In: V. Teichrieb, F. L. S. Nunes, L. S. Machado, R. Tori (orgs), Realidade Virtual e Aumentada na Prática – Livro dos Minicursos SVR 2008, p.34-59, Gráfica e Copiadora Nacional, Recife, PE, 2008 (ISBN 978-85-7669-179-2).

OpenSceneGraph: conceitos básicos e aplicações em realidade virtual e aumentada com ARToolKit

Luiz Gonzaga da Silveira Jr
Feevale - UNISINOS
gonzaga <at> acm.org

Alberto Barbosa Raposo
Tecgraf / PUC-Rio
abraposo <at> tecgraf.puc-rio.br

Abstract

In this chapter, we give a short introduction to OpenSceneGraph (OSG), through basic concepts and practical examples. Besides, we highlight the benefits of their use in the development of applications in Virtual Reality (VR) and Augment Reality (AR), through the conjunction with the ARToolKit. This synergy provides support for the rapid applications development (RAD), with high-quality rendering and leaving transparent the complex recognition of the markers to the developers.

Resumo

Este capítulo apresenta a OpenSceneGraph (OSG) através de conceitos básicos, ilustrados com exemplos práticos, destacando os benefícios da sua utilização no desenvolvimento de aplicações em Realidade Virtual (RV) e aumentada (RA) em conjunto com a ARToolKit. Essa sinergia oferece o suporte para o desenvolvimento rápido de aplicações com renderização de alta qualidade e deixando transparente para o desenvolvedor a complexidade no reconhecimento de marcadores.

1.1. Introdução

Este documento apresenta uma introdução ao Open Scene Graph (OSG) [Kuehne and Martz 2007, Martz 2007], um *toolkit* de código aberto e multi-plataforma para aplicações gráficas de alto desempenho. Escrito em C++ padrão

e OpenGL, o OSG pode ser executado em todas as plataformas Windows, além dos sistemas operacionais OSX, GNU/Linux, IRIX, Solaris, HP-Ux, AIX e FreeBSD. O OSG é voltado para áreas como simulação visual, jogos, realidade virtual, modelagem e visualização científica, dentre outras.

O OSG é uma camada de abstração de alto-nível e extensível desenvolvida sobre o OpenGL, provendo uma interface de programação mais produtiva para o desenvolvimento de aplicações de computação gráfica. Uma estrutura hierárquica para renderização eficiente, o emprego de técnicas de gerenciamento de memória, e capacidade para lidar como modelos 2D/3D provêem os recursos para fazer do OSG uma boa escolha para programadores de aplicações gráficas.

Antes de discutirmos o Open Scene Graph (OSG), é interessante explorarmos um pouco uma questão fundamental.

1.1.1. A questão é: "o que é um grafo de cena?"

Um grafo de cena é uma estrutura de dados usada para organizar a cena em uma aplicação gráfica. Parte-se do princípio que uma cena é normalmente decomposta em várias partes diferentes, que precisam ser "ligadas" de alguma forma. Um grafo de cena é então um grafo onde cada nó representa uma das partes em que a cena pode ser dividida. Sendo mais rigoroso, um grafo de cena é um grafo acíclico direcionado que estabelece uma relação entre os nós (partes da cena).

Suponha que você queira renderizar uma cena contendo uma estrada e um caminhão. Um grafo representando essa cena é ilustrado na Figura 1.1.

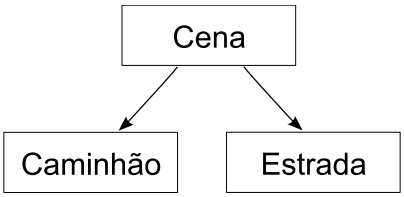

Figura 1.1. Um grafo de cena contendo uma estrada e um caminhão.

Na maioria das vezes é necessário transladar os objetos da cena para que eles sejam renderizados na posição desejada. Dessa forma, os nós do grafo de cena não devem se limitar a representar geometrias.[1] Nesse caso, pode-se incluir um nó representando a translação, levando ao grafo de cena mostrado na Figura 1.2.

Até agora, os exemplos apresentados foram de fato árvores, e não grafos, mas esse não é sempre o caso. Vamos adicionar duas caixas à cena: uma no caminhão e outra na estrada. Ambas as caixas terão nós de translação sobre elas, de forma que possam ser posicionadas no lugar correto. Além disso, a caixa do caminhão também deve ser transladada com a translação do caminhão, pois ela se move junto com ele. A novidade é que, uma vez que as duas caixas são objetos idênticos, você não precisa criar um nó para cada uma delas. Um nó "referenciado" duas vezes faz esse truque, como a Figura 1.3

[1] Na verdade, o nó "Cena" na Figura 1.1 não representa um objeto geométrico, ele representa um grupo de nós (no caso, "Caminhão" and "Estrada").

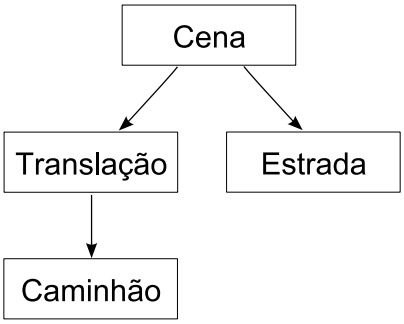

Figura 1.2. Um grafo de cena contendo uma estrada e um caminhão transladado.

ilustra. Durante a renderização, o nó "Caixa" será visitado (e renderizado) duas vezes, mas o gasto de memória é reduzido, visto que o modelo só é carregado uma vez.

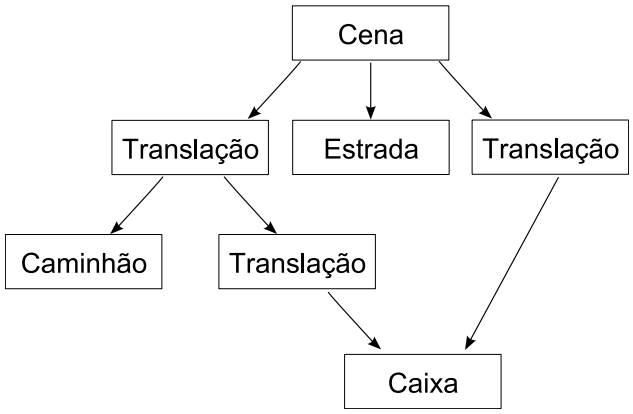

Figura 1.3. Um grafo de cena contendo uma estrada, um caminhão e duas caixas.

Claro que grafos de cena podem se tornar muito mais complexos que os exemplos acima. Porém, a noção aqui apresentada já é suficiente por enquanto. Então, é hora de discutir uma segunda questão fundamental.

1.1.2. A questão é: "a quem interessa?"

A resposta é simples: interessa a qualquer um que queira uma estrutura de dados "limpa" para organizar uma cena de Computação Gráfica e uma renderização eficiente. O OSG não só expõe a geometria dos modelos e estado de renderização relacionados ao OpenGL, mas também provê recursos adicionais. A estrutura do grafo de cena é adequada para uma organização espacial intuitiva, descarte por *frustum* de visualização e oclusão, níveis de detalhes, minimização de mudanças de estado, suporte a efeitos de renderização, dentre outros aspectos.

A mais simples implementação de grafo de cena provê mecanismos para armazenagem de geometria e aparência, e desenha a cena percorrendo a estrutura do grafo. Assim, a principal tarefa deste percorrimento do grafo é enviar o dado armazenado para a placa gráfica, por meio de chamadas OpenGL. O OSG realiza o percorrimento considerando atualizações da cena, descartes e finalmente o desenho. É importante ressaltar que em alguns casos como o de aplicações estéreo, é necessário mais de um percorrimento

do grafo por quadro gerado.

1.1.3. Finalmente algo relacionado ao OSG

Até aqui, nossa discussão era sobre grafos de cena "genéricos". De agora em diante, os exemplos usarão exclusivamente grafos de cena OSG ou seja, ao invés de usarmos um nó "Translação" genérico, vamos usar uma instância de uma classe real definida na hierarquia do OSG.

Um nó na hierarquia do OSG é representado pela classe `osg::Node`. Apesar de tecnicamente possível, não é muito útil a instanciação de `osg::Node`s. As coisas ficam interessantes quando olhamos as subclasses de `osg::Node`. Nesta seção, três dessas subclasses serão apresentadas: `osg::Geode`, `osg::Group` e `osg::PositionAttitudeTransform`.

Objetos renderizáveis são representados como instâncias da classe `osg::Drawable`. Porém, `osg::Drawable`s não são nós, de modo que não podemos colocá-las diretamente no grafo de cena. É necessário usar um nó geométrico para isso: `osg::Geode` (*geometry node*).

Nem todo nó de um grafo OSG pode ter outros nós ligados a ele como filhos. De fato, só nós que são instâncias de `osg::Group` ou de uma de suas subclasses podem ter nós filhos.

Usando `osg::Geode`s e um `osg::Group`, é possível recriar o grafo da Figura 1.1 usando classes reais do OSG. O resultado é mostrado na Figura 1.4.

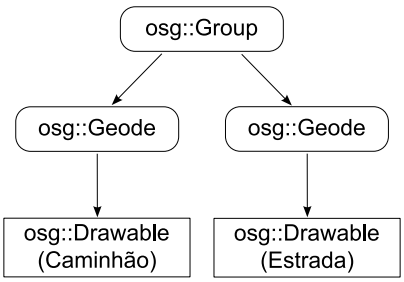

Figura 1.4. Um grafo de cena OSG contendo uma estrada e um caminhão. Instâncias de classes OSG derivadas de `osg::Node` são desenhadas em caixas arredondadas. `osg::Drawable`s são representados como retângulos.

O grafo da Figura 1.4 não é a única maneira de traduzir o grafo da Figura 1.1 em um grafo OSG real. Mais de um `osg::Drawable` pode estar ligado a um único `osg::Geode`, de modo que o grafo de cena da Figura 1.5 também é uma representação OSG do mesmo grafo.

Os grafos das Figuras 1.4 e 1.5 têm os mesmo problema que o da Figura 1.1: o caminhão provavelmente precisará ser transladado para aparecer na posição correta. A solução é a mesma de antes: transladar o caminhão. Em OSG, provavelmente a maneira mais simples de transladar um nó é adicionar um nó `osg::PositionAttitudeTransform` sobre ele. Um `osg::PositionAttitudeTransform` tem associado a ele não só uma translação, mas também uma orientação e uma escala. Apesar de não ser exatamente a mesma coisa,

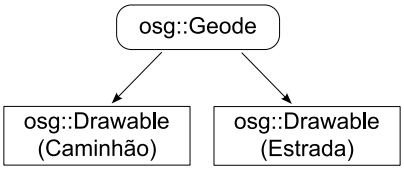

Figura 1.5. Grafo OSG alternativo representando a mesma cena da Figura 1.4.

pode-se imaginá-lo como o equivalente OSG das chamadas OpenGL `glTranslate()`, `glRotate()` e `glScale()`. A Figura 1.6 é a versão OSG da Figura 1.2.

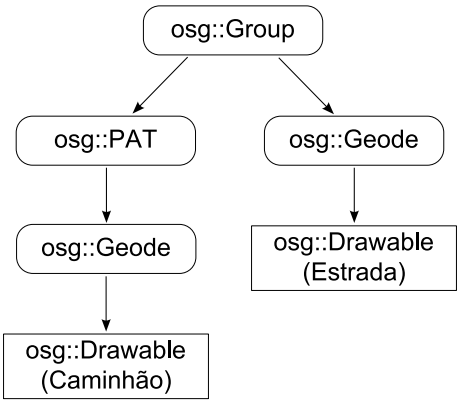

Figura 1.6. Grafo de cena OSG contendo uma estrada e um caminhão transladado. Por questão de compactação, o `osg::PositionAttitudeTransform` é escrito como `osg::PAT`.

Para completar, a Figura 1.7 mostra a representação OSG do grafo "genérico" da Figura 1.3.

1.1.4. Gerenciamento de Memória

Uma vez que o OSG faz uso extensivo de *smart pointers*[2], é válido gastarmos algum tempo com uma explicação a respeito. Não ouse pular esta seção se "smart pointers" lhe parece grego (e você não é grego!).

Começamos com uma definição: um *recurso* é alguma coisa que deve ser alocada antes de ser usada e desalocada quando não for mais necessária. Possivelmente, o recurso mais comum que usamos quando programamos é a memória heap, mas existem muitos outros exemplos. Um caso comum são arquivos, que devem ser fechados depois de abertos. Também em OpenGL há alguns exemplos de recursos. Um exemplo são os nomes de texturas gerados por `glGenTextures()`, que devem ser liberados por `glDeleteTextures()`).

O que há de mais fascinante sobre os recursos é o fato de que há muitos programadores que acreditam que são capazes de escrever "no braço" código capaz de liberá-los em qualquer caso *e* que nunca vão esquecer de escrever o código para isso. Esse pensamento só leva a "vazamentos" de recursos. A boa notícia é que, com alguma disciplina,

[2] Aliás, todo programa deveria fazê-lo.

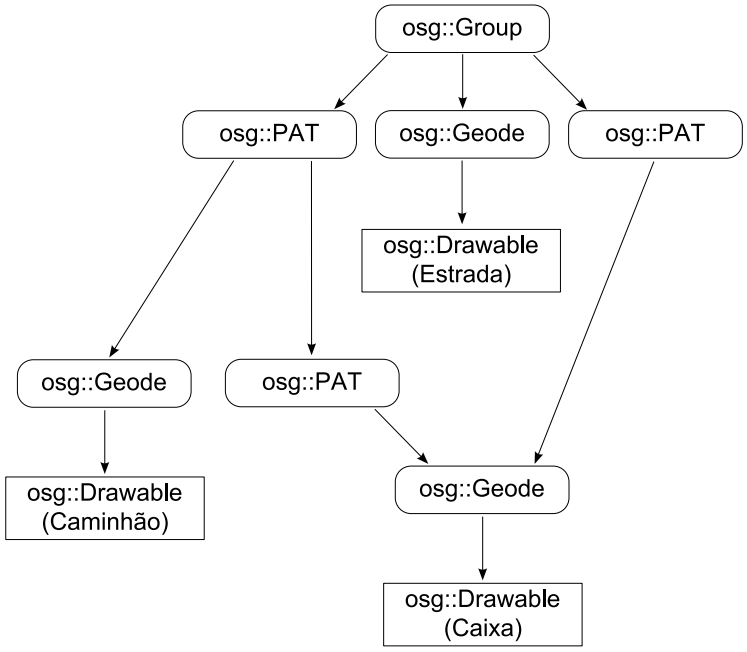

Figura 1.7. Grafo OSG contendo uma estrada, um caminhão e duas caixas.

podemos deixar a tarefa de liberar o recurso para o compilador C++, que é bem mais confiável que a gente para tarefas como essa.

Vamos discutir algumas das principais idéias por trás do gerenciamento de recursos em C++ mas uma discussão completa a respeito foge do escopo deste texto. E por falar em "escopo", o escopo das variáveis "automáticas" (i.e., variáveis alocadas na pilha) tem papel central no gerenciamento de recursos em C++: as regras da linguagem garantem que o destrutor de um objeto alocado na pilha será chamado quando ele ficar fora do escopo. E como isso ajuda a evitar o vazamento de recursos? Observe a classe a seguir:

```
class ThingWrapper
{
   public:
      ThingWrapper() { handle_ = AllocateThing(); }
      ~ThingWrapper() { DeallocateThing (handle_); }
      ThingHandle& get() { return handle_; }
   private:
      ThingHandle handle_;
};
```

A classe aloca algo (`Thing`) no construtor e o libera no destrutor. Então, sempre que precisarmos dessa `Thing` podemos fazer algo como:

```
ThingWrapper thing;
UseThing (thing.get());
```

Ao instanciarmos um `ThingWrapper` a Thing é alocada (pelo construtor do `ThingWrapper`). E a parte boa é que a Thing será automaticamente liberada quando

ela ficar fora de escopo, uma vez que seu destrutor garantidamente será executado quando isso acontecer. Voilà! Gerenciamento automático de recurso!

