



**CENTRO UNIVERSITÁRIO DA CIDADE DO RIO DE JANEIRO  
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO  
NÚCLEO DE PROJETOS E PESQUISAS EM APLICAÇÕES  
COMPUTACIONAIS**

**Grafo de Cena Distribuído em Aglomerados  
Computacionais De Baixo Custo**

**Pedro Boechat de Almeida Germano**

**RIO DE JANEIRO  
Dezembro de 2011**

**Grafo de Cena Distribuído em Aglomerados Computacionais De Baixo  
Custo**

**Trabalho de Graduação apresentado à  
disciplina de Projeto Final II, do Curso de  
Bacharelado em Ciência da Computação do  
Centro Universitário da Cidade.**

**Professor Orientador: Luciano Pereira Soares**

**Composição da Banca Examinadora:**

**Prof.<sup>a</sup> Rachel de Camargo, M.Sc.**  
Centro Universitário da Cidade

**Prof. Rafael Castaneda, M.Sc.**  
Centro Universitário da Cidade

RIO DE JANEIRO  
**Dezembro de 2011**

Resumo do projeto apresentado ao Centro Universitário da Cidade do Rio de Janeiro  
como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência  
da Computação

## **Grafo de Cena Distribuído em Aglomerados Computacionais De Baixo Custo**

Pedro Boechat de Almeida Germano

Dezembro / 2011

**Orientador: Prof. Luciano Pereira Soares, D.Sc.**

Este trabalho propõe a implementação de um grafo de cena distribuído em aglomerados de computadores pessoais (*commodity clusters*). Ele atende à questão da crescente demanda por poder computacional dos sistemas de realidade virtual através da distribuição do processo de renderização por múltiplos computadores. A escolha do uso de aglomerados de computadores convencionais é consoante com uma tendência do mercado mundial, como por exemplo a computação em nuvens (*cloud computing*), de escalar poder computacional através de máquinas de baixo custo.

Abstract of Dissertation presented to Centro Universitário da Cidade do Rio de Janeiro  
as a partial fulfillment of the requirements for the degree of Bachelor in Computer  
Science

## **Distributed Scene Graph in a Low-Cost Computer Cluster**

Pedro Boechat de Almeida Germano

December / 2011

**Advisor: Prof. Luciano Pereira Soares, D.Sc.**

This work proposes the development of a distributed scene graph that works in clusters of personal computers (commodity clusters). It answers the question of the increasing demand for computational power of real-time virtual reality systems by distributing the rendering process across multiple computers. The choice for commodity clusters is consonant to a global tendency – for instance, the cloud computing - of scaling computational power with low-cost machines.

Dedico este trabalho à Deus,  
infinitamente misericordioso,  
e à minha família.

## **Agradecimentos**

Ao meu orientador, Luciano Pereira Soares, pelo seu apoio e amizade. Sua participação e experiência foram fundamentais para a conclusão deste trabalho. Não poderia deixar de mencionar nossas conversas, através das quais pude conhecê-lo melhor e aumentar minha estima pela sua pessoa.

Aos ex-colegas de trabalho: Flávio Faria de Paula, pelas dicas de redação, Leo Moreira Soares, pelas conversas sobre vida acadêmica, e Jorge Alcino Neto, pelo incentivo ao comparecer às minhas apresentações.

Por último, mas não menos importante, agradeço imensamente à minha namorada, Priscilla Martins de Araújo, por seu sacrifício e compreensão durante os fins de semana e noites de estudo.

