# O Engine Gráfico OGRE

[1]Thiago Souto Maior Cordeiro de Farias, [1]Saulo Andrade Pessoa,
[2]Veronica Teichrieb, [1]Judith Kelner

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)

Av. Prof. Moraes Rego S/N, Prédio das Incubadoras, 1º Andar, Cidade Universitária, 50670-901, Recife, Pernambuco

[2]Departamento de Sistemas Computacionais – Escola Politécnica de Pernambuco – Universidade de Pernambuco

Rua Benfica n° 455, Bairro Madalena, 50720-001, Recife, Pernambuco

{mouse, sap, jk}@cin.ufpe.br, vt@dsc.upe.br

***Abstract***

*OGRE is a scene oriented, flexible 3D rendering engine. It is written in C++ and designed to turn easier and more intuitive applications development using hardware accelerated 3D graphics. This text describes this technology, introduces OGRE's main concepts and gives a view of its powerful capabilities as an engine for the development of graphics applications and games.*

***Resumo***

*O OGRE é um engine gráfico 3D orientado à cena e flexível. Foi desenvolvido em C++ e designado a tornar a implementação de aplicações mais fácil e intuitiva para os desenvolvedores, utilizando gráficos 3D acelerados por hardware. Este texto apresenta essa tecnologia, introduzindo os principais conceitos do OGRE e dando uma visão geral das suas potencialidades como engine de desenvolvimento de aplicações gráficas e de jogos.*

# 1. Introdução ao Engine Gráfico OGRE

## 1.1. O que é o OGRE?

OGRE (*Object-oriented Graphics Rendering Engine*) é um *engine* gráfico *open-source* que funciona na maioria das plataformas existentes. Ele provê uma vasta gama de *plugins*, ferramentas e *add-ons*, que favorecem a criação de vários tipos de aplicações gráficas, empregando diversos conceitos inerentes ao desenvolvimento destas.

O *engine* funciona em inúmeras configurações de *hardware* 3D (GPUs) disponíveis no mercado, desde as mais obsoletas às mais sofisticadas. Através da interface provida pelo OGRE, o desenvolvedor pode verificar se o *hardware* oferece suporte aos requisitos mínimos para execução da aplicação, e conseqüentemente pode programar técnicas alternativas para requisitos mais sofisticados. Este conceito de técnicas alternativas pode ser aplicado também aos materiais e efeitos, incluindo versões de *shaders* e texturas multicamada.

A interface de programação oferecida nativamente pelo OGRE é escrita em C++, o que conseqüentemente requer do desenvolvedor um bom conhecimento da linguagem e de conceitos de orientação a objetos (abstração, encapsulamento, polimorfismo). Atualmente, existem alguns *wrappers* para o OGRE em Java, .NET e Python, mas eles ainda estão em fase de desenvolvimento.

O propósito do OGRE não é ser um *game engine*; ele é um *rendering engine* genérico que pode ser incorporado a bibliotecas de tratamento de entradas, de processamento de som e as plataformas que disponibilizem algoritmos de inteligência artificial, compondo assim um *kit* de desenvolvimento mais completo que dê suporte ao desenvolvimento de jogos 3D.

## 1.2. Licença

O OGRE é licenciado sob os termos da GNU *Lesser Public License* (LGPL), o que basicamente quer dizer que o código fonte pode ser adquirido e utilizado em aplicações (comerciais ou não) gratuitamente. A abertura do código fonte só é obrigatória mediante alterações feitas no próprio *engine*, sendo também necessário divulgar as alterações para a comunidade. Por conveniência, é recomendado que se mostre o ícone do OGRE por um período mínimo de 2 segundos ao longo da execução do programa.

Mais informações relativas à licença podem ser encontradas em:

**http://www.gnu.org/copyleft/lgpl.html**

## 1.3. Instalação

A instalação do OGRE pode ser realizada de duas formas, sendo elas através da versão pré-compilada, ou a partir do código fonte. A instalação da versão pré-compilada é a mais indicada para iniciantes.

### 1.3.1. SDK pré-compilado

A instalação do SDK (*Software Development Kit*) pré-compilado é feita de maneira bastante intuitiva e rápida (através de um *wizard*), de forma que, seguindo apenas alguns

passos, a operação é finalizada sem necessidade de muita intervenção do usuário. Os primeiros passos do *wizard* podem ser vistos na Figura 1.

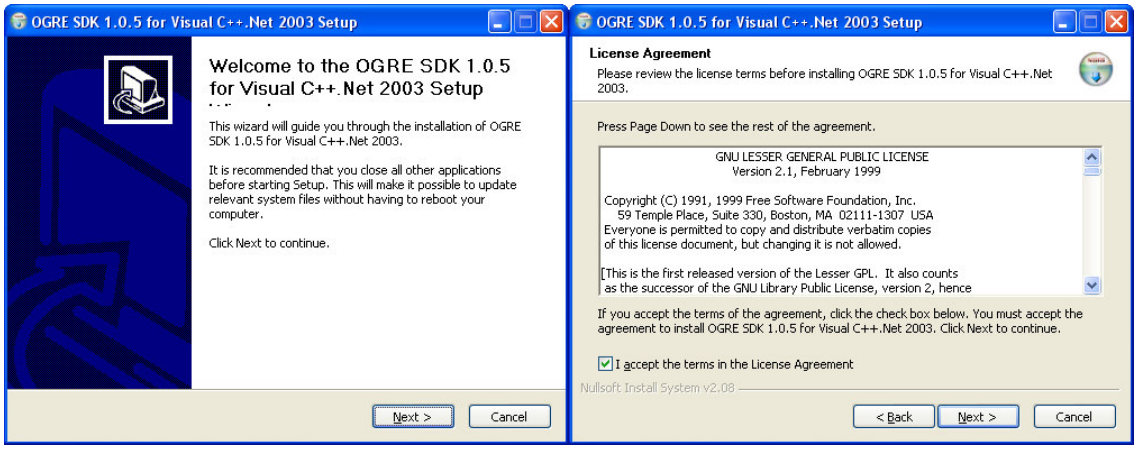



**Figura 1. Primeiros passos da instalação do SDK pré-compilado.**

Ao fim do processo, um diretório é criado contendo as bibliotecas e os arquivos de cabeçalho necessários para desenvolver as aplicações. Juntamente com a biblioteca são encontradas algumas aplicações-exemplo e a documentação do OGRE, que é composta pelo manual do usuário e pela especificação da API (*Application Programming Interface*).

Após a instalação, o *wizard* se encarrega de criar uma variável de ambiente chamada **OGRE_HOME**, que contém o caminho do diretório onde o SDK foi instalado. Este será posteriormente utilizado para configuração do ambiente de desenvolvimento utilizado pelo usuário, através de *Makefile* (arquivo de construção utilizado em plataformas UNIX-*Like*) ou de alguma IDE (*Integrated Development Enviroment*).

### 1.3.2. Código Fonte

A instalação através do código fonte do OGRE é um pouco mais complexa, pois envolve uma seqüência maior de passos e, conseqüentemente, um tempo maior até que o *engine* esteja completamente pronto para ser utilizado. Ela é extremamente recomendada para quem pretende fazer alterações no código fonte da biblioteca, a fim de modificar algum comportamento. Também é bastante útil ter o código fonte do OGRE em mãos para fazer um *debug* mais apurado, uma vez que é permitido verificar o que acontece dentro da biblioteca.

O primeiro passo a ser realizado é o *download* do código fonte, a partir da página oficial do OGRE (**http://www.ogre3d.org**[1]). Na página pode-se encontrar uma seção de *downloads*, na qual existem *links* que apontam para o código fonte suportado por diversas plataformas. A plataforma que servirá de exemplo neste tutorial é a Windows, utilizando como ambiente de desenvolvimento o Visual C++ .NET 2003 (versão 7.1). É necessário ainda o *download* de algumas bibliotecas que são pré-requisitos para a compilação do OGRE e das suas ferramentas. Encontraremos estas

[1] Este site é a única referência oficial sobre a biblioteca gráfica OGRE.

dependências juntamente com o código fonte do OGRE (Dependencies v1.0.6 For Visual C++ .Net 2003 (7.1)). O OGRE também tem como pré-requisito a instalação do DirectX 9.0 SDK, que está disponível para *download* no *site* da Microsoft (**http://msdn.microsoft.com/directx/sdk/**). A última atualização data de Fevereiro de 2006, e ainda está disponível para *download* livre (sem a autenticação através da chave do Windows).

Após o *download* do código fonte e de suas dependências, deve-se extrair os arquivos compactados e transferir as pastas contidas no arquivo de dependências para a pasta criada pelo arquivo dos códigos fonte, chamada "**ogrenew**". Nesta mesma pasta encontraremos arquivos que pertencem aos projetos e ao serem compilados darão origem à biblioteca e aos exemplos. A árvore de diretórios deve incluir todos os diretórios mostrados na Figura 2.

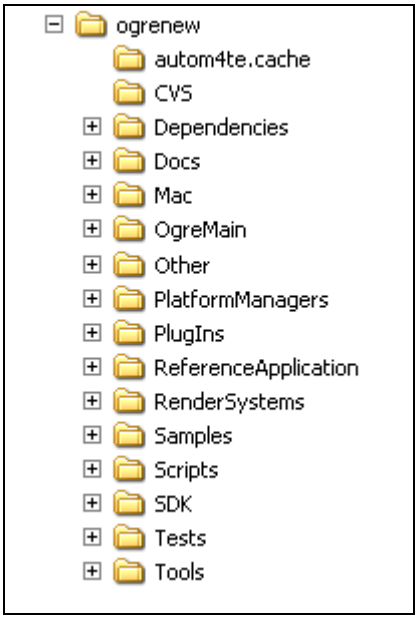



**Figura 2. Árvore de diretórios do código fonte.**

O próximo passo é abrir o arquivo "**Ogre.sln**", que contém todos os projetos a serem compilados bem como as suas dependências. Tendo aberto o arquivo a partir do Visual Studio .NET 2003, pode-se então construir a biblioteca a partir do menu "**Build**", e em seguida, "**Build Solution**", como ilustra a Figura 3. O processo pode demorar um pouco, dependendo do poder de processamento e da memória do computador utilizado.

Todos os binários gerados pela solução serão copiados no diretório "**PATH\ogrenew\Samples\Common\bin**", separados apenas pelos diretórios "**Release**" e "**Debug**". Os arquivos de biblioteca "**.lib**" são gerados dentro das pastas de seus respectivos projetos; por exemplo: "**PATH\ogrenew\OgreMain\lib**".