A classe `ThingWrapper` é um exemplo de uma técnica de programação em C++ comumente chamada de RAII (resource acquisition is initialization). Um *smart pointer* é simplesmente uma classe (ou mais comumente, um template de classe) que usa a técnica RAII para gerenciar automaticamente a memória heap. Bem parecido com o `ThingWrapper`, mas ao invés de chamar hipotéticos `AllocateThing()` e `DeallocateThing()`, um *smart pointer* tipicamente recebe um ponteiro para a nova memória alocada em seu construtor e usa o operador C++ `delete` para liberar essa memória no destrutor.

No exemplo do `ThingWrapper`, `thing` é considerado o dono da Thing alocada com `AllocateThing()`, e portanto é responsável por desalocá-la. Em OSG há um detalhe adicional para complicar um pouco mais: às vezes o objeto tem mais de um dono.[3] Por exemplo, no grafo da Figura 1.7, o `osg::Geode` conectado à caixa tem dois pais. Qual dos dois deve ser o responsável por desalocá-lo?

Nesses casos, o recurso não deve ser desalocado enquanto houver pelo menos uma referência apontando para ele. Por isso, a maioria dos objetos em OSG tem um contador interno para registrar o número de referências apontando para ele.[4] O recurso (i.e., o objeto) será destruído apenas quando seu contador de referências interno chegar a zero.

Felizmente, os programadores não precisam cuidar manualmentes desses contadores: é para isso que existem os *smart pointers*. Em OSG, os *smart pointers* são implementados como um template de classe chamado `osg::ref_ptr<>`. Sempre que um objeto OSG recebe um ponteiro para um outro objeto OSG isso é imediatamente armazenado em um `osg::ref_ptr<>`. Desse modo, o contador de referências do objeto em questão é gerenciado automaticamente, e o mesmo será automaticamente desalocado quando não for mais referenciado por ninguém.

O exemplo abaixo mostra *smart pointers* do OSG em ação.

SmartPointers.cpp

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <osg/Geode>
4  #include <osg/Group>
5
6  void MayThrow()
7  {
8     if (rand() % 2)
9        throw "Aaaargh!";
10 }
11
12 int main()
13 {
14    try
15    {
16       srand(time(0));
```

[3] Essa complicação adicional não é exclusividade do OSG. Posse compartilhada (*shared ownership*) é uma situação razoavelmente comum na prática.

[4] Sendo mais exato, os objetos com um contador de referências embutido são todos aqueles que são instâncias de classes derivadas de `osg::Referenced`.

```
17       osg::ref_ptr<osg::Group> group (new osg::Group());
18
19       // Isso está OK, apesar de um pouco loquaz.
20       osg::ref_ptr<osg::Geode> aGeode (new osg::Geode());
21       MayThrow();
22       group->addChild (aGeode.get());
23
24       // Isso é seguro, também.
25       group->addChild (new osg::Geode());
26
27       // Isso é perigoso! Não faça!
28       osg::Geode* anotherGeode = new osg::Geode();
29       MayThrow();
30       group->addChild (anotherGeode);
31
32       // Diga tchau!
33       std::cout << "Oh, muito bom. Sem exceções. sem vazamentos.\n";
34    }
35    catch (...)
36    {
37       std::cerr << "'anotherGeode' possivelmente vazado!\n";
38    }
39 }
```

Sobre o exemplo acima, a primeira coisa a notar é que ele dá uma primeira idéia de como criar grafos de cena como os mostrados nas figuras da Seção 1.1.3. (Por enquanto, é questão de curiosidade. A próxima seção tratará disso.) A intenção do exemplo é na verdade mostrar duas maneiras seguras de usar os *smart pointers* do OSG e uma maneira perigosa para NÃO usá-los:

- As linhas 20 a 22, mostram uma maneira segura de usar os *smart pointers*: um `osg::ref_ptr<>` (chamado `aGeode`) é explicitamente criado e inicializado com um recém alocado `osg::Geode` (o recurso) na linha 20. Nesse ponto, o contador de referências do geode alocado na memória heap fica igual a 1 (uma vez que há apenas um `osg::ref_ptr<>`, o `aGeode`, apontando para ele.)

  Um pouco abaixo, na linha 22, o geode é colocado como filho de um nó grupo. Quando isso ocorre, o grupo incrementa as referências ao geode para 2.

  Agora, o que acontece se algo de ruim ocorrer? O que acontece se a chamada a `MayThrow()` na linha 21 realmente executar o *throw*? O `aGeode` ficará fora de escopo e será destruído. Seu destrutor irá decrementar o contador de referências do geode. E, uma vez que ele foi decrementado para zero, ele também libera o geode adequadamente, sem vazamento de memória.

- A linha 25 faz mais ou menos a mesma coisa do caso anterior. A difereça é que o geode é alocado com `new` e colocado como filho do nó grupo em uma única linha de código. Isso é seguro também, pois não há nada de ruim que possa acontecer nesse momento.

- A maneira errada, perigosa e desaconselhável de gerenciar a memória é mostrada nas linhas 28 a 30. Se parece com o primeiro caso, porém o geode é alocado com `new` mas armazenado num ponteiro "dumb". Se o `MayThrow()` na linha 29 executar o *throw*, ninguém vai chamar o `delete` no geode, e ele vai "vazar".

Há uma outra coisa que pode ser dita aqui: o destrutor de `osg::Referenced` não é sequer público, de modo que não se pode dizer `delete anotherGeode`. Instâncias de classes derivadas de `osg::Referenced` (e.g., `osg::Geode`) são simplesmente feitas para serem gerenciadas automaticamente usando `osg::ref_ptr<>`s.

Então, faça a coisa certa e nunca escreva código como no terceiro caso acima!

1.2. Dois Visualizadores 3D

Nesta seção, finalmente teremos programas OSG que mostram alguma coisa na tela. Serão apresentados 2 visualizadores de modelos 3D para ilustrar vários conceitos.

1.2.1. Um visualizador muito simples

O primeiro visualizador é muito simples. Basicamente, o que ele faz é carregar um arquivo passado como parâmetro na linha de comando e mostrá-lo na tela. Sem maiores delongas, segue seu código:

VerySimpleViewer.cpp

```
1  #include <iostream>
2  #include <osgDB/ReadFile>
3  #include <osgProducer/Viewer>
4
5  int main (int argc, char* argv[])
6  {
7     // Checa parâmetros da linha de comando
8     if (argc != 2)
9     {
10       std::cerr << "Uso: " << argv[0] << " <arquivo com modelo>\n";
11       exit (1);
12    }
13
14    // Cria um visualizador baseado em um Producer
15    osgProducer::Viewer viewer;
16    viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);
17
18    // Carrega o modelo
19    osg::ref_ptr<osg::Node> loadedModel = osgDB::readNodeFile(argv[1]);
20
21    if (!loadedModel)
22    {
23       std::cerr << "Problema ao abrir '" << argv[1] << "'\n";
24       exit (1);
25    }
26
27    viewer.setSceneData (loadedModel.get());
28
29    // Entra no loop de renderização
30    viewer.realize();
31
32    while (!viewer.done())
33    {
34       // Espera todas as threads de descarte e de desenho completarem.
35       viewer.sync();
36
37       // Atualiza a cena percorrendo-a com o update visitor, que
38       // chamará todos os callbacks de atualziação e animações do nó.
39       viewer.update();
40
41       // Dispara os cull e draw traversals da cena.
```

