Introdução a Java 3D™

Isabel Harb Manssour1

Resumo: A API Java 3D™ consiste em uma hierarquia de classes Java™ que serve como interface para o desenvolvimento de sistemas gráficos tridimensionais. O programador, apenas, necessita trabalhar apenas com construtores de alto nível para criar, manipular e visualizar objetos geométricos, tendo os detalhes de visualização gerenciados automaticamente. Um programa Java 3D cria instâncias de objetos gráficos que são colocados em um grafo de cena, que também pode conter luz, som e outros elementos que possibilitam a criação de universos virtuais. O grafo de cena consiste em uma combinação de objetos 3D numa estrutura de árvore que especifica o conteúdo do universo virtual e como este deve ser visualizado. Este artigo apresenta uma introdução a API Java 3D, descrevendo seus principais conceitos, sua estrutura e os tipos de funcionalidades disponíveis. São ainda identificadas as recentes tendências para o desenvolvimento de aplicações em Java 3D

Palavras-chave: Java 3D, grafo de cena, representação de objetos, realismo.

Abstract: Java 3D™ API is composed by a hierarchy of Java™ classes used as an interface for the development of three-dimensional graphics systems. The programmer needs to work just with high level constructors to create, manipulate and visualize geometrical objects, with the visualization details being automatically handled. A Java 3D program creates graphical object instances that are placed in a scene graph that can also have lights, sound and other elements that allow the creation of virtual universes. A scene graph is composed by a group of 3-D objects combined in a tree structure that determines the contents of the virtual universe and how it should be visualized. This paper presents an introduction to the Java 3D API, describing its main concepts, its structure and the available functionalities. We also identify the current trends in the development of Java 3D applications.

Keywords: Java 3D, scene graph, objects representation, realism.

1 Faculdade de Informática - PUCRS, Av. Ipiranga 6681, P. 30, 90619-900 Porto Alegre/RS, Brasil
{manssour@inf.pucrs.br}

1 Introdução

A API Java 3D consiste em uma hierarquia de classes Java que serve como interface para o desenvolvimento de sistemas gráficos tridimensionais [1]. Possui construtores de alto nível que permitem a criação e manipulação de objetos geométricos, especificados em um universo virtual. Também possibilita a criação de universos virtuais com uma grande flexibilidade, onde as cenas são representadas através de grafos, e os detalhes de sua visualização são gerenciados automaticamente. Assim, o desenvolvimento de um programa Java 3D se resume na criação de objetos e no seu posicionamento em um grafo de cena, que os combina em uma estrutura de árvore. Os grafos de cena são responsáveis pela especificação do conteúdo do universo virtual e pela forma como este é visualizado.

Java 3D foi construída com o objetivo de criar uma API que fosse independente de plataforma, semelhante a VRML (*Virtual Reality Modeling Language* [2]). Então, a *Intel*, a *Silicon Graphics*, a *Apple* e a *Sun Microsystems*, em colaboração, definiram a especificação Java 3D. Em 1998, a *Sun* colocou a sua implementação para *download* [3]. Atualmente, Java 3D consiste em uma API baseada nas bibliotecas gráficas OpenGL [4] e DirectX [5], e os programas podem ser escritos como aplicação, *applet*, ou ambas.

Nos últimos anos, várias aplicações foram desenvolvidas usando Java 3D, tais como jogos, comércio eletrônico, visualização de dados e elaboração de interfaces. Elaboração de lojas virtuais, bem como visualização 3D de produtos, são exemplos de aplicações de Java 3D no comércio eletrônico. Na visualização de dados destacam-se os *toolkits*. Um exemplo é o VisAD (*Visualization for Algorithm Development*) [6], que possibilita a visualização e análise interativa e colaborativa de dados numéricos. O VisAD tem sido usado nas áreas de biologia e meteorologia, entre outras.

Este artigo apresenta uma introdução a API Java 3D, descrevendo, essencialmente, seus principais conceitos, sua estrutura e os tipos de funcionalidades disponíveis. Esta seção aborda a linguagem Java (seção 1.1), alguns conceitos básicos de Computação Gráfica (seção 1.2) e os passos para a instalação e utilização de Java 3D (seção 1.3). O restante do artigo apresenta uma descrição de como criar universos virtuais (seção 2) e uma revisão das funcionalidades para inclusão de realismo, interação e animação (seção 3). Na seção 4, Java 3D é comparada com outras ferramentas gráficas e são identificadas algumas tendências para o desenvolvimento de aplicações em Java 3D.

1.1 Linguagem Java

Java é uma linguagem de programação orientada a objetos, independente de plataforma, que foi desenvolvida pela *Sun Microsystems, Inc.* Atualmente, é uma das linguagens mais utilizadas para o desenvolvimento de sistemas, e pode ser obtida gratuitamente em http://java.sun.com. Java é tanto compilada como interpretada: o compilador transforma o programa em *bytecodes*, que consiste em um tipo de código de

máquina específico da linguagem Java; o interpretador, disponível na JVM (*Java Virtual Machine*) que pode ser instalada em qualquer plataforma, transforma os *bytecodes* em linguagem de máquina para execução, sem que seja necessário compilar o programa novamente.

Portanto, o ambiente Java engloba tanto um compilador, quanto um interpretador. Somente para executar programas Java é utilizado o JRE (*Java Runtime Environment*), que normalmente é instalado junto com as versões mais recentes dos navegadores para Internet, para possibilitar a execução de *applets*. Entretanto, para o desenvolvimento de novos programas, é preciso instalar o J2SE (*Java 2 Platform, Standard Edition*) que inclui o compilador, a JVM e a API (*Application Programming Interface*).

A API básica que é instalada com o J2SE, engloba os pacotes que contêm as classes responsáveis pelas funcionalidades de entrada e saída, interface gráfica, coleções, entre outras. Além disso, existe a *Java Standard Extension API*, que inclui outros pacotes, tais como acesso a banco de dados e o *Java Media Framework*, que suporta tecnologias gráficas e multimídia. Java 3D é um dos componentes deste pacote.

Como o objetivo deste trabalho não é ensinar a programar na linguagem Java, as funcionalidades desta linguagem não são abordadas. Para aprender esta linguagem de programação, sugere-se o estudo da documentação disponível no *site* da *Sun* (http://developer.java.sun.com/developer/infodocs/), bem como de livros especializados [7].

1.2 Conceitos Básicos de Computação Gráfica

Para entender a API Java 3D e aproveitar melhor as suas funcionalidades, é importante conhecer alguns conceitos básicos de Computação Gráfica. Quando se trabalha com síntese de imagens 3D, deve-se considerar que a maioria dos periféricos de entrada e saída é 2D. Portanto, várias técnicas são utilizadas para contornar estas limitações dos dispositivos e possibilitar a visualização com realismo de um objeto modelado, por exemplo, através de uma malha de triângulos. Transformações geométricas de escala, rotação e translação, manipulação de uma câmera sintética, projeção perspectiva, iluminação, cor, sombra e textura são algumas das técnicas utilizadas para contribuir na geração de imagens de alta qualidade.

Está fora do escopo deste trabalho descrever tais conceitos, entretanto é importante conhecer a nomenclatura que será utilizada. Vários livros de Computação Gráfica apresentam uma descrição detalhada de cada técnica citada e podem ser consultados [8, 9, 10]. A descrição de alguns termos importantes é apresentada a seguir.

As transformações geométricas consistem em operações matemáticas que permitem alterar uniformemente o aspecto de objeto(s), mas não a sua topologia. São usadas para posicionar, rotacionar e alterar o tamanho dos objetos no universo. O conceito de câmera sintética é usado para definir de que posição o objeto será visualizado, como se fosse obtida uma “foto” quando a câmara estava numa dada posição direcionada para o objeto. Neste

processo, torna-se necessário aplicar uma projeção, que é o procedimento utilizado para se obter representações 2D de objetos 3D.

Várias técnicas também foram desenvolvidas para tentar reproduzir a realidade em termos de aparência, por exemplo, efeitos de iluminação e de textura de materiais. Portanto, diversas fontes de luz, tais como pontual e *spot*, podem ser incluídas em um universo para permitir a simulação da reflexão dos objetos, que possibilita descrever a interação da luz com uma superfície, em termos das propriedades da superfície e da natureza da luz incidente.