# Índice

<b>1 Introdução .....</b>	<b>12</b>
<b>1.1. Posicionamento e Justificativa.....</b>	<b>12</b>
<b>1.2. Objetivos .....</b>	<b>13</b>
<b>1.3. Organização do Trabalho .....</b>	<b>13</b>
<b>2. Fundamentos Teóricos .....</b>	<b>14</b>
<b>2.1. Grafos de Cena.....</b>	<b>14</b>
2.1.1. View Frustum Culling.....	17
2.1.2. Occlusion Culling.....	18
<b>2.2. Renderização Distribuída .....</b>	<b>19</b>
2.2.1. Sort-First .....	20
2.2.2. Sort-Middle .....	22
2.2.3. Sort-Last.....	23
<b>2.3. Sincronismo .....</b>	<b>25</b>
2.3.1. Frame-Lock .....	26
2.3.2. Rate- Lock .....	28
2.3.3. Data-Lock.....	28
<b>2.4. Z-Buffering.....</b>	<b>29</b>
<b>2.5. Composição de Imagens .....</b>	<b>32</b>
<b>3. Trabalhos Relacionados .....</b>	<b>35</b>
<b>3.1. OpenSG.....</b>	<b>35</b>
<b>3.2. Chromium (WireGL) .....</b>	<b>37</b>
<b>3.3. Syzygy .....</b>	<b>40</b>
<b>3.4. FlowVR .....</b>	<b>42</b>
<b>3.5. jReality .....</b>	<b>44</b>
<b>3.6. Comparativo.....</b>	<b>47</b>
<b>4. Solução Proposta (XithCluster).....</b>	<b>48</b>
<b>4.1. Introdução .....</b>	<b>48</b>
<b>4.2. Componentes de Arquitetura .....</b>	<b>48</b>
<b>4.3. Sessão de Colaboração .....</b>	<b>50</b>
<b>4.4. Hierarquia de Nós do Grafo de Cena .....</b>	<b>51</b>
<b>4.5. Serialização de Dados .....</b>	<b>54</b>
<b>4.6. Camada de Comunicação.....</b>	<b>55</b>

4.6.1. Protocolo de Comunicação .....	57
<b>4.7. Extensão ao Xith3D .....</b>	<b>60</b>
4.7.1. Distribuição da Renderização .....	60
4.7.2. Atualização de Estado dos Nós do Grafo de Cena.....	61
4.7.3. Distribuição da Geometria e Replicação dos Nós do Grafo de Cena	
63	
<b>4.8. Aplicação Renderizadora.....</b>	<b>65</b>
<b>4.9. Aplicação Compositora .....</b>	<b>67</b>
<b>4.10. Dependências de Plataforma .....</b>	<b>69</b>
<b>5. Experimentos e Resultados.....</b>	<b>70</b>
<b>5.1. Métricas do teste .....</b>	<b>70</b>
<b>5.2. Coleta de dados .....</b>	<b>71</b>
<b>5.3. Análise dos resultados .....</b>	<b>72</b>
5.3.1. Mestre.....	72
5.3.2. Renderizador .....	73
5.3.3. Compositor.....	74
<b>6. Conclusões .....</b>	<b>76</b>
<b>6.1. Contribuições .....</b>	<b>77</b>
<b>6.2. Trabalhos Futuros .....</b>	<b>77</b>
<b>Apêndice A .....</b>	<b>78</b>
<b>Apêndice B .....</b>	<b>85</b>
<b>Apêndice C .....</b>	<b>114</b>
<b>Referências .....</b>	<b>122</b>

# **Lista de Figuras**

<b>Figura 2.1.</b> Uma estrutura de dados do tipo "árvore".....	13
<b>Figura 2.2.</b> Nós representando o relacionamento todo-parte .....	14
<b>Figura 2.3.</b> Uma cena com os limites volumétricos hierárquicos visíveis.....	15
<b>Figura 2.4.</b> O view frustum culling consiste no descarte de objetos que estão completamente fora do sólido de visualização (ECKEL, 1998).....	17
<b>Figura 2.5.</b> O occlusion culling consiste no descarte dos objetos que estão ocultos por estarem atrás de objetos mais próximos do visualizador (ECKEL, 1998).....	18
<b>Figura 2.6.</b> Imagem adaptada do modelo canônico.....	19
<b>Figura 2.7.</b> O esquema de trabalho no método Sort-First .....	20
<b>Figura 2.8.</b> No Sort-First as primitivas são distribuídas por região de tela.....	20
<b>Figura 2.9.</b> O esquema de trabalho no método Sort-Middle .....	22
<b>Figura 2.10.</b> Modelo de divisão de tela entre rasterizadores:	
(a) Por fragmentos contíguos	
(b) Por fragmentos entrelaçados .....	23
<b>Figura 2.11.</b> O esquema de trabalho do método Sort-Last.....	24
<b>Figura 2.12.</b> No Sort-Last as imagens transmitidas para a camada compositora possuem o tamanho da tela toda .....	24
<b>Figura 2.13.</b> Os sistemas de RV distribuídos precisam manter seus dados consistentes, caso contrário podem sofrer de falta de coerência visual (SOARES, 2005) .....	26
<b>Figura 2.14.</b> A condição de corrida é um problema clássico de alteração de dados em paralelo .....	26
<b>Figura 2.15.</b> O swap-buffer muda a exibição entre o buffer frontal e um dos buffers traseiros (Baseado em (ECKEL, 1998)).....	27
<b>Figura 2.16.</b> O frame-lock garante que todos as partes de um sistema distribuído de RV terminem o novo quadro antes que ele seja exibido	
<b>Figura 2.17</b> A renderização das primitivas é realizada na ordem correta através do algoritmo do pintor (Baseado em (CONCI)).....	29
<b>Figura 2.18.</b> As distâncias dos pixels no z-buffer, após o desenho de três primitivas (Baseado em (CONCI)) .....	30
<b>Figura 2.19.</b> Nos z-buffers de 8 ou 16-bits, partes das primitivas podem se sobrepor indesejavelmente quando estão muito próximas (Z-Fighting) .....	30
<b>Figura 2.20.</b> Círculo transparente desenhado sobre triangulo azul com e sem o z-buffer habilitado .....	31
<b>Figura 2.21.</b> A imagem de fundo e a imagem frontal, com cor chave azul, resultando na imagem final a direita .....	33

