VI Symposium on Virtual Reality (SVR 2003), p.117-128. Ribeirão Preto, SP, Brasil.

Experiência de Portais em Ambientes Arquitetônicos Virtuais

Romano J. M. Silva, Gustavo N. Wagner, Alberto B. Raposo, Marcelo Gattass

TeCGraf - Grupo de Tecnologia em Computação Gráfica / PUC-Rio
{romano,gustavow,abraposo,gattass}@tecgraf.puc-rio.br

Abstract. This paper presents an implementation of a portal algorithm to improve the perfomance of real time walkthroughs in architectural virtual environments with high realism. The algorithm was implemented with the OpenSceneGraph library, allowing its use with several virtual reality tools and models exported from commercial applications.

Resumo. Este artigo apresenta uma implementação de algoritmo de portais para melhorar o desempenho da navegação em tempo real em ambientes arquitetônicos virtuais com alto realismo gráfico. O algoritmo foi implementado sobre a biblioteca OpenSceneGraph, permitindo seu uso com várias ferramentas de realidade virtual e modelos provenientes de aplicações comerciais.

1 Introdução

Mesmo com o crescente progresso nas áreas de aplicação de ambientes virtuais, a navegação em modelos arquitetônicos utilizando as tecnologias de realidade virtual continua sendo uma tarefa constantemente aprimorada. Seu emprego permite, por exemplo, que arquitetos, engenheiros e usuários possam prever e verificar o interior de um apartamento antes da sua conclusão. Decoradores e projetistas podem, ainda, analisar a iluminação do ambiente e o posicionamento de objetos levando em consideração critérios de ergonometria.

A fim de aprimorar o resultado das análises, os usuários têm exigido maior realismo gráfico em tempo real. Maior realismo gráfico significa aumento do número de primitivas do modelo. Além disso, efeitos de iluminação, como sombras e radiosidade, devem ser exibidos no ambiente virtual.

As placas gráficas atuais têm aumentado seu poder de processamento de forma exponencial, porém, o problema também tem se tornado mais complexto. Embora algumas placas já permitam ótimas otimizações no cálculo de sombras, nenhuma delas é capaz de processar dados de radiosidade ou sombras de muitas fontes de luz em tempo real. O número de primitivas agrava ainda mais o problema: a utilização de texturas e *shaders* (como *bump-mapping*, por exemplo), são necessidades típicas no modelo. Se as imagens não forem bem organizadas, um número muito grande de texturas pode resultar numa degradação de desempenho da cena.

Outro fator complicante é que o usuário ou projetista quer ver o resultado de uma mudança feita no projeto imediatamente, ou no menor tempo possível, ou seja, a etapa de pré-processamento do modelo deve ser curta.

Diversas são as técnicas de otimização existentes em computação gráfica a fim de evitar o envio de primitivas desnecessárias para a placa gráfica. A principal delas é a técnica de descarte por campo de visão (*view-frustum culling*): objetos que estejam fora do campo de visão não são enviados para a placa gráfica, pois os mesmos não contribuirão para a imagem final.

Outra otimização existente é o descarte por oclusão (*occlusion culling*). A idéia básica desses algoritmos é o descarte de objetos que estão ocultos por outros objetos da cena. Dentro dessa categoria de algoritmos, existe uma solução adequada para ambientes fechados: o algoritmo de célula/portal.

O algoritmo de célula/portal trabalha sobre um cena organizada em salas (células), onde cada célula possui um conjunto de objetos renderizáveis e uma lista de portais, que são portas, janelas ou outras conexões com células vizinhas. O algoritmo encontra a célula onde o observador se localiza e renderiza os objetos dessa célula usando descarte por campo de visão. Após isso, ele procura todos os portais visíveis dessa célula e renderiza suas células adjacentes recursivamente, reduzindo o tamanho do campo de visão à medida que atravessa cada portal.

Os primeiros algoritmos de célula/portal foram propostos por Airey [1], e Teller e Séquin [2]. As duas implementações exigem uma demorada etapa de pré-processamento e são restritos a ambientes 2-D ou 2.5-D (mapa de alturas). Luebke e Georges [3] inovaram ao criar um algoritmo rápido para o cálculo de oclusão dos portais.

A implementação de Luebke e Georges executa um rápido pré-processamento para montagem da estrutura de dados de portais. Essa estrutura é inserida em um grafo de cena. Durante o percorrimento do grafo, os portais, ao serem atravessados, diminuem o campo de visão. O teste de oclusão se reduz, portanto, a um simples teste de descarte com base no novo campo de visão do portal.

No presente trabalho foi feita a implementação do algoritmo de Luebke e Georges [3] estendendo a biblioteca de grafo de cena OpenSceneGraph [4]. O OpenSceneGraph (OSG) é uma biblioteca com código aberto, independente de plataforma, escrita na linguagem de programação C++ e sobre a biblioteca gráfica OpenGL[1]. Entre os seus usos, estão a simulação visual, a modelagem científica, realidade virtual e jogos. Para realização da navegação no ambiente virtual foi utilizada a biblioteca de realidade virtual VR Juggler [5, 6].