Além disso, cada tipo de material tem características próprias, que permitem sua identificação visual ou táctil. Por exemplo, microestruturas podem produzir rugosidade na superfície dos objetos, ou estes podem ser compostos por um material como mármore ou madeira. As técnicas usadas para simular estes efeitos em Computação Gráfica são conhecidas como mapeamento de textura. Todas estas técnicas foram implementadas na API Java 3D, e a seção 3 descreve como utilizá-las.

1.3 Instalação e Utilização

Todo software necessário para trabalhar com Java 3D é gratuito e pode ser obtido no *site* da *Sun* (http://java.sun.com). A primeira etapa consiste em instalar o J2SE, versão 1.2 ou posterior (ver seção 1.1). Num segundo momento, deve-se instalar a API Java 3D, que está disponível em http://java.sun.com/products/java-media/3D/download.html. Como Java 3D é baseada nas bibliotecas OpenGL ou DirectX (figura 1), estas também devem estar instaladas. Além disso, para executar os programas em um navegador, é necessário ainda instalar um *plug-in*.

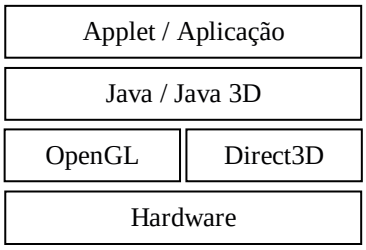


Figura 1. Camadas de software existentes quando se utiliza Java 3D.

O processo de compilar e de executar é o mesmo que para aplicações e *applets* Java, isto é, utiliza-se o comando *javac FileName.java* para compilar, e *java FileName* para executar através do interpretador. Entretanto, outros pacotes, referentes a API Java 3D, devem ser importados no código-fonte das classes (por exemplo, *com.sun.j3d.utils.** e *javax.media.j3d.**). Para facilitar o desenvolvimento, recomenda-se a utilização de um ambiente de programação, tal como o BlueJ [11].

2 Criando Universos Virtuais

Para criar e manipular objetos geométricos 3D, os programadores trabalham apenas com construtores de alto nível, pois os detalhes para a geração das imagens são gerenciados automaticamente. Os objetos 3D, juntamente com luzes, som e outros elementos, estão em um universo virtual. Este universo é composto por um ou mais grafos de cena, que consistem na organização dos objetos em uma estrutura do tipo árvore [1]. Detalhes sobre a criação de grafos de cena são apresentados na seção 2.1, e as principais classes da API Java 3D utilizadas são abordadas na seção 2.2. Alguns elementos que podem ser inseridos em um grafo de cena são apresentados nas seções 2.3 e 2.4.

2.1 Grafo de Cena

Um grafo de cena é criado a partir de instâncias de classes Java 3D que definem som, luz, orientação, geometria, localização e aparência visual dos objetos. Estes objetos correspondem aos nodos (ou vértices), e os seus relacionamentos são identificados por arcos (ou arestas) que representam dois tipos de relacionamentos: referência, que simplesmente associa um objeto com o grafo de cena; e herança (pai-filho), onde um "nodo do tipo grupo" pode ter um ou mais filhos, mas apenas um pai, e um "nodo do tipo folha" não pode ter filhos. Num grafo de cena, os nodos do tipo grupo são identificados por círculos, e do tipo folha por triângulos. Como ilustra a Figura 2 [12, 1], os objetos são organizados em uma estrutura de árvore em um grafo de cena. Assim, um nodo é considerado raiz, e os demais são acessados seguindo os arcos a partir da raiz.

Recomenda-se que cada grafo de cena possua um único objeto *VirtualUniverse*. Este objeto define um universo e possui pelo menos um objeto *Locale*, que é responsável pela especificação do ponto de referência no universo virtual e serve como raiz dos sub-grafos de um grafo de cena. Na Figura 2, por exemplo, existem dois sub-grafos cujas raízes são objetos do tipo grupo (*BranchGroup*). O objetivo destes objetos é agrupar nodos relacionados através de alguma associação comum ou através de um conjunto de características. Existem duas categorias diferentes de sub-grafos que geralmente são incluídos em um grafo de cena. A primeira, representada pela sub-árvore da esquerda na Figura 2, descreve o conteúdo do universo virtual (*content branch graphs* ou sub-grafo de conteúdo), tais como geometrias, aparências, comportamentos, localizações, sons e luzes. A segunda, representada pela sub-árvore da direita na Figura 2, especifica os parâmetros de controle da visualização da cena (*view branch graphs* ou sub-grafo de visualização), tal como direção de observação [12, 1].

Na Figura 2 se pode observar que todos os nodos do tipo grupo são representados graficamente por círculos, nodos folhas são identificados por triângulos e os demais objetos por retângulos. O nodo *TransformGroup* é usado para especificar a posição (relativa a *Locale*), orientação e escala dos objetos geométricos no universo virtual. Os nodos folha no exemplo da Figura 2 são *Behavior*, *Shape3D* e *ViewPlatform*. *Behavior* contém o código Java necessário para manipular a matriz de transformação associada com a geometria do objeto. *Shape3D* refere-se a dois objetos: *Geometry*, que fornece a forma geométrica do

objeto, e *Appearance*, que descreve a aparência da geometria, isto é, sua cor, textura, características de reflexão, entre outras. Finalmente, *ViewPlatform* define a visão final do usuário dentro do universo.

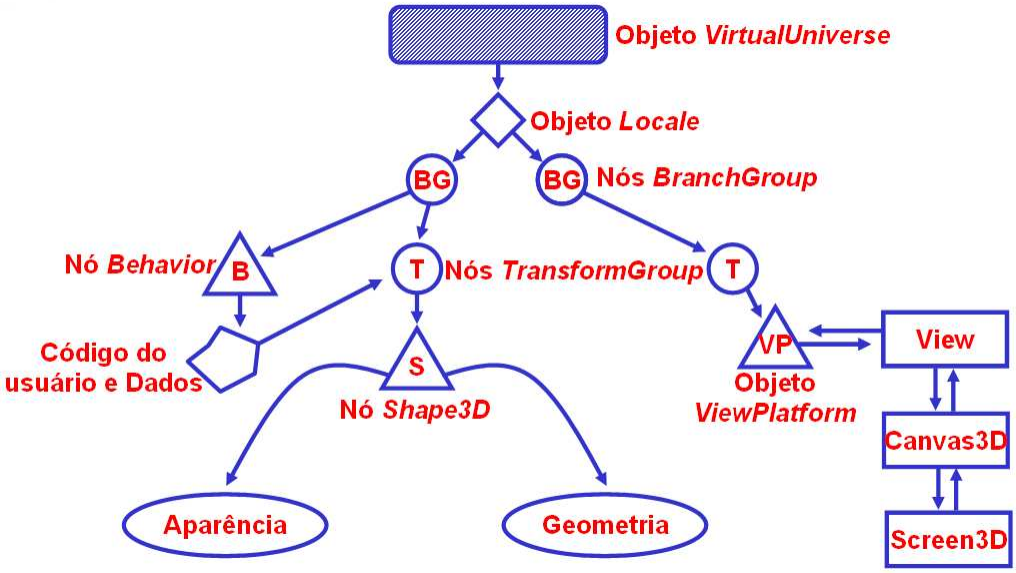


Figura 2. Exemplo de representação gráfica de um grafo de cena [12, 1].

De forma simplificada, os passos para a criação de um programa Java 3D são [1]:

1. Criar um objeto *GraphicsConfiguration*;
2. Criar um objeto *Canvas3D*;
3. Construir e compilar pelo menos um sub-grafo de conteúdo;
4. Criar um objeto *SimpleUniverse*, que referencia o objeto *Canvas3D* criado e automaticamente cria os objetos *VirtualUniverse* e *Locale*, e constrói o sub-grafo de visualização;
5. Inserir o sub-grafo de conteúdo no universo virtual.