<b>Figura 2.22. Na composição de imagens por prioridade fixa não é possível haver enlace de figuras .....</b>	<b>33</b>
<b>Figura 2.23. Ao compor imagens usando o z-buffer as imagens que contém transparência devem ser compostas por último .....</b>	<b>34</b>
<b>Figura 3.1. Mothership configurada para realizar o Sort-First.....</b>	<b>40</b>
<b>Figura 3.2. Mothership configurada para realizar o Sort-Last .....</b>	<b>40</b>
<b>Figura 3.3. Configuração que combina características de Sort-First com Sort-Last de maneira única.....</b>	<b>40</b>
<b>Figura 4.1. Os componentes de arquitetura do XithCluster .....</b>	<b>50</b>
<b>Figura 4.2. O XithCluster fica em estado ocioso quando não há pelo menos um componente de cada tipo rodando no aglomerado .....</b>	<b>51</b>
<b>Figura 4.3. O XithCluster permite que um renderizador entre no aglomerado durante a renderização distribuída .....</b>	<b>51</b>
<b>Figura 4.4. A hierarquia completa de nós folha do Xith3D, com os nós suportados pelo XithCluster pintados de verde.....</b>	<b>56</b>
<b>Figura 4.5. A hierarquia completa de nós grupo do Xith3D, com os nós suportados pelo XithCluster pintados de verde.....</b>	<b>57</b>
<b>Figura 4.6. A leitura transacional determina que uma mensagem só seja processada caso todos os seus campos sejam lidos .....</b>	<b>60</b>
<b>Figura 4.7. O diagrama ilustra a troca de mensagens durante a criação de uma nova sessão.....</b>	<b>62</b>
<b>Figura 4.8. O diagrama ilustra a troca de mensagens durante a sinalização de um novo quadro .....</b>	<b>63</b>
<b>Figura 4.9. A estratégia padrão de distribuição garante que todas as geometrias da cena sejam distribuídas pelos renderizadores .....</b>	<b>66</b>
<b>Figura 4.10. Cada geometria tem seu caminho replicado e enviado para um renderizador.....</b>	<b>67</b>
<b>Figura 4.11. Composição com erro pois a ordem de composição do renderizador com a primitiva transparente não foi a última .....</b>	<b>70</b>
<b>Figura 4.12. Composição sem erro pois a ordem de composição do renderizador com a primitiva transparente foi usada a última.....</b>	<b>70</b>
<b>Figura 5.1. A coleta de dados feita através da ferramenta JVisualVM.....</b>	<b>73</b>
<b>Figura 5.2. Tempo médio de execução das operações do componente Mestre .....</b>	<b>75</b>
<b>Figura 5.3. Tempo médio de execução das operações do componente Renderizador.</b>	<b>76</b>
<b>Figura 5.4. Tempo médio de execução das operações do componente Compositor ....</b>	<b>77</b>
<b>Figura I - Diagrama de classes geral.....</b>	<b>80</b>
<b>Figura II – Diagrama de classes do subsistema de composição .....</b>	<b>81</b>
<b>Figura III – Diagrama de classes do subsistema de serialização.....</b>	<b>82</b>