O trabalho está organizado da seguinte maneira: as ferramentas utilizadas serão descritas na seção 2. Na seção 3, será detalhada a implementação do algoritmo de portais no OpenSceneGraph. Na seção 4, serão mostrados os resultados obtidos seguidos da conclusão, na seção 5.

2 Ferramentas utilizadas

2.1 OpenSceneGraph

Um grafo de cena é uma estrutura de dados que permite a organização hierárquica de objetos que constituem uma cena. Por exemplo, quando se representa um veículo com quatro rodas, é desejável que a alteração da posição ou da orientação do carro seja transmitida para as rodas. Outra relação explorada no grafo de cena é a hierarquia de volumes envolventes, onde objetos próximos são agrupados para que, durante a etapa de descarte, sejam eliminados com apenas um teste feito no topo da hierarquia, evitando a necessidade de visitar cada um de seus filhos.

O OpenSceneGraph (OSG) é uma biblioteca que implementa várias funções de um grafo de cena: descarte por campo de visão, plano de oclusão (utilizando o algoritmo de *Shadow Frusta* [7]), descarte de pequenos objetos, suporte a níveis de detalhe discretos, ordenação por estado[2], suporte a diversos tipos de arquivos, *vertex arrays* e *display lists*, sendo os dois últimos otimizações específicas do OpenGL.

Nesse projeto, dois fatores foram fundamentais para a escolha da biblioteca OSG. De imediato,

[1] http://www.opengl.org

[2] Um estado define as características de um objeto: material, textura, iluminação, etc.

a navegação que funcionava diretamente sobre o OpenGL, no OSG teve um ganho de desempenho considerável. A extensibilidade da biblioteca permitiu que funcionalidades não existentes, como o sistema de portais, fossem implementadas sem que nenhuma linha de código fosse adicionada ou modificada no núcleo da biblioteca.

A favor do OSG também temos o fato dele ser independente de plataforma, ter código aberto e ser gratuito. Apesar da inexistência de publicações, vários projetos comerciais e acadêmicos já o utilizam. As bibliotecas de realidade virtual VR Juggler[3] e Vess[4], o motor de física da CM Labs[5] e a biblioteca de animação de personagens Cal3D[6] são alguns dos exemplos de contribuição da sua comunidade de usuários.

No OSG existem exemplos para demonstrar toda a funcionalidade da biblioteca, o que compensa a falta de documentação adequada. Essa pequena falha, junto à necessidade de conhecimentos avançados de orientação a objetos, torna o aprendizado demorado.

Estrutura e otimizações

O OpenSceneGraph divide a tarefa de renderização em três etapas: a etapa de atualização (UPDATE), de descarte (CULL) e a de desenho propriamente dita (DRAW). Na etapa de atualização são executadas, por exemplo, simulações ou atualizações da aplicação e do grafo de cena quando o mesmo tem objetos dinâmicos. Durante o descarte, os objetos que estão fora do campo de visão ou que pouco contribuam para a cena (ocupam menos que 1 *pixel*, por exemplo) são eliminados. Os objetos restantes formam uma lista de objetos renderizáveis.

A lista de objetos renderizáveis pode ser ordenada da maneira que o desenvolvedor definir. O OSG pré-define duas maneiras de ordenar essa lista. Quando os objetos são opacos, eles são ordenados por estado e colocados na frente da lista. Dessa forma, minimiza-se a troca de estado do OpenGL, já que objetos com o mesmo estado serão renderizados em sequência. Quando os objetos são transparentes, os mesmos são ordenados de trás para frente e colocados no final da lista de renderização. Assim, todos os objetos opacos são renderizados primeiro, seguidos dos objetos transparentes, evitando que o teste de *Z-Buffer* feito pela placa gráfica descarte objetos que estejam atrás dos objetos transparentes. O OSG fornece ainda um sistema de *callbacks* para o desenvolvedor implementar o seu próprio algoritmo de ordenação.

O OSG possui, ainda, uma classe que percorre o grafo de cena removendo os nós vazios, pré-calculando as transformações realizadas em objetos estáticos e agregando estados semelhantes, aumentando o desempenho da aplicação em termos de quadros por segundo.

Padrões de projeto

O OSG utiliza vários conceitos de orientação a objeto e padrões de projeto [8]. Entre os diversos padrões de projeto que o OSG implementa estão a cadeia de responsabilidade (*chain of responsability*), a composição (*composite*) e o visitador (*visitor*).

A cadeia de responsabilidade é empregada na criação de leitores dos arquivos de modelo e imagem que compõem a cena. Os leitores são criados como *plug-ins* dinâmicos. Essa técnica foi utilizada para

[3] http://www.vrjuggler.org
[4] http://vess.ist.ucf.edu
[5] http://www.cm-labs.com
[6] http://cal3d.sourceforge.net

criação do leitor para um novo tipo de arquivo utilizado neste trabalho, sem que houvesse necessidade de recompilar toda a biblioteca.

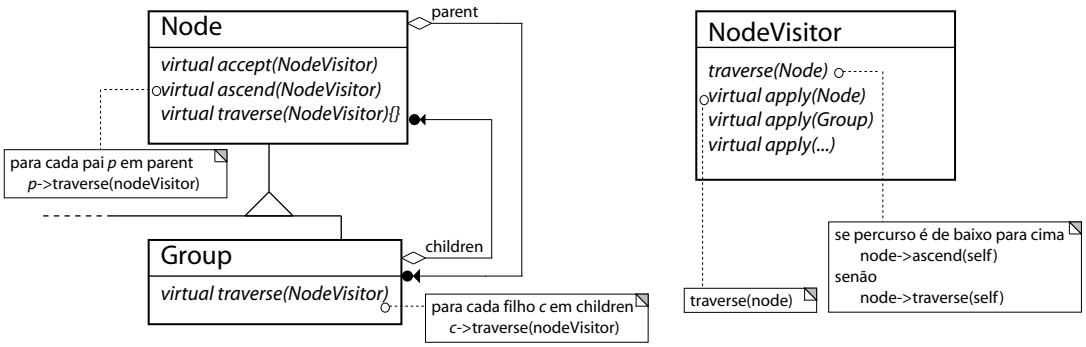

Figura 1: Classes do OpenSceneGraph: utilização de padrões de projeto

No OSG, qualquer percorrimento do grafo é feito através do padrão visitador. A figura 1 ilustra as classes abstratas que compõem este padrão. A classe *Node* é a classe raiz de todos os objetos que podem ser inseridos no grafo de cena. Dela herda a classe *Group*, empregando o padrão composição. Um grupo contém uma lista de nós filhos. E cada nó mantém sua lista de pais. No OSG é permitido que um nó tenha vários pais a fim de haver compartilhamento de nós em locais diferente do grafo.

A classe *NodeVisitor* é a classe mãe de todos os visitadores. Ela permite percorrer o grafo de cima para baixo, de baixo para cima ou em percurso redefinido pelo método virtual *traverse* das subclasses de *Node*. Alguns grafos de cena, como o Performer[7], somente permitem o percurso de cima para baixo ou vice-versa. Uma das vantagens do OSG é facilitar a redefinição do percurso que, apesar de não ser necessária para o algoritmo de célula/portal, facilita a sua implementação.

2.2 VR Juggler

O VR Juggler é uma biblioteca de realidade virtual, de código aberto, multi-plataforma e independente de dispositivo [5]. Ela fornece uma abstração dos dispositivos de entrada e saída. Junto a ela, qualquer biblioteca de grafo de cena (OpenGL Performer, OpenSceneGraph, OpenSG) pode ser usada para renderizar a cena.

O desenvolvedor não precisa se preocupar com o dispositivo de entrada que o usuário irá utilizar para interagir com a aplicação, ou como ele irá visualizá-la. A escolha dos dispositivos é feita através de arquivos de configuração. Baseado nesses arquivos, a biblioteca carrega os *drivers* adequados. A grande vantagem dos arquivos de configuração é permitir que diferentes dispositivos sejam combinados de várias maneiras sem necessidade de recompilar a aplicação. Por exemplo, o usuário pode configurar a aplicação para rodar com um joystick em uma CAVE ou, apenas mudando alguns parâmetros do arquivo, utilizar um mouse 3-D imerso em um capacete de realidade virtual.

A biblioteca VR Juggler foi escolhida devido a sua facilidade de extensão, dado que a mesma fornece seu código fonte. Durante o nosso trabalho, a biblioteca foi estendida para suportar alguns dispositivos extras, como o projetor de imagem estéreo por coluna da VRex [9], o mouse 3-D SpaceBall

[7] http://www.sgi.com/software/performer

4000 FLX [10] da Logitech, o sensor de movimento InterTrax^2 da Intersense [11] e dois joysticks padrão Microsoft Windows ™.

A fim de permitir a visualização estéreo por coluna, um pedaço do núcleo do VR Juggler foi alterado. Este tipo de estéreo é feito renderizando a imagem do olho esquerdo nas colunas ímpares da tela e a do olho direito nas colunas pares. A sua implementação utiliza os recursos de *stencil buffer* do OpenGL para alternar a renderização da imagem entre as colunas pares e ímpares. O projetor VRex reduz a resolução horizontal da imagem e se encarrega de alternar os quadros entre os olhos esquerdo e direito. Durante a renderização, o VR Juggler verifica o modo de visualização (olho esquerdo, olho direito, estéreo *quad-buffer* ou estéreo por coluna) e chama a função de desenho da aplicação.

Os *drivers* para os dispositivos de entrada citados foram criados através da interface de extensão do VR Juggler e integrados à biblioteca. A interface de extensão define uma série de métodos que devem ser implementados pelos *drivers* facilitando bastante o trabalho do desenvolvedor.

Os dispositivos de entrada podem ser digitais ou posicionais. Os digitais retornam verdadeiro ou falso caso um evento digital tenha ocorrido. Um exemplo de evento digital é o botão do joystick. Os dispositivos posicionais retornam uma matriz que representa a posição e a orientação do observador no mundo. Esses dados são utilizados pela classe que implementa a aplicação.

3 Portais

Os modelos arquitetônicos são compostos, de maneira geral, por salas, paredes, portas e janelas. Analisando a figura 2, percebe-se que muitos objetos da cena estão ocultos pelas paredes, ou seja, somente os objetos vistos através das portas deveriam ser enviados para a placa gráfica.