O código apresentado no Anexo 1 (*HelloUniverseJFrame.java*) ilustra o processo de criação de um pequeno programa Java 3D, que tem como objetivo apresentar um cubo com faces coloridas que é constantemente rotacionado ao redor do eixo y (Figura 3) [1].

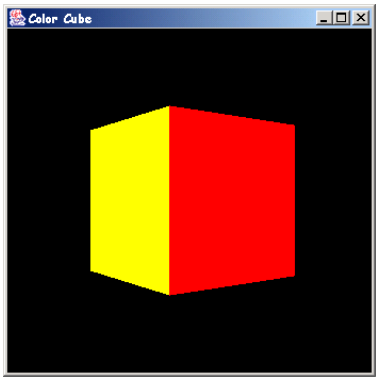


Figura 3. Resultado da execução do programa do Anexo 1 [1].

2.2 Classes Principais

A API Java 3D fornece um grande número de classes para especificação, posicionamento e visualização de objetos gráficos. Nesta seção são apresentadas algumas classes fundamentais para o desenvolvimento de programas gráficos, que aparecem no exemplo do Anexo 1. Detalhes sobre as demais classes, bem como informações sobre seus construtores e métodos, podem ser obtidos consultando a documentação da API Java 3D (http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_2_API/j3dapi/).

Uma das classes mais importantes é a *SimpleUniverse*, pois é responsável pela configuração de um ambiente mínimo para executar um programa Java 3D, fornecendo as funcionalidades necessárias para a maioria das aplicações. Quando uma instância de *SimpleUniverse* é criada (linha 80 do Anexo 1), automaticamente são criados todos os objetos necessários para o sub-grafo de visualização, tais como *Locale*, *ViewingPlatform* e *Viewer*.

GraphicsConfiguration (linha 72 do Anexo 1) é uma classe que faz parte do pacote *awt* responsável pela descrição das características do dispositivo gráfico (impressora ou monitor). Sua estrutura varia de plataforma para plataforma, fornecendo, por exemplo, a resolução do dispositivo. A classe *Canvas3D* fornece o *canvas* (linha 75 do Anexo 1), ou seja, uma área de desenho, onde é realizada a visualização 3D.

A classe *BranchGroup* serve como ponteiro para a raiz de um sub-grafo de cena (linha 102 do Anexo 1). Instâncias desta classe são os únicos objetos que podem ser inseridos em *Locale*. Um sub-grafo de cena que tem um *BranchGroup* como raiz pode ser considerado como uma *compile unit*, podendo ser compilado, inserido em um universo virtual (associando-o com *Locale*) e desassociado deste universo em tempo de execução.

As transformações geométricas de escala, rotação e translação, são especificadas através de uma instância de *Transform3D*, que representa uma matriz 4x4 de números reais (*float*). Objetos da classe *TransformGroup*, por sua vez, especificam uma transformação, através de um objeto *Transform3D*, que será aplicada a todos os seus filhos. Ao serem aplicadas as transformações, deve-se considerar que os efeitos num grafo de cena são cumulativos.

A classe *BoundingSphere* define uma região (ou volume) limitada por uma esfera que é especificada a partir de um ponto central e um raio. No exemplo do Anexo 1 (linhas 122 à 124), a esfera é associada com o limite para o objeto *RotatorInterpolator*, que é usado para fazer a animação do cubo (ver seção 3.4). As demais classes que aparecem no exemplo do Anexo 1 são abordadas nas seções 2.3 e 2.4.

2.3 Geometrias

Em Computação Gráfica, modelos são utilizados para representar entidades físicas ou abstratas e fenômenos no computador. Portanto, a etapa de modelagem consiste em todo o processo de descrever um modelo, objeto ou cena, de forma que se possa desenhá-lo. A representação de um objeto deve ser feita de forma que seja fácil de usar e analisar. Atualmente, existem várias técnicas para a representação de modelos 3D. Nesta seção, são apresentadas algumas formas de representar um modelo em Java 3D, que é composto por: **geometria**, que define a "forma geométrica" do modelo; **aparência**, que define as propriedades do material que compõe a geometria, tais como cor, transparência e textura.

2.3.1. Representação de objetos

A maneira mais simples de definir a geometria de um modelo é através das primitivas gráficas *Box*, *Sphere*, *Cylinder* e *Cone*, disponíveis no pacote *com.sun.j3d.utils.geometry*. A especificação destas primitivas é feita através da criação de instâncias de classes que possuem estes mesmos nomes. O código a seguir contém a assinatura de alguns dos construtores disponíveis nestas classes [13], e a Figura 4 ilustra a criação destas primitivas.

```
1. Box();       // Box default com todas as dimensões = 1.0
2. Box (float xdim, float ydim, float zdim, Appearance ap);
3.
4. Cone();      // Cone default com raio = 1.0 e altura = 2.0
5. Cone (float radius, float height);
6. Cone (float radius, float height, Appearance  ap);
7.
8. Cylinder (); // Cilindro default com raio = 1.0 e altura = 2.0
9. Cylinder(float radius, float height);
10.Cylinder(float radius, float height, Appearance  ap);
11.
12.Sphere();    // Esfera default com raio = 1.0
13.Sphere(float radius);
14.Sphere(float radius, Appearance  ap);
```

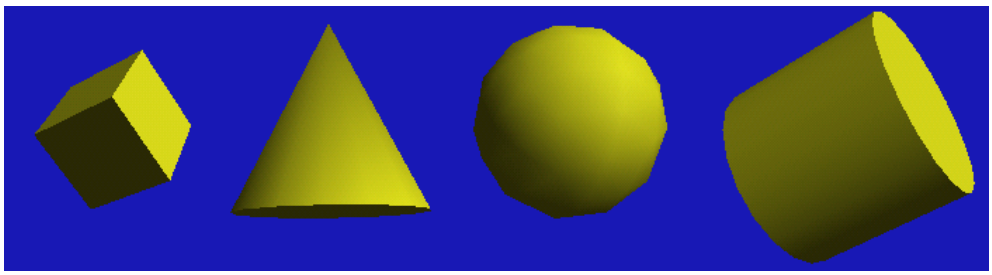


Figura 4. Resultado da criação de instâncias de *Box*, *Cone*, *Sphere* e *Cylinder*.

Uma forma comum de representar objetos na Computação Gráfica 3D é através de uma lista de vértices e uma lista de arestas ou de faces poligonais. Neste caso, uma malha de polígonos representa uma superfície composta por faces planas, que podem ser triângulos (preferencialmente) ou quadrados. A maioria dos objetos, desde um simples triângulo até o complexo modelo de um avião, é modelado desta maneira.

O nodo *Shape3D* é usado para definir um objeto em Java 3D. Instâncias desta classe referenciam um nodo *Geometry* e um nodo *Appearance* (seção 2.3.2). *Geometry* é uma superclasse abstrata que tem como subclasses, por exemplo, *GeometryArray* e *Text3D*. A classe *GeometryArray* também é uma classe abstrata, e suas subclasses são usadas para especificar pontos, linhas e polígonos preenchidos, tal como um triângulo. Algumas de suas subclasses são: *QuadArray*, *TriangleArray*, *LineArray*, *PointArray* e *GeometryStripArray*, que, por sua vez, tem *LineStripArray*, *TriangleStripArray* e *TriangleFanArray* como subclasses. Cada uma destas classes possui uma lista de vértices que podem ser conectados de diferentes maneiras, como mostra a Figura 5. Além disso, objetos *GeometryArray* também podem armazenar coordenadas do vetor normal, de cores e de texturas [1].