<b>Figura IV – Diagrama de classes do subsistema de mensagens .....</b>	<b>83</b>
<b>Figura V - Diagrama de atividades da criação de uma nova sessão .....</b>	<b>84</b>
<b>Figura VI - Diagrama de atividades do envio de atualizações .....</b>	<b>85</b>
<b>Figura VII - Diagrama de atividades da estratégia de distribuição simples.....</b>	<b>86</b>

## **Lista de Tabelas**

**Tabela 3.1. Comparativo de funcionalidades entre as diferentes soluções analisadas** 51

**Tabela 4.1. Descrição das mensagens enumeradas no diagrama de nova sessão .....** 65

**Tabela 4.2. Descrição das mensagens enumeradas no diagrama de novo quadro .....** 66

**Tabela 4.3. Parâmetros de configuração da aplicação rendererApp.....** 67

**Tabela 4.4. Parâmetros de configuração da aplicação composerApp .....** 69

## **Lista de Abreviaturas**

API *Application Programming Interface* (Interface de Programação de Aplicativos)

JAR *Java Archive* (Arquivo Java)

JDK *Java Development Kit* (Kit de Desenvolvimento Java)

JRE *Java Runtime Environment* (Ambiente de Execução Java)

LOD *Level Of Detail* (Nível de Detalhe)

RPC *Remote Procedure Call* (Chamada a Procedimento Remoto)

RV Realidade Virtual

SO Sistema Operacional

UML *Unified Modeling Language* (Linguagem de Modelagem Unificada)

VE *Virtual Environment* (Ambiente Virtual)

# 1 Introdução

Foi possível observar, nos últimos anos, a popularização do uso de aglomerados de computadores pessoais (*commodity clusters*) em aplicações científicas, educacionais e de entretenimento. No final da década de 90, com o surgimento dos cartões gráficos pessoais e o avanço das tecnologias de rede, os *commodity clusters* também se tornaram uma opção para as aplicações de realidade virtual (RV). As vantagens do seu uso são muitas: custo reduzido, extensibilidade, escalabilidade, adesão à padrões da indústria e grande disponibilidade de softwares gratuitos.

## 1.1. Posicionamento e Justificativa

Todos os sistemas de RV em tempo real estão em um dilema: aumentar o realismo das imagens ou manter a taxa de atualização em níveis aceitáveis. Isso acontece pois a busca pelo realismo das imagens concorre com a velocidade de atualização, já que ambos aumentam a demanda por recursos do computador.

Este trabalho propõe uma arquitetura de sistema de RV que usa grafo de cena para reduzir a demanda por recursos do computador, enquanto aumenta a disponibilidade de recursos através da distribuição por aglomerado. Nele é desenvolvida uma solução simples e de baixo custo para o aumento do desempenho em sistemas de RV em tempo real.

## **1.2. Objetivos**

A solução desenvolvida neste trabalho serve para construir sistemas de RV que requeiram alto desempenho e possuam limitações orçamentárias. Este trabalho aplica técnicas avançadas de programação em computação gráfica e computação distribuída para que seja possível renderizar de imagens mais realistas mantendo a taxa de atualização em níveis aceitáveis.

## **1.3. Organização do Trabalho**

Este trabalho é composto de 6 Seções e 3 Apêndices. A Seção 1 introduz o leitor ao problema e aos objetivos do trabalho. A Seção 2 apresenta os fundamentos teóricos mais relevantes estudados durante a revisão da literatura especializada. A Seção 3 analisa algumas ferramentas similares e as compara com a implementação proposta. A Seção 4 delineia essa implementação, detalhando seus pontos mais importantes. A Seção 5 analisa os resultados das avaliações experimentais feitas sobre a implementação. Por fim, a Seção 6 conclui este trabalho, apresentando suas contribuições e apontando caminhos para sua continuação. O Apêndice A apresenta 7 diagramas UML da implementação desenvolvida. O Apêndice B apresenta o seu manual de uso. Por fim, o Apêndice C apresenta uma publicação realizada a partir deste trabalho.

## 2. Fundamentos Teóricos

### 2.1. Grafos de Cena

Grafos de cena são estruturas de dados usadas para relacionar os objetos de uma cena virtual. Eles geralmente são representados por grafos acíclicos completamente conectados também conhecidos como árvores. Conforme ilustrado na Figura 2.1, as árvores são compostas por nós e possuem um nó “raiz” e um ou mais nós “folha”.

