild(new SoCone);

   // configura o viewer
   SoXtExaminerViewer *myViewer =
            new SoXtExaminerViewer(myWindow);
   myViewer->setSceneGraph(root);
   myViewer->setTitle("Examiner Viewer");
   myViewer->show();
```

```
    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

Observe pela Figura 12 que neste exemplo a janela principal oferece um conjunto de ferramentas para exploração da cena, através de botões e *dials* que captam eventos externos (normalmente do usuário) e repassa para a Open Inventor.

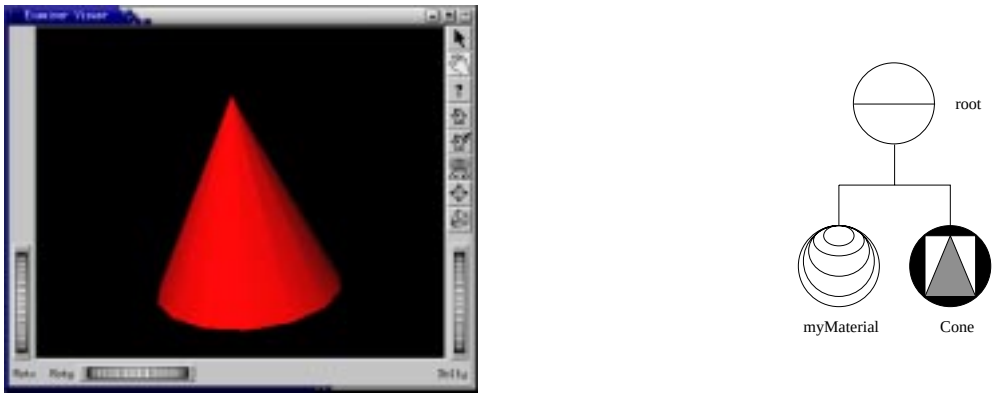

Figura 12: Componente *Examiner Viewer* e o grafo da cena.

3.4 Formato de Dados 3D

O formato de dados (de arquivo) da Open Inventor é um padrão da *Silicon Graphics* para troca de informação 3D entre diferentes aplicações e plataformas. Ele suporta tanto arquivo binário quanto arquivo ASCII. A extensão usada para arquivos da Open Inventor é **.iv**. Qualquer arquivo da Open Inventor deve possuir um cabeçalho padrão para identificá-lo (primeira linha do arquivo):

```
#Inventor V2.0 ascii
```

ou

```
#Inventor V2.0 binary
```

Os nós no arquivo contêm os seguintes elementos: nome, abre-chaves ({), campos (se houver) seguidos de seus nós filhos (se houver) e fecha-chaves (}). Por exemplo,

```
 Separator {
        Transform {
            translation 0 0 0
            rotation -0.43643582 -0.21821791 -0.87287164  0.49999997
        }
        Cube {
        }
    }
```

Note a presença do *Separator*, utilizado para agrupar alguns nós dentro da cena. Neste caso, as transformações de rotação e translação estarão restritas ao escopo dos objetos daquele nó e dos seus filhos, se existirem.

Além destes elementos descritos acima, o arquivo da Open Inventor suporta todos os elementos presentes no grafo de cena. Esta forte correlação é extremamente importante e interessante, pois permite-nos pensar indistintamente sobre o grafo de cena e o formato de arquivo 3D da Open Inventor. Ou seja, o conteúdo de um arquivo nada mais é do que o conteúdo de um grafo de cena.

A Open Inventor possui um conjunto de objetos (e métodos) que lêem grafos de cena de um arquivo, armazenando-os na base de dados e vice-versa, conseguindo também gravar em arquivo grafos (ou sub-grafos) de cena armazenados na base de dados. Vejamos, por exemplo, um fragmento de código usado para ler um arquivo Open Inventor:

```
SoSeparator * readFile(const char *filename)
{
   // Abre o arquivo
   SoInput mySceneInput;
   if (!mySceneInput.openFile(filename)) {
      fprintf(stderr, "Problema ao abrir arquivo %s\n", filename);
      return NULL;
   }

   // Ler o arquivo inteiro e armazena as informacoes
   // lidas na base de dados
   SoSeparator *myGraph = SoDB::readAll(&mySceneInput);
   if (myGraph == NULL) {
      fprintf(stderr, "Problema na leitura de arquivo \n");
      return NULL;
   }

   mySceneInput.closeFile();
   return myGraph;
}
```

Observe que a função `readFile` retorna um apontador para o nó raiz da cena contida no arquivo lido. Assim, a cena pode ser visualizada facilmente, bastando associar o nó em questão a uma área de renderização ou a um componente *examiner viewer*.