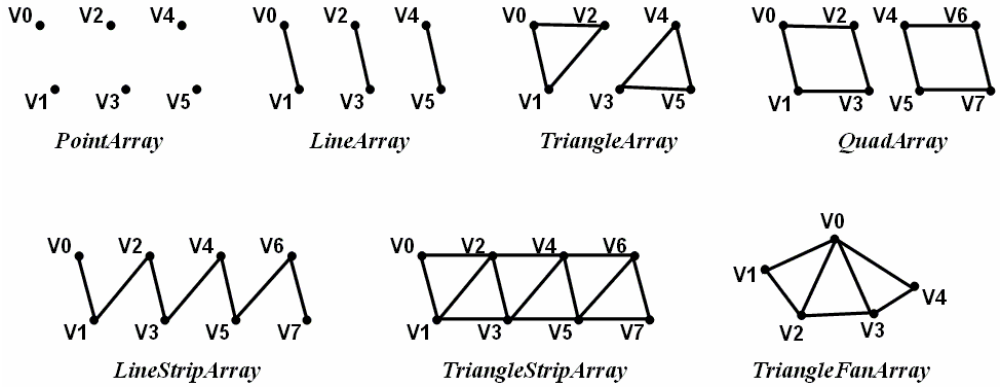


Figura 5. Subclasses de *GeometryArray* e *GeometryStripArray* [1].

Exemplos de construtores para as subclasses de *GeometryStripArray* são apresentados no código a seguir. Os parâmetros correspondem, respectivamente, ao número total de

vértices; a uma máscara indicando quais componentes estão presentes em cada vértice (coordenadas, normais, cor e/ou textura); e ao vetor de vértices.

```
1. LineStripArray(int vtxCount, int vtxFormat, int stripVtxCounts[])
2. TriangleStripArray(int vtxCount, int vtxFormat,
3.                                                 int stripVtxCounts[]))
4. TriangleFanArray(int vtxCount, int vtxFormat,
5.                                                 int stripVtxCounts[]))
```

Java 3D também permite importar dados geométricos criados por outras aplicações. Neste caso, um arquivo de formato padrão é importado através de um *loader* e a cena armazenada é representada em código Java 3D. O pacote *com.sun.j3d.loaders* fornece a interface que deve ser implementada. Entre os arquivos que podem ser importados estão 3D Studio (.3ds), *Wavefront* (.obj), VRML (.wrl) e AutoCAD (.dfx). Por exemplo, para utilizar um *loader* no exemplo apresentado no Anexo 1, seria necessário incluir o código a seguir no método *criaGrafoDeCena*. A imagem resultante está ilustrada na Figura 6.

```
1. ObjectFile f = new ObjectFile(ObjectFile.RESIZE,
2.                                  (float)(60.0 * Math.PI / 180.0));
3. Scene s = null;
4.
5. try
6. {
7.    s = f.load( new
8.       java.net.URL(getCodeBase().toString() + "./teapot.obj"));
9. }
10.catch (FileNotFoundException e) { … }
11.catch (ParsingErrorException e) { … }
12.catch (IncorrectFormatException e) { … }
13.catch (java.net.MalformedURLException ex) { … }
14.
15.objRaiz.addChild(s.getSceneGroup());
```

2.3.2. Aparência

As classes que definem as primitivas gráficas não especificam cor, pois a sua aparência é determinada através do nodo *Appearance*. Se este nodo não for instanciado, a geometria é desenhada com a cor branca, sem a possibilidade de determinar suas propriedades, tais como textura, transparência, cor, estilo de linha e material. Alguns de seus métodos são listados a seguir, mostrando que este nodo faz referência a vários outros objetos.

```
1. void setColoringAttributes (ColoringAttributes coloringAttributes)
2. void setLineAttributes (LineAttributes lineAttributes)
3. void setTexture (Texture texture)
4. void setMaterial (Material material)
5. Material getMaterial ()
```

Figura 6. Visualização do arquivo *teapot.obj* utilizando um *loader*.

O nodo *Material* é muito utilizado para definir a aparência de um objeto considerando que existem fontes de luz. As propriedades que devem ser especificadas são: cor ambiente refletida da superfície do material, cujo valor *default* é (0.2, 0.2, 0.2); cor difusa, que consiste na cor do material quando iluminado e possui como valor *default* (1.0, 1.0, 1.0); cor especular do material, que tem branco como *default*; cor emissiva, isto é, a cor da luz que o material emite (preto por *default*); e *shininess*, que indica a concentração do brilho do material, que varia entre 1 e 128, sendo o *default* 64.

O código a seguir ilustra a utilização destes nodos para criar um cone amarelo com brilho, considerando a estrutura de programa definida no exemplo do Anexo 1. A utilização de textura e fontes de luz, que influenciam a aparência de uma geometria, está descrita na seção 3.

```
1. Appearance app = new Appearance();
2.
3. //Parâm.:ambColor,emissiveColor,diffuseColor,specColor,shininess
4. Material material = new Material(new Color3f(0.8f,0.8f,0.1f),
5.                                  new Color3f(0.0f,0.0f,0.0f),
6.                                  new Color3f(0.8f,0.8f,0.1f),
7.                                  new Color3f(1.0f,1.0f,1.0f), 100.0f);
8. app.setMaterial(material);
9. Cone cone = new Cone(0.4f, 0.8f);
10.cone.setAppearance(app);
```

2.4 Texto e *Background*

Existem duas classes que podem ser instanciadas para adicionar um texto em uma cena Java 3D:

- *Text2D*: consiste na representação de um conjunto de caracteres em polígonos retangulares que são aplicados como textura, isto é, a textura para o retângulo apresenta os caracteres com uma determinada cor, sendo o fundo transparente;
- *Text3D*: o conjunto de caracteres é convertido para objetos geométricos 3D que os representam.

As transformações geométricas de escala, rotação e translação podem ser aplicadas em objetos *Text2D* e *Text3D* para posicioná-los no universo. Diferentes tipos de fonte e cores também podem ser aplicados. O próximo trecho de código ilustra a utilização destas classes, e a imagem gerada a partir do mesmo é apresentada na Figura 7. Nas linhas 9 e 10 está sendo criada uma instância de *Text2D*, onde os parâmetros passados para o construtor são, respectivamente: o texto que será exibido, a sua cor, a fonte, o tamanho e o estilo da fonte [13]. A criação de um objeto *Text3D* envolve a criação de outros objetos, como pode ser visto nas linhas 21 a 25. Este código pode ser inserido no método *criaGrafoDeCena* do exemplo do Anexo 1.

```
1.  TransformGroup text2DTrans = new TransformGroup();
2.  TransformGroup text3DTrans = new TransformGroup();
3.  Transform3D trans = new Transform3D();
4.  Transform3D t1 = new Transform3D();
5.  text2DTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
6.  objRaiz.addChild(text2DTrans);
7.  trans.setTranslation(new Vector3d(-0.6,-0.4,0.0));
8.  text2DTrans.setTransform(trans);
9.  Text2D text2D=new Text2D("TEXTO !!",new Color3f(0.8f,0.1f,0.8f),
10.                          "Helvetica", 50, Font.BOLD | Font.ITALIC);
11. text2DTrans.addChild(text2D);
12.
13. text3DTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
14. objRaiz.addChild(text3DTrans);
15. t1.rotY(Math.toRadians(-10.0));
16. t1.rotX(Math.toRadians(45.0));
17. trans.mul(t1);
18. trans.setScale(0.35);
19. trans.setTranslation(new Vector3d(-0.55,0.3,0.0));
20. text3DTrans.setTransform(trans);
21. Font3D font3d = new Font3D(new Font("Helvetica", Font.PLAIN, 1),
22.                            new FontExtrusion());
23. Text3D textGeom = new Text3D(font3d, new String("TEXTO 3D!"),
24.                              new Point3f(-1.0f, 0.0f, 0.0f));
25. Shape3D textShape = new Shape3D(textGeom);
26. textShape.setAppearance(app);
27. text3DTrans.addChild(textShape);
```

Figura 7. Exemplo de objetos *Text3D* e *Text2D*.

Caso não seja especificada uma cor diferente, a cor de fundo da janela (ou *background*) é preta. A API Java 3D fornece algumas alternativas para alterar o *background* de uma janela. É possível, de uma maneira simples, determinar uma cor sólida qualquer, colocar uma imagem, colocar uma geometria, ou fazer uma combinação destas duas últimas abordagens. Os passos para que seja possível determinar um *background* diferente são [1]:

1. Criar um objeto *Background* especificando uma cor ou imagem;
2. Adicionar geometria (opcional);
3. Fornecer um objeto *Application Boundary* ou *BoundingLeaf*;
4. Adicionar o objeto *Background* no grafo de cena.

Para simplesmente alterar a cor para azul, por exemplo, é necessário criar uma instância de *Background*, como ilustra a linha 4 do próximo trecho de código. Objetos deste tipo possuem *ApplicationBounds* (linha 5) para permitir que diferentes *backgrounds* sejam especificados para diferentes regiões do universo virtual.

```
1.      BoundingSphere bounds =
2.         new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
3.      Color3f bgColor = new Color3f(0.1f, 0.1f, 0.7f);
4.      Background bg = new Background(bgColor);
5.      bg.setApplicationBounds(bounds);
6.      objRaiz.addChild(bg);//objRaiz definido na linha 23 do Anexo1
```

Para permitir que uma textura seja colocada como *background* de uma janela é necessário passar para o construtor uma instância de *ImageComponent2D*. Um exemplo de como criar um *background* com textura em uma *applet* é apresentado a seguir (considere o código do Anexo 1).

```
1. BoundingSphere bounds =
2.              new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
3.
4. java.net.URL imagemBg = null;
5. try {
6.      imagemBg = new java.net.URL(getCodeBase().toString() +
7.                                  "bg.jpg");
8. }
9. catch (java.net.MalformedURLException ex) { … }
10. TextureLoader texturaBg = new TextureLoader(imagemBg, this);
11. Background bg = new Background(texturaBg.getImage());
12. bg.setApplicationBounds(bounds);
13. bg.setImageScaleMode(Background.SCALE_REPEAT);
14. objRaiz.addChild(bg);
```

3 Realismo, Interação e Animação

Para gerar imagens com realismo, é necessário implementar várias técnicas que permitem reproduzir no computador a "aparência" dos objetos do mundo real. Entre estas técnicas encontram-se os efeitos de iluminação e textura, descritos nas seções 3.1 e 3.2, respectivamente. As seções 3.3 e 3.4 descrevem como trabalhar com interação e animação em Java 3D. Algoritmos e técnicas de realismo não são descritos nesta seção. Apenas alguns termos são apresentados. Para uma descrição detalhada sugere-se a leitura de livros de Computação Gráfica [8, 9].

3.1 Iluminação

Uma das primeiras etapas para trabalhar com modelos de iluminação consiste em definir a fonte de luz que será incluída no ambiente, isto é, o objeto que emite energia radiante. Pontual, direcional e *spot* são os tipos de fonte de luz utilizados. Uma fonte de luz pontual é aquela cujos raios emanam uniformemente em todas as direções a partir de um único ponto. O brilho do objeto varia de uma parte para outra, dependendo da direção e da distância da fonte de luz. Uma fonte de luz direcional é aquela cujos raios vêm da mesma direção, e *spot* é uma luz que emana de um ponto com uma intensidade variável que diminui conforme a direção desvia de um dado eixo.

Posteriormente é necessário definir o modelo de Reflexão (ou Modelo de Iluminação) que descreve a interação da luz com uma superfície, em termos das propriedades da superfície e da natureza da luz incidente. O principal objetivo neste caso é exibir os objetos tridimensionais no espaço de tela bidimensional que se aproximem da realidade. Através de um modelo de reflexão também é possível fazer com que objetos do tipo espelho apresentem em sua superfície a imagem de outros objetos do universo. Os três tipos principais são: ambiente, que é uma luz que vem de todas as direções, resultante da luz refletida no ambiente; difusa, luz que vem de uma direção, atinge a superfície e é refletida em todas as direções, fazendo que o objeto possua o mesmo brilho independente de onde está sendo

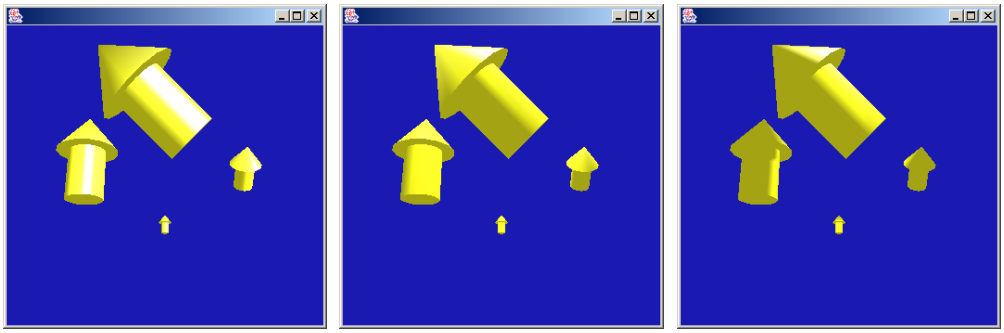
visualizado; especular, luz que vem de uma direção e tende a ser refletida numa única direção.

Em Java 3D a classe abstrata *Light* define um conjunto de parâmetros comum para todos os tipos de fonte de luz. Estes parâmetros são usados para definir cor da luz, se está "ligada" ou não e qual é a sua região de influência. As suas subclasses são *AmbientLight, DirectionalLight* e *PointLight*, que por sua vez tem *SpotLight* como subclasse [13].

A classe *AmbientLight* possui um componente de reflexão ambiente. Através de seus construtores é possível definir a sua cor e se está ativa ou não. Para especificar uma fonte de luz em um ponto fixo que espalha raios de luz igualmente em todas as direções se utiliza a classe *PointLight*. Já a classe *DirectionalLight* determina uma fonte de luz "orientada" com origem no infinito. Estas fontes de luz contribuem para as reflexões difusa e especular. Nos seus construtores se podem definir a cor, a posição e o coeficiente de atenuação da fonte de luz. Nos construtores da classe *SpotLight* os parâmetros são diferentes, pois é preciso especificar a direção, o ângulo de expansão e a concentração da luz, ou seja, o quanto a intensidade da luz é atenuada em função do ângulo de expansão [13]. Os trechos de código a seguir ilustram a utilização destas classes para criar diferentes efeitos de iluminação. Em cada exemplo é utilizada uma fonte de luz diferente (direcional, pontual e *spot*), e as reflexões ambiente, difusa e especular são combinadas. A figura 8 ilustra a imagem gerada a partir de cada trecho de código.

```
1. /////////// Exemplo 1: fonte de luz direcional
2. Color3f corLuz = new Color3f(0.9f, 0.9f, 0.9f);
3. Vector3f direcaoLuz  = new Vector3f(-1.0f, -1.0f, -1.0f);
4. Color3f corAmb = new Color3f(0.2f, 0.2f, 0.2f);
5. AmbientLight luzAmb = new AmbientLight(corAmb);
6. luzAmb.setInfluencingBounds(bounds);
7. DirectionalLight luzDir= new DirectionalLight(corLuz,direcaoLuz);
8. luzDir.setInfluencingBounds(bounds);
9. objRaiz.addChild(luzAmb);
10. objRaiz.addChild(luzDir);
11. //Parâm.:ambColor,emissiveColor,diffuseColor,specColor,shininess
12. Material material = new Material(new Color3f(0.8f,0.8f,0.1f),
13.         new Color3f(0.0f,0.0f,0.0f), new Color3f(0.8f,0.8f,0.1f),
14.         new Color3f(1.0f,1.0f,1.0f), 100.0f);
15.
16. /////////// Exemplo 2: fonte de luz pontual
17. Color3f corLuz = new Color3f(0.9f, 0.9f, 0.9f);
18. Point3f posicaoLuz   = new Point3f(0.0f, 2.0f, 0.0f);
19. Point3f atenuaLuz = new Point3f(0.2f, 0.2f, 0.2f);
20. Color3f corAmb = new Color3f(0.2f, 0.2f, 0.2f);
21. AmbientLight luzAmb = new AmbientLight(corAmb);
22. luzAmb.setInfluencingBounds(bounds);
23. objRaiz.addChild(luzAmb);
24. PointLight luzPont = new PointLight(corLuz,posicaoLuz,atenuaLuz);
25. luzPont.setInfluencingBounds(bounds);
26. objRaiz.addChild(luzPont);
27. Material material = new Material(new Color3f(0.8f,0.8f,0.1f),
28.         new Color3f(0.0f,0.0f,0.0f), new Color3f(0.8f,0.8f,0.1f),
29.         new Color3f(1.0f,1.0f,1.0f), 100.0f);
```

```
30.
31. /////////// Exemplo 3: fonte de luz spot
32. Color3f corAmb = new Color3f(0.2f, 0.2f, 0.2f);
33. AmbientLight luzAmb = new AmbientLight(corAmb);
34. luzAmb.setInfluencingBounds(bounds);
35. objRaiz.addChild(luzAmb);
36. Color3f corLuz       = new Color3f(1.0f, 1.0f, 1.0f);
37. Point3f posicaoLuz   = new Point3f(0.0f, 2.0f, 0.0f);
38. Point3f atenuaLuz = new Point3f(0.1f, 0.1f, 0.1f);
39. Vector3f direcaoLuz  = new Vector3f(0.0f, -1.0f, 0.0f);
40. SpotLight luzSpot = new SpotLight(corLuz, posicaoLuz, atenuaLuz,
41.                    direcaoLuz, (float)Math.toRadians(12), 60.0f);
42. luzSpot.setInfluencingBounds(bounds);
43. objRaiz.addChild(luzSpot);
44. Material material = new Material(new Color3f(0.8f,0.8f,0.1f),
45.         new Color3f(0.0f,0.0f,0.0f), new Color3f(0.8f,0.8f,0.1f),
46.         new Color3f(1.0f,1.0f,1.0f), 100.0f);
```



a **b** **c**

Figura 8. Exemplo da utilização de luz direcional (a), pontual (b) e *spot* (c).

3.2 Textura

Cada tipo de material tem características próprias que permitem sua identificação visual ou táctil. Estas características se traduzem como microestruturas que produzem rugosidade na superfície dos objetos. Exemplos são o plástico, a areia e o mármore, que podem ser simulados com a aplicação de imagens digitalizadas ou outras técnicas. Em Computação Gráfica, estes detalhes da superfície de um objeto são chamados de textura.

Uma técnica de mapeamento de texturas consiste simplesmente no mapeamento de uma imagem (mapa de textura/padrão de textura) para a superfície de um objeto. Para aplicar textura na API Java 3D é necessário criar uma aparência (seção 2.3.2), armazenar a imagem da textura e fazer a associação entre estes objetos. Também é preciso definir o posicionamento da textura na geometria, bem como os seus atributos. Resumindo, os passos para especificação de uma textura são [1]:

1. Preparar a imagem de textura;
2. Carregar a textura;
3. Associar a textura com a aparência;
4. Determinar as coordenadas de textura da geometria.

A etapa de preparação da imagem deve ser feita em um programa externo à API Java 3D, pois consiste na sua criação e edição. Porém, é obrigatório que a imagem esteja num formato compatível para leitura, por exemplo, JPG ou GIF, e que o seu tamanho seja múltiplo de dois em cada dimensão. Esta imagem pode estar armazenada em um arquivo local ou em uma URL. Para carregar a textura utiliza-se uma instância da classe *TextureLoader*, que deve ser associada com um objeto *Appearance*. No final, devem-se especificar as coordenadas de textura da geometria. Nesta etapa, o programador determina a posição da textura em uma geometria através de coordenadas de textura. Estas coordenadas são definidas por vértices que definem um ponto de textura que deve ser aplicado. Dependendo da especificação dos pontos, a imagem pode ser rotacionada, "esticada" ou duplicada.

O próximo trecho de código ilustra a aplicação de uma textura em um cilindro. A figura 9 mostra a imagem da textura e a mesma imagem mapeada no cilindro que é visualizado de dois ângulos diferentes.

```
1. Appearance app = new Appearance();
2.
3. java.net.URL texImage = null;
4.
5. try {
6.  texImage = new java.net.URL(getCodeBase().toString() +
7.                              deserto.jpg");
8. }
9. catch (java.net.MalformedURLException ex) { … }
10.
11.TextureLoader loader = new TextureLoader(texImage, this);
12.
13.app.setTexture(loader.getTexture());
14.
15.Material material = new Material(new Color3f(0.2f,0.2f,0.2f),
16.                            new Color3f(0.0f,0.0f,0.0f),
17.                            new Color3f(1.0f,1.0f,1.0f),
18.                            new Color3f(0.5f,0.5f,0.5f), 100.0f);
19.
20.app.setMaterial(material);
21.
22.Cylinder cilindro = new Cylinder(0.4f, 0.7f,
23.    Cylinder.GENERATE_NORMALS | Cylinder.GENERATE_TEXTURE_COORDS,
24.    20, 10, app);
25.
26.cilindro.setAppearance(app);
27.
28.objTrans.addChild(cilindro);
```

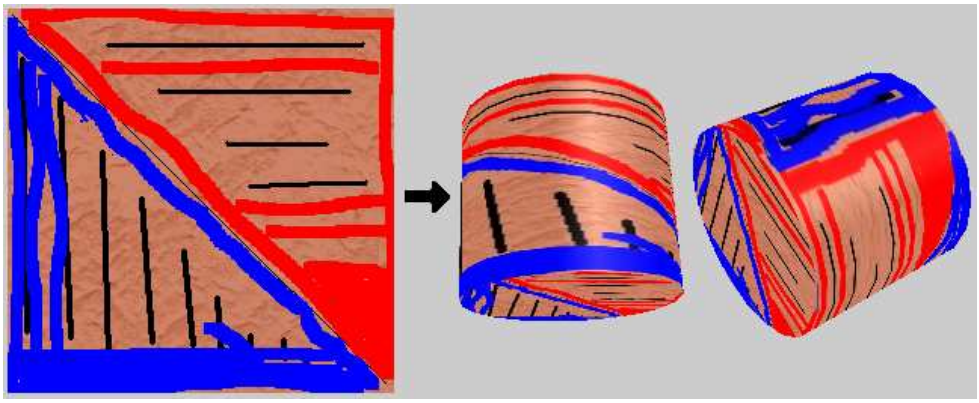


Figura 9. Resultado do mapeamento de textura em um cilindro.

3.3 Interação

Interação consiste na ocorrência de alterações em resposta a ações do usuário, isto é, seu objetivo é mudar o grafo de cena, ou os seus objetos, em resposta a estímulos do usuário, tais como pressionar uma tecla ou mover o *mouse*. A interação, bem como a animação, é especificada através de subclasses da classe abstrata *Behavior*. Esta classe fornece mecanismos para incluir o código necessário para alterar o grafo de cena, por exemplo, removendo objetos ou trocando seus atributos [1].

A classe *Behavior* faz uma conexão entre um estímulo e uma ação, sendo que um estímulo pode resultar em uma ou mais alterações. Existe uma hierarquia de subclasses desta classe, sendo que algumas delas são usadas para interação (por exemplo, *MouseBehavior* e *KeyNavigatorBehavior*) e outras para animação (por exemplo, *Interpolator* e *Billboard*).

Os passos para criar uma subclasse de *Behavior* são:

1. Implementar pelo menos um construtor para a subclasse que armazene uma referência para o objeto que será alterado;
2. Sobrescrever o método *initialization()* especificando um critério para começar a interação;
3. Sobrescrever o método *processStimulus()*, responsável pela execução da ação.

Estes passos estão ilustrados no próximo trecho de código, extraído de *Sun Microsystems* [1]. Neste exemplo, o objeto *TransformGroup* passado como parâmetro para o construtor sofre uma rotação sempre que o usuário clicar com o botão do *mouse*.

```
1. /*
2.  *  @(#)SimpleBehaviorApp.java
3.  *
4.  * Copyright (c) Sun Microsystems, Inc.
5.  * 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A
6.  * All Rights Reserved.
7.  */
8. private class SimpleBehavior extends Behavior
9. {
10.  private TransformGroup targetTG;
11.  private Transform3D rotation = new Transform3D();
12.  private double angle = 0.0;
13.
14.  SimpleBehavior(TransformGroup targetTG)
15.  {
16.    this.targetTG = targetTG;
17.  }
18.
19.  public void initialize()
20.  {
21.    this.wakeupOn(new WakeupOnAWTEvent(MouseEvent.MOUSE_CLICKED));
22.  }
23.
24.  public void processStimulus(Enumeration criteria)
25.  {
26.    angle += 0.1;
27.    rotation.rotY(angle);
28.    targetTG.setTransform(rotation);
29.    this.wakeupOn(new WakeupOnAWTEvent(MouseEvent.MOUSE_CLICKED));
30.  }
31. }
```

A classe *Behavior* fornece um conjunto de funcionalidades para incluir ações definidas pelo usuário no grafo de cena, além de especificar um critério para começar a interação. O método abstrato *initialize* é chamado apenas uma vez, e o método *processStimulus*, também abstrato, faz o processamento de uma ação em resposta a um estímulo. O método *wakeupOn* define o critério para fazer interação, que é recebido como parâmetro através de um objeto *WakeupOnAWTEvent*. A classe *WakeupOnAWTEvent*, usada para indicar que uma interação deve ser realizada quando um determinado evento AWT ocorre, possui um construtor que recebe a constante que identifica este evento AWT (linhas 21 e 29 do trecho de código anterior) [13].

Outra forma de interagir com os objetos em Java 3D é através da classe *OrbitBehavior*. Neste caso, a *View* é movida em torno do ponto de interesse quando o mouse é arrastado com o botão pressionado. Inclui ações de rotação, translação e *zoom*, obtidas, respectivamente, da seguinte maneira: arrastar o mouse com o botão esquerdo pressionado; arrastar o mouse com o botão direito pressionado; arrastar o mouse com o botão do meio pressionado (ou alt+botão esquerdo). O próximo trecho de código ilustra como o exemplo do Anexo 1 pode ser alterado para incluir este tipo de interação.

```
1. :
2. BranchGroup scene = criaGrafoDeCena();
3. universe = new SimpleUniverse(canvas);
4.
5. ViewingPlatform viewingPlatform = universe.getViewingPlatform();
6. viewingPlatform.setNominalViewingTransform();
7.
8. // Adiciona "mouse behaviors" à "viewingPlatform"
9. OrbitBehavior orbit = new OrbitBehavior(canvas,
10.                                        OrbitBehavior.REVERSE_ALL);
11.BoundingSphere bounds = new BoundingSphere
12.                                  (new Point3d(0.0,0.0,0.0), 100.0);
13.orbit.setSchedulingBounds(bounds);
14.viewingPlatform.setViewPlatformBehavior(orbit);
15.universe.addBranchGraph(scene);
16.:
```

3.4 Animação

De forma simplificada, animação consiste na exibição de imagens em seqüência. Em Java 3D, a interação corresponde à execução de alterações no grafo de cena em resposta a ações do usuário, e a animação é definida como a execução de alterações com a passagem do tempo, e não com uma ação do usuário. A API Java 3D fornece um grande número de subclasses da classe *Behavior* que são úteis para criar animações: *Billboard*, *Interpolator* e *LOD*.

As subclasses *Billboard* e *LOD* animam objetos em resposta a mudanças de posição e orientação de visualização, e não de acordo com o tempo. A primeira é usada para orientar de forma automática um polígono com textura para fique ortogonal à posição de observação. No caso da classe *LOD*, objetos complexos são representados por um conjunto de objetos que possuem diferentes níveis de detalhe, que variam de acordo com a distância do observador virtual (quanto mais longe, menos detalhes) [1].

Interpolator possui um conjunto de subclasses que, em conjunto com objetos *Alpha*, manipulam alguns parâmetros de um grafo de cena para criar uma animação baseada no tempo. Algumas de suas subclasses são: *ColorInterpolator*, *PathInterpolator*, *PositionInterpolator*, *RotationInterpolator* e *ScaleInterpolator*. O processo de animação através da classe *RotationInterpolator* está exemplificado no programa do Anexo 1, onde o cubo é constantemente rotacionado ao redor do eixo y (linhas 122-130).

A classe *Alpha* fornece métodos para converter um valor de tempo em um valor entre 0.0 e 1.0 (*f(t) = [0.0,1.0]*). Estes valores são úteis para as subclasses de *Interpolator*. As constantes INCREASING_ENABLE e DECREASING_ENABLE, que servem para indicar se os valores serão gerados na ordem crescente ou decrescente, são definidas nesta classe. Seus construtores criam e inicializam o objeto com valores pré-definidos, e também permitem que o usuário especifique valores diferentes para seus atributos. Por exemplo, com o construtor *Alpha (int loopCount, long increasingAlphaDuration)*, no primeiro parâmetro é possível definir o número de vezes que o objeto será executado (-1 para um laço) e o período de tempo durante o qual *Alpha* vai de zero para um [13].

A classe *RotationInterpolator* define uma animação que modifica o componente de rotação do *TransformGroup* através de uma interpolação linear entre um par de ângulos especificados (usando o valor gerado por um objeto *Alpha*). O ângulo interpolado é usado para indicar uma transformação de rotação sobre o eixo Y. De maneira análoga, *PositionInterpolator* e *ScaleInterpolation* modificam, respectivamente, o componente de translação e de escala através de uma interpolação linear entre um par de posições ou de valores de escala (ambos usam o valor gerado por um objeto *Alpha*). A posição interpolada para *PositionInterpolator* é usada para gerar uma transformação de translação sobre o eixo X, e para *ScaleInterpolation* é usada para gerar uma transformação de escala sobre o sistema de coordenada do *interpolator*. [13].

4 Comentários Finais

Neste artigo foi apresentada a estrutura e algumas funcionalidades da API Java 3D, uma ferramenta (ou *toolkit*) orientada a objetos que utiliza o conceito de grafo de cena para representar objetos 3D [14]. Conceitos básicos para a criação de universos virtuais foram abordados na seção 2, e algumas alternativas para gerar imagens com maior realismo foram descritas na seção 3. Java 3D é uma poderosa API que disponibiliza diversas funcionalidades, tais como colisões, sensores e som 3D, que não foram mencionadas aqui. Para obter mais informações recomenda-se consultar a documentação da *Sun* [13, 1] ou livros já publicados [12, 15, 16].

Java 3D não é apenas um formato de arquivo 3D, tal como DXF (*Auto-CAD Drawing Interchange File*). Ao contrário de VRML, que consiste, basicamente, numa linguagem de descrição de ambientes virtuais, Java 3D é uma API de programação completa. Por outro lado, é uma API de mais alto nível de abstração quanto à programação gráfica quando comparada com OpenGL e DirectX. Similares a Java 3D, Open Inventor™ [17] e OpenGL Performer™ [18] oferecem uma solução para problemas de programação de ambientes gráficos interativos 3D, baseada na criação de um grafo de cena. Apesar das primitivas também serem geradas através de rotinas OpenGL, a programação em Open Inventor e OpenGL Performer é feita utilizando a linguagem C++.

Resumindo, apesar da API Java 3D fornecer várias funcionalidades para a criação de ambientes gráficos 3D, ainda necessita muita implementação para o desenvolvimento de novas aplicações. Além disso, OpenGL e DirectX devem estar instaladas para permitir a visualização do universo virtual. Se comparada a Open Inventor, além da diferença na linguagem de programação utilizada [14], Java 3D gerencia o grafo de cena de maneira mais otimizada, obtendo, assim, um melhor desempenho [16]. OpenGL Performer também possui um desempenho superior, mas não está disponível gratuitamente.

Conforme mencionado na seção 1, Java 3D pode ser usada em diferentes tipos de aplicações, de visualização científica [6] ao desenvolvimento de interfaces [19] e ambientes virtuais para navegação. Recentemente, a tecnologia Java, incluindo a API Java 3D, também foi utilizada em uma CAVE construída para pesquisa em Bioinformática [20]. Estas

aplicações mostram que a API Java 3D pode ser utilizada para diferentes finalidades, incorporando todas as vantagens da programação na linguagem Java, que facilita a construção de interfaces gráficas. Sua desvantagem, entretanto, está no desempenho: a execução de aplicações implementadas na linguagem de programação C/C++ que utilizam OpenGL ou DirectX geralmente é mais rápida. A opção pela sua utilização irá depender dos objetivos da aplicação e do nível de abstração no qual se deseja trabalhar. Por apresentar um grande número de funcionalidades, a programação em Java 3D pode ser mais produtiva quando comparada com OpenGL.

Referências

[1] Sun Microsystems Java 3D Engineering Team. *Java 3D API Tutorial*. Disponível em http://developer.java.sun.com/developer/onlineTraining/java3d/ (setembro 2003).

[2] A.L. Ames, D.R. Nadeau, J.L. Moreland. *VRML 2.0 Sourcebook*. 2nd ed. New York: John Wiley, 1997. 654 p.

[3] K. Brown. *Ready-to-run Java 3D*. New York, NY: John Wiley & Sons, 1999. 400 p.

[4] R.S. Wright Jr., M. Sweet. *OpenGL SuperBible*. 2nd ed. Indianapolis, Indiana: Waite Group Press, 2000. 696 p.

[5] R. Dunlop. *Sams teach yourself directx 7 in 24 hours*. Indianapolis, IN: Sams, 2000. 583 p.

[6] B. Hibbard, A. Donaldson, L. Matthews et al. *VisAD*. Disponível em http://www.ssec.wisc.edu/~billh/visad.html (julho 2003).

[7] H.M. Deitel. *Java How to Program*. Upper Saddle River, NJ: Prentice Hall, 2002. 1546 p.

[8] J.D. Foley et al. *Computer Graphics: Principles and Practice*. New York, Addison Wesley, 1990, 2nd ed.

[9] D. Hearn. *Computer Graphics*. 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1997. 652 p.

[10] O. Lathrop. *The Way Computer Graphics Works*. Wiley Computer Publishing, 1997.

[11] M. Kölling. *BlueJ - The Interactive Java Environment*. Disponível em http://www.bluej.org (julho 2003).

[12] H. Sowizral, K. Rushforth, M. Deering. *The JavaTM 3D API Specification*. 2nd Edition. Addison-Wesley. 1998. 482 p.

[13] Sun Microsystems, Inc. *Java 3D 1.2 API Documentation*. Disponível em http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_2_API/j3dapi/ (setembro 2003).

[14] A.L. Bicho, L.G. da Silveira Jr, A.J.A. da Cruz e A.B. Raposo. Programação Gráfica 3D com OpenGL, Open Inventor e Java 3D. *REIC - Revista Eletrônica de Iniciação Científica*. v. II, n. I, março, 2002. Disponível em http://www.sbc.org.br/reic/edicoes/2002e1/tutoriais/ ProgramacaoGrafica3DcomOpenGLOpenInventoreJava3D.pdf (julho 2003).

[15] D. Selman. *Java 3D Programming.* Manning Publications Company. 2002. 400 p.

[16] A.E. Walsh. *Java 3D: API Jump-start.* Upper Saddle River, NJ: Prentice Hall, 2002. 245 p.

[17] J. Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor.* 1st ed. Addison-Wesley Pub Co., 1994. 514 p.

[18] Silicon Graphics, Inc. *OpenGL Performer Overview.* Disponível em http://www.sgi.com/software/performer/overview.html/ (setembro 2003).

[19] J. Barrilleaux. *3D User Interfaces with Java 3D.* Manning Publications Co. 2001. 499 p.

[20] C.W. Sensen et al. *Establishment of the first Java 3D™ enabled CAVE® – a whitepaper*. Disponível em http://java.sun.com/products/java-media/3D/ calgary.whitepaper.pdf (setembro 2003).

Anexo 1 - Exemplo de um programa Java 3D completo [1]

```
1. /*
32.  *  @(#)HelloUniverse.java 1.55 02/10/21 13:43:36
33.  *
34.  * Copyright (c) 1996-2002 Sun Microsystems, Inc.
35.  * All Rights Reserved.
36.  *
37.  * Redistribution and use in source and binary forms,with
38.  * or without modification, are permitted provided that
39.  * the following conditions are met:
40.  * - Redistributions of source code must retain the above
41.  *   copyright notice, this list of conditions and the
42.  *   following disclaimer.
43.  * - Redistribution in binary form must reproduce the
44.  *   above copyright notice, this list of conditions and
45.  *   the following disclaimer in the documentation and/or
46.  *   other materials provided with the distribution.
47.  */
48.
49.//////////////////////////////////////////////////////
50.// Isabel Harb Manssour - Junho de 2003
51.// HelloUniverseJFrame.java ilustra como criar uma
52.// aplicação simples em Java 3D.
53.// Este código está baseado no demo HelloUniverse.java
54.
55.import javax.swing.*;
56.import java.awt.*;
57.import com.sun.j3d.utils.geometry.ColorCube;
58.import com.sun.j3d.utils.universe.*;
59.import javax.media.j3d.*;
60.import javax.vecmath.*;
61.
62.public class HelloUniverseJFrame extends JFrame
63.{
64.   ////////////////////////////////////////////////
65.   // Atributo da classe HelloUniverseJFrame
66.
67.   private SimpleUniverse universe = null;
68.
69.
70.   ////////////////////////////////////////////////
71.   // Construtor da classe HelloUniverseJFrame
72.   public HelloUniverseJFrame()
73.   {
74.     Container container = getContentPane();
75.     container.setLayout(new BorderLayout());
76.     setTitle("Color Cube");
77.     GraphicsConfiguration config =
78.        SimpleUniverse.getPreferredConfiguration();
79.
80.     Canvas3D canvas = new Canvas3D(config);
```

```
81.     container.add("Center", canvas);
82.
83.     // Cria um sub-grafo de conteúdo
84.     BranchGroup scene = criaGrafoDeCena();
85.     universe = new SimpleUniverse(canvas);
86.
87.     // O código abaixo faz com que a ViewPlatform seja
88.     // movida um pouco para trás, para que os objetos
89.     // possam ser visualizados.
90.     universe.getViewingPlatform().
91.                           setNominalViewingTransform();
92.
93.     // Anexa o sub-grafo no universo virtual
94.     universe.addBranchGraph(scene);
95.
96.     setSize(350,350);
97.     setVisible(true);
98.   }
99.
100.
101.     ////////////////////////////////////////////////
102.     // Método responsável pela criação do grafo de cena
103.     // (ou sub-grafo)
104.     public BranchGroup criaGrafoDeCena()
105.  {
106.    // Cria o nó raiz
107.    BranchGroup objRaiz = new BranchGroup();
108.    // Cria o nó TransformGroup e permite que ele possa
109.    // ser alterado em tempo de execução(TRANSFORM_WRITE).
110.    // Depois, adiciona-o na raiz do grafo de cena.
111.    TransformGroup objTrans = new TransformGroup();
112.    objTrans.setCapability(
113.                          TransformGroup.ALLOW_TRANSFORM_WRITE);
114.    objRaiz.addChild(objTrans);
115.
116.    // Cria um cubo colorido (Shape3D) e o adiciona no grafo.
117.    objTrans.addChild(new ColorCube(0.4));
118.    // Cria um novo objeto Behaviour que irá executar as
119.    // operações desejadas no "transform" especificado
120.    // e adiciona-o no grafo.
121.    Transform3D eixoY = new Transform3D();
122.    Alpha rotacaoAlpha = new Alpha(-1, 4000);
123.
124.    RotationInterpolator rotator =
125.         new RotationInterpolator(rotacaoAlpha, objTrans, eixoY,
126.                                  0.0f, (float) Math.PI*2.0f);
127.    BoundingSphere bounds =
128.         new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
129.    rotator.setSchedulingBounds(bounds);
130.    objRaiz.addChild(rotator);
131.
132.    // Para o Java 3D realizar otimizações no grafo de cena
133.    objRaiz.compile();
134.
135.    return objRaiz;
136.  }
```

```
137.
138.   ////////////////////////////////////////////////
139.   // Método principal que permite executar a aplicação
140.   public static void main(String[] args)
141.   {
142.      HelloUniverseJFrame h = new HelloUniverseJFrame();
143.   }
144. }
```