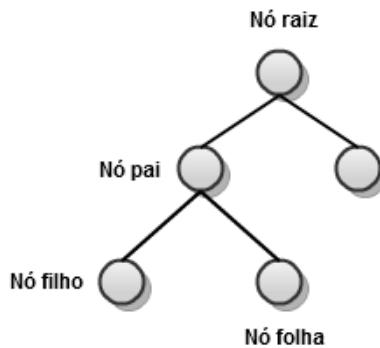
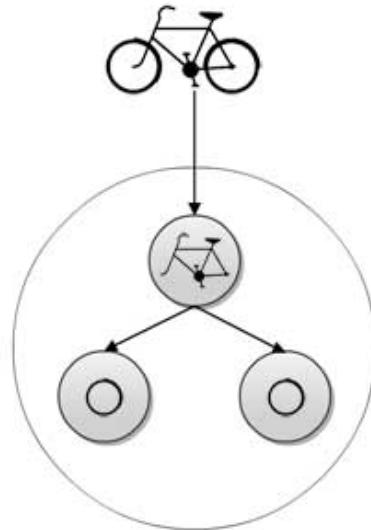


Figura 2.1. Uma estrutura de dados do tipo "árvore"

O propósito de um grafo de cena é permitir que algum processamento seja realizado enquanto os nós são visitados. O processo de visitação de nós também é conhecido como percurso. Os tipos de percursos realizados sobre os grafos de cena são os mesmos tipos de percurso realizados sobre outra árvore (pré-ordem, pós-ordem e in-ordem) (DROSDEK, 2002).

O uso mais comum dos grafos de cena é para a representar o relacionamento *todo-parte* entre os objetos da cena. Por exemplo, um modelo 3D de uma bicicleta representado como o quadro as e rodas (Figura 2.2) ilustra este conceito. Esse tipo de organização permite que objetos filhos possuam características relativas ao objeto pai. Por exemplo, pode-se permitir que operações como translação, rotação e escala, quando aplicadas a um objeto pai, sejam imediatamente repercutidas nos seus objetos filhos.



**Figura 2.2.** Nós representando o relacionamento todo-parte

Em aplicações onde se testa a colisões entre objetos, é comum usar os grafos de cena para construir limites volumétricos hierárquicos (*Hierarchical Bounding Volumes*) (FOLEY, 1990). Limites volumétricos são formas geométricas usadas para simplificar o teste de colisão entre objetos da cena. Construindo-os de maneira hierárquica, esses limites passam a representar não só objetos, mas vizinhanças de objetos (Figura 2.3). Essa técnica otimiza o número de testes de colisão, pois o teste de colisão pode ser feito com um grupo de objetos de uma vez. Por exemplo, ao testar a colisão de um determinado objeto com

outros objetos da cena, testa-se primeiro a colisão do limite volumétrico do objeto com os limites volumétricos das suas vizinhanças. Caso o teste de colisão com uma vizinhança passe, é necessário testar a colisão com as vizinhanças contidas dentro daquela vizinhança. Os testes de colisão se repetem até que sejam feitos efetivamente contra um objeto. Contudo, caso o teste de colisão contra uma vizinhança não passe, não é necessário testar a colisão com os objetos contidos pela vizinhança, reduzindo o número de testes necessários.



**Figura 2.3. Uma cena com os limites volumétricos hierárquicos visíveis (SIMPLESCENEGRAPH)**

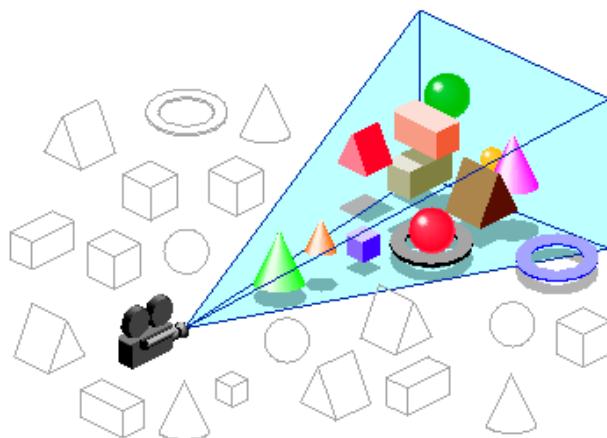
Também é comum a combinação de grafos de cena com técnicas de particionamento espacial (WAND, 2004). Técnicas de particionamento espacial dividem recursivamente o espaço em setores representados por sólidos, que são usados para aumentar a eficiência da renderização da cena ou para facilitar consultas sobre um determinado lugar no espaço.

Por fim, os nós do grafo de cena podem compartilhar atributos de renderização entre si. Os atributos de renderização (como cor, transparência, material, etc.) de um nó são usados para configurar os estados de renderização durante o percurso do grafo. Se um nó compartilhar seus atributos de renderização com seus filhos, de maneira similar ao

relacionamento *todo-parte*, pode-se reduzir o uso da memória secundária (pois os atributos pertencem apenas a um único nó), e pode-se reduzir o número de chamadas ao hardware gráfico (pois se configura a penas uma vez a máquina de estados do OpenGL durante o percurso do grafo).

### 2.1.1. View Frustum Culling

Os grafos de cena também colaboram com a redução do envio de primitivas para o hardware gráfico. Através do uso dos limites volumétricos hierárquicos pode-se verificar se um grupo inteiro de objetos está visível ou não (Figura 2.4). Por exemplo, se o limite volumétrico de uma vizinhança estiver totalmente fora do sólido de visualização, os objetos contidos por ela não precisarão ser enviados para o hardware gráfico a fim de serem renderizados. Isso aumenta significativamente a performance de um sistema de RV (CLARK, 1976).

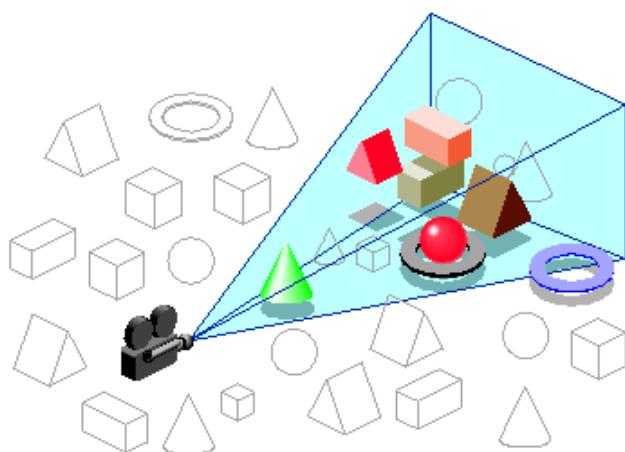


**Figura 2.4.** O view frustum culling consiste no descarte de objetos que estão completamente fora do sólido de visualização (ECKEL, 1998)

## 2.1.2. Occlusion Culling

Os grafos de cena também podem ser usados no auxílio da execução de algoritmos de PVS (*Potentially Visible Sets*). Os algoritmos de PVS determinam os objetos que estão ocultos por se situarem atrás de objetos mais próximos do visualizador (*occlusion culling*) (Figura 2.5).

Sem o uso do *occlusion culling*, o hardware gráfico usa o *z-buffer* para evitar o redesenho desnecessário de primitivas (Seção 2.4). Cenas com muitos objetos geralmente se beneficiam dessa técnica, pois o envio desnecessário de primitivas ao hardware gráfico é, geralmente, muito custoso que a execução de um algoritmo de PVS (STANEKER, 2005).



**Figura 2.5.** O occlusion culling consiste no descarte dos objetos que estão ocultos por estarem atrás de objetos mais próximos do visualizador (ECKEL, 1998)

Por esses motivos, o uso de grafos de cena é muito comum nos sistemas de RV. Hoje em dia existem inúmeros grafos de cena, proprietários e *open source*, sendo usados para os mais diversos fins, como simulação militar, visualização científica ou jogos eletrônicos.

## 2.2. Renderização Distribuída

A renderização pode ser definida como a síntese de imagens a partir da descrição computacional de uma cena 3D. Ao observar o processo de renderização, percebe-se que ele pode ser dividido em três partes: a leitura dos dados, o processamento das geometrias e a rasterização. Nos sistemas de RV distribuídos, essas três partes podem ser pensadas como independentes. O modelo canônico de renderização distribuída (Figura 2.6) mostra como cada uma dessas partes se relