XithCluster: Uma Biblioteca de Grafo de Cena Distribuído

Pedro Boechat *

* UniverCidade (Centro Universitário da Cidade do Rio de Janeiro)
Núcleo de Projetos e Pesquisa em Aplicações Computacionais
Av. Rio Branco e Largo da Carioca s/nº, Rio de Janeiro, Brasil
e-mail: pboechat@gmail.com

Luciano Pereira Soares * ǂ

ǂ Tecgraf, DI - Departamento de Informática
PUC-Rio (Pontifícia Universidade Católica do Rio de Janeiro)
Rio de Janeiro, Brasil
e-mail: lpsoares@tecgraf.puc-rio.br

Resumo - Existe uma crescente demanda por recursos computacionais nos sistemas de realidade virtual e uma das abordagens a esta questão é através da distribuição do processo de renderização por múltiplos computadores. O presente trabalho propõe a implementação de um grafo de cena distribuído em aglomerados de computadores pessoais convencionais chamado de XithCluster, que é consoante com a tendência do mercado mundial de escalar poder computacional através de máquinas de baixo custo, como na computação nas nuvens. O XithCluster é uma solução de renderização distribuída desenvolvida em Java, sendo assim naturalmente multi-plataforma. Baseada no grafo de cena Xith3D, foi desenvolvido um protocolo de mensagens leves e diretas, apoiado sobre uma camada de comunicação de alta performance. Sua arquitetura implementa o método de ordenação Sort-Last, utilizando também os mecanismos de sincronismo Frame-Lock, Rate-Lock e Data-Lock.

Palavras-chave: renderização distribuída; composição de imagens; aglomerados de computadores

I. Introdução

Nos últimos anos é possível observar a popularização do uso de aglomerados de computadores convencionais (*Commodity Clusters*) em aplicações científicas, educacionais e até de entretenimento. Nos anos 90 com a popularização dos cartões gráficos 3D e o avanço das tecnologias de rede, tornou o uso dos *clusters* viável também para as aplicações de realidade virtual (RV). As vantagens do seu uso são diversas: custo reduzido, extensibilidade, escalabilidade, adesão à padrões da indústria e grande disponibilidade de aplicações.

Os sistemas de RV em tempo real possuem uma característica paradoxal: gerar imagens cada vez mais reais enquanto mantêm uma alta taxa de atualização. A busca pela melhoria da qualidade das imagens concorre inversamente com o tempo de resposta, pois ambos aumentam a demanda por recursos computacionais do sistema. Existem medidas conhecidas para se buscar o balanço entre esses dois objetivos: o aumento do poder computacional bruto e a otimização por software. Este trabalho utiliza ambas medidas, conjugando técnicas modernas de grafos de cenas otimizados e o aumento de recursos computacionais através de aglomerados de computadores. A implementação desenvolvida é uma solução simples e de baixo custo para a distribuição de processamento de sistemas de RV em tempo real inteiramente baseada em ferramentas de software livre.

Este artigo está organizado da seguinte forma: A primeira seção apresenta sua introdução e motivação. A seção 2 enumera os trabalhos relacionados considerados mais relevantes. A seção 3 apresenta um embasamento teórico necessário para a apresentação mais detalhada da solução. A seção 4 discorre sobre a arquitetura da implementação, explicitando as decisões mais importantes tomadas em sua elaboração. Por fim, as seções 5 e 6 expõem alguns resultados e considerações finais.

II. Trabalhos Relacionados

A. OpenSG

O OpenSG [3] é um grafo de cena de código aberto, escrito em C++ e desenvolvido com a biblioteca OpenGL. Oferecendo portabilidade, extensibilidade, suporte a *multi-threading* e renderização distribuída. Seu modelo de distribuição dita que haja apenas um cliente para múltiplos servidores. Isso ocorre pois seu conceito de servidor está diretamente ligado ao de nó renderizador, enquanto que o cliente é aquele que detém e divide os dados da cena. O OpenSG implementa dois dos métodos de distribuição: o Sort-First e o Sort-Last. O desenvolvimento de aplicações distribuídas com qualquer um destes métodos é transparente, pois toda a separação de primitivas, balanceamento de carga, transferência de imagens e composição são gerenciados pela ferramenta.

B. Chromium (WireGL)

O Chromium [4] é uma ferramenta que substitui dinamicamente as bibliotecas do OpenGL. Sendo totalmente não invasiva, ela pode transformar aplicações gráficas locais em aplicações distribuídas sem a necessidade de alterá-las. Isso é feito através de uma série de componentes, responsáveis não só pela interceptação das chamadas ao OpenGL, como também por processar fluxos de dado de uma maneira inovadora. Além disso, o Chromium também é parte de uma pilha de tecnologias utilizadas para dar maior suporte à paralelização de sistemas gráficos. Seu núcleo, chamado de *Mothership,* é bastante flexível, podendo ser configurado para realizar os métodos clássicos de distribuição de renderização, (Sort-First e Sort-Last), como também para mesclá-los em combinações exclusivas [5].

C. jReality

A jReality [6] é uma biblioteca utilizada na criação de aplicações científicas em Java. Concebida originalmente como uma ferramenta para visualização matemática, seu crescimento a fez tomar um caminho

mais abrangente. Seu principal diferencial é a capacidade de fazer aplicações de VR executarem tanto em estações de trabalho quanto em ambientes virtuais sem a necessidade de alteração de código. A jReality também dá suporte a diferentes bibliotecas de renderização, diversos dispositivos de entrada e saída e a algumas funcionalidades inéditas no âmbito das aplicações matemáticas, como o uso intermitente dos espaços euclidiano, hiperbólico e elíptico.

III. Renderização Distribuída

O problema principal da renderização paralela é a redistribuição ou ordenação das primitivas geométricas[1]. Os três principais métodos para resolução deste problema são: a redistribuição sendo realizada antes do processamento da geometria (*Sort-First*), a redistribuição entre o processamento da geometria e a rasterização (*Sort-Middle*) e a redistribuição após a rasterização (*Sort-Last*).

A. Sort-Last

Como o método de *Sort-First* é extremamente explorado na literatura, e o *Sort-Middle* depende de interrupções no pipeline gráfico ineficientes nos cartões gráficos convencionais, o método adotado por esta solução foi o *Sort-Last*. Neste método cada nó do aglomerado deve tanto processar a geometria quanto realizar a rasterização. Em uma etapa prévia ao início da renderização, as primitivas deverão ser divididas entre os nós seguindo um critério livre (Figura 1). A liberdade na definição deste critério constitui uma vantagem do *Sort-Last* sobre outros métodos, onde há a necessidade de se dividir as primitivas de acordo com regiões de tela.

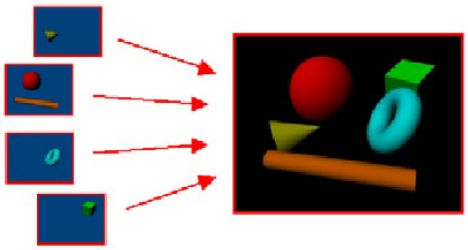

Figura 1. Redistribuição por Sort-Last [2]

Além disso, o *Sort-Last* é mais estável que os outros métodos, pois não precisa retransmitir as primitivas caso alguma delas cruze os limites de uma região de tela. Contudo, uma desvantagem sua é o elevado consumo de banda, dada a necessidade de envio constante das imagens geradas a uma próxima camada de software, onde serão combinadas para formar uma imagem final. Neste trabalho, abordaremos as técnicas desenvolvidas na otimização desse processo.

B. Composição de Imagens

A composição de imagens pode ser analisada por duas abordagens distintas [1]: a composição por prioridade fixa e a por prioridade de *pixels*.

Na abordagem da priorização por *pixels* o sistema atribui um valor de prioridade para cada *pixel* da imagem. Esse valor geralmente coincide com o valor de profundidade do *pixel* (*Z-buffer*). Com este nível granular de priorização é possível que imagens geradas de maneira independente se sobreponham (ou entrelacem), o que não seria possível em uma abordagem onde a priorização é dada por imagem (prioridade fixa).