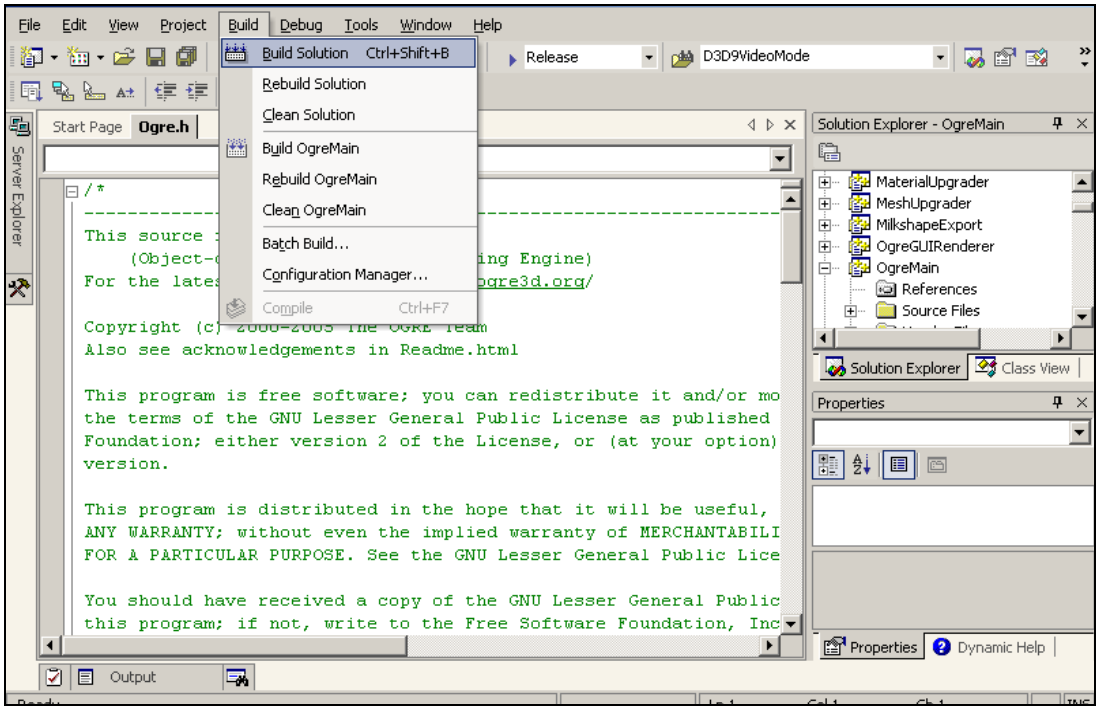



**Figura 3. Processo de compilação através do código fonte.**

## 2. A Arquitetura do OGRE

A arquitetura adotada pelo OGRE utiliza vários padrões de projeto (*Factories*, *Singletons*, *Observers*, etc.), que a tornam mais simples e fácil de usar. Vários conceitos e abstrações são introduzidos pela arquitetura, sendo os principais: *Root*, *SceneManager*, *RenderSystem*, *Entity*, *Mesh*, *SceneNode* e *Material*. O relacionamento entre estes componentes pode ser observado na Figura 4.

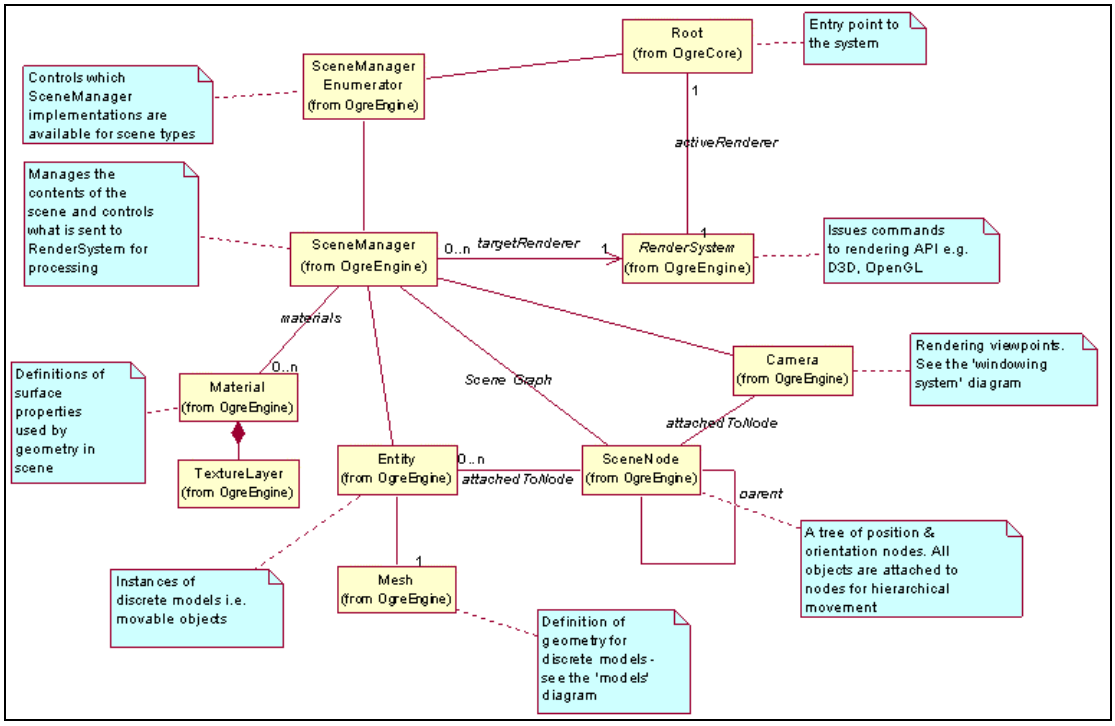



**Figura 4. Relacionamento entre os principais componentes do OGRE.**

## 2.1. O Objeto Root

O objeto *Root* é o ponto de entrada do OGRE. Ele deve ser instanciado antes de todos os outros objetos, pois é responsável por dar origem a todo o sistema e administrar o acesso ao mesmo. Desta forma, ele também deve ser o último a ser destruído, pois sua destruição significa a finalização do uso da biblioteca.

A configuração do sistema pode ser realizada através do objeto *Root*, chamando o método **Root::showConfigDialog()**, que detecta todas as opções disponibilizadas pelos sistemas de renderização (OpenGL e DirectX) e exibe uma caixa de diálogo para que o usuário configure a resolução da tela, se a exibição será em tela cheia, entre outras opções, como ilustra a Figura 5.

A partir do *Root* é possível obter a referência para outros objetos importantes do sistema, tais como o *SceneManager*, o *RenderSystem* e outros gerenciadores de recursos. Fazem parte também das tarefas desempenhadas pelo *Root*: habilitar o modo de renderização contínua (modo utilizado em jogos) e o uso de *FrameListeners*, que serão descritos na seção 7.

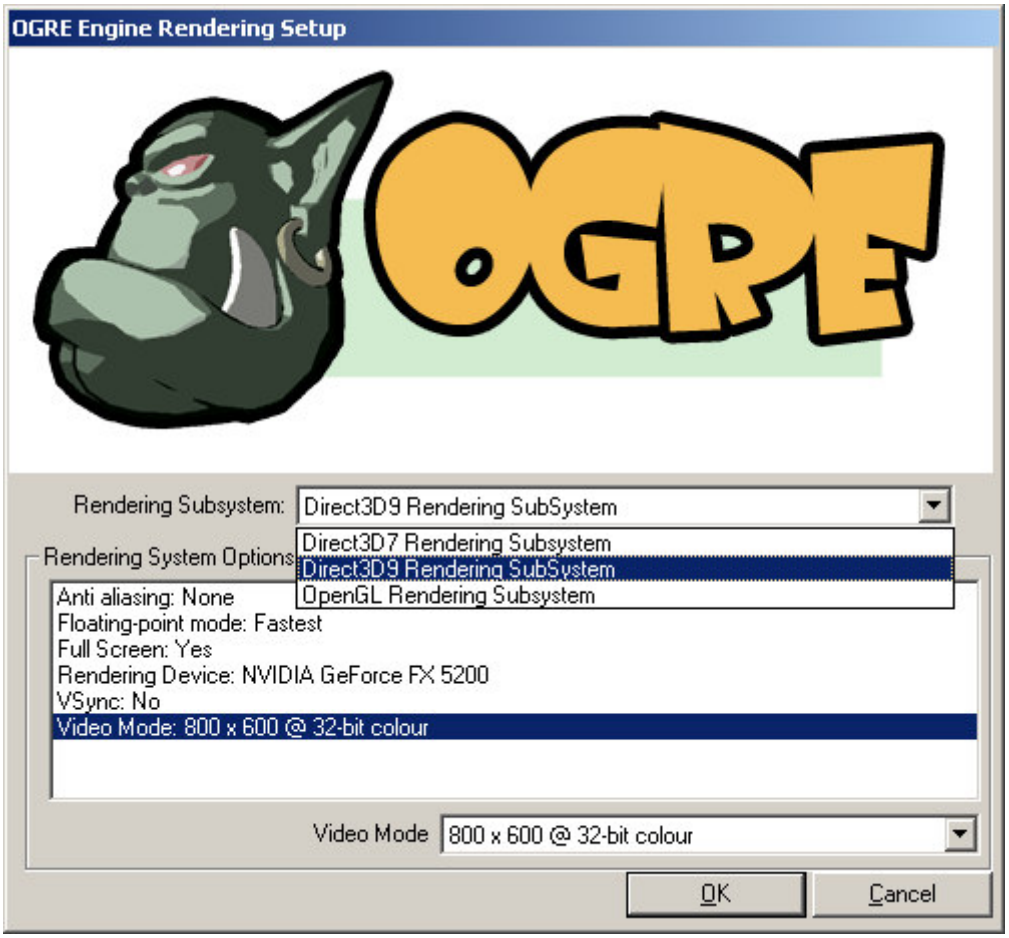



**Figura 5. Janela de configuração inicial do OGRE.**

## 2.2. O Objeto SceneManager

O *SceneManager* é o objeto que reúne e organiza todo o conteúdo da cena que será renderizado pelo *engine*. Ele é responsável por aplicar algoritmos para otimizar a exibição da cena, por criar e gerenciar todas as câmeras, os objetos móveis (*Entities*), as luzes, a malha que representa o mundo e as partes estáticas da cena.

A partir do *SceneManager* é possível criar e acessar nós e objetos móveis, a partir de nomes atribuídos a eles através dos seguintes métodos:

- **SceneManager::createSceneNode/getSceneNode();**
- **SceneManager::createCamera/getCamera();**
- **SceneManager::createLight/getLight().**

Os métodos listados acima fornecem acesso respectivamente a nós de controle da cena, câmeras e luzes.

O *SceneManager* também é responsável por enviar toda a cena para o *RenderSystem* quando necessário. Isso é feito automaticamente através do método **SceneManager::_renderScene()**, que é chamado pelo *engine* a cada quadro.

A maior parte da interação com o *SceneManager* se dá ao longo da criação da cena. Durante este processo uma grande quantidade de objetos é criada, havendo a possibilidade de modificação dinâmica durante o ciclo de renderização, através de um *FrameListener*.

Diferentes tipos de cenas requerem diferentes algoritmos para decidir que objeto pode ser enviado para o *RenderSystem* de forma a manter um bom desempenho na renderização. Por causa disso, o *SceneManager* foi projetado para ser especializado (através de subclasses) em um determinado tipo de cena. O *SceneManager* padrão tem um bom desempenho em cenas relativamente pequenas, pois faz pouca ou nenhuma otimização nelas. A intenção é que seja criado um *SceneManager* para cada tipo de cena, que faça as devidas otimizações na sua organização e seleção de objetos a serem renderizados, assumindo algumas características existentes de acordo com cada tipo. Um exemplo de especialização do *SceneManager* é o *BspSceneManager*, que otimiza cenas contendo grandes ambientes fechados baseando-se numa árvore BSP (*Binary Space Partition*). Este tipo de cena é comumente encontrado em jogos, como Quake 3. Outro exemplo de especialização é o *PagingLandscapeSceneManager*, utilizado em grandes cenas abertas e terrenos extensos. Ele é capaz de otimizar a memória e a velocidade do programa através de técnicas de compressão de vértices e compartilhamento de coordenadas de textura.

As aplicações que utilizam o OGRE não precisam saber quais subclasses do *SceneManager* estão disponíveis para utilização. A aplicação pode simplesmente chamar o método **Root::getSceneManager()** passando o tipo da cena, que pode assumir valores representados pelas constantes **ST_GENERIC**, **ST_EXTERIOR_CLOSE**, **ST_EXTERIOR_FAR**, **ST_EXTERIOR_REAL_FAR** e **ST_INTERIOR**. Durante a chamada do método **Root::getSceneManager()**, o OGRE se encarregará de procurar a subclasse mais próxima da sugerida no tipo, ou retornar a subclasse genérica caso a especialista não esteja disponível. Isto permite ao desenvolvedor criar posteriormente um *SceneManager* especializado que otimizará um tipo de cena antes não otimizado, excluindo a necessidade de compilar novamente a aplicação.

### 2.3. O Objeto RenderSystem

O *RenderSystem* é uma classe abstrata que define a interface entre o OGRE e a API gráfica utilizada. Ele é responsável por executar comandos para renderização e configurar opções próprias da API gráfica. Esta classe precisa ser abstrata porque todos os comandos realizados por ela dependem da API de renderização utilizada, fazendo-se necessária a presença de subclasses específicas para cada uma delas (*D3D9RenderSystem*, *GLRenderSystem*, etc.).

Após a inicialização do sistema, através do método **Root::initialise()**, o objeto *RenderSystem* referente ao tipo escolhido na configuração pode ser acessado a partir do método **Root::getRenderSystem()**.

Mesmo sendo um dos componentes mais importantes do *engine*, o *RenderSystem* não deve ser acessado diretamente. Tudo que for preciso para renderizar objetos e definir opções está disponível a partir do *SceneManager*, do *Material* e de outras classes provenientes da cena. O acesso ao *RenderSystem* é somente justificado quando se deseja criar múltiplas janelas de renderização (janelas totalmente separadas, excluindo alguns

casos como múltiplos *viewports* para dividir uma mesma janela, que podem ser criados através da classe *RenderWindow*) ou quando se deseja acessar outros recursos avançados presentes somente no *RenderSystem*.

### 2.4. O Objeto Entity

Uma entidade é uma instância de um objeto móvel na cena. Por exemplo, pode ser um carro, um personagem, um avião, enfim, qualquer objeto relativamente pequeno e móvel na cena.

Toda entidade possui uma malha associada, ou seja, uma geometria que descreve o objeto graficamente e que é representada pelo objeto *Mesh*. Uma malha é composta por um conjunto de vértices e uma série de outras informações associadas a cada um deles, tais como normais, mapeamento de coordenadas de textura e índices de posicionamento nas faces.

Várias entidades podem referenciar o mesmo objeto *Mesh*. Tal fato ocorre quando existem mais de uma cópia de um objeto na cena. Desta forma, mesmo com vários objetos do mesmo tipo sendo exibidos na cena, existirá apenas uma cópia do objeto *Mesh* para todos eles, otimizando assim o tempo de carregamento ou replicação das malhas e a memória utilizada pela aplicação.

As entidades são criadas através do método **SceneManager::createEntity()**, fornecendo-se um nome para a entidade e o nome do arquivo "**.mesh**" que contém as informações da malha. O *SceneManager* não carregará diretamente o arquivo, pois tal operação será executada pelo *MeshManager*, responsável por carregar a malha apenas uma vez, mesmo sendo requisitado inúmeras vezes.

Para que uma entidade seja renderizada, é necessário associá-la a um *SceneNode*, que será visto na próxima subseção. Quaisquer informações relativas à posição, à orientação e à escala não serão armazenadas diretamente na entidade, mas sim no *SceneNode* ao qual ela está associada.

Quando uma *Mesh* é inicializada, ela automaticamente carrega os materiais que foram referenciados no arquivo "**.mesh**". É possível se ter mais de um material associado a uma *Mesh*, sendo cada um deles associado a um subconjunto da malha (*SubMesh*). Normalmente uma *Mesh* é formada por várias *SubMeshes* e cada uma destas recebe originalmente um material diferente. Sendo assim, se uma *Mesh* possuir apenas um material, ela será composta por uma única *SubMesh*.

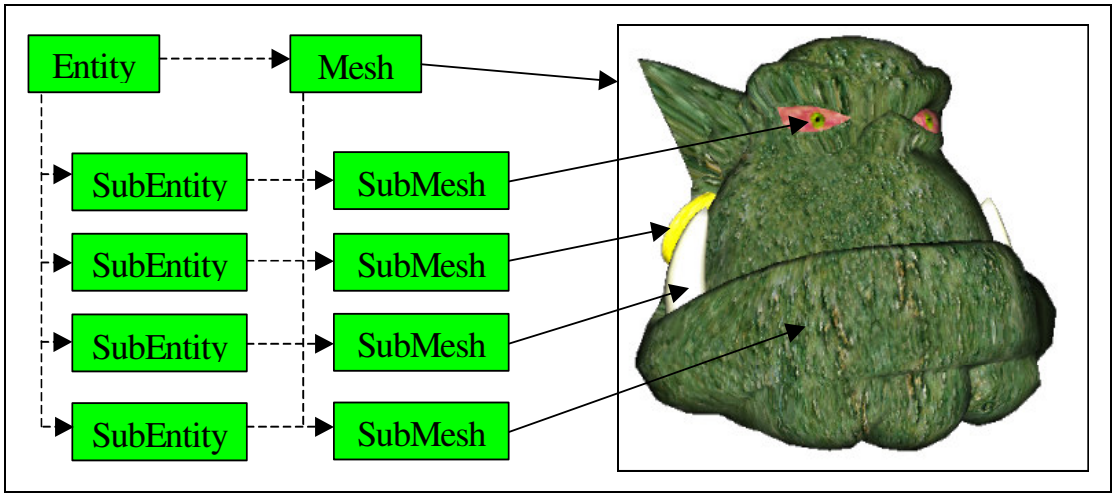



**Figura 6. Relação entre *Entity*, *SubEntity*, *Mesh*, *SubMesh* e *Material*.**

Uma entidade é composta por uma ou mais sub-entidades (*SubEntity*), dependendo apenas da quantidade de *SubMeshes* contidas na *Mesh* a qual ela referencia. Cada *SubMesh* terá uma *SubEntity* relacionada, que armazenará informações adicionais como o *Material* que pode ser modificado através do método **subEntity::setMaterialName()**, como mostrado na Figura 6. Desta forma, pode-se modificar a entidade dando-se uma aparência diferente da *Mesh* que a originou.

## 2.5. O Objeto SceneNode

*SceneNode* é um objeto utilizado para agrupar entidades, luzes, câmeras e objetos móveis e armazenar informações relativas a posição, orientação e escala destas.

Toda entidade, para tornar-se visível, deve estar associada a um *SceneNode*, que deve fazer parte da estrutura hierárquica que compõe a cena. Esta estrutura é composta inicialmente por um nó raiz, disponibilizado pelo *SceneManager* através do método **SceneManager::getRootSceneNode()**, e deve ser expandida criando-se nós que serão filhos deste nó raiz. Cada nó pode ter mais de um filho, podendo assim compor um ambiente hierárquico qualquer. Pode também fazer parte desta hierarquia nós como câmeras e luzes. Estes, particularmente, não precisam estar associados a *SceneNodes*, uma vez que também armazenam informações de orientação e posicionamento.

Para a criação de uma hierarquia de nós deve-se utilizar o método **SceneNode::createChildSceneNode()**, o qual cria nós filhos, como mostrado na Figura 7.

```
SceneNode *rootNode = this->sceneManager->getRootSceneNode();

SceneNode *childNode = rootNode->createChildSceneNode();
```

**Figura 7. Criação de uma hierarquia simples a partir do nó raiz.**

Pode-se utilizar o *SceneNode* também para aplicar transformações espaciais às entidades associadas. Estas transformações são aplicadas hierarquicamente a todos os nós descendentes, facilitando a construção e animação de objetos hierárquicos. A Figura

8 ilustra uma hierarquia de nós em um carro, na qual os pneus são filhos da carroceria. Assim, as transformações realizadas sobre a carroceria também afetarão os pneus.

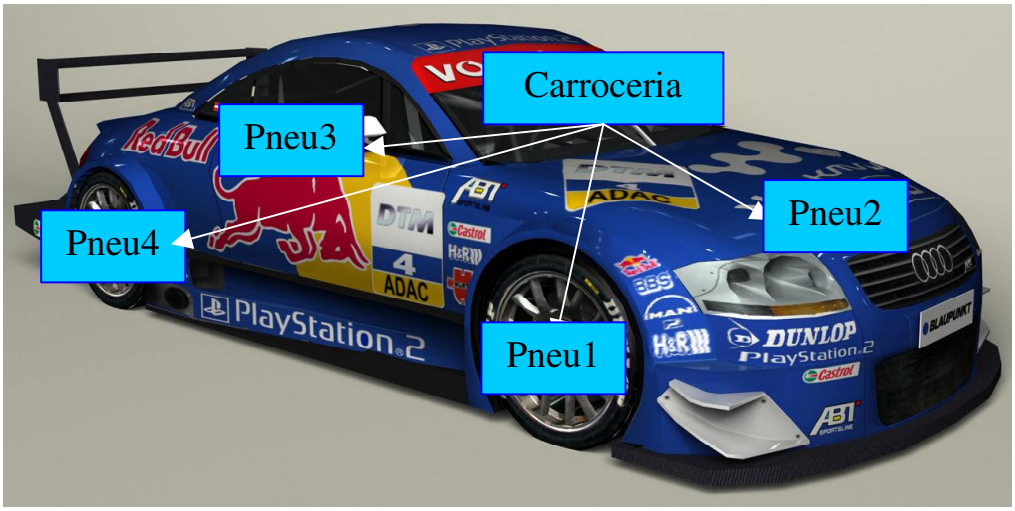



**Figura 8. Hierarquia de *SceneNodes*.**

Tais conjuntos de transformações podem ser traduzidos em operações, como translações e rotações, sendo estas últimas em relação a um determinado eixo de coordenadas (*row*, *pitch*, *yaw*) ou a um eixo qualquer (utilizando quatérnios).

Outros comandos também estão disponíveis através do *SceneNode*. Entre eles estão o **setVisible**, que alterna a visibilidade do grupo de entidades associado ao nó e o **showBoundingBox**, que mostra a caixa que delimita o volume das entidades.

Cada *SceneNode* pode ser identificado através de um nome dado no momento de sua criação. Sendo assim, é possível fazer buscas utilizando como parâmetro o nome do *SceneNode* desejado, a partir do objeto *SceneManager*, como ilustrado na Figura 9.

```
SceneNode *node = this->sceneManager->getSceneNode("cabecaNode");
```

**Figura 9. Buscando um *SceneNode*.**

## 2.6. O Objeto Camera & Viewport

O objeto *Camera* representa um ponto de vista da cena, simbolizado por um nó com propriedades de *Frustum*. O conceito de *Frustum* está ligado a uma área restrita de visualização, que possui atributos como campo de vista (ângulo de abertura), aspecto (relação entre largura e altura), e os planos que delimitam o volume de visualização (*near plane* e *far plane*), como mostra a Figura 10.

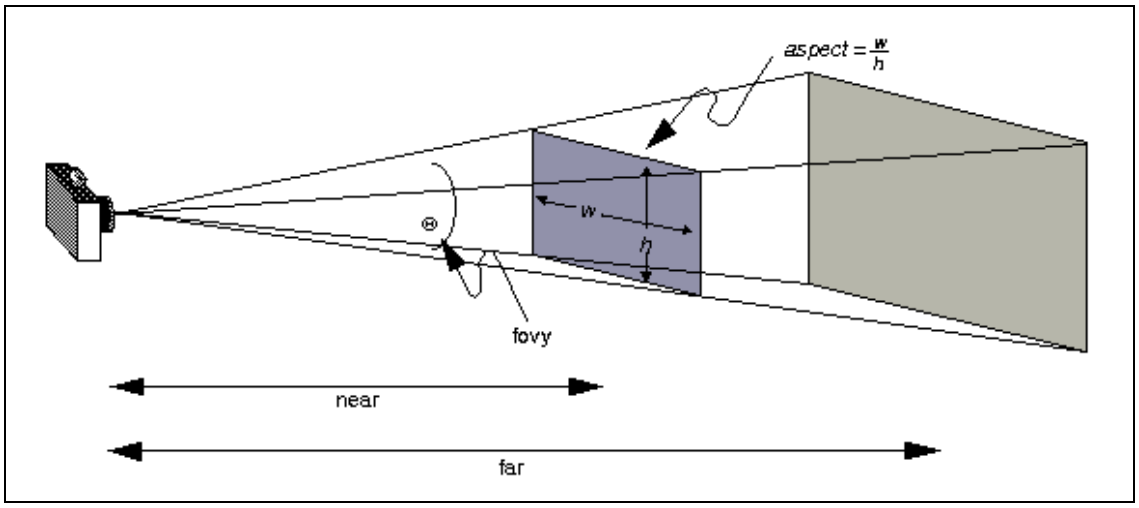



**Figura 10. Pirâmide de visualização (Frustum) e seus atributos.**

```
SceneManager* sceneMgr = root->getSceneManager(ST_GENERIC);
//-----------------------------------------------------
// Cria a câmera
//-----------------------------------------------------
Camera* camera = sceneMgr->createCamera("SimpleCamera");
//-----------------------------------------------------
// cria um viewport que preenche a janela toda
//-----------------------------------------------------
Viewport* viewPort = root->getAutoCreatedWindow()->addViewport(camera);
//-----------------------------------------------------
// Ajusta a cor de fundo para “preto”
//-----------------------------------------------------
Viewport->setBackgroundColour(ColourValue::Black)
```

**Figura 11. Exemplo de como uma câmera é criada e anexada a um *viewport*.**

A câmera possui ainda uma posição e uma orientação, podendo ser movimentada dinamicamente. Sua criação deve ser feita sempre através do *SceneManager*, o qual guarda referências para as câmeras e pode retorná-las a partir de um nome associado a elas no momento de sua criação, como ilustrado na Figura 11.

Uma câmera só deverá ser utilizada como ponto de vista se associada a um alvo de visualização (*RenderTarget*), como uma janela ou uma textura. No caso mais comum adicionamos uma câmera a uma *RenderWindow* e criamos uma área de visualização dentro da janela chamada *Viewport*. O *viewport* tem informações como altura, largura, cor do fundo e posicionamento no alvo de visualização, que podem ser utilizadas para criar mecanismos como visualizações diferentes em uma mesma janela (recurso bastante utilizado em jogos *multiplayer* para consoles).

A partir de *Cameras* e *Viewports* pode-se também criar efeitos de reflexão do ambiente utilizando outro tipo de *RenderTarget*, chamado *RenderTexture*, que será melhor descrito na seção 6.2.

## 2.7. O Objeto Material

O objeto *Material* controla como os objetos da cena serão renderizados, o que reflete no modo de visualização posterior dos mesmos. No objeto *Material* são especificadas quais

as propriedades da superfície do material, como os componentes do modelo de iluminação (ambiente, difuso, especular e emissivo). Também fazem parte do *Material* as camadas de texturas associadas a ele, suas imagens correspondentes, e como elas devem interagir para formar a superfície final (*blending*, composição aditiva, composição modulativa, etc.). A Figura 12 ilustra o resultado da composição de duas texturas. Além disso, alguns efeitos podem ser adicionados através do *Material*. Fazem parte deste conjunto de efeitos técnicas como mapeamento de ambiente (*Environment Mapping*), filtros de textura, animações e modo de *culling* (se o material deve ser aplicado nas duas faces dos triângulos da malha). Os *Materials* são identificados unicamente por um nome e podem ser atribuídos a qualquer *Entity* ou *SubEntity* através dos métodos **Entity::setMaterialName()** ou **SubEntity::setMaterialName()**.


**Figura 12. Resultado da composição de duas texturas.**

Um *Material* é composto de um conjunto de *Techniques* e *Passes* que podem ser alteradas através do código C++ ou carregadas em um *script* de configuração de material. Esta característica torna a edição e modificação do material independente do código fonte, podendo inclusive haver uma modificação sem que seja necessária uma recompilação. Os *scripts* e seus atributos e facilidades serão melhores explorados na seção 5.

## 3. Overlays

*Overlays* são estruturas que permitem a renderização de elementos 2D e 3D na frente dos elementos da cena, criando assim efeitos como HUDs (*Head-Up Displays*), menus e painéis de *status*. O painel de estatísticas e taxa de exibição de quadros que vem por padrão nas aplicações de demonstração do OGRE é um exemplo de utilização de *Overlays* e pode ser visto na Figura 13.


**Figura 13. Exemplo de utilização de *Overlays*.**

Os *Overlays* podem ser criados, assim como os *Materials*, tanto através do *OverlayManager* quanto a partir de um *script* "**.overlay**". Normalmente, os *Overlays* são criados a partir de *scripts*, pela facilidade de edição e independência do código fonte. Logo após sua criação todos os *Overlays* estão desabilitados, ou seja, escondidos até que seja chamado o método **Overlay::show()** referente a ele. Vários *Overlays* podem ser exibidos de uma só vez e se eles se sobrepuserem, a ordem de aparição será computada levando em conta o atributo **Z-Order**, que pode ser determinado a partir do método **Overlay::setZOrder()**.

A classe *OverlayElement* representa os elementos 2D que são adicionados a um *Overlay*. Todos os itens que podem ser adicionados a *Overlays* derivam desta classe, contendo atributos como tamanho, posição, nome do material, entre outros. Uma subclasse importante de *OverlayElement* é a *OverlayContainer*, que adiciona a característica de agrupamento de outros *OverlayElements*, permitindo movimentação e alinhamento conjunto de elementos.

A classe *OverlayManager* é utilizada para a criação e adição de *Overlays* ao sistema. Isto é feito através do método **OverlayManager::createOverlayElement()**, que pode ser utilizado para criar tipos predefinidos de *Overlays*, tais como os *Panels* que podem ser criados a partir de chamadas, como mostra a Figura 14.

```
OverlayElement *element = OverlayManager::getSingleton().createOverlayElement("Panel", "myNewPanel");
```

**Figura 14. Criação de um *OverlayElement* do tipo *Panel*.**

## 4. Tipos Suportados

Esta seção trata dos tipos suportados pelo OGRE. A subseção Malhas apresenta o formato *mesh* e mostra como as malhas podem ser exportadas a partir de programas de modelagem 3D, como o 3DS Max. A seção Texturas apresenta a utilização das mesmas pelo *engine*.

### 4.1. Malhas

Por padrão, o OGRE oferece suporte ao carregamento de malhas através dos arquivos "**.mesh**". Estes arquivos podem ser facilmente exportados através de inúmeros *softwares* de modelagem 3D, como 3DS Max, Maya, Blender, etc. Além dos arquivos "**.mesh**", existem os arquivos "**.xml.mesh**" para armazenar dados da malha, sendo esses diferenciados dos primeiros apenas por não serem binários.

O processo de exportação das malhas é realizado em dois passos. Primeiramente um arquivo "**.xml.mesh**" é gerado pela ferramenta de modelagem 3D. Este arquivo, por não ser binário, pode ser lido, entendido e editado diretamente em qualquer editor de texto, possibilitando assim que pequenas alterações possam ser realizadas diretamente. Em seguida, o "**.xml.mesh**" deve ser traduzido para um "**.mesh**", sendo este último o arquivo final que será carregado pelo *engine*. Este último passo é abstraído caso o modelo tenha sido exportado pelo 3DS Max, pois a ferramenta o executa implicitamente (o 3DS utiliza automaticamente o tradutor). A Figura 15 mostra a interface do "**.mesh**" *exporter* do 3DS Max.

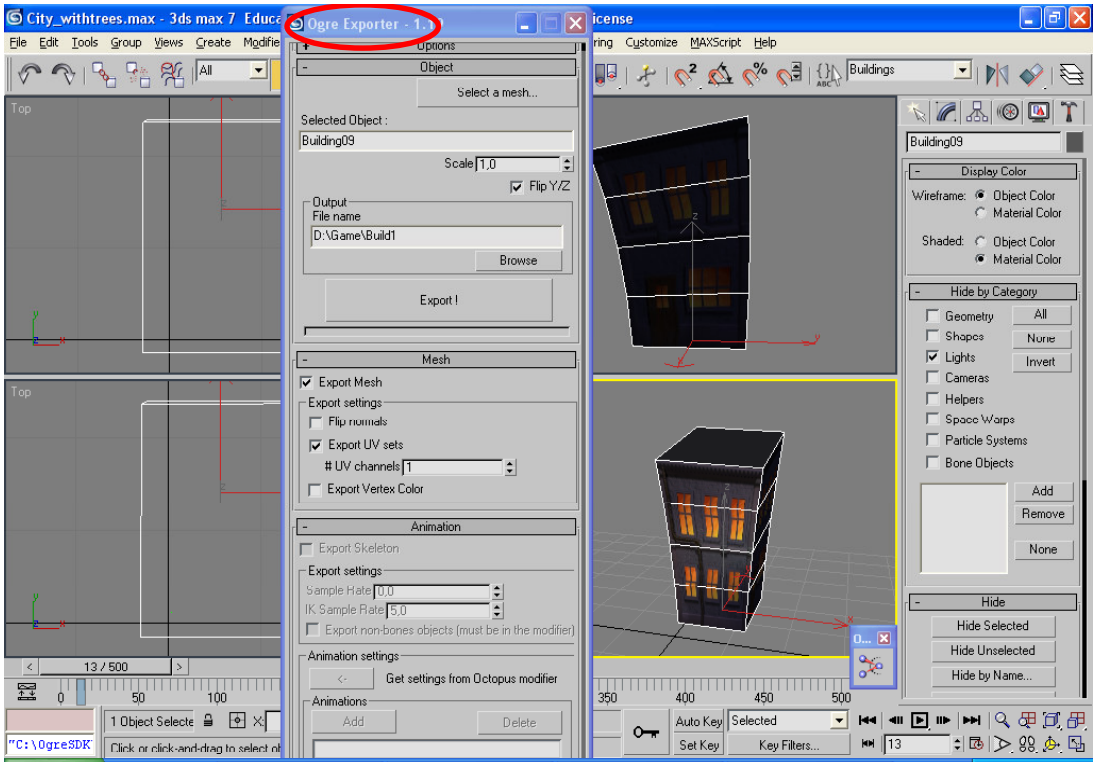



**Figura 15. Interface do *exporter* de *mesh* no 3DS Max.**

A tradução dos arquivos "**.xml.mesh**" para os "**.mesh**" é feita por uma ferramenta chamada *OgreXMLConverter*. Ela está disponível para *download* gratuito no

*site* oficial do OGRE. Outra ferramenta disponibilizada no *site* oficial é o *MeshViewer*, que possibilita que os arquivos de malha possam ser visualizados antes de serem carregados pelo *engine*.

### 4.2. Texturas

O OGRE utiliza uma biblioteca chamada DevIL para o carregamento de texturas em memória, o que o faz bastante versátil com relação a diversidade de tipos de imagens suportadas. Esta biblioteca também é multiplataforma, o que garante a homogeneidade de implementação nas versões do OGRE para Macintosh, Windows e Linux.

A utilização da biblioteca DevIL garante ao OGRE suporte às imagens dos tipos **.bmp**, **.cut**, **.dcx**, **.dds**, **.ico**, **.gif**, **.jpg**, **.lbm**, **.lif**, **.mdl**, **.pcd**, **.pcx**, **.pic**, **.png**, **.pnm**, **.psd**, **.psp**, **.raw**, **.sgi**, **.tga**, **.tif**, **.wal**, **.act** e **.pal**.

Caso exista um formato de imagem que deva ser carregado e não esteja na lista de tipos suportados pelo OGRE, existe um recurso chamado *ManualLoader*, que consiste na implementação de uma classe a qual o OGRE se reportará quando for necessário carregar uma imagem de um determinado tipo, tornando possível a adição de texturas neste novo formato.

## 5. Scripts

*Scripts* são simples arquivos de texto que podem ser criados e alterados em qualquer editor de texto, e que têm o propósito de agilizar o processo de desenvolvimento. Eles servem como um caminho alternativo ao código fonte para criação de alguns recursos suportados pelo *engine*. Os *scripts* suportados pelo OGRE são: *Initialization Scripts, Material Scripts*, *Particle Scripts*, *Overlay Scripts* e *Font Definition Scripts*.

Os *scripts*, além de tornarem o código fonte da aplicação mais enxuto, também descartam a necessidade de recompilação do código quando houver a necessidade de alteração de parâmetros dos recursos. Esta característica torna o processo de experimentação mais rápido e menos cansativo. Outra característica positiva dos *scripts* é que eles podem ser reusados facilmente por outras aplicações, já que são independentes do código fonte.

### 5.1. Inicialização

Fazem parte dos *scripts* de inicialização os arquivos "**plugins.cfg**", "**ogre.cfg**" e "**resources.cfg**". Eles são responsáveis, respectivamente, pelo carregamento dos *plugins*, auto-preenchimento das opções de configuração e definição de caminhos para os recursos utilizados pela aplicação. Este último é utilizado obrigatoriamente apenas para aplicações que derivam da classe *ExampleApplication*, descrita detalhadamente na seção 7.

Cada arquivo tem seu formato específico. No caso do arquivo "**plugins.cfg**" existe uma configuração chamada de *PluginFolder*, que aponta para a pasta onde estarão localizados todos os *plugins* listados no arquivo. As demais configurações são do tipo *Plugin* e devem ser seguidas de nomes de *plugins* a serem carregados (os nomes das bibliotecas sem a extensão "**dll**"). O arquivo "**resources.cfg**" é formado de diretivas *Zip* e *FileSystem*, que apontam respectivamente para arquivos compactados no formato *zip* e

para pastas onde o OGRE procurará por mais arquivos de *scripts* e recursos utilizados na aplicação. O arquivo "**ogre.cfg**" é dispensável, pois é gerado sempre que a tela de configurações exibida na inicialização da aplicação é fechada. Ele contém os últimos valores configurados pelo usuário.

### 5.2. Materiais

*Material Scripts* possibilitam a definição de materiais complexos de forma fácil e rápida, os quais podem ser reusados posteriormente. Na definição de materiais através de *scripts*, o desenvolvedor pode escrever seus *scripts* e posteriormente carregá-los quando necessário.

Arquivos contendo *scripts* de materiais geralmente são identificados pela extensão "**.material**". Esses arquivos são analisados (*parsed*) automaticamente no momento da inicialização da aplicação. Por padrão, a aplicação busca em todos os diretórios de recursos comuns pelos arquivos "**.material**", e analisa a estrutura de cada script. Ao final da análise, os *scripts* que possuírem erro na sintaxe não serão carregados, e conseqüentemente os objetos que utilizem este material aparecerão brancos.

É possível definir inúmeros materiais em um único arquivo "**.material**". Essa é uma opção bastante utilizada quando se deseja agrupar materiais relativos a uma única entidade da cena. Por exemplo, todos os materiais aplicados em um personagem de um jogo podem ser agrupados em um único arquivo "**.material**", facilitando a busca dos mesmos caso seja necessária alguma alteração.

A estrutura de um *script* de material é relativamente simples; ela segue uma estrutura hierárquica de seções. Como mostra a Figura 16, as seções de um *script* de material são: *material*, *technique*, *pass* e *texture_unit*. Cada seção possui um conjunto específico de atributos e, caso um atributo não apareça na sua respectiva seção, a este será atribuído um valor padrão.

```
// Isto é um comentário
material Rio/Agua
{
    // Técnicas com maior prioridade ficam em cima
    technique
    {
        // Primeiro passo
        pass
        {
            // Aqui ficam o atributos do passo
            ambient 0.5 0.5 0.5
            diffuse 1.0 1.0 1.0

            // Texture unit 0
            texture_unit
            {
                texture agua1.jpg
                scroll_anim 0.1 0.0
                wave_xform scale sine 0.0 0.7 0.0 1.0
            }
            // Texture unit 1 (é um passo com multitexture)
            texture_unit
            {
                texture agua2.png
                rotate_anim 0.25
            }
        }
    }

    // Segunda técnica, pode ser utilizada como fallback ou LOD
    technique
    {
        // … etc
    }

}
```

**Figura 16. Estrutura básica de um *script* de *material*.**

O valor padrão dos atributos pode variar de atributo para atributo; por exemplo, o valor padrão do atributo *ambient* é 1.0 1.0 1.0 1.0, enquanto o do atributo **cull_hardware** é *clockwise*. A Tabela 1 descreve estes atributos.

**Tabela 1. Tabela de atributos presentes no *script* de material.**

| Seção | Atributo | Valor Padrão | Descrição |
|---|---|---|---|
| **material** | lod_distances | - | Indica a distância de atuação das técnicas |
| | receive_shadows | On | Indica se o material recebe sombras |
| | transparency_casts_shadows | Off | Indica se um material transparente projeta sombra |
| **technique** | lod_index | 0 | Define o índice do LOD ao qual essa technique pertence |
| **pass** | ambient | 1.0 1.0 1.0 1.0 | Define a componente ambiental do material |
| | diffuse | 1.0 1.0 1.0 1.0 | Define a componente difusa do material |
| | specular | 0.0 0.0 0.0 0.0 0.0 | Define a componente especular do material |
| | emissive | 0.0 0.0 0.0 0.0 0.0 | Define a componente emissora do material |
| | scene_blend | one zero (opaco) | Indica o tipo de blending |
| | depth_check | On | Indica se o z-buffer será checado na renderização |
| | depth_write | On | Indica se o z-buffer será atualizado na renderização |
| | depth_func | less_equal | Define a função da checagem do z-buffer |
| | depth_bias | - | Define o valor aplicado à profundidade do pixel |
| | alpha_rejection | always_pass | Define o valor de descarte de pixels no alphatest |
| | cull_hardware | Clockwise | Define o modo de hadware culling |
| | cull_software | Back | Define o mode de software culling |
| | Lighting | On | Define se iluminação dinâmica está habilitada |
| | Shading | Gouraud | Define o tipo de shading |
| | fog_override | False | Indica se material deve ser afetado por fog |
| | colour_write | On | Indica se será renderizado com escrita de cor |
| | max_lights | 8 | Define o número máximo de luzes do passo |
| | Iteration | Once | Indica se o passo é executado mais de uma vez |
| **texture_unit** | texture | None | Indica a imagem usada com textura |
| | anim_texture | None | Indica a imagem usada com textura animada |
| | cubic_texture | None | Indica a imagem usada com textura cúbica |
| | tex_coord_set | 0 | Indica que conjunto de coordenadas UV será usado |
| | tex_address_mode | Wrap | O que ocorre quando a coordenada UV ultrapassar 1.0 |
| | filtering | Bilinear | Indica o filtro utilizado |
| | max_anisotropy | 1 | Define o nível máximo de anisotropia |
| | colour_op | Modulate | Define como as cores das texturas serão combinadas |
| | colour_op_ex | none | Extensão do atributo colour_op |
| | colour_op_multipass_fallback | - | Define a operação de fallback multi passo |
| | alpha_op_ex | - | Define como os valores do alpha serão combinados |
| | env_map | Off | Habilita/Desabilita o efeito de coordenadas de textura |
| | scroll | - | Define o deslocamento da rolagem da textura |
| | scroll_anim | - | Define a animação da rolagem da textura |
| | rotate | - | Ângulo da rotação da textura |
| | rotate_anim | - | Define a animação da rotação da textura |
| | scale | - | Define o fator de escala aplicado à textura |
| | wave_xform | - | Define a função na qual a animação será baseada |

O identificador de um material é definido na declaração da seção material, seguindo o formato material **<identificador>**. Todo material deve ser identificado por uma *string* globalmente única, caso contrário, a aplicação será finalizada devido ao conflito entre nomes. Os identificadores podem conter caracteres '/', de modo que a noção de árvore possa ser expressa.

As técnicas são definidas através da palavra *technique* e diferentemente de um material, não requerem identificador. Uma *technique* pode ser vista como uma forma de se chegar a um dado efeito, desta forma, um material pode ser composto por várias técnicas. Materiais com múltiplas *techniques* são bastante úteis nos casos em que a aplicação será executada em diferentes plataformas e configurações de *hardware*. O *engine* escolherá (em tempo de execução) uma das *techniques* definidas no *script*, dando maior prioridade as que forem primeiramente declaradas. Assim, é recomendado que se disponham as *techniques* das mais sofisticadas para as menos sofisticadas. Múltiplas *techniques* também podem ser utilizadas em casos onde se pode utilizar um efeito

menos preciso em objetos que, por exemplo, estejam distantes do observador (*Level Of Detail* - LOD).

### 5.3. Programas

São considerados programas trechos de código utilizados para modificar o *pipeline* gráfico da placa de vídeo, de forma a personalizar o modo como os objetos são renderizados, iluminados ou transformados.

Existem dois tipos de programas, chamados programas de vértices (*vertex programs* ou *vertex shaders*) e programas de fragmentos (*fragment programs* ou *pixel shaders*). Programas de vértices podem modificar a posição dos vértices para fins de animação ou modificar a renderização através de iluminação por vértices (método Gouraud). Já os programas de fragmentos modificam basicamente a cor de cada *pixel* da cena, dando um aspecto visual personalizado ao material ao qual eles são aplicados.

A alteração do *pipeline* gráfico é uma tarefa extremamente complexa e exige conhecimento específico de computação gráfica e álgebra linear. Sendo assim, nesta seção daremos enfoque apenas na utilização de *scripts* previamente escritos.

Para utilizar um programa de vértice ou fragmento em um determinado material deve-se primeiro defini-lo, dizendo que tipo de programa é, identificando-o com um nome e especificando o tipo da linguagem de descrição, como ilustra a Figura 17.

```
fragment_program myCgFragmentProgram cg
{
    source myCgFragmentProgram.cg
    entry_point main
    profiles ps_2_0 arbfp1
}
```

**Figura 17. Exemplo de *fragment program*.**

No OGRE são suportadas três linguagens de alto nível para *shading*, entre elas a HLSL (*High Level Shading Language*, utilizada pelo DirectX), a GLSL (GL *Shading Language*, utilizada pelo OpenGL) e a CG (C for *Graphics*, criada pela NVidia, porém utilizada em qualquer plataforma). Além das linguagens de alto nível também são suportados programas de *shading* escritos em linguagem Assembly.

Após a definição do programa deve-se referenciá-lo dentro de algum *pass*, fazendo com que ele seja executado juntamente com o passo de renderização, como ilustrado na Figura 18.

```
...
pass
{
      vertex_program_ref myVertexProgram
      {
            param_indexed_auto 0 worldviewproj_matrix
            param_indexed      4 float4  10.0 0 0 0
      }
}
...
```

**Figura 18. Referência de um *vertex program* utilizada em uma seção *pass*.**

## 5.4. Partículas

*Scripts* de partículas permitem definir sistemas de partículas sem que sejam necessárias alterações no código fonte. Eles são carregados (como nos *scripts* de material) no momento da inicialização da aplicação, verificando por padrão em todos os locais de recursos comuns (**Root::addResourceLocation()**) e fazendo o *parsing* dos arquivos "**.particle**". Uma vez que os *scripts* tenham sido analisados, a aplicação estará apta a inicializar sistemas baseados neles usando o método **ParticleSystemManager::getSingleton().createSystem()**.

Muitos sistemas de partículas podem ser escritos em um único *script*. Um exemplo de *script* de partícula pode ser visto na Figura 19.

```
// A sparkly purple fountain
Examples/PurpleFountain
{
    material Examples/Flare2
    particle_width 20
    particle_height 20
    cull_each false
    quota 10000
    billboard_type oriented_self

    // Area emitter
    emitter Point
    {
        angle 15
        emission_rate 75
        time_to_live 3
        direction 0 1 0
        velocity_min 250
        velocity_max 300
        colour_range_start 1 0 0
        colour_range_end 0 0 1
    }

    // Gravity
    affector LinearForce
    {
        force_vector 0 -100 0
        force_application add
    }

    // Fader
    affector ColourFader
    {
        red -0.25
        green -0.25
        blue -0.25
    }
}
```

**Figura 19. Exemplo de um *script* de partículas.**

Um sistema pode ter um conjunto de atributos no nível mais alto, como por exemplo *quota*, que define a quantidade máxima de partículas geradas. Abaixo desta seção, existem basicamente duas outras, *emitter* e *affector*, que criam e modificam as

partículas, respectivamente. Os atributos disponíveis nos *affectors* e *emitters* são dependentes dos tipos de cada um. A Figura 19 mostra dois tipos de *affectors* diferentes, cada um com atributos diferentes. *Emitters* podem ser classificados como *Point*, *Box*, *Cylinder* ou *Ellipsoid*. Esta classificação define a forma da fonte de emissão.

### 5.5. Overlays

Os *scripts* de *Overlay* possibilitam que se defina *Overlays* em um arquivo de *script* e que os mesmos sejam reusados facilmente. Ao invés de se usar os métodos das classes *OverlayManager*, *Overlay* e *OverlayElement*, o que é um pouco inviável na prática devido ao grande volume de código, pode-se definir *Overlays* em arquivos de texto que podem ser carregados sempre que necessário.

Os *scripts* de *Overlay* são carregados pelo sistema durante a inicialização. Uma busca é feita nas pastas de recursos comuns da aplicação e os arquivos "**.overlay**" encontrados são interpretados e processados. Caso se queira processar um arquivo individual, deve-se usar o método **OverlayManager::getSingleton().parseScript()**.

Um único *script* pode conter vários *Overlays*. O formato do *script* é pseudo-C++, com as seções delimitadas por {} (chaves), comentários indicados por // (comentários aninhados não são permitidos) e herança através do uso de *templates*. O formato genérico desse tipo de *script* é mostrado na Figura 20.

```
// O nome do overlay deve ser colocado primeiro
MyOverlays/ANewOverlay
{
    zorder 200

    container Panel(MyOverlayElements/TestPanel)
    {
        // centraliza horizontalmente, colocando no topo
        left 0.25
        top 0
        width 0.5
        height 0.1
        material MyMaterials/APanelMaterial

        // Outro panel aninhado com esse
        container Panel(MyOverlayElements/AnotherPanel)
        {
            left 0
            top 0
            width 0.1
            height 0.1
            material MyMaterials/NestedPanel
        }
    }

}
```

**Figura 20. Formato genérico de definição de *script* de *Overlay*.**

O exemplo anterior define um único *Overlay* chamado "**MyOverlays/AnewOverlay**", com dois *panels*, sendo que um está aninhado dentro do

outro. O *script* utiliza métricas relativas (valor padrão se nenhuma opção **metrics_mode** for encontrada).

Pode-se incluir dentro de um *Overlay* qualquer número de elementos 2D/3D. Para isso, basta definir um bloco aninhado iniciado por "**element**" ou "**container**", caso se queira elementos 2D, ou "**entity**", no caso de se desejar incluir uma dimensão a mais.

*Templates* podem ser usados para criar vários elementos com as mesmas propriedades. Um *template* é um elemento abstrato e não é adicionado a um *Overlay*. Ele funciona como uma classe base que pode ser herdada por algum elemento para compartilhar suas propriedades padrão. Para se criar um *template*, a palavra "**template**" deve ser a primeira a estar presente na definição do elemento (antes mesmo de *container*, *element* ou *entity*) e a herança funciona de forma similar à sintaxe de C++, usando-se "**:**" após a declaração do elemento.

Os atributos de um *Overlay* são válidos dentro do escopo delimitado pelas {} (chaves) e cada um deve estar unicamente presente na linha. Uma lista dos atributos mais utilizados pode ser encontrada na Tabela 2.

**Tabela 2. Atributos do *Overlay* e suas descrições.**

| Atributo | Valor Padrão | Descrição |
|---|---|---|
| metrics_mode | relative | indica a unidade que será usada para dimensionar e posicionar o elemento |
| horz_align | left | indica o alinhamento horizontal do elemento |
| vert_align | top | indica o alinhamento vertical do elemento |
| left | 0 | indica a posição horizontal do elemento de acordo com seu componente pai |
| top | 0 | indica a posição vertical do elemento de acordo com o seu componente pai |
| width | 1 | indica a largura do elemento proporcionalmente ao tamanho da tela |
| height | 1 | indica a altura do elemento proporcionalmente ao tamanho da tela |
| material | none | indica o nome do material a ser usado pelo componente |
| caption | blank | indica uma legenda textual para ser usada pelo componente |
| rotation | none | indica a rotação do elemento |

Os elementos comuns que já vêm por padrão no *engine* são o *Panel*, o *BorderPanel*, o *TextArea* e o *TextBox*, sendo os dois primeiros *containers* e os dois últimos *elements*. O *Panel* é o *container* mais básico que pode ser usado. Um de seus atributos indica se o *container* será transparente ou não, por exemplo. O *BorderPanel* é um tipo de *Panel* mais complexo, no qual pode-se definir valores de tamanho das bordas e material usado nas mesmas. O *TextArea* é usado quando se deseja renderizar texto na tela, e possui atributos como nome da fonte, cor e altura dos caracteres. O elemento *TextBox*, utilizado para entrada de texto, é formado por uma *TextArea* e um *Panel* de fundo. Sua *TextArea* define as características do texto que será digitado e o *Panel* de fundo o local onde a entrada será colocada.

## 6. Facilidades do Engine

Nesta seção serão apresentadas algumas das facilidades oferecidas pelo OGRE. Elas estão descritas em três subseções intituladas Geração de Sombras, *Render-to-Texture* e *Mouse Picking*. Estas características possibilitam que os desenvolvedores possam implementar aplicações mais agradáveis para o usuário, tanto do ponto de vista visual quanto do de interação.

### 6.1. Geração de Sombras

Geração de sombras é um aspecto importante na busca pelo realismo de uma cena, pois ajuda na percepção da distância entre os objetos. Existem inúmeras formas de gerar sombras, cada uma com suas vantagens e desvantagens. De uma forma geral, quanto mais próxima da realidade for a simulação de sombras, mais ela será computacionalmente custosa. O OGRE oferece várias implementações de sombra, de modo que o desenvolvedor possa escolher qual implementação é a mais apropriada para a sua aplicação. Até a versão 1.0.7, o OGRE suporta três tipos de cálculos de sombra:

- *Texture Modulative* – é a menos custosa das três, porém não gera efeitos tão bons. Ela funciona criando um *Render-to-Texture* (ver detalhes na próxima seção), que então gera os mapas de textura que serão aplicados a cena simulando as sombras;
- *Stencil Modulative* – não é tão precisa quanto a *Stencil Additive*, mas é mais rápida. Ela funciona renderizando os volumes das sombras após a renderização dos objetos não transparentes;
- *Stencil Additive* – esta técnica renderiza cada luz como um passo aditivo separado na cena. É a mais custosa das três, pois para cada luz adicional ela requer um novo passo de renderização.

Para se ter sombras em uma cena, primeiramente o cálculo das mesmas deve ser habilitado através do método **SceneManager::setShadowTechnique()**. É importante que este passo seja executado antes de qualquer outro, pois a técnica escolhida pode alterar a forma como as malhas são carregadas. Em seguida, uma ou mais luzes devem ser criadas, pois sem elas as sombras não podem ser geradas. Após isso, desabilita-se a projeção de sombra nos objetos que não devem projetá-las. Este passo é realizado pelo método **Entity::setCastShadows()**, que recebe como parâmetro um valor booleano. E, por final, define-se quais objetos receberão ou não sombras. Tal definição (diferentemente da anterior) é realizada em cada *Material*, podendo ser através do *script* (no atributo **receive_shadows**), ou diretamente no código (**Material::setReceiveShadows()**).

A Figura 21 mostra a diferença de resultado visual gerado pelos diferentes tipos de sombra.

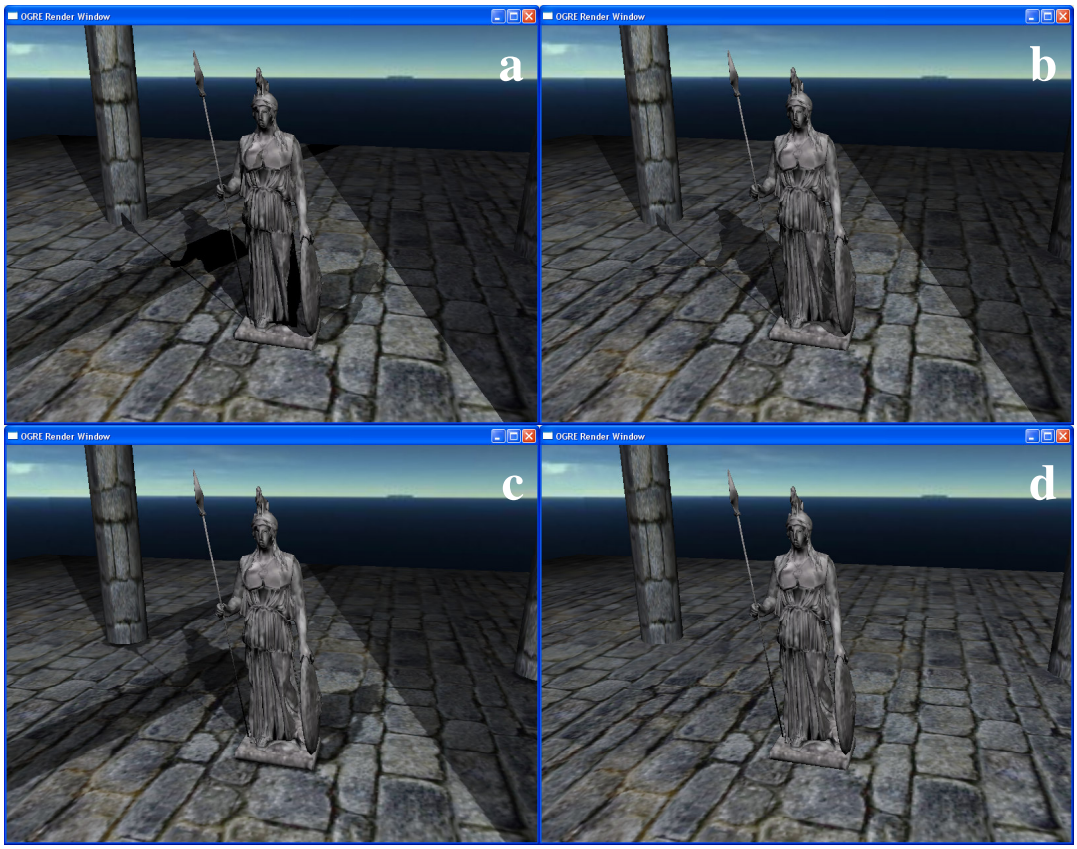



**Figura 21. Geração de sombras: (a) *Stencil Additive*; (b) *Stencil Modulative*; (c) *Texture Modulative*; (d) sem sombra.**

## 6.2. Render-to-Texture

*Render-to-Texture* consiste numa técnica para renderizar cenas em texturas que são utilizadas em materiais para torná-los reflexivos. Esta técnica pode ser usada, por exemplo, para compor o material da carroceria de um carro, que tem uma cor base e também reflete o ambiente.

Para utilizar esta técnica no OGRE temos que entender o conceito de *RenderTargets*. Estes servem como "alvos" de renderização para uma determinada câmera, que é associada a um *viewport* (porção do alvo que será utilizada para renderização) deste alvo. Desta forma, podemos classificar como *RenderTargets* os objetos *RenderWindow* e *RenderTexture*, sendo o último o alvo utilizado na renderização para uma textura.

Os passos necessários para aplicar um material que contenha uma *RenderTexture* são os seguintes:

1. criação do *script* de material (como visto na Figura 22);
2. adição do nome da *RenderTexture* como **texture_unit** de um material (como visto no primeiro **texture_unit** da Figura 22);

```
material RenderTextureMaterial
{
     technique
     {
          pass
          {
               texture_unit
               {
                    texture RTName
                    tex_addr_mode clamp
               }
               texture_unit
               {
                    texture fundoMaterial.jpg
               }
          }
     }
}
```

**Figura 22. *Material* que mistura uma textura básica com um componente reflexivo.**

3. criação (no código) de um *RenderTexture* com o mesmo nome utilizado no *script* de material (ilustrado na Figura 23);

```
RenderTexture* rttTex =
    mRoot->getRenderSystem()->createRenderTexture( "RTName", 512, 512 );
```

**Figura 23. Criação de um objeto *RenderTexture*.**

4. adição de um *viewport* através de uma câmera que vai definir que parte da cena será renderizada no *RenderTexture* (como visto na Figura 24);

```
Viewport *v = rttTex->addViewport( mCamera );
```

**Figura 24. Criação de um *viewport* em uma *RenderTexture*.**

5. aplicação do material criado em alguma entidade (visto na Figura 25).

```
this->entity->setMaterialName("RenderTextureMaterial");
```

**Figura 25. Aplicação de um material, que utiliza uma *RenderTexture*, a uma entidade.**

Pode-se, a partir da seqüência de comandos vista acima, obter resultados interessantes e significativos, do ponto de vista de mapeamento do ambiente em objetos reflexivos, como ilustrado na Figura 26, onde percebe-se a reflexão do ambiente (grama, árvores e céu) na cabeça do ogro.

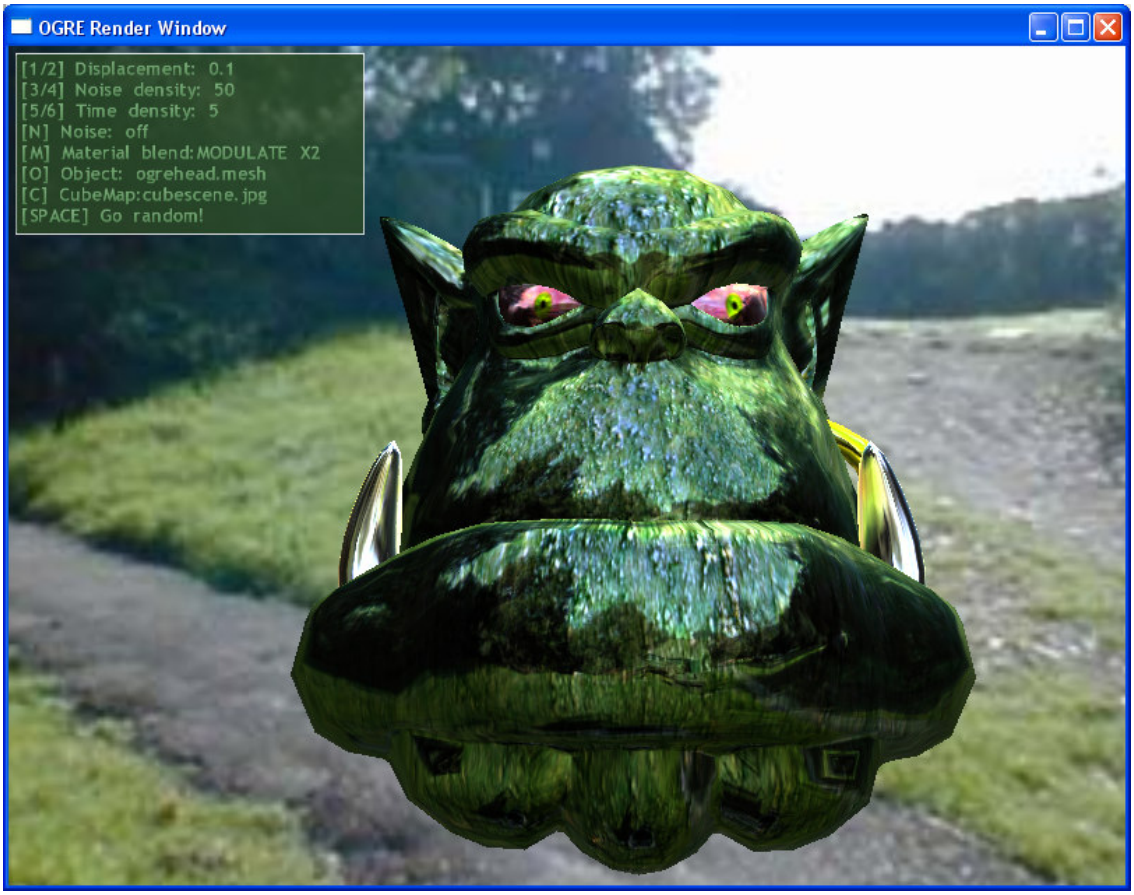



**Figura 26. Ilustração de material reflexivo.**

### 6.3. Mouse Picking

Uma rotina bastante comum em aplicativos 3D é a seleção de objetos da cena utilizando o mouse, a qual é denominada *Picking*. Tal rotina é suportada nativamente pelo OGRE, de modo que aplicações (principalmente editores 3D) que necessitem desse tipo de requisito possam ser desenvolvidas.

```
// Cria o objeto RaySceneQuery
this->raySceneQuery = this->sceneManager->createRayQuery( Ray() );

// Computa o raio originado da posição onde ocorreu o click do mouse
Ray mouseRay=this->camera->getCameraToViewportRay(e->getX(),e->getY());

// Atualiza o RaySceneQuery com o novo raio
this->raySceneQuery->setRay( mouseRay );

// Executa o query
RaySceneQueryResult &result = raySceneQuery->execute();
RaySceneQueryResult::iterator itr = result.begin();
```

**Figura 27. Objeto *RaySceneQuery* sendo criado e executado.**

O *Picking* é realizado através do objeto *RaySceneQuery*. Tal objeto é criado através do método **SceneManager::createRaySceneQuery()** recebendo como

parâmetro um raio (*Ray*). Este raio pode ser inicialmente nulo, pois ele pode ser redefinido posteriormente. Após a definição do raio, o *RaySceneQuery* deverá ser executado, retornando assim o conjunto de objetos interceptados pelo mesmo. Este raio é usualmente computado utilizando o método **Camera::getCameraToViewportRay()**, que recebe como parâmetro a posição relativa do mouse à janela. Um exemplo de código na qual um objeto *RaySceneQuery* é criado e executado pode ser visto na Figura 27.

O *Picking* padrão implementado no OGRE detecta os objetos com base no *AxisAlignedBox* dos objetos, o que faz com que os resultados sejam imprecisos, porém rápidos. Um *AxisAlignedBox* é um tipo de *BoudingBox* que sempre está alinhado com o eixo de coordenadas do mundo, assim, mesmo que os objetos sejam rotacionados seus *BoudingBoxes* continuarão alinhados. Se a aplicação requisitar uma forma mais acurada de *Picking*, esta deverá ser implementada pelo desenvolvedor. A implementação padrão do *Picking* no OGRE, por ser mais rápida, pode ser utilizada como um filtro inicial, diminuindo assim a área de busca na qual um algoritmo mais complexo atuará. A Figura 28 mostra os *AxisAlignedBoxes* (observe que eles sempre estão alinhados com o mundo) dos peixes nadando.