```
42          viewer.frame();
43      }
44
45      // Espera todas as threads de descarte e de desenho completarem antes de sair.
46      viewer.sync();
47  }
```

Esse exemplo é bastante simples, mas há algumas coisas a se considerar. Primeiro, observe que o OSG, tal como o OpenGL, é independente do sistema de janelas. Então, a tarefa de criar um janela com um contexto OpenGL próprio para desenho não é tratada pelo OSG. Nesse primeiro exemplo (e na maioria dos outros que virão), este trabalho é realizado por uma biblioteca chamada Open Producer (ou simplesmente Producer). O Producer é projetado para ser eficiente, portável, escalável e pode ser facilmente usado com o OSG.

Portanto, este primeiro exemplo usa um visualizador baseado em Producer, instanciado na linha 15. A linha 16 inicializa o visualizador com configurações padrão, o que inclui muitos recursos úteis. Não há espaço para descrevermos estes recursos aqui, mas tente apertar teclas e mover o mouse enquanto executa um exemplo para descobrir alguns desses recursos.

O OSG sabe como ler (e escrever) vários formatos de modelos 3D e imagens, e todas as funções e classes relacionadas a isso são declaradas no namespace `osgDB`. A linha 19 usa uma dessas funções, a `osgDB::readNodeFile()`, que recebe como parâmetro o nome de um arquivo contendo um modelo 3D e retorna um ponteiro para um `osg::Node`. O nó retornado contém toda a informação necessária para renderizar a cena 3D adequadamente, incluindo, por exemplo, vértices, polígonos, normais e mapas de texturas.

O nó retornado por `osgDB::readNodeFile()` está pronto para ser adicionado a um grafo de cena. De fato, esse é um exemplo bem simples, e o nó é todo o grafo de cena: observe que na linha 27 dizemos ao visualizador o que se espera visualizar, e é exatamente o nó que recebemos ao chamar a `osgDB::readNodeFile()`.

A chamada na linha 30 "materializa" a janela do visualizador, i.e., cria uma janela com um contexto OpenGL próprio. E isso é tudo que o programa faz. Desse ponto em diante, há apenas algumas chamadas adicionais para garantir que o programa continua executando para sempre.[5] O loop da linha 32 até a 43 é o típico loop principal de uma aplicação baseada em OSG e Producer. A chamada final da linha 46 simplesmente espera que eventuais threads remanescentes terminem, antes do programa prosseguir.

1.2.2. Um visualizador simples (porém *bugado*)

O próximo exemplo carrega *n* modelos passados como parâmetros na linha de comando. Eles são atachados a um `osg::Switch`, e as coisas são feitas como se apenas um deles estivessem ativos a cada instante. O exemplo também possui um *event handler* que permite ao usuário selecionar qual deles está ativo. Além disso, cada modelo tem um `osg::PositionAttitudeTransform` sobre ele, de modo que o usuário pode mu-

[5] Ou até que o usuário aperte ESC.

dar suas escalas. O modelo se tornará mais escuro ou mais claro à medida que a escala varia porque as normais não estão normalizadas (este é o *bug* do visualizador). Mas esse é o assunto da seção seguinte, onde este problema será resolvido usando um `StateSet`.

A Figura 1.8 mostra o grafo de cena para o visualizador simples porém *bugado*. Observe o "???": deve haver coisas abaixo do nó retornado por `osgDB::readNodeFile()` (pelo menos há um `osg::Drawable`. Um geode também é mandatório, mas talvez o nó retornado seja este geode. Mas no momento isso não interessa.). Observe também a elipse indicando que todos os modelos passados como parâmetros na linha de comando são adicionados ao grafo de cena.

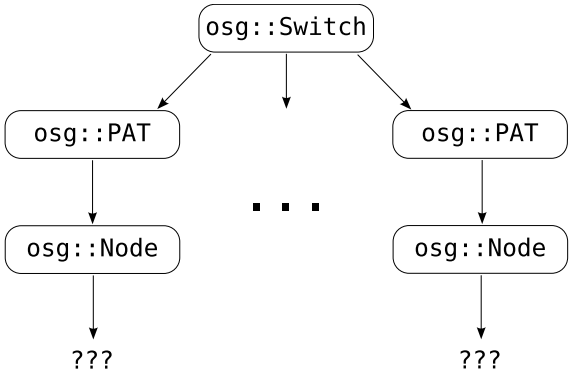

Figura 1.8. O grafo de cena OSG usado no "visualizador simples porém *bugado*".

SimpleAndBuggyViewer.cpp

```
1  #include <iostream>
2  #include <osg/PositionAttitudeTransform>
3  #include <osg/Switch>
4  #include <osgDB/ReadFile>
5  #include <osgGA/GUIEventHandler>
6  #include <osgProducer/Viewer>
7
8  osg::ref_ptr<osg::Switch> TheSwitch;
9  unsigned CurrentModel = 0;
10
11 class ViewerEventHandler: public osgGA::GUIEventHandler
12 {
13    public:
14       virtual bool handle (const osgGA::GUIEventAdapter& ea,
15                            osgGA::GUIActionAdapter&)
16       {
17          if (ea.getEventType() == osgGA::GUIEventAdapter::KEYUP)
18          {
19             switch (ea.getKey())
20             {
21                // Left key: seleciona modelo anterior
22                case osgGA::GUIEventAdapter::KEY_Left:
23                   if (CurrentModel == 0)
24                      CurrentModel = TheSwitch->getNumChildren() - 1;
25                   else
26                      --CurrentModel;
27
28                   TheSwitch->setSingleChildOn (CurrentModel);
29
30                   return true;
31
32                // Right key: seleciona próximo modelo
```

```
33                case osgGA::GUIEventAdapter::KEY_Right:
34                   if (CurrentModel == TheSwitch->getNumChildren() - 1)
35                      CurrentModel = 0;
36                   else
37                      ++CurrentModel;
38
39                   TheSwitch->setSingleChildOn (CurrentModel);
40
41                   return true;
42
43                // Up key: aumenta escala do modelo atual
44                case osgGA::GUIEventAdapter::KEY_Up:
45                {
46                   osg::ref_ptr<osg::PositionAttitudeTransform> pat =
47                      dynamic_cast<osg::PositionAttitudeTransform*>(
48                         TheSwitch->getChild (CurrentModel));
49                   pat->setScale (pat->getScale() * 1.1);
50
51                   return true;
52                }
53
54                // Down key: diminui escala do modelo atual
55                case osgGA::GUIEventAdapter::KEY_Down:
56                {
57                   osg::ref_ptr<osg::PositionAttitudeTransform> pat =
58                      dynamic_cast<osg::PositionAttitudeTransform*>(
59                         TheSwitch->getChild (CurrentModel));
60                   pat->setScale (pat->getScale() / 1.1);
61                   return true;
62                }
63
64                // Não trata outras teclas
65                default:
66                   return false;
67             }
68          }
69          else
70             return false;
71       }
72 };
73
74
75 int main (int argc, char* argv[])
76 {
77    // Checa parâmetros da linha de comando
78    if (argc < 2)
79    {
80       std::cerr << "Uso: " << argv[0]
81                 << " <arquivo com modelo> [ <arquivo com modelo> ... ]\n";
82       exit (1);
83    }
84
85    // Cria visualizador baseado em Producer
86    osgProducer::Viewer viewer;
87    viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);
88
89    // Cria o event handler e o atacha ao visualizador
90    osg::ref_ptr<osgGA::GUIEventHandler> eh (new ViewerEventHandler());
91         viewer.getEventHandlerList().push_front (eh);
92
93    // Constrói o grafo de cena
94    TheSwitch = osg::ref_ptr<osg::Switch> (new osg::Switch());
95
96    for (int i = 1; i < argc; ++i)
97    {
98       osg::ref_ptr<osg::Node> loadedNode = osgDB::readNodeFile(argv[i]);
99       if (!loadedNode)
100         std::cerr << "Problem opening '" << argv[i] << "'\n";
101      else
```

```
102         {
103            osg::ref_ptr<osg::PositionAttitudeTransform> pat(
104               new osg::PositionAttitudeTransform());
105            pat->addChild (loadedNode.get());
106            TheSwitch->addChild (pat.get());
107         }
108      }
109
110      // Garante que temos pelo menos 1 modelo antes de prosseguir
111      if (TheSwitch->getNumChildren() == 0)
112      {
113         std::cerr << "Nenhum modelo 3D foi carregado. Abortando...\n";
114         exit (1);
115      }
116
117      viewer.setSceneData (TheSwitch.get());
118
119      TheSwitch->setSingleChildOn (0);
120
121      // Entra no loop de renderização
122      viewer.realize();
123
124      while (!viewer.done())
125      {
126         viewer.sync();
127         viewer.update();
128         viewer.frame();
129      }
130
131      // Espera todas as threads de descarte e de desenho completarem antes de sair
132      viewer.sync();
133   }
```

1.3. `StateSet`s

Há uma classe muito importante no OSG que ainda não mencionamos: a `osg::StateSet`. Ela é tão importante que esta seção é dedicada a ela. No entanto, para se entender a importância das `osg::StateSet`s, é importante termos algum conhecimento sobre como o OpenGL funciona. Esse pequeno *background* em OpenGL é discutido na próxima subseção. Se você já conhece o assunto "OpenGL como máquina de estados", pode pular para a Seção 1.3.2. Caso contrário, continue lendo.

1.3.1. OpenGL como máquina de estados

De uma maneira grosseira, OpenGL pode ser visto como algo que transforma vértices em pixels. Essencialmente, o programador diz: "Hey, OpenGL, por favor processe esta lista de pontos no espaço 3D para mim". E, logo depois, o OpenGL responde: "Feito! Os resultados estão na sua tela 2D". Essa não é uma descrição completamente acurada do OpenGL, mas está boa para os propósitos desta seção.

Portanto, assumimos que o OpenGL pega vértices e gera pixels. Suponha que os vértices sejam v_1, v_2, v_3 e v_4. Quais pixels eles devem originar? Ou, em outras palavras, como eles devem ser renderizados? Para começar, o que esses vértices representam? Quatro pontos isolados? Um quadrilátero? Dois segmentos de reta (v_1–v_2 e v_3–v_4)? Talvez 3 segmentos (v_1–v_2, v_2–v_3 e v_3–v_4)? Ou algo ainda diferente?

Indo em outra direção, que cores os pixels devem ter? Os objetos renderizados são afetados por alguma fonte de luz? Se forem, quantas fontes há, onde elas estão e quais

são suas características? Elas são mapeadas em textura?

Podemos continuar com estas questões por vários parágrafos, mas vamos parar por aqui. O importante é observar que, embora o OpenGL essencialmente transforme vértices em pixels, há inúmeras maneiras de realizar esta transformação. De alguma forma teremos que ser capazes de "configurar" o OpenGL para ele fazer o que queremos. Mas como configurar tudo isso?

A tática é dividir para conquistar. Há inúmeras configurações, mas elas são ortogonais. Isso significa que podemos, por exemplo, alterar a iluminação sem mexer nas configurações de mapeamento de textura. Claro que há interação entre essas configurações, no sentido de que a cor final do pixel depende tanto da configuração da iluminação quando do mapeamento de textura (e de outros). O importante é saber que eles podem ser configurados independentemente.

De agora em diante, vamos chamar essas configurações OpenGL pelo seu nome correto: atributos e modos (a diferença entre atributo e modo não é importante no momento). O conjunto de atributos e modos define precisamente como o OpenGL se comporta. No entanto, logo se percebeu que a expressão "definir atributos e modos" é longa, e então se criou um nome mais simples: "estado".

Isso explica o título desta seção. O OpenGL pode ser visto como uma máquina de estados. Todos os detalhes que definem exatamente como vértices são transformados em pixels são parte do estado OpenGL. Se estamos desenhando coisas verdes e queremos desenhar coisas azuis, temos que mudar o estado OpenGL. Se estamos desenhando coisas com a iluminação habilitada e queremos desenhar coisas com iluminação desabilitada, temos que mudar o estado OpenGL. O mesmo acontece com mapeamento de textura e tudo o mais.

A questão que surge é "como mudamos o estado OpenGL quando usamos o OSG?". É isso que responderemos na seqüência.

1.3.2. OSG e o estado OpenGL

O OSG provê um mecanismo para manipular o estado OpenGL diretamente no grafo de cena. No percorrimento do grafo para o descarte, as geometrias com mesmo estado são agrupadas para minimizar as mudanças de estado, enquanto no percorrimento para o desenho o estado atual é "capturado" para evitar mudanças de estado redundantes.

A classe **StateSet** armazena um conjunto de valores do estado, identificados como *mode* ou *attribute*. Assim, cada nó do grafo pode ser associado a um **StateSet**. Os *modes* são análogos às chamadas *glEnable* e *glDisable*, enquanto *attributes* permitem a especificação de parâmetros, tais como cor de fog, propriedades de materiais, etc.

O OSG provê um mecanismo para controlar como o estado é herdado no grafo de cena. Por default, todos os filhos herdam os parâmetros de estado dos seus pais, mas pode sobrescrevê-los também. Porém, você pode forçar o estado do pai sobrescrever o estado do filho ou proteger o filho da sobrescrita do pai.

`ON`, `OFF`, `OVERRIDE`, `PROTECTED` e `INHERIT` são parâmetros para `osg::StateSet::setAttribute()`.

1.3.3. Um visualizador 3D simples (e sem *bugs*)

A idéia aqui é corrigir o *bug* do exemplo anterior chamando `ss->setMode(GL_NORMALIZE,osg::StateAttribute::ON)`, conforme mostrado no trecho a seguir, que representa a alteração no laço para a construção do grafo de cena para os diferentes modelos – linhas 93 a 108 do exemplo anterior.

SimpleAndBuglessViewerReduced.cpp

```
1    // Constrói o grafo de cena
2    TheSwitch = osg::ref_ptr<osg::Switch> (new osg::Switch());
3
4    for (int i = 1; i < argc; ++i)
5    {
6       osg::ref_ptr<osg::Node> loadedNode = osgDB::readNodeFile(argv[i]);
7       if (!loadedNode)
8          std::cerr << "Problema ao abrir '" << argv[i] << "'\n";
9       else
10      {
11         osg::ref_ptr<osg::StateSet> ss (loadedNode->getOrCreateStateSet());
12         ss->setMode (GL_NORMALIZE, osg::StateAttribute::ON);
13         osg::ref_ptr<osg::PositionAttitudeTransform> pat(
14            new osg::PositionAttitudeTransform());
15         pat->addChild (loadedNode.get());
16         TheSwitch->addChild (pat.get());
17      }
18   }
19
20
```

A única diferença com relação ao exemplo anterior são as linhas 11 e 12. Porém, esta pequena diferença gera melhores resultados, principalmente quando modelos 3D, modelados em escalas distintas, são importados para a aplicação.

1.4. Arquivos: Carregando e Salvando

Para desenvolver aplicações "sérias", como jogos ou cenas complexas de realidade virtual, precisamos corrigir as limitações de APIs de baixo nível relacionadas à criação de modelos usando algumas poucas primitivas. Precisamos ser capazes de ler coisas complexas dos arquivos com os modelos, e a biblioteca osgDB vai nos ajudar. Ela provê estas funcionalidades, permitindo que nosso programa leia e escreva modelos 3D. Uma grande variedade de formatos é suportada, incluindo os nativos OSG (.osg - ASCII) e (.ive - binário), OpenFlight (.flt), TerraPage (.txp) com suporte multi-threading, LightWave (.lwo), Alias Wavefront (.obj), Carbon Graphics GEO (.geo), 3D Studio MAX (.3ds), Peformer (.pfb), Quake Character Models (.md2). Direct X (.x), Inventor Ascii 2.0 (.iv)/ VRML 1.0 (.wrl), Designer Workshop (.dw) e AC3D (.ac). Além disso, o OSG inclui *image loaders* para ler e escrever imagens 2D. Dentre os formatos suportados estão: .rgb, .gif, .jpg, .png, .tiff, .pic, .bmp, .dds (inclui compressed mip mapped imagery), .tga e quicktime (no MacOSX).

O OSG provê um mecanismo de plugin dinâmico e extensível para ler e escrever modelos 3D e 2D de/para arquivos. Você só precisa adicionar os seguintes cabeçalhos às aplicações:

```
#include <osgDB/ReadFile>
```

```
#include <osgDB/WriteFile>
```

Vamos ler uma simpática vaquinha:

```
osg::ref_ptr<osg::Node> lmodel = osgDB::readNodeFile(``cow.osg'');
```

Agora, vamos salvar uma nova vaca:

```
bool osgDB::writeNodeFile(lmodel,``cow.osg'');
```

Se esta operação falhar, a função retorna *false*. Caso contrário, ela retorna *true*. Bingo! O local onde o arquivo será salvo pode ser absoluto ou relativo, e os arquivos existentes serão sobrescritos. Para evitar problemas, lembre-se de checar os arquivos existentes.

1.5. Um pouco de realidade aumentada com ARToolKit

Realidade Aumentada (RA) é uma tecnologia que acrescenta elementos virtuais à percepção do mundo real pelo usuário, através da combinação (ou sobreposição) da cena do mundo real com elementos virtuais [Azuma 1997]. O principal objetivo é suplementar o mundo real com objetos virtuais coexistindo no mesmo espaço (visual). Por exemplo, objetos renderizados computacionalmente poderiam ser sobrepostos a imagens capturadas por uma câmera digital convencional ou webcam, de forma a enriquecer visualmente a cena. A Figura 1.9 ilustra uma sobreposição do modelo 3D (logotipo da OpenSceneGraph) em uma folha de papel.

Para que o modelo seja colocado corretamente sobre a folha, o computador precisa rastrear (*tracking*) o campo de visão (*viewpoit*) da câmera em relação à folha. Uma forma de fazer isso é através da colocação de padrões (marcadores) na cena, no caso, sobre a folha. Assim, quando os marcadores são reconhecidos nas imagens, pode-se determinar a posição e o ângulo da câmera em relação aos marcadores de forma barata e relativamente rápida. Como conseqüência, pode-se alinhar a câmera virtual e a câmera real, tornando possível a sobreposição dos objetos renderizados aos marcadores reais.

A ARToolKit é uma API C/C++ que facilita a construção dessas aplicações em realidade aumentada. Ela resolve o problema do rastreamento do campo de visão da câmera (usuário), determinando a posição/orientação relativas entre câmera real e os marcadores, através do emprego de algoritmos de visão computacional. Isso possibilita não somente adicionar objetos virtuais à cena, como também acompanhar mudanças de posição dos marcadores, dando a impressão que os objetos estão presos a eles.

A ARToolKit fornece um *framework* completo para a prototipação de aplicações RA em tempo-real, destacando-se além do *tracking* e sobreposição dos modelos 3D aos marcadores reais; uma biblioteca multi-plataforma de video com suporte a múltiplas interfaces de entrada de vídeo, formatos e câmeras; e suporte a renderização eficiente de objetos 3D e biblioteca gráfica (GUI) baseadas em OpenGL e GLUT, respectivamente. A biblioteca gráfica herda um conjunto de funcionalidade da GLUT para facilitar a criação de janelas, manipulação de eventos externos (usuário), interface com sistemas de janela

Figura 1.9. Um modelo 3D da terra sobreposto.

(X11, Win32) e API de vídeos. Além disso permite a exibição de cenas renderizadas com a OpenGL, já que esta é totalmente independente do sistema de janela.

O suporte a OpenGL e também a VRML permite a renderização de objetos virtuais nas cenas reais. No entanto, o desenvolvimento de aplicações produtivas fica comprometido pela restrição da OpenGL em prover suporte direto a importação de modelos 3D em diferentes formatos, ausência de gerenciamento através de grafo de cena, programação orientada a objetos. Por outro lado, VRML acrescenta algumas funcionalidades relativas a importação de modelo e animação, porém restritivas e com grande dependência do formato .wrl. Além disso, a baixa qualidade visual, a ausência de otimização na representação de superfícies, a carência em acompanhar os avanços promovidos pelo hardware gráfico moderno, e a grande falha em promover uma sinergia com a Web (objetivo da sua concepção), provocam uma enorme restrição da comunidade em empregar VRML em visualização, jogos e simuladores em geral.

Na próxima seção apresentamos uma solução para esses problemas, através da combinação da OpenSceneGraph com a ARTollKit. A OSG é uma API C++, desenvolvida sobre a OpenGL, supre as lacunas da OpenGL no tocante a funcionalidade, através de um grafo de cena otimizado e a VRML no que concerne a eficiência e importação de uma vasta gama de formatos 3D. Além disso, oferece um enorme ferramental para a produção de aplicações com renderização de alta qualidade visual, plug-ins para bibliotecas de animação baseada em esqueleto (Cal3D e ReplicantBody), física (ODE) e áudio (OpenAL), além de estatísticas da renderização e uma arquitetura orientada a objetos altamente extensível e multi-plataforma.

1.6. Combinando ARToolKit e OSG

De certo, o suporte da ARToolKit à OpenGL e a mediocridade da VRML incentivaram o desenvolvimento de um componente para combinar a ARToolKit com a OpenSceneGraph. Tanto que foram desenvolvidos alguns, sendo dois com maior sucesso, OSGAR [Coelho et al. 2004] (`http://www.cc.gatech.edu/ael/resources/osgar.html`) e a OSGART [Looser et al. 2007] (`http://www.artoolworks.com/community/osgart/index.html`). A Figura 1.10 ilustra a combinação da ARToolKit com a OSG, através da OSGART. Este *framework* é a escolha natural para essa apresentação, por oferecer mais funcionalidade e acompanhar as diretrizes de projeto da ARToolkit.