4 Java 3D[13]

Java 3D é uma interface criada para o desenvolvimento de aplicações gráficas tridimensionais em Java, executada no topo de bibliotecas gráficas de mais baixo nível, tais como OpenGL e Direct3D, conforme ilustra a Figura 13. De forma mais precisa, Java 3D é um componente da *Sun Microsystems Inc.*, 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.

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 Open Inventor. 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 [Sowirzal, 1998]. Descrever uma cena usando um grafo é tarefa mais simples do que construir a

[13]Biblioteca desenvolvida pela *Sun Microsystems Inc.*

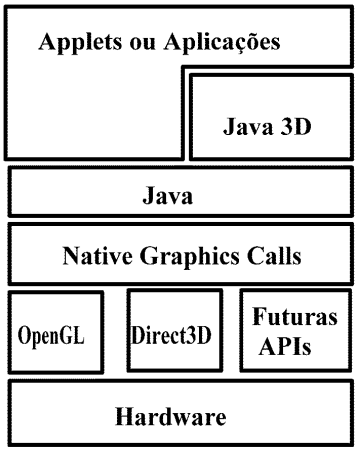

Figura 13: Relação entre as várias camadas de software no contexto de Java 3D.

mesma usando linhas de comando que especificam primitivas gráficas, tais como as da 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, assim como faz a Open Inventor. 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 [Sowizral, 1999].

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

4.1 Java

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 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 14 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 [Sun

Microsystems, 2001]. Na seção 4.2 mostra-se um procedimento básico que pode ser empregado na construção de programas Java 3D e um primeiro exemplo de código.

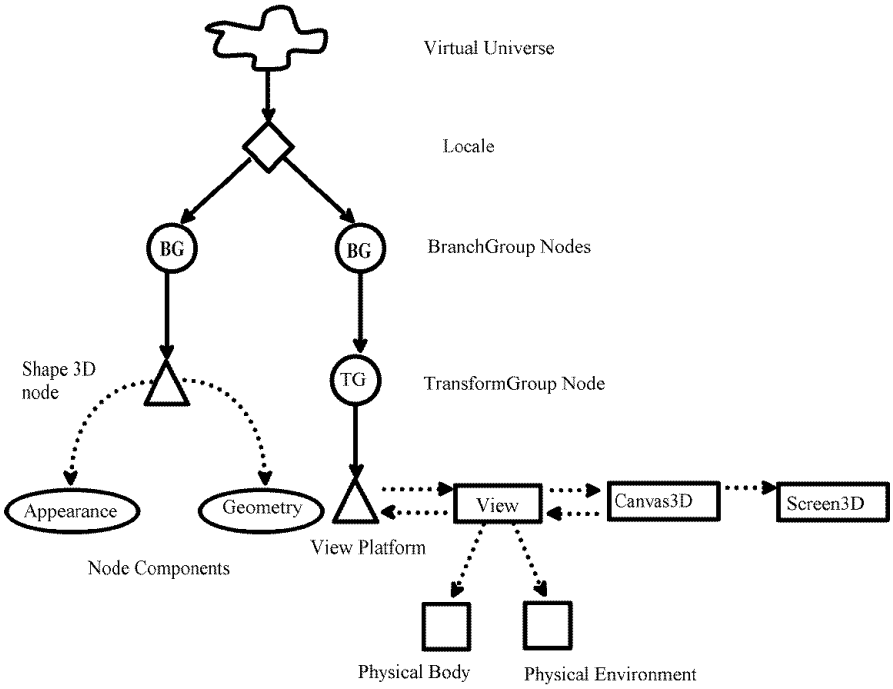

Figura 14: Grafo de uma cena em Java 3D.