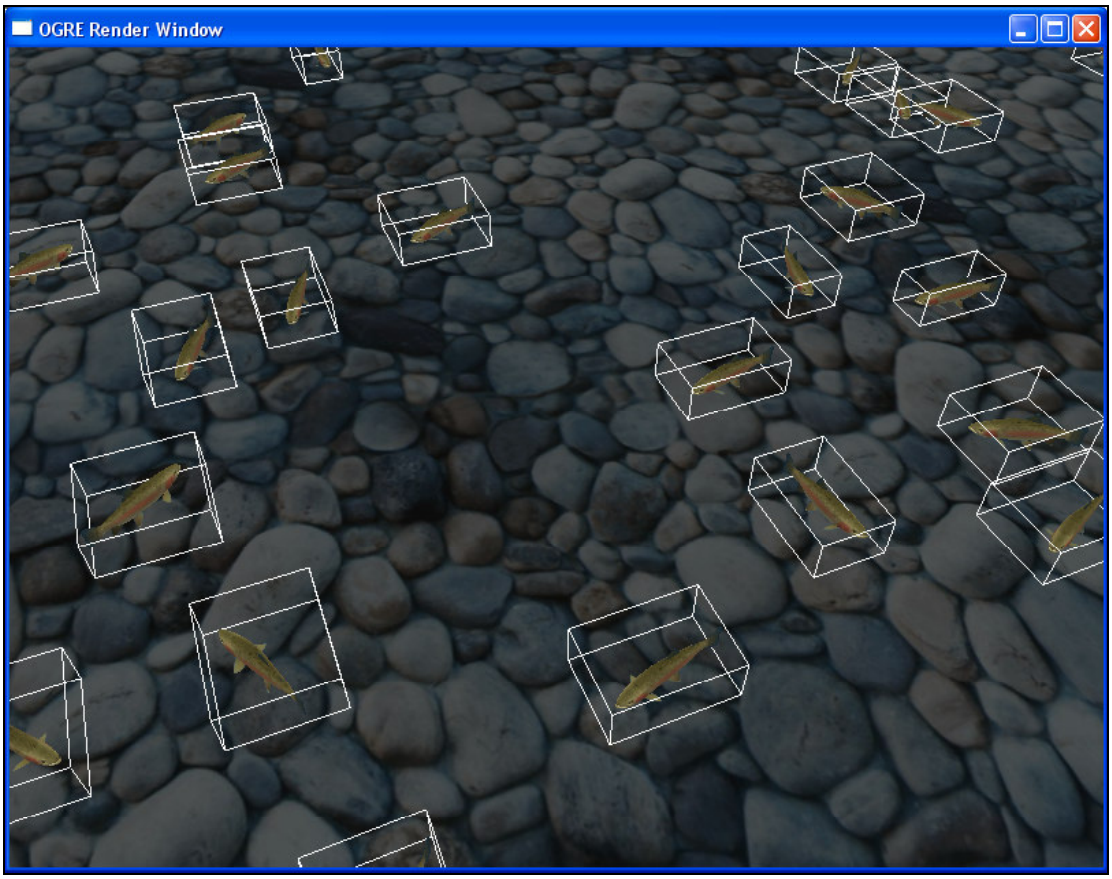



**Figura 28. *AxisAlignedBoxes* das entidades da cena.**

## 7. Aplicação Básica

O OGRE fornece um modelo básico de aplicação, que é seguido por todos os seus exemplos, composto por algumas classes básicas que representam aplicações e formas

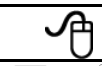
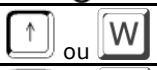
de interação padrão. Dentre elas encontram-se as classes *ExampleApplication* e *ExampleFrameListener*.

A classe *ExampleApplication* representa uma aplicação padrão, onde a configuração, a criação de janela, a criação de *viewports*, o carregamento de arquivo de inicialização, a criação do *SceneManager* e do *Root* são feitos automaticamente, podendo o programador interferir no que achar mais prudente, apenas sobrescrevendo o método correspondente à atividade escolhida para ser especializada nas classes filhas. Fazem parte da classe *ExampleApplication* os seguintes métodos:

- **createScene**: responsável por alocar e organizar os objetos e nós da cena;
- **createFrameListener**: responsável por criar e adicionar os *FrameListeners* ao *engine*;
- **createCamera**: responsável por criar e configurar a câmera;
- **chooseSceneManager**: responsável por criar o *SceneManager*;
- **configure**: configura a aplicação exibindo uma caixa de diálogo para o usuário e cria a janela de visualização;
- **createViewports**: cria os *viewports* da aplicação e corrige o aspecto (relação entre largura e altura) da câmera;
- **destroyScene**: responsável por destruir todo o conteúdo criado no *createScene*;
- **go**: responsável pela configuração e por iniciar o processo de renderização. A função *main* deve criar uma classe filha da *ExampleApplication* e invocar o método **ExampleApplication::go()** desta classe para dar início à aplicação.

A classe *ExampleFrameListener* é a implementação padrão da interface *FrameListener*, descrita no SDK do OGRE. Ela dispõe de vários recursos de interação com câmera e exibição de estatísticas bastante úteis durante a fase de construção e teste de uma aplicação. Também é implementado por ela um esquema de tratamento de eventos para o teclado e o mouse, criando alguns comandos básicos que estão descritos na Tabela 3.

**Tabela 3. Eventos tratados pelo *ExampleFrameListener*.**

| Entrada | Descrição |
|---|---|
|  | movimentação padrão do ponto de vista |
|  | movimenta para frente |
|  | movimenta para trás |
|  | movimenta para a esquerda |
|  | movimenta para a direita |
|  | movimenta para cima |
|  | movimenta para baixo |
|  | troca a visibilidade do quadro de estatísticas |
|  | modifica o modo de renderização (shaded, wireframe, points ) |
|  | modifica a filtragem da textura (bilinear, trilinear ou anisotrópica) |
|  | exibe e oculta a visualização das coordenadas da câmera |

Para que uma classe filha de *ExampleFrameListener* funcione plenamente é necessário que ao final do método **ExampleFrameListener::frameStarted()** (que deve ser sobrescrito para adicionar alguma ação específica ao *FrameListener*) seja chamado o método correspondente da classe pai, como mostra a Figura 30 da próxima subseção.

## 7.1. Modelo de Aplicação

A maneira mais simples e fácil de se criar uma aplicação no OGRE é herdando a classe *ExampleApplication* e sobrescrevendo o método **SceneManager::createScene()**. Essa classe vem com as configurações iniciais de cena, câmera e janela. A única ação que o desenvolvedor deve tomar é criar a cena a ser renderizada.

A Figura 29 descreve um exemplo de aplicação básica que utiliza a classe *ExampleApplication*, fornecida no SDK do OGRE. Define a classe *BasicApp* que contém os seguintes métodos:

- **createScene**: descreve os nós e entidades que estarão presentes nas cenas. As três primeiras linhas desse método modificam a cor da luz ambiente para preto, indicando que o ambiente está em completa escuridão e o tipo de técnica de sombras para o **SHADOWTYPE_STENCIL_ADDITIVE**, respectivamente. As dezesseis linhas seguintes definem um plano de resolução de 1500 por 1500 e de 20 segmentos de largura por 20 segmentos de profundidade, uma entidade e um nó para o plano, uma entidade e um nó para uma esfera com centro alinhado ao chão. Por fim, as cinco linhas seguintes definem um ponto de luz e a última linha situa a câmera numa posição do espaço;

- **createFrameListener**: descreve os *FrameListeners* da aplicação. É criado um objeto do tipo *BasicFrameListener* e adicionado à lista de *FrameListeners* do objeto *mRoot*, que é obtido através da classe *ExampleApplication*.

```
#include "ExampleApplication.h"

using namespace Ogre;

class BasicApp : public ExampleApplication {

private:

      Light* mLight;

protected:

      void createScene() {
            mSceneMgr->setAmbientLight(ColourValue(0, 0, 0));
            mSceneMgr->setShadowTechnique(
                  Ogre::SHADOWTYPE_STENCIL_ADDITIVE);

            Plane plane(Vector3::UNIT_Y, 0);
            MeshManager::getSingleton().createPlane("Ground",
                  ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
                  plane, 1500, 1500, 20, 20, true, 1, 5, 5,
                  Vector3::UNIT_Z);

            Entity* groundEntity = mSceneMgr->createEntity("GroundEntity",
                  "Ground");
            mSceneMgr->getRootSceneNode()->createChildSceneNode()
                  ->attachObject(groundEntity);
            groundEntity->setMaterialName("Examples/Rockwall");
            groundEntity->setCastShadows(false);

            Entity* sphereEntity = mSceneMgr->createEntity("Sphere",
                  "sphere.mesh");

            SceneNode* sphereNode = mSceneMgr->getRootSceneNode()
                  ->createChildSceneNode("SphereNode");
            sphereNode->attachObject(sphereEntity);

            mLight = mSceneMgr->createLight("PointLight");
            mLight->setType(Light::LT_POINT);
            mLight->setPosition(Vector3(10, 150, 0));
            mLight->setDiffuseColour(0.2f, 0.2f, 0.9f);
            mLight->setSpecularColour(0.2f, 0.2f, 0.9f);

            mCamera->setPosition(0, 100, 500);
      }

      void createFrameListener() {
            BasicFrameListener* listener = new BasicFrameListener(
                  mWindow, mCamera, mLight);
            listener->showDebugOverlay(true);
            mRoot->addFrameListener(listener);
      }
};
```

**Figura 29. Exemplo de uma aplicação básica utilizando a classe *ExampleApplication*.**

A Figura 30 descreve um exemplo de *FrameListener* que utiliza a classe *ExampleFrameListener* como base, também fornecida no SDK do OGRE. Ela define a classe *BasicFrameListener*, que recebe principalmente a luz criada na classe *BasicApp*. O seguinte método é sobrecarregado:

- **frameStarted**: recebe um *FrameEvent* que possui informações de tempo decorrido entre este e o último evento e entre este e o último quadro renderizado. Retorna um valor booleano indicando se a renderização de quadros deve continuar ou se a aplicação deve ser abandonada. As linhas seguintes alteram a posição da luz a cada quadro, definindo uma trajetória circular no plano XZ em que a luz se encontra.

```
#include "ExampleFrameListener.h"

using namespace Ogre;

class BasicFrameListener : public ExampleFrameListener {

private:

      Vector3 mLightPosition;
      Light* mLight;
      Real angle;

public:

      BasicFrameListener(RenderWindow* win, Camera* cam, Light* light)
      : ExampleFrameListener(win, cam, false, false), mLight(light) {
            mLightPosition = mLight->getPosition();
            angle = 0;
      }

      bool frameStarted(const FrameEvent& evt) {
            mLightPosition.x += Math::Cos(angle);
            mLightPosition.z += Math::Sin(angle);
            mLight->setPosition(mLightPosition);

            angle += evt.timeSinceLastFrame * Math::PI / 2;

            return ExampleFrameListener::frameStarted(evt);
      }
};
```

**Figura 30. Exemplo de um *FrameListener* para aplicação utilizando a classe *ExampleFrameListener*.**