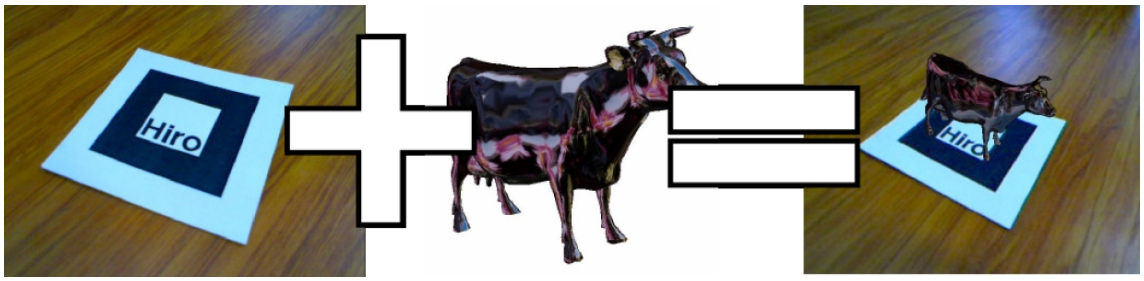

Figura 1.10. ARToolkit + OSG = OSRGAT

A OSGART foi desenvolvida como uma extensão da OSG. Ela implementa uma estrutura hierárquica, baseada em grafo de cena da OSG, para ARToolKit introduzir parâmetros do vídeo e o outro vídeo na cena. Como conseqüência, as demais funcionalidades da OSG podem fornecer elementos gráficos de alta qualidade visual para serem sobrepostos aos marcadores identificados pela ARToolKit. Esta estreita relação com a OSG, credencia a OSGART como uma valorosa ferramenta para desenvolvimento rápido de aplicações virtuais imersivas e de realidade aumentada [Milgram and Kishino 1994]. A OSGART generaliza os componentes de hardware através de camadas, tanto para *mouse* e teclado, quanto para dispositivos de RV com suporte a *tracking* e interfaces tangíveis e hápticas.

Uma grande necessidade em aplicações com fortes requisitos funcionais em RA é a importação conteúdos existentes sem a necessidade de conversão. A OSGART, através da OSG, fornece um suporte direto a várias mídias, por exemplo imagens, vídeo, modelos 3D (dezenas de formatos) e animação. Além disso, sob o ponto de vista de manipulação e interação, a OSGART oferece metáforas baseadas em WIMP e pós-WIMP (fala, gestos) além de tradicionais técnicas de interação em RV. Por exemplo, interação baseada em raio e distintos modos de navegação podem ser agregados. Essa gama de funcionalidades supre a pobreza da GLUT e são oferecidas em conjunto pela OSG e OpenProducer (`http://www.andesengineering.com/Producer/`). Adicionalmente, componentes para colaboração podem ser agregados às aplicações, através do *middleware* ICE de forma transparente à rede. Dentre as possibilidades mais evidentes, seriam tele-apontadores, tele-*draggers* e tele-manipuladores, que dariam suporte ao desenvolvimento de aplicações com múltiplos espaços de interação.

A OSGART foi desenvolvida em C++, assim como OpenSceneGraph, portável

entre diferentes plataformas. Além disso, os módulos *osgBindings* e *osgIntrospection* oferecem a possibilidade de se criar conteúdos em RA, empregando diversas linguagens *scripts*, como Python, Ruby e LUA. LUA pode ainda ser integrada de modo a mudar o comportamento da computação em tempo de execução, sem comprometer o desempenho da computação, ideal para desenovolvimento de ferramentas de testes.

Um dos principais problemas em RA é a aquisição de vídeo para registro visual, mas um módulo dentro da OSGART cobre fontes de vídeo como arquivos, *streams* e vídeo gerado. Essas funcionalidades são herdadas diretamente da ARToolKit. A Figura 1.11 mostra a estrutura geral da OSGART.

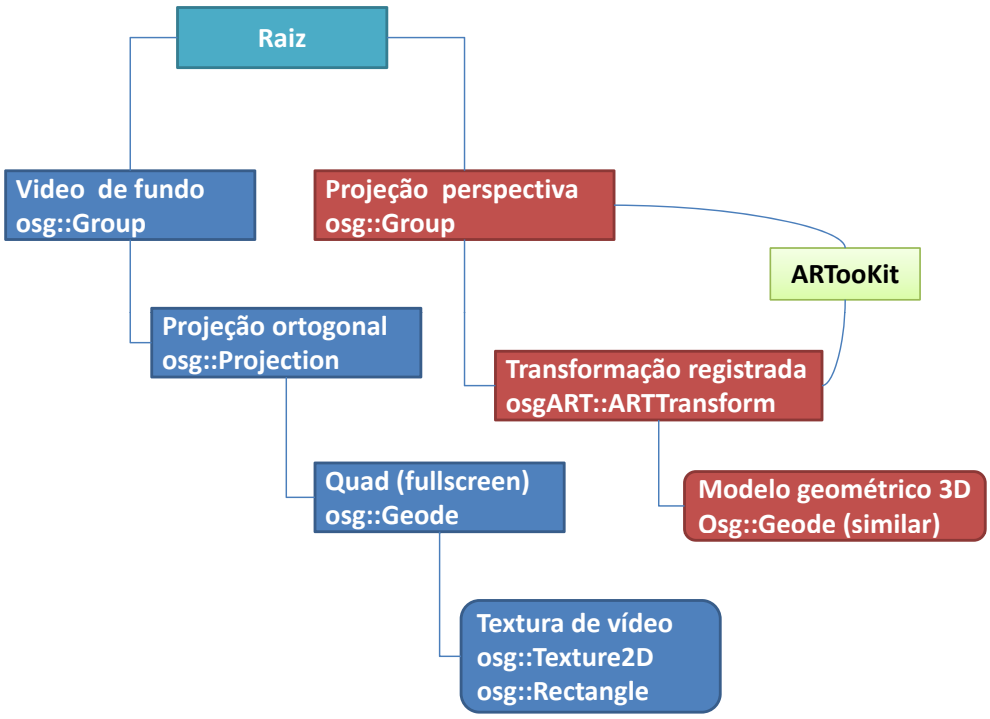

Figura 1.11. Estrutura geral da OSGART

A OSGART fornece um conjunto de classes que facilitam o desenvolvimento de aplicações RA, sendo três os componentes principais:

- O *vídeo de fundo* necessita ser desenhado embaixo dos demais objetos geométricos, para dar a ilusão de que todos os objetos 3D estão de fato incorporados ao mundo real.
- A *matriz de projeção* utilizada pela ARToolKit, necessita ser aplicada dentro da OSG. Essa matriz é determinada a partir de parâmetros intrínsecos da câmera real e os seus elementos são coletados no processo de calibração e passados para a câmera virtual.

- As *transformações dos marcadores*, também computadas pela ARToolKit, são mapeadas para transformações OSG. Assim, a OSG utiliza os nodos de transformações para posicionar os objetos geométricos 3D dentro da cena real, Figura 1.12.

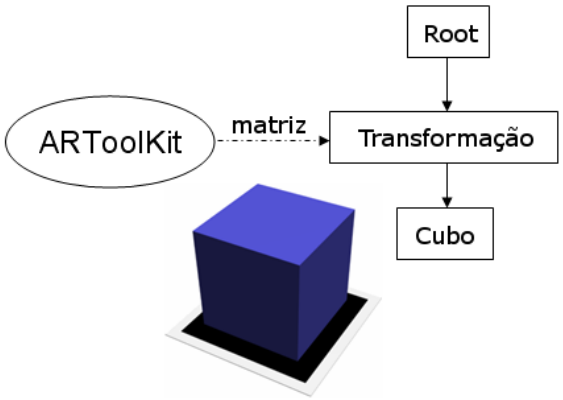

Figura 1.12. Matriz de transformação da OSGART, destacando o mapeamento de transformações da ARToolKit na OSG

Para melhorar a qualidade do trabalho de integração é desejável que os objetos reais possam ocluir os objetos virtuais, conferindo maiior realismo à cena. Para se obter esse efeito, modelos rudimentares do ambiente real têm que ser construídos, para modificar a informação de profundidade do sistema de renderização e assim gerar o efeito de oclusão.

1.7. OpenSceneGraph e ARToolKit: juntas em ação

Para realizar as medidas com mais precisão é importante que a câmera esteja calibrada. A ARToolKit fornece valores *default* de calibração, no entanto, para obter melhores resultados com a sua câmera, recomendamos proceder a calibração da mesma. A página da ARToolKit oferece dois métodos para calibração, veja em `http://www.hitl.washington.edu/artoolkit/documentation/usercalibration.htm`.

1.7.1. Exibindo um vídeo simples (passo-a-passo)

Este exemplo simples demonstra o display de um video de fundo com OSGART. A classe *VideoBackground* exibe um vídeo simples de entrada (webcam). A escolha do vídeo é feita através de um plug-in de vídeo.

A primeira providência é iniciar a OSGART (iniciar a ARToolKit) e criar um *viewer* usando o *Producer*. Esse processo é feito pelo seguintes comandos:

```
osgARTInit(&argc, argv);
osgProducer::Viewer viewer;
viewer.setUpViewer(osgProducer::Viewer::ESCAPE_SETS_DONE);
```

Observe que a última linha do código anterior tão somente possibilita sair da aplicação usando a tecla ESC. Agora, precisa-se carregar o vídeo, através de um *plug-in*, e fazer o mesmo para o *tracker*:

```
osg::ref_ptr<osgART::GenericVideo> video =
osgART::VideoManager::createVideoFromPlugin("osgart_artoolkitd");

if (!video.valid()){
 osg::notify(osg::FATAL) << "Nao pode inicializar vídeo!" <<
 std::endl;
 exit(1);
}
osg::ref_ptr<osgART::GenericTracker> tracker=
 osgART::TrackerManager::createTrackerFromPlugin
                  ("osgart_artoolkit_trackerd");
```

A validação do *tracker* foi omitida acima, mas também é interessante fazê-la. Com os *plug-ins* carregados, são obtidas informações sobre o vídeo, essenciais para conectar o *tracker*:

```
video->open();
tracker->init(video->getWidth(), video->getHeight());
```

Lembrando que o comando $video->open()$ não inicia a stream de vídeo, tão somente obtém as informações de formato do vídeo, afim de inicializar o *tracker* com as dimensões corretas.

Até agora, realizamos o *setup* do sistema para captura do vídeo e seus parâmetros essenciais para uma aplicação RA. Agora vamos construir a cena propriamente dita e agregar a ARToolkit à OSG, começando com os nodos que serão vídeo de fundo (OSGART) e raiz do vídeo (OSG):

```
osgART::VideoBackground* videoBackground=
  new osgART::VideoBackground(video.get());

videoBackground->setTextureMode
     (osgART::GenericVideoObject::USE_TEXTURE_RECTANGLE);

videoBackground->init();

osg::Group* foregroundGroup        = new osg::Group();
foregroundGroup->addChild(videoBackground);
```

A OSG possui uma forma otimizada de renderização, agrupando objetos com os mesmos estados para serem renderizados juntos e assim, mudanças freqüentes no estado da OpenGL são evitadas. Além disso, permite controlar a seqüência de renderização, usando os chamados *RenderBins*. Os "bins" com números mais baixos são renderizados antes daqueles com um número maior. Colocar objetos no Renderbin=1 assegura que são renderizados primeiro, portanto, no fundo (algoritmo do pintor). Esse procedimento é necessário quando trabalhamos com objetos especiais, como por exemplo, objetos com certo grau de transparência e nesse exemplo também, pois precisamos deixar o vídeo real no fundo. No caso, ajustamos os parâmetros para o vídeo ser desenhado primeiro:

```
foregroundGroup->getOrCreateStateSet()
                      ->setRenderBinDetails(1, "RenderBin");
```

Agora, vamos casar os parâmetros da câmera virtual com os parâmetros da câmera real, obtida para ARToolKit:

```
osg::Projection* projectionMatrix =
    new osg::Projection
      (osg::Matrix(tracker->getProjectionMatrix()));
```

Assim, os objetos reais e virtuais utilizam os mesmos parâmetros de visualização. Para isso o nodo dos parâmetros de visualização (projeção) deve conter o nodo com a cena completa. Esse nodo, por sua vez, deve conter o vídeo e objetos 3D (que nesse exemplo, não estão presentes):

```
osg::MatrixTransform* modelViewMatrix =
     new osg::MatrixTransform();
modelViewMatrix->addChild(foregroundGroup);
projectionMatrix->addChild(modelViewMatrix);
```

Finalmente, o nodo com ambas as cenas transformadas coerentemente são agrupados e anexados ao *viewer*, com isso a janela OpenGL é criada e o vídeo iniciado:

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild(projectionMatrix);
viewer.setSceneData(root.get());
viewer.realize();
video->start();
```

Por fim, o programa entra no laço do Producer, até que seja apertada a tecla ESC ou a janela seja fechada. O laço do Producer pouco difere do usual, apenas pela atualização do vídeo antes da cena completa, com o intuito de instruir a ARToolKit a capturar o *frame* e deixá-lo disponível para ser adicionado à cena. Na Figura 1.13 é mostrado um *snapshot* da cena.

1.7.2. Adicionando um modelo geométrico 3D

O procedimento é uma continuação da seção 1.7.1, porém ao invés de adicionar um nodo de transformação da OSG inalterado, adiciona-se um nodo com as informações geométricas (transformações) dos marcadores:

```
osg::ref_ptr<osg::MatrixTransform> markerTrans =
     new osgART::ARTTransform(marker.get());
```

Assim, pode-se adicionar o modelo 3D a este nodo, inclusive podendo transformá-lo localmente (no caso, modificando-se a posição e escala, usando os recursos da OSG – ver códigos C++). A adição de iluminação pode enriquecer os exemplos e evitar a exibição de modelos sem sombreamento. Todo esse trabalho é feito de forma fácil e rápida usando a OSG. Na Figura 1.14, há um *snapshot* deste exemplo, utilizando um modelo no formato AutoDesk .3DS e contendo modelos 3D sobrepostos ao vídeo.

Figura 1.13. Vídeo capturado usando ARToolKit e exibido com o auxílio da OSG e Producer.

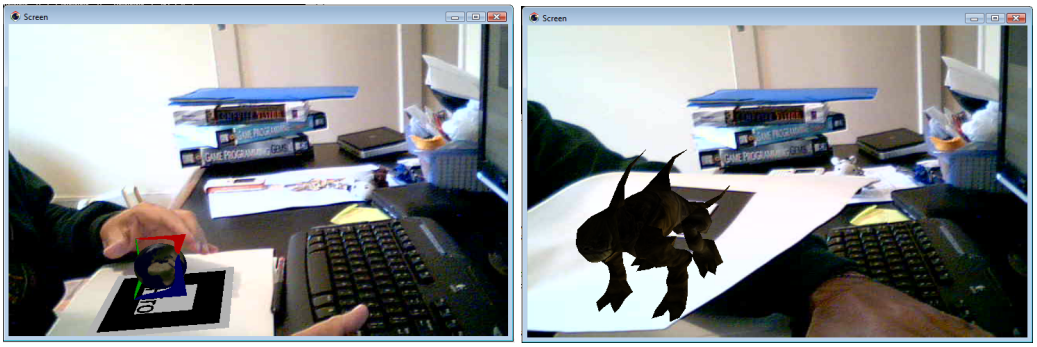

Figura 1.14. Modelos 3D (formatos .osg e .3DS) sobre vídeo usando OSGART.

1.8. Conclusão

Nesse capítulo, apresentamos os conceitos fundamentais da OpenSceneGraph, ilustrados com exemplos. Essa fundamentação serviu de base para mostrar como desenvolver aplicações em Realidade Aumentada, utilizando OSG e ARToolkit. Por isso, apresentamos alguns conceitos fundamentais de RA, da própria ARToolKit, e do *framework* criado para promover a interligação entre essas tecnologias, a OSGART. Por fim, ilustramos a sinergia das tecnologias com exemplos práticos.

Desta forma, buscamos incentivar a adoção destas tecnologias, já largamente empregadas no cenário internacional, por nossos pesquisadores e alunos no Brasil. Com esse intuito, manteremos os códigos fontes completos mostrados trabalho na página `http://www.v3d.com.br/osgart`, incluindo futuras atualizações. Também está na página os slides do mini-curso dado no SVR'2008, em João Pessoa-PB. Consulte regularmente o site ou envie email para os autores diretamente: `gonzaga@acm.org` ou `abraposo@tecgraf.puc-rio.br`, em caso de dúvidas, sugestões ou simplesmente para trocar idéias.

Referências

[Azuma 1997] Azuma, R. (1997). A survey of augmented reality. *Presence Teleoperators and Virtual Environments*, 6(4):355–385.

[Coelho et al. 2004] Coelho, E. M., MacIntyre, B., and Julier, S. (2004). Osgar: A scene graph with uncertain transformations. In *ISMAR*, pages 6–15, Los Alamitos, CA, USA. IEEE Computer Society.

[Kuehne and Martz 2007] Kuehne, B. and Martz, P. (2007). *OpenSceneGraph Reference Manual v1.2*. Blue Newt Software LLC and Skew Matrix Software LLC.

[Looser et al. 2007] Looser, J., Billinghurst, M., Grasset, R., and Cockburn, A. (2007). An evaluation of virtual lenses for object selection in augmented reality. In *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 203–210, New York, NY, USA. ACM.

[Martz 2007] Martz, P. (2007). *OpenSceneGraph Quick Start Guide*. PMARTZ Computer Graphics Systems.

[Milgram and Kishino 1994] Milgram, P. and Kishino, F. (1994). A taxonomy of mixed reality visual displays. *IEICE Transactions on Information Systems*, E77-D(12).