Além disso, como o *Z-buffer* é um subproduto da renderização, ele pode ser utilizado sem grandes custos na composição da imagem final. Um algoritmo que itera pelas posições das matrizes de profundidade obtendo os menores valores para cada índice possui uma complexidade linear O(n), o que leva a um atraso tolerável geralmente inferior a um quadro [2].

Contudo, uma limitação desta abordagem é a sua imprecisão com geometrias transparentes. Isso ocorre devido a ordem em que as imagens são combinadas, o que pode influenciar no aspecto final das destas geometrias, como ilustrado na Figura 2.

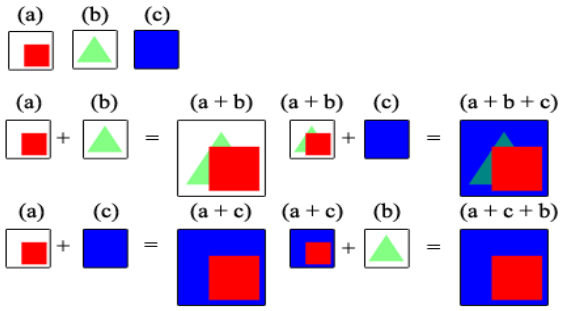

Figura 2. Ordem da composição influenciando imagem final

IV. Arquitetura do Sistema

A solução proposta possibilita a construção de aplicações de RV distribuídas de maneira rápida e simples. Esta pesquisa se apoiou sobre a ferramenta Xith3D [7], que é um grafo de cena construído em Java, de código aberto e com uma grande variedade de funcionalidades. Toda a camada de comunicação foi desenvolvida utilizando unicamente *sockets*.

O protótipo implementado se restringe a prover acesso aos elementos básicos para a construção de uma cena, como a criação de formas geométricas e o uso de texturas. Porém futuros trabalhos poderão facilmente complementar esta implementação, adicionando novas capacidades através dos pontos de extensão.

O sistema é compreendido por uma biblioteca de classes e duas aplicações independentes (rendererApp e composerApp). Toda a configuração da solução pode ser feita através de arquivos textos (*property files*) ou linha de comando.

A. Componentes de Arquitetura

A arquitetura do XithCluster apresenta três componentes: mestre (Master), renderizadores (Renderers) e compositor (Composer). A interação entre eles está representada na Figura 3.

O nó mestre é o componente que gerencia o aglomerado e os recursos da cena. Trata-se de uma aplicação usuária construída com a biblioteca de classes disponibilizada pelo XithCluster. Através dos métodos de sua API o desenvolvedor pode criar uma cena virtual, tratar as entradas do usuário e realizar algum processamento entre os ciclos da aplicação. No entanto, as tarefas essenciais para a distribuição do processo de renderização são realizadas transparentemente, como é o caso da divisão da geometria entre os renderizadores, da manutenção da coerência dos quadros produzidos (*Frame-Lock*) e da transmissão das atualizações do grafo de cena (*Data-Lock*).

Os nós renderizadores são os responsáveis pela geração das imagens da cena. Eles são processos da aplicação *rendererApp* instanciados, geralmente, em máquinas distintas. O conjunto de geometrias sob sua responsabilidade é determinado tão logo ingressam no aglomerado. A eles também cabe enviar as imagens produzidas ao nó compositor. Essa submissão pode ser antecedida opcionalmente pela compressão dos dados da imagem, como pelo uso do formato PNG [8].

O último componente, o nó compositor, é o responsável por montar e exibir a imagem final. Implementado através da aplicação *composerApp*, ele deve executar em uma máquina conectada a um dispositivo de exibição. O mecanismo de composição adotado utiliza, além do *z-buffer,* valores de ordenação preestabelecidos a fim de minimizar problemas com geometrias transparentes. O nó compositor também é responsável, juntamente com o nó mestre, por manter uma razão de atualização de quadros mínima (*Rate-Lock*). Um representação é apresentada na Figura 3.

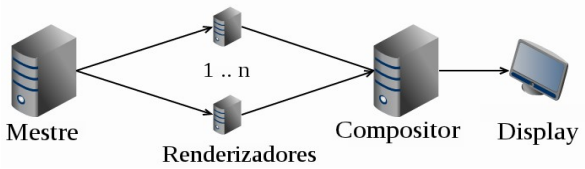

Figura 3. Modelo arquitetural simplificado

B. Serialização de Dados

O mecanismo de serialização desenvolvido é o principal responsável pela otimização da comunicação dos nós, pois através dele é possível enviar apenas as informações essenciais para a reconstrução de uma primitiva transmitida pela rede.

A serialização de dados é baseada nos mecanismos de manipulação de fluxo de dados do Java *DataInputStream* e *DataOutputStream,* dispondo dos métodos de leitura e escrita de tipos primitivos (inteiros, decimais, booleanos, etc.) de maneira independente de plataforma. Além de prover meios para a transmissão de alguns tipos de nó do grafo de cena, este mecanismo permite a leitura e escrita de estruturas matemáticas (vértices, matrizes, tuplas, transformações, etc.) e atributos de renderização (cores, texturas, atributos de renderização, etc.). Assim, a adição de suporte a novos tipos de nós do grafo de cena pode ser facilmente implementado.

C. Camada de Comunicação

A camada de comunicação proposta por esta solução é baseada na biblioteca xSockets [9]. Essa biblioteca utiliza as novas classes de E/S da plataforma Java para possibilitar operações de leitura e escrita de dados em alta performance. Além disso a camada de comunicação foi desenvolvida de maneira otimizada, transmitindo apenas dados binários e evitando a instanciação desnecessária de objetos (*Garbage Collection*).

A camada de comunicação está presente em todos os componentes de arquitetura da solução e em todos eles atua de forma transparente. Sua responsabilidade é intermediar a troca de mensagens entre os componentes da arquitetura, reforçando o uso do protocolo desenvolvido. Ela atua tanto sincronismo da troca de mensagens quanto no empacotamento dos dados enviados. A criação dos pacotes de dados é realizada por uma categoria de classes chamadas *Packagers* - classes que são fortemente acopladas ao mecanismo de serialização.

A camada de comunicação é concretizada pela classes *NetworkManagers*. Essas classes internas expõem interfaces públicas que representam as operações, em alto nível, realizadas pelo nós. Por conta de suas naturezas *multi-threaded*, já que devem se comunicar assíncronamente enquanto realizam suas tarefas principais, essas classes são construídas utilizando conceitos de *thread-safety* [10 e 11].

i. Protocolo de Comunicação

O protocolo de comunicação é a regra que rege a troca de mensagens dos componentes de arquitetura. Descreveremos essa regra em termos do conteúdo e da ordenação das mensagens a serem trocadas. É importante ressaltar que o protocolo de comunicação se situa no nível equivalente ao da camada de aplicação no modelo OSI [12]. Todas as mensagens trocadas são transmitidas em pacotes TCP/IP. A escolha pelo protocolo TCP para o transporte é justificado pelo uso de conexões *full duplex* confiáveis.

Apresentamos abaixo diagramas de troca de mensagens de dois possíveis cenários: a divisão de primitivas (Figura 4) e a renderização de um novo quadro (Figura 5). Os componentes de arquitetura foram separados em raias próprias e representados por retângulos com iniciais: M (Master), R (Renderer) e C (Composer). As setas direcionadas representam uma troca de mensagem. A seta pontilhada significa que a mensagem é não mandatória, mas pode ocorrer em algumas circunstâncias. Todas as setas foram enumeradas de acordo com a ordem em que as mensagens devem ser trocadas para a execução correta da cena.

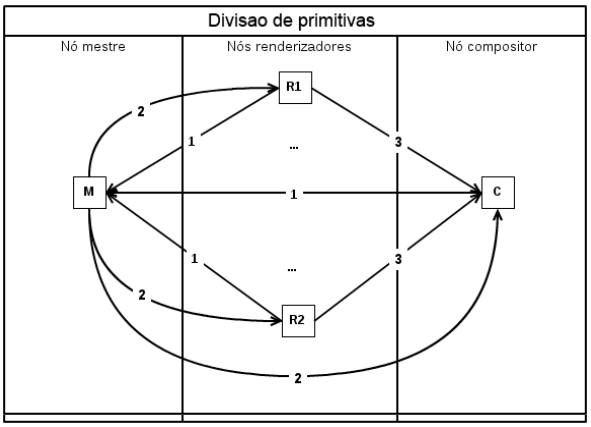

Figura 4. Troca de mensagens para distribuição de primitivas

Nº	Descrição	Emissor	Rcpt.	Conteúdo
1	Solicitação de ingresso no aglomerado	R ou C	M	Flag SYN do protocolo TCP
2	Divisão das primitivas e ajustes de aplicação	M	R e C	Dimensões de tela, razão de quadros alvo, ponto de vista, fontes de luz e geometrias da cena
3	Ordem de composição	R	C	Prioridade da imagem no processo de composição

Tabela 1. Legenda das mensagens da divisão de primitivas

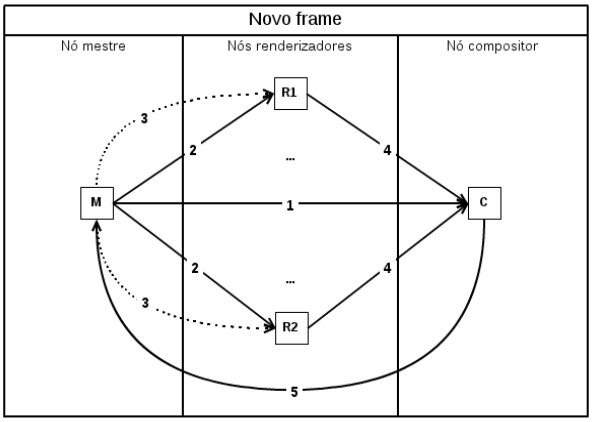

Figura 5. Troca de mensagens para renderização de *frame*

Nº	Descrição	Emissor	Rcpt.	Conteúdo
1	Quantidade quadros a descartar	M	C	Quantidade de quadros
2	Sinal de novo quadro	M	R e C	Número do quadro corrente
3	Atualizações de cena (opcional)	M	R	Descrição das últimas alterações na cena pertinentes ao nó
4	Imagem rasterizada	R	C	*Buffers* de cor, transparência, profundidade e método de compressão (opcional)
5	Sinal de fim de quadro	C	M	Número do quadro corrente

Tabela 2. Legenda das mensagens da renderização de *frame*

V. Resultados

Os resultados apresentados traçam uma relação entre o aumento da quantidade de polígonos e a performance da solução desenvolvida. A ferramenta utilizada para a coleta de dados se chama jvisualvm[13] e foi utilizada no monitoramento de diversos parâmetros da máquina virtual java (Figura 6). Ela é de uso gratuito e vem embutida juntamente com o kit de desenvolvimento java da Oracle.

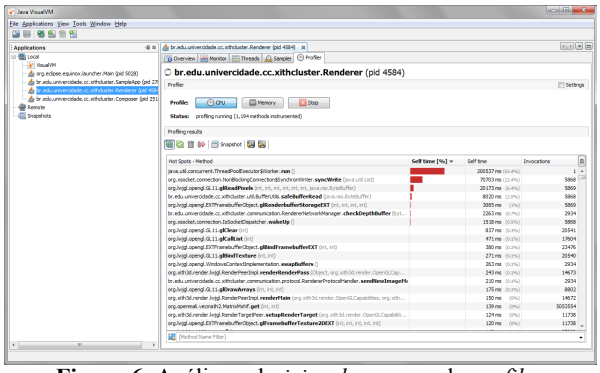

Figura 6: Análise pelo *jvisualvm* no modo *profiler*

Para a implementação desta avaliação foram necessárias três aplicações de teste. As aplicações visam representar diferentes cenários de dados: um com cem, outro com mil e outro dez mil (Figura 7) polígonos.

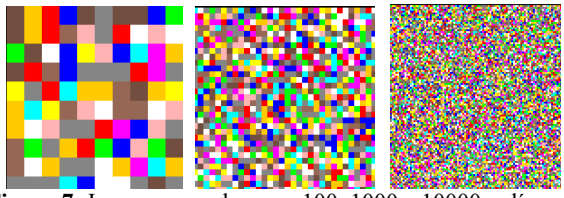

Figura 7: Imagens geradas com 100, 1000 e 10000 polígonos renderizados simultaneamente

Os gráficos abaixo nas Figuras 8, 9 e 10 apresentam o tempo das operações mais importantes em cada um dos cenários anteriores. Os testes procedidos tiveram como parâmetro de equidade a geração de de mil quadros. Os testes foram realizados em uma única máquina e com apenas uma instância de renderizador executando por

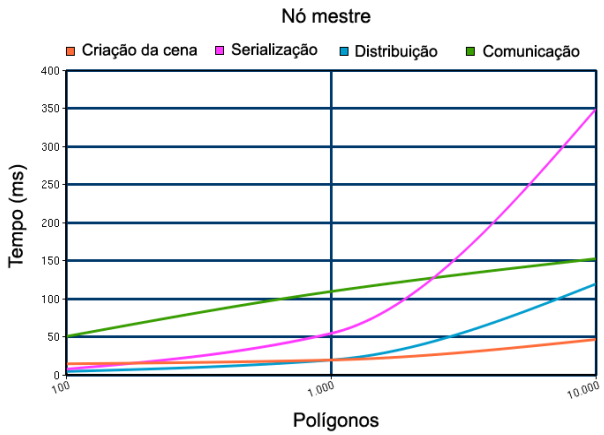

Figura 8: Forte relação entre a operação de serialização e o número de polígonos da cena

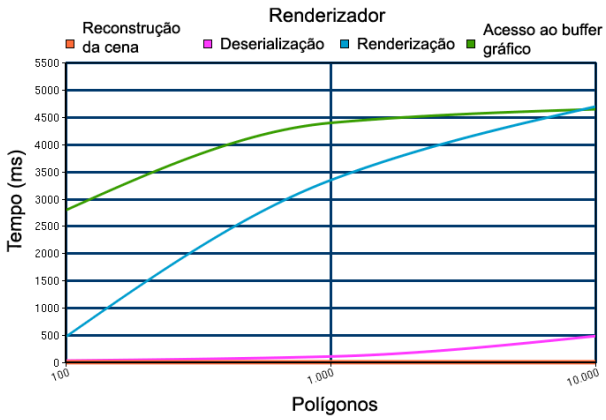

Figura 9: Dependência observada entre a deserialização e o número de polígonos é baixa

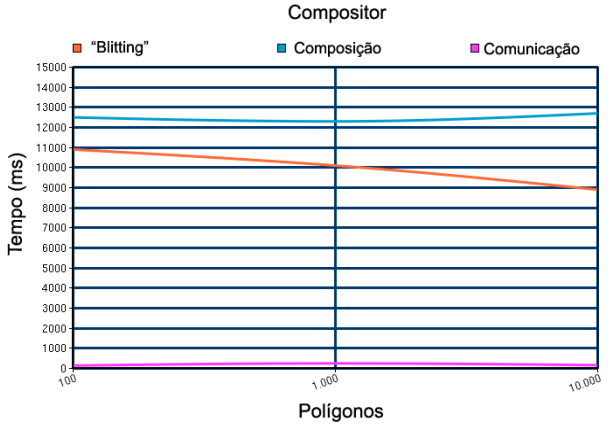

Figura 10: Compositor aponta a inexistência de uma relação entre performance e o número de polígonos

VI.Conclusões e Trabalhos Futuros

O XithCluster provê meios para a construção de aplicações de RV distribuídas em Java sendo então naturalmente multiplataforma (Figura 11) Ela disponibiliza um modelo simples e extensível, facilitando a prototipação rápida de sistemas de renderização distribuída, como através da criação de provas de conceito. Por ser escrita em uma linguagem interpretada, tem a vantagem de poder ser utilizada em diversos domínios porém pode ter algumas perdas de desempenho que foram contornadas.

Os primeiros testes apresentaram resultados mais limitados devida a todo o processamento ser realizado em uma só CPU e GPU. Como trabalho futuro se deseja utilizar sistemas de aglomerados de computadores convencionais com todas as ferramentas de integração [14] para verificar quais são os ganhos e perdas com a distribuição.

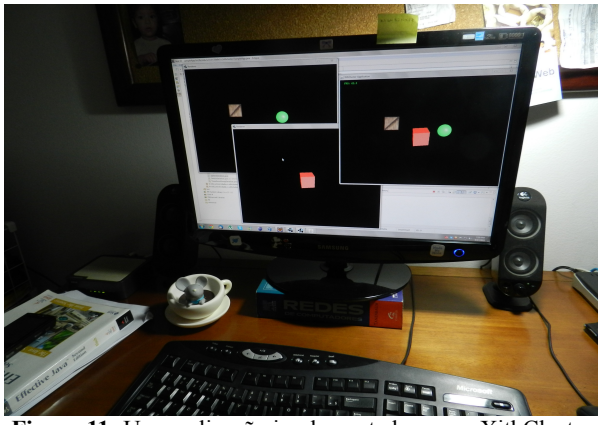

Figura 11: Uma aplicação implementada com o XithCluster executando na plataforma *Windows*

Referências

[1] MOLNAR, S., 1991, *Image-Composition Architectures for Real-Time Image Generation*. Tese de D.Sc., Universidade da Carolina do Norte, EUA.

[2] SOARES, L. P., 2005, *Um Ambiente de Multi projeção Totalmente Imersivo baseado em Aglomerados de Computadores*. Tese de D.Sc., Escola Politécnica, Universidade de São Paulo, São Paulo.

[3] OpenSG Official Website. Disponível em: <http://www.opensg.org>. Acesso em 13 jun. 2011

[4] Chromium Homepage. Disponível em: <http://chromium.sourceforge.net>. Acesso em: 13 jun. 2011.

[5] HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P., *Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters*. Universidade da Califórnia, EUA.

[6] BRINKMANN, P. GUNN, C. WEIBMANN, S., 2010, *jReality: Interactive Audiovisual Applications Across Virtual Environments*.

[7] Xith3D Homepage. Disponível em: <http://xith.org>. Acesso em: 13 de jun. 2011.

[8] Portable Network Graphics (PNG) Specification. Disponível em: <http://www.w3.org/Graphics/PNG/>. Acesso em: 13 de jun. 2011.

[9] xSocket Project Homepage. Disponível em: <http://sourceforge.net/projects/xsocket/>. Acesso em: 13 jun. 2011.

[10] COLOURIS, G. DOLLIMORE, J. KINDBERG, T., 2007, *Sistemas Distribuídos*: Conceitos e Projeto. 4 ed. Bookman.

[11] The Java Tutorials: Concurrency. Disponível em: <http://download.oracle.com/javase/tutorial/essential/concurrency/index.html>. Acesso em: 13 jun. 2011.

[12] TANENBAUM, A., 2004, *Redes de Computadores*. 4 ed. Campus.

[13] Java VisualVM Documentation. Disponível em: <http://download.oracle.com/javase/6/docs/technotes/guides/visualvm/index.html>. Acesso em: 13 jun. 2011.

[14] SOARES, L. P., CABRAL, M. C., BRESSAN, P. A., LOPES, R. D. and ZUFFO, M. K. 2002 . Powering Multiprojection Immersive Environments with Cluster of Commodity Computers . In *Proceding of The 1st Ibero-American Symposium in Computer Graphics* (Guimarães , Portugal , Julho, 2002).