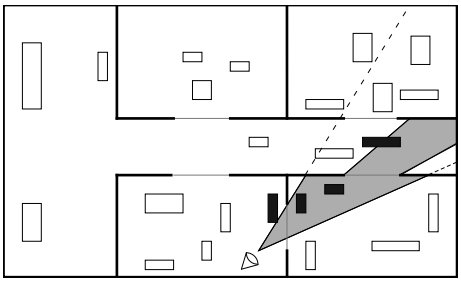

Figura 2: Modelo célula/portal: somentes os retângulos preenchidos são enviados para a placa gráfica

Ao dividir o modelo em células (salas) e portais (portas e janelas), é possível determinar, em cada quadro, quais os objetos visíveis ao observador. Como cada célula conecta-se a outra através de um portal, o conjunto potencialmente visível é aquele composto pela célula inicial, onde encontra-se o observador, e por cada célula adjacente, conectada por um portal.

Luebke e Georges [3] criaram um solução dinâmica que executa em tempo linear no número de células, para determinar quais estão visíveis em uma cena. Seu algoritmo é integrável a sistemas de grafos de cena existentes, como OpenGL Performer ou OpenSceneGraph. O único pré-processamento necessário é para a construção da estrutura de dados de cada célula (geralmente feito durante a leitura do arquivo do modelo).

Existem diversos motores de jogos que fazem uso de portais, porém são restritos a modelos criados em ferramentas desenvolvidas especificamente para eles. A maioria desses motores não possui suporte

a dispositivos de realidade virtual, como sensores de movimento, multi-projeção ou visão estereoscópica. Um dos objetivos desse trabalho é a incorporação do algoritmo de portais de Luebke ao grafo de cena da biblioteca OpenSceneGraph, que é capaz de ser integrada ao VR Juggler.

3.1 Integração com o OpenSceneGraph

Como foi mencionado anteriormente, o OpenSceneGraph percorre o grafo de cena e durante a etapa de CULL, descarta os objetos fora do campo de visão. A fim de estender esse processo de descarte, foram criadas três novas classes de objetos dentro do OpenSceneGraph: *Localizador*, *Célula* e *Portal*. A figura 3 mostra a organização dessas classes.

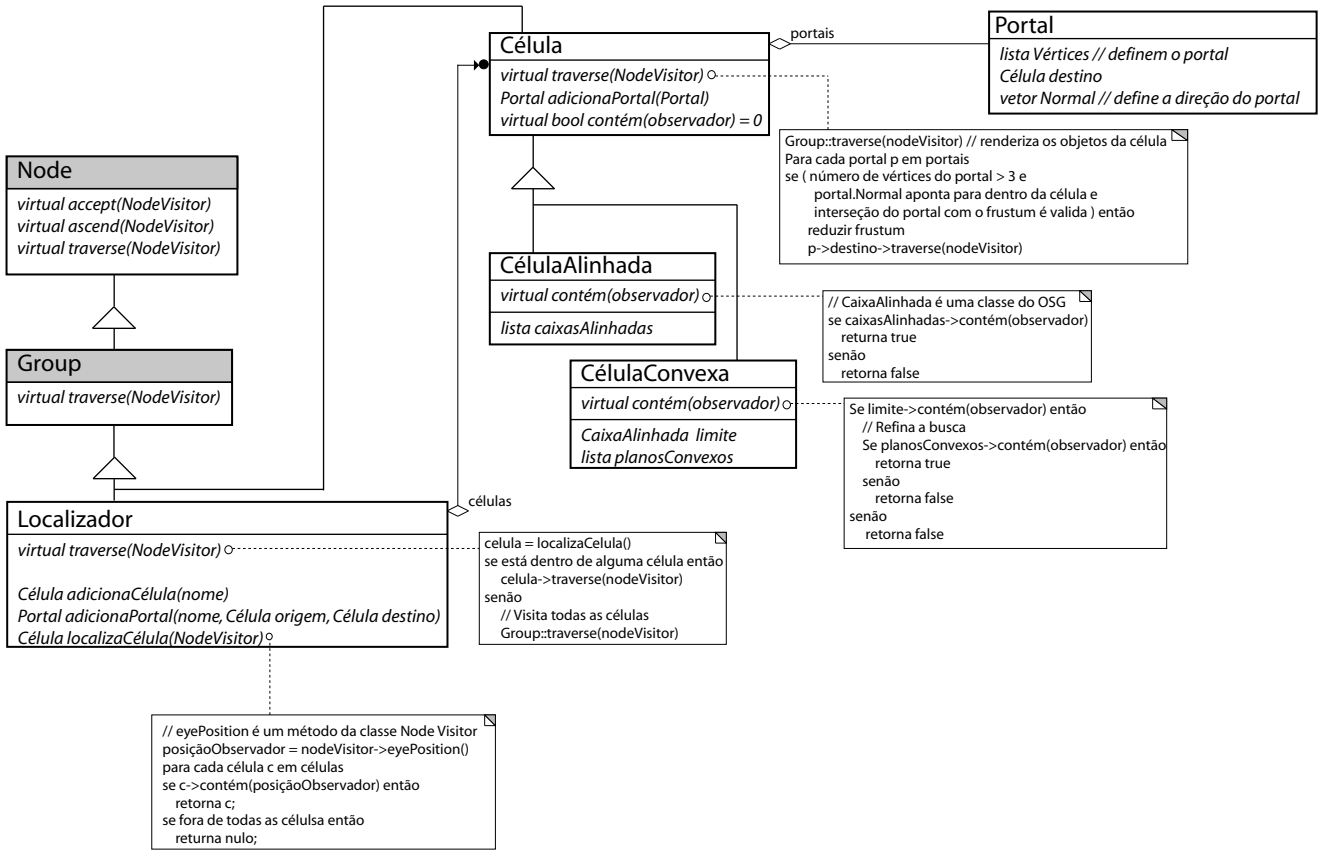

Figura 3: Organização das classes *Localizador*, *Célula* e *Portal* no OSG

A classe *Localizador* é responsável pela inserção de células e portais no grafo de cena. Ela sobrecarrega o método *traverse* de *Group*. Nesse método, a classe localiza a célula onde se encontra o observador utilizando o método *localizaCélula*. Se o observador estiver dentro de alguma célula, o visitador continua seu percurso a partir dela, senão ele deve visitar todas as células da cena, como se faria em uma cena tradicional, sem portais.

A classe *Célula* também herda de *Group* e sobrecarrega seu método *traverse*. Quando um visitador passa por uma célula, seus objetos são renderizados e o visitador continua seu percurso pelas célula adjacentes através da sua lista de portais. O método *contém* de célula é um método abstrato chamado pela

classe *Localizador* para teste de pertinência. Ele é implementado nas suas subclasses *CélulaAlinhada* e *CélulaConvexa*.

Na *CélulaAlinhada*, o método faz um teste de pertinência entre a posição do obsevador e a caixa alinhada com os eixos que define o volume da célula. Esse teste é muito rápido e pouco influencia no desempenho da etapa de CULL. Porém ele só pode ser aplicado se a célula for definida por caixas alinhadas com os eixos principais.

Caso o limite da célula não seja alinhado com os eixos, a classe *CélulaConvexa* deve ser utilizada. Essas células são limitadas por planos que definem um volume convexo. A classe ainda contém um atributo que é a caixa alinhada que envolve a célula (figura 4). Essa caixa alinhada permite que seja feito um teste inicial para detectar a possibilidade do observador não estar dentro do volume convexo. Se ele estiver fora dessa caixa envolvente, logo ele estará fora da célula. Se ele estiver dentro da caixa envolvente, devem ser feitos testes com todos os planos para saber se o observador está no interior do volume convexo da célula.

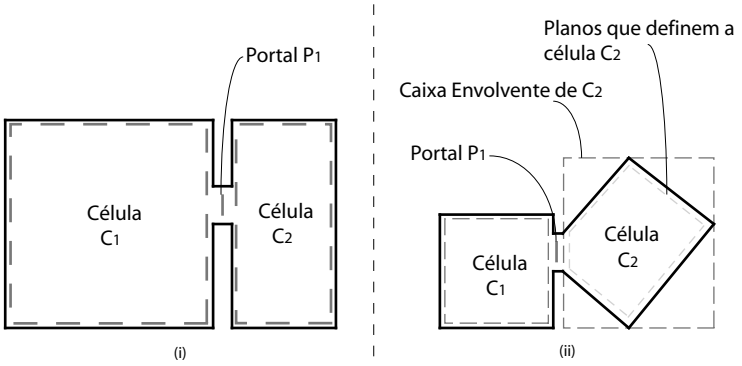

Figura 4: Visão plana das células: (i) os limites são alinhados com os eixos; (ii) os planos formam um volume convexo

Quando o visitador atravessa alguma célula, ela deve determinar por quais portais o visitador deve continuar. Para isso, o método *traverse* da classe *Célula* executa os seguintes passsos:

1. Caso a célula C não tenha sido visitada anteriormente, marcá-la como visitada
2. Renderizar os objetos de C usando descarte por campo de visão
3. Para cada portal P de C dentro do campo de visão, faça
   (a) Se P estiver de frente para o observador, faça
      i. Encontrar a interseção I do campo de visão com P
      ii. Percorrer a célula adjacente a P usando I como novo campo de visão
      iii. Recursivamente, voltar ao passo 1

A interseção do portal com o campo de visão é calculada da seguinte maneira: o portal é projetado na tela e o retângulo envolvente dessa projeção é calculado. A interseção é feita entre esse retângulo e a projeção do campo de visão atual, sendo o campo de visão inicial a própria tela. Esse procedimento faz com que o tamanho do campo de visão calculado seja um pouco maior do que o real, porém, isso não é problema, pois o que se deseja é apenas uma estimativa aproximada dos objetos visíveis.

A figura 5 mostra como os objetos das classes *Localizador*, *Célula* e *Portal* são inseridos dentro de um grafo de cena. O cenário externo é renderizado sem portais, enquanto o restante da cena, que está abaixo de Localizador, utilizará os recursos de portais.

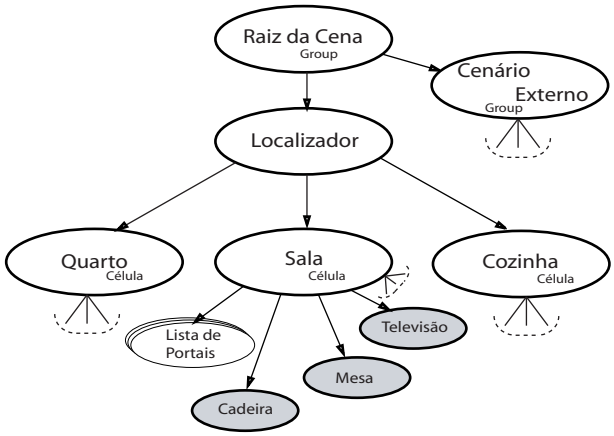

Figura 5: Objetos das classes *Localizador*, *Célula* e *Portal*

3.2 Exportação do modelo com portais

Os modelos arquitetônicos criados para esse trabalho foram feitos no 3D Studio Max 5.1. As definições de células e portais ficam codificadas no nome dos objetos criados no programa de edição, logo o projetista terá que seguir algumas convenções para definir as células e os portais. Os objetos contidos nas células devem estar ligados a elas como filhos. Essa codificação, usando apenas nomes e relações hierárquicas, permite que modelos com portais sejam lidos de outros formatos além dos exportados pelo 3D Studio.

Quando o modelo é lido pela aplicação, as informações contidas no nome dos objetos são decodificadas e as células e os portais são criados. Para isso, um visitador percorre todo o grafo de cena extraindo essa informação e retornando, ao fim, um objeto *Localizador*.

Uma próxima etapa do trabalho é o cálculo automático das células e portais a partir das informações contidas no modelo. A partir da estrutura de uma casa, por exemplo, será possível detectar as paredes, portas e janelas automaticamente e, com essa informação, montar a estrutura de dados para o algoritmo de célula/portal.

4 Resultados

O algoritmo de portais foi implementado em C++ utilizando a versão 0.9.3 do OpenSceneGraph. Os testes foram feitos com as seguintes configurações: resolução de tela de 1280x1024, amostragem de textura bilinear, sem filtragem anisotrópica, sem recursos de *anti-aliasing*, sincronismo vertical desabilitado e uma fonte de luz. O computador utilizado foi um Pentium 4, 2.53 GHz, com 3GB de memória equipado com uma placa de vídeo NVIDIA QuadroFX 1000 com 128MB.

O modelo utlizado para teste foi um apartamento com cerca de 1 milhão de faces. O apartamento foi dividido em 7 células, uma para cada cômodo mais a célula externa. Foram utilizados 16 portais,

sendo uma média de 2,28 portais por célula.

O desempenho de visualização foi medido em quadros por segundo (figura 6ii) em um percurso pré-definido (figura 6i). A tabela 1 apresenta em detalhes os valores para os pontos destacados do percurso.

Ponto	QPS sem portal	QPS com portal
1	27,56	44.33
2	18,15	36,42
3	60,10	158,62
4	48,23	66,93
5	41,71	122,05
6	125,98	162,40
7	55,12	59,11
8	246,31	250,25
9	24,83	106,30

Tabela 1: Resultados obtidos

No primeiro ponto destacado, o observador está voltado para a sala. No seu campo de visão, existe apenas um portal que conecta a sala a varanda, dessa forma o desempenho utilizando portal é bem maior. A sala possui vários objetos complexos e a varanda possui um vaso de planta bastante detalhado. Isso fez com que o desempenho, mesmo com portal, não passasse de 45 quadros por segundo (qps).

No segundo ponto, o observador acabou de retornar e, posicionado na varanda, olha para o interior do apartamento. Nesse ponto, temos o portal que liga a varanda à sala e o que liga a sala ao corredor. O portal do corredor leva ao quarto superior a direita. O desempenho é semelhante ao caso anterior devido aos objetos complexos da sala. Sem a utilização do portal, quase todo o interior da cozinha e do quarto é renderizado sem necessidade, reduzindo à metade o desempenho.

No terceiro ponto, o observador está no corredor voltado para os dois quartos na parte inferior da planta. Como o algoritmo reduz o campo de visão, a cama do quarto à esquerda não é renderizada. Por ser bastante complexa, ela influencia substancialmente no desempenho, saltando de 60 qps para 160 qps com o uso de portais.

No ponto quatro, o observador está no quarto mais à direita, ligeiramente voltado para a parede. O algoritmo de portais evita que objetos do quarto ao lado sejam renderizados. No quinto ponto, o observador está no corredor, voltado para o quarto de cima. Com o algoritmo de portais, as camas não são renderizadas e, pelo mesmo motivo do terceiro ponto, o desempenho passa de 40 qps para 120 qps.

No sexto ponto, após fazer o retorno no quarto de cima, o observador vira-se para a porta. Sem o algoritmo de portais, quase toda a sala foi renderizada, por isso o desempenho de 125 qps contra os 160 qps com o uso de portais.

Nos sétimo e oitavo pontos observa-se pouco ganho com a utilização de portais. O mesmo se deve ao fato do observador estar voltado para o interior da cozinha. Na direção de visualização não há nenhum portal que reduza o número de objetos da cena.

No nono ponto, o algoritmo de portais evita a renderização do conteúdo da sala, por isso o ganho de desempenho foi substancial, saltando de 25 qps para 106 qps.

Através dos resultados apresentados, comprovamos que houve ganho de desempenho com o algoritmo de portais para o modelo utilizado. Observamos que em alguns pontos, o desempenho com e sem

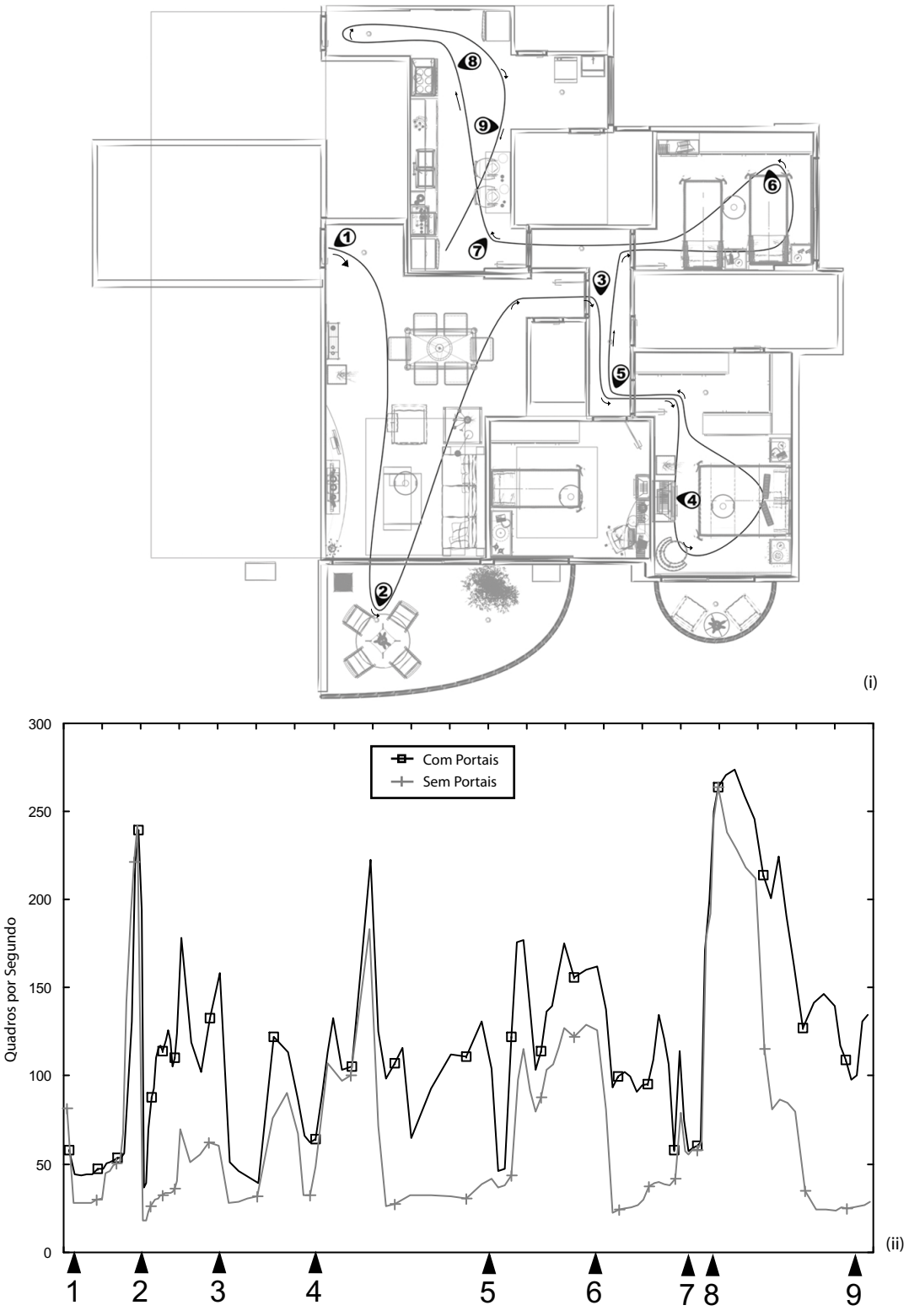

Figura 6: (i) Pontos principais do percurso e o (ii) Desempenho em quadros por segundo

portais foram equivalentes. Esse fato ocorre por dois motivos: objetos muito complexos localizados em uma única célula ou células que foram mal divididas.

As camas, as almofadas da sala, as frutas da cozinha e os vasos de planta das varandas eram objetos muito detalhados que prejudicavam o desempenho da cena, mesmo com portais. A sala e a cozinha poderiam ser subdivididas para formar células menores, cujos portais diminuiriam ainda mais o volume de visão do observador.

O processamento extra necessário para escolha dos portais e células a serem percorridos pouco influenciou no desempenho total da aplicação. Podemos observar esse fato, notando que o desempenho sem portais não ultrapassa, em nenhum momento, o desempenho com portais.

A figura 7 mostra a visão da sala. Vale ressaltar que o modelo testado é relativamente pequeno. Ganhos muito mais expressivos poderiam ser obtidos com um modelo mais complexo.

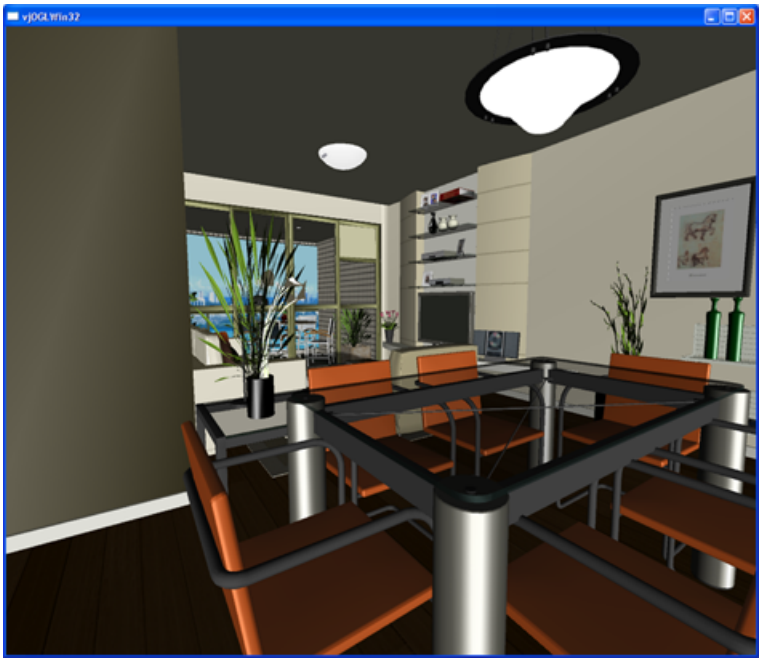

Figura 7: Visualização da sala do modelo do apartamento usado nos testes

5 Conclusão

Este trabalho mostrou uma implementação do algoritmo de portais [3] no OpenSceneGraph como ferramenta para navegação em ambientes virtuais arquitetônicos.

Como o OSG é uma biblioteca de código aberto, sendo utilizada por muitas ferramentas de RV, como o VR Juggler, a implementação deste trabalho é mais abrangente do que as de motores de jogos 3D. Essa característica permite que modelos sejam criados em ferramentas comerciais, como o 3DStudio Max, e visualizados de forma eficiente em tempo real. Além disso, a facilidade de extensão do OSG contribuiu para o rápido desenvolvimento do trabalho e sua integração com o VR Juggler.

O VR Juggler atendeu diversos requisitos do projeto, como a simplicidade da interface e o desempenho. Mesmo com essas características, a equipe de desenvolvimento encontrou alguns problemas,

dentre os quais, a dificuldade de implementação de algoritmos de detecção de colisão e uma comunicação limitada entre a aplicação e os dispositivos de entrada. Algumas informações de configuração dos dispositivos de entrada eram necessárias para a aplicação, mas inacessíveis pela *interface* do VR Juggler. Por esta razão, pretende-se, como trabalho futuro, resolver este problema através da substituição da ferramenta de abstração de dispositivos de realidade virtual.

Agradecimentos

O TeCGraf é um laboratório prioritariamente financiado pela Petrobras. Agradecemos a QUEM? pelo modelo do apartamento cedido para esse trabalho. Alberto Raposo é bolsista de produtividade em pesquisa do CNPq, processo n^o 305015/02-8.

Referências

[1] J. Airey, J. Rohlf, F. Brooks, *Towards image realism with interactive update rates in complex virtual building environments*, Symposium on Interactive 3D Graphics, 24(2):41-50, March 1990.

[2] S. Teller, C. Séquin, *Visibility preprocessing for interactive walkthroughs*, Computer Graphics (Proceedings of SIGGRAPH 91), 25(4):61-69, 1991.

[3] D. Luebke, C. Georges, *Portals and mirrors: Simple, fast evaluation of potentially visible sets*, ACM Interactive 3D Graphics Conference, Monterey, CA, 1995.

[4] R. Osfield, D. Burns, *OpenSceneGraph*, http://www.openscenegraph.org.

[5] A. Bierbaum, *VR Juggler: A Virtual Platform for Virtual Reality Application Development*, Master Thesis, Iowa State University, 2000.

[6] A. Bierbaum, C. Just, P. Hartling, C. Cruz-Neira, *Flexible Application Design Using VR Juggler*, SIGGRAPH, 2000.

[7] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, H. Zhang, *Accelerated Occlusion Culling using Shadow Frusta*, Symposium on Computational Geometry, pp. 1-10, 1997

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - elements of reusable object-oriented software*, Addison-Wesle Longman, Inc., 1995.

[9] VRex, Inc., *3D Stereoscopic imaging products and services*, http://www.vrex.com.

[10] 3Dconnexion, *A Logitech Company*, http://www.3dconnexion.com.

[11] InterSense, Inc., *The New Standard in Motion Tracking* http://www.intersense.com.