Os grafos de cenas (ou subgrafos) são conectados ao universo virtual (representado na Figura 14 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 14 é o ramo à esquerda do nó `Locale`). Os *view branch graphs* especificam as atividades e os 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.

Um caminho entre o nó raiz do grafo de cenas até um de seus nós folha 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 folha 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 [Brown, 1999].

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 15 mostra alguns níveis da hierarquia de classes dos componentes que compõem um grafo de cenas.

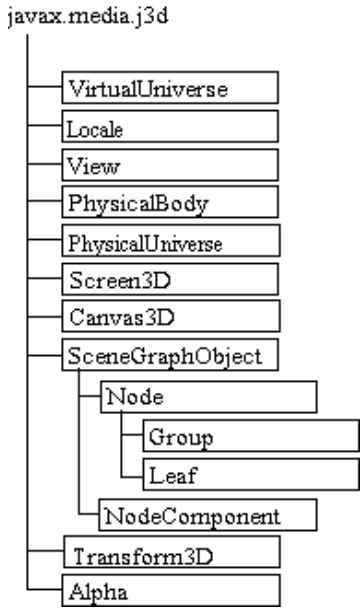

Figura 15: Hierarquia de classes de alguns componentes de uma cena Java 3D.

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 14, 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 [Sun Microsystems, 2001].

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 super-estruturas para anexar os `BranchGroup`), construir um *view branch* e um *content branch*. Um bom ponto de partida é a seqüência de passos sugerida por [Sun Microsystems, 2002]:

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 16, que desenha um cubo rotacionado de $\pi/4$ radianos no eixo *x* e $\pi/5$ radianos no eixo *y* (ver Figura 17). Procura-se evidenciar os passos da seqüência dada anteriormente e fornecer comentários que contribuam para o entendimento do Exemplo01 (adaptado de [Sun Microsystems, 2002]):

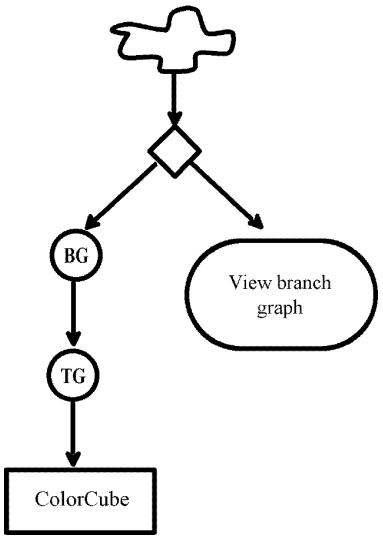

Figura 16: Grafo de cena do Exemplo01.

```
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);
    }
}
```

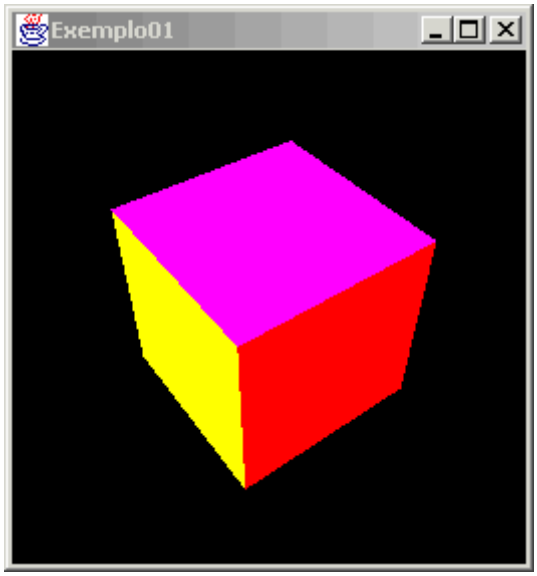

Figura 17: Imagem produzida pelo Exemplo01.

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
  }
}
```

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 18).

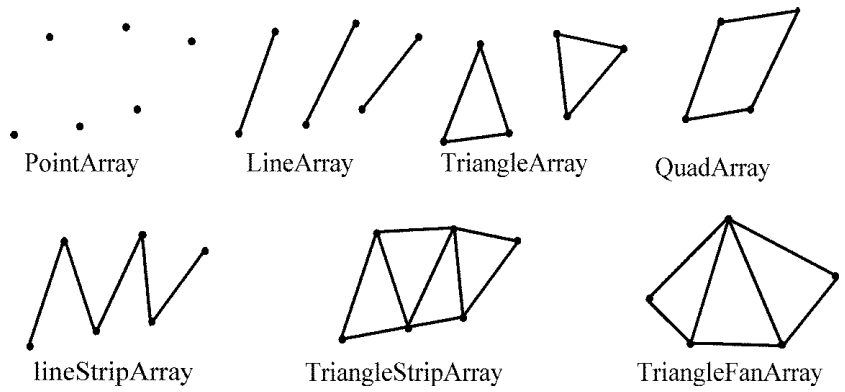

Figura 18: Objetos *array* e *strip*.

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, COORDINATES | 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
        }
```

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. Este baixo desempenho 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 [Sun Microsystems, 2002].

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 19).

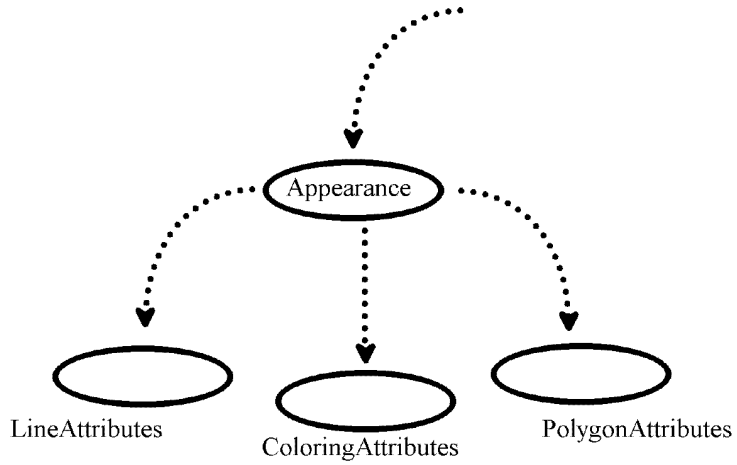

Figura 19: Subgrafo da cena relativa ao Exemplo02 (descrito a seguir).

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 19.

```
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* [Sun Microsystems, 2002].

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 `TransformGroup`, que os *behaviors* promovem as alterações no mundo virtual.

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 últimos 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 *branch graph* anexado a um nó `Locale` (ver Seção 4.2). 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 [Sun Microsystems, 2002]) 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;
   }
```

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 20). 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 uma questão de desempenho, se ele não for definido o *behavior* associado nunca será executado [Sun Microsystems, 2001].

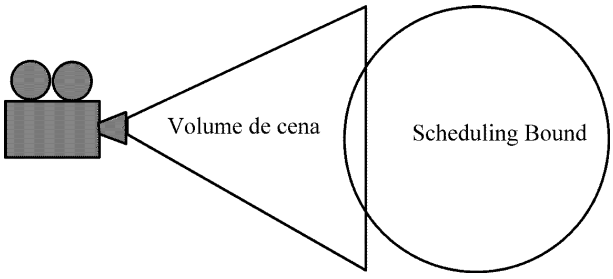

Figura 20: *Scheduling bound*.

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) 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.

Mapeamento do tempo em `Alpha`

As características de um objeto `Alpha` são determinadas por uma lista de parâmetros que podem variar até um total de dez. A Figura 21 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.

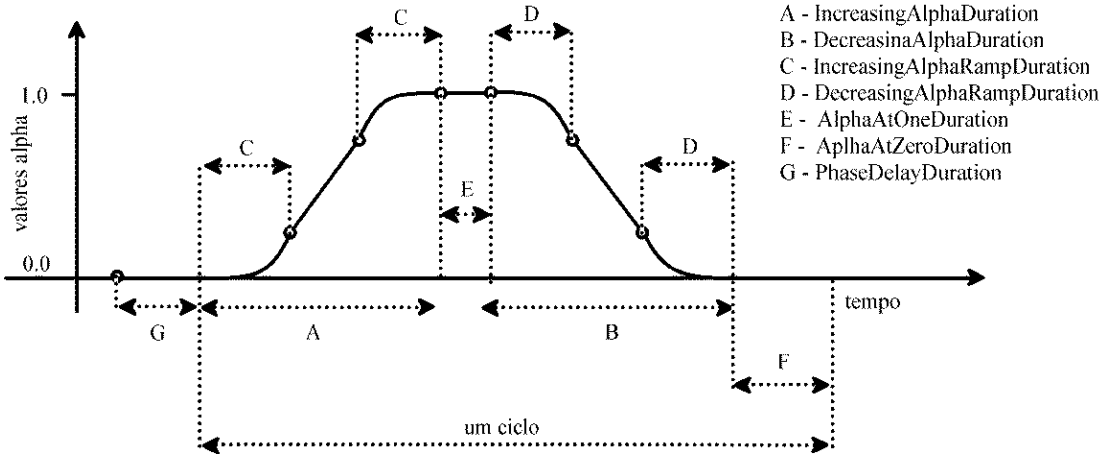

Figura 21: Gráfico Alpha: tempo x valores alpha.

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 (`alphaAtZeroDuration`) e uma fase com valores alpha constantes em um (`alphaAtOneDuration`). 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 a Java 3D. Imagine, 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 22 mostra o gráfico relativo a este exemplo.

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

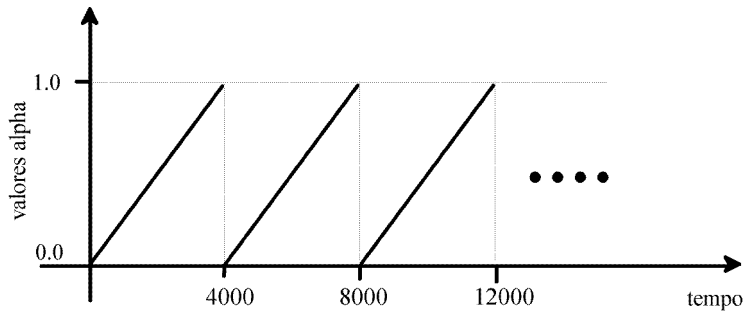

Figura 22: Gráfico Alpha com infinitos ciclos.

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 [Brown, 1999] e [Sun Microsystems, 2001].

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, 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 [Sun Microsystems, 2002]) 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 [Sun Microsystems, 2002].

5 Comparando OpenGL, Open Inventor e Java 3D

Open Inventor e Java 3D são *toolkits* orientadas a objetos e de alto nível de abstração no que tange à programação gráfica 3D. Ambas foram projetadas de forma muito semelhante, incorporando o conceito de grafo de cena para representação de objetos 3D. Além de utilizar a OpenGL para renderização da cena, estas *toolkits* também dispõem de funcionalidades para construção de interface gráfica 2D com o usuário (janelas,botões, menus, etc.) (Figura 23).

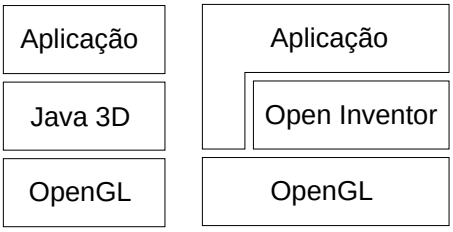

Figura 23: Relação entre OpenGL, Java 3D e Open Inventor.

Normalmente, utiliza-se a linguagem C++ para programação com Open Inventor, Java para programação com Java 3D e a linguagem C para programação de aplicações com OpenGL. No entanto, existem implementações destas bibliotecas e de suas APIs em outras linguagens de programação, como Delphi, Ada e Modula-3. Além disso, a interoperabilidade natural entre C e C++ possibilita o desenvolvimento de aplicações em C++ utilizando a OpenGL.

A OpenGL não faz qualquer referência à interface gráfica com o usuário (isto fica a cargo do programador). Desta forma, Open Inventor e Java 3D oferecem grandes facilidades para a construção de aplicações gráficas 3D interativas e tiram proveito da OpenGL para renderização da cena. Já a OpenGL oferece uma API com mais baixo nível de abstração, tendo a seu favor a independência do sistema de janelas e a possibilidade de aceleração da renderização através do acesso à implementação de rotinas em hardware (placas de vídeo aceleradoras).

A escolha de qual ferramenta utilizar depende da natureza do seu projeto. Vejamos algumas dicas: se você está querendo exercitar os conceitos básicos de computação gráfica ou, se o aplicativo a ser desenvolvido é simples, utilize a OpenGL. Senão, se o seu aplicativo é de médio ou grande porte, Open Inventor e Java 3D oferecem APIs

de programação mais produtivas, dispondo de funcionalidades para programação de interfaces gráficas e o grafo de cena para representação da informação gráfica. Além disso, tanto Open Inventor quanto Java 3D possuem rotinas para converter o grafo de cena em primitivas da OpenGL de forma otimizada.

Entre Open Inventor e Java 3D, a escolha passa pela linguagem a ser utilizada, respectivamente C++ ou Java. Programas em C++ geralmente são mais rápidos do que em Java, isto quando falamos de interface gráfica 2D. No entanto, C++ é uma linguagem mais complexa do que Java. Em relação a cena 3D não há diferença entre as *toolkits*, pois ambas utilizam OpenGL para a renderização.

Para melhor ilustrar esta comparação, apresentamos a seguir um aplicativo, que exibe um cilindro vermelho na tela, implementado utilizando OpenGL, Open Inventor e Java 3D (Figura 24).

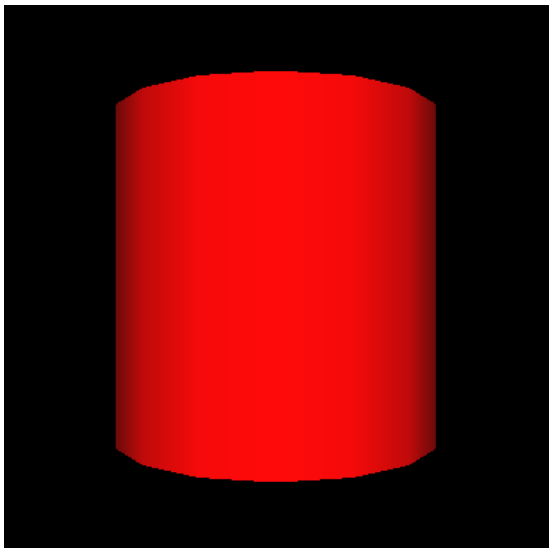

Figura 24: Um cilindro vermelho.

Implementação com OpenGL

```
#include <stdio.h>
#include <stdlib.h>
#include <GL/glut.h>

static void do_resize(int w,int h);
static void draw_scene(void);
static void draw_cylinder(GLfloat radius,GLfloat height,int slices);

void main(int argc,char *argv[])
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(300,300);
   glutInitWindowPosition(10,10);
   glutCreateWindow("Cilindro");
   glutDisplayFunc(draw_scene);
   glutReshapeFunc(do_resize);
   glutMainLoop();
   return 0;
```

```
}

static void do_resize(int w,int h)
{
    glViewport(0,0,(GLsizei) w,(GLsizei) h);
}

static void draw_scene(void)
{
    GLfloat mat_cylinder[] = {1.0,0.0,0.0,1.0};
    GLfloat direction[] = {0.0,0.0,-1.0,0.0};

    glLoadIdentity();
    glEnable(GL_LIGHTING);

    glLightfv(GL_LIGHT0,GL_POSITION,direction);
    glEnable(GL_LIGHT0);

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);

    glClearColor(0.0,0.0,0.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glFrustum(-0.4,0.4,-0.4,0.4,1.0,1000.0);
    gluLookAt(0,0,4,0,0,0,0,1,0);
    glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_cylinder);
    draw_cylinder(1.0,2.0,12);
    glFlush();
}

static void draw_cylinder(GLfloat radius,GLfloat height,int slices)
#define PI 3.1415926535
{
   typedef GLfloat VDATA[2];
   VDATA *vdata;
   double angle;

   vdata = (VDATA *) malloc(sizeof(VDATA)*slices);

   for (int i = 0; i < slices; i++) {
       angle = 2*PI*radius*i/slices;
       vdata[i][0] = (GLfloat) cos(angle);
       vdata[i][1] = (GLfloat) sin(angle);
   }

   glBegin(GL_QUAD_STRIP);
   for (int i = 0; i <= slices; i++) {
       glNormal3f(vdata[i%slices][0],0,vdata[i%slices][1]);
       glVertex3f(vdata[i%slices][0],height/2.0,vdata[i%slices][1]);
       glNormal3f(vdata[i%slices][0],0,vdata[i%slices][1]);
```

```
        glVertex3f(vdata[i%slices][0],-height/2.0,vdata[i%slices][1]);
    }
    glEnd();

    glBegin(GL_POLYGON);
    for (int i = 0; i < slices; i++) {
        glNormal3f(0,1,0);
        glVertex3f(vdata[i][0],height/2.0,vdata[i][1]);
    }
    glEnd();

    glBegin(GL_POLYGON);
    for (int i = 0; i < slices; i++) {
        glNormal3f(0,-1,0);
        glVertex3f(vdata[i][0],-height/2.0,vdata[i][1]);
    }
    glEnd();

    free(vdata);
}
```

Implementação com Open Inventor

```
#include <stdlib.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoCylinder.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

int
main(int argc, char **argv)
{
  // Inicializa o Open Inventor e retorna uma janela principal
  // para ser utilizada. Se nao obtiver sucesso, o programa e' abortado
   Widget myWindow = SoXt::init("cilindro"); // passa o nome do programa
   if (myWindow == NULL) exit(1);

   // constroi uma cena com um cilindro vermelho
   SoSeparator *root = new SoSeparator;
   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   SoMaterial *myMaterial = new SoMaterial;
   root->ref();
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);
   myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);   // Rgb
   root->addChild(myMaterial);
   root->addChild(new SoCylinder);

   // Cria uma area de renderizacao, para mostrar o grafo de cena.
```

```
   // A area de renderizacao ira' ser colocada na janela principal
   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);

   // constroi a camera para visualizar a cena
   myCamera->viewAll(root, myRenderArea->getViewportRegion());

   // coloca a cena na area de renderizacao, mudando o titulo
   myRenderArea->setSceneGraph(root);
   myRenderArea->setTitle("Cilindro");
   myRenderArea->show();


   SoXt::show(myWindow);  // mostra a janela principal
   SoXt::mainLoop();      // Main event loop do Inventor
}
```

Implementação com Java 3D

```
import com.sun.j3d.utils.behaviors.mouse.*;
import com.sun.j3d.utils.geometry.Cylinder;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;

/**
 * MyCylinder cria um cilindro simple (vermelho)
 */
public class MyCylinder extends Applet {
  public BranchGroup createSceneGraph()
  {
    // Cria no' raiz do grafo de cena
    BranchGroup objRoot = new BranchGroup();

    // cria um Transformgroup para fazer escala em todos os
    // objetos que aparecem na cena
    TransformGroup objScale = new TransformGroup();
    Transform3D t3d = new Transform3D();
    t3d.setScale(0.55);
    objScale.setTransform(t3d);
    objRoot.addChild(objScale);

    // cria um cilindro e adiciona
    // ao grafo de cena
    Cylinder CylinderObj = new Cylinder();
    objScale.addChild(CylinderObj);

    BoundingSphere bounds =
```

```
      new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);

    // luzes
    Color3f lColor1 = new Color3f(1.0f, 0.0f, 0.0f);
    Color3f lColor2 = new Color3f(1.0f, 0.0f, 0.0f);
    Vector3f lDir1  = new Vector3f(-1.0f, -1.0f, -1.0f);
    Vector3f lDir2  = new Vector3f(0.0f, 0.0f, -1.0f);
    DirectionalLight lgt1 = new DirectionalLight(lColor1, lDir1);
    DirectionalLight lgt2 = new DirectionalLight(lColor2, lDir2);
    lgt1.setInfluencingBounds(bounds);
    lgt2.setInfluencingBounds(bounds);
    objScale.addChild(lgt1);
    objScale.addChild(lgt2);

    // otimiza conversao do grafo de cena para renderizacao
    // pela OpenGL
    objRoot.compile();

    return objRoot;
  }

  public MyCylinder (){
    setLayout(new BorderLayout());
    GraphicsConfiguration config =
       SimpleUniverse.getPreferredConfiguration();

    Canvas3D c = new Canvas3D(config);
    add("Center", c);

    // Cria um grafo simple e o anexa ao  virtual universe
    BranchGroup scene = createSceneGraph();
    SimpleUniverse u = new SimpleUniverse(c);

    // Move ViewPlatform para tras, para que
    // os objetos da cena sejam vistos
    u.getViewingPlatform().setNominalViewingTransform();

    u.addBranchGraph(scene);
  }


  public static void main(String argv[])
  {

    BranchGroup group;

    new MainFrame(new MyCylinder(), 400, 400);
  }
}
```

6 Conclusão

Este tutorial mostrou uma visão geral de algumas tecnologias para programação gráfica. As tecnologias aqui apresentadas foram escolhidas por serem populares e de fácil acesso (gratuitas e de código aberto). Existe, no entanto, um amplo espectro de ferramentas similares e em níveis de abstração ainda mais baixos (por exemplo, o X Toolkit [McCormack , 1994]) e mais altos (por exemplo, a VRML [Web 3D Consortium, 2001]) que as aqui apresentadas.

Para a apresentação das tecnologias selecionadas, foi adotada 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. Por outro lado, 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 existentes e permite mostrar que, apesar de diferentes, elas não são antagônicas, pelo contrário, são complementares, como fica evidenciado pelos elos que as unem.

7 Onde encontrar os *softwares*

GNU - `http://www.gnu.org`
Java -`http://java.sun.com`
Java 3D - `http://java.sun.com/products/java-media/3D`
Mesa 3D (clone da OpenGL) - `http://www.mesa3d.org`.
Motif/Lesstif - `http://lesstif.org`
Open Inventor - `http://oss.sgi.com/projects/inventor`
OpenGL - `http://www.opengl.org` e `http://www.sgi.com/software/opengl`

Referências

[ANSI, 1985a] ANSI. Computer Graphics-Graphical Kernel System (GKS) Function Description. Technical report, American National Standards Institute for Information Processing Systems, 1985a.

[ANSI, 1985b] ANSI. Programmer's Hierarchical Interactive Graphics System (PHIGS). Technical report, American National Standards Institute for Information Processing Systems, 1985b.

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

[GTK+, 2002] GTK+ The GIMP Toolkit. `http://www.gtk.org/`, 2002.

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

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

[Neider, 1993] J. Neider, T. Davis and W. Mason. *OpenGL Programming Guide* (Addison Wesley, 1993).

[Wernecke, 1994] J. Wernecke. *The Inventor Mentor* (Addison Wesley, 1994).

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

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

[Kilgard, 1996] M. J. Kilgard. The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3. Technical report, SGI, `http://www.opengl.org/developers/documentation/glut/spec3/spec3.html`, 1996.

[Segal, 1994] M. Segal and K. Akeley. The OpenGL Graphics Interface. Technical report, Silicon Graphics Inc, http://www.opengl.org/developers/documentation/white_papers/oglGraphSys/opengl.html, 1994.

[Segal, 1996] M. Segal and K. Akeley. The Design of the OpenGL Graphics Interface. Technical report, Silicon Graphics Inc, [on-line] http://www.opengl.org/developers/documentation/white_papers/opengl/index.html, 1996.

[Strauss, 1992] P. S. Strauss and R. Carey. An Object−Oriented 3D Graphics Toolkit. In *Siggraph'92*, pages 341–352 (ACM, Chicago, 1992).

[Strauss, 1993] Paul S. Strauss. IRIS Inventor, A 3D Graphics Toolkit. In *ACM OOPSLA*, pages 192–200 (ACM, 1993).

[Sowizral, 1999] Sowizral, H. A. and Deering, M. The Java 3D API and Virtual Reality. *IEEE Computer Graphics and Applications*, 19(3):192–200, 1999.

[Sun Microsystems, 2002] Sun Microsystems. *Java 3D API Collateral*. http://java.sun.com/products/java-media/3D/collateral, 2002.

[Sun Microsystems, 2001] Sun Microsystems. *Java 3D API Specification*. http://java.sun.com/products/java-media/3D/forDevelopers/j3dguide/j3dTOC.doc.html, 2001.

[Web 3D Consortium, 2001] Web 3D Consortium. The Virtual Reality Modeling Language, International Standard ISO/IEC DIS 14772-1. http://www.web3d.org/technicalinfo/specifications/vrml97/index.htm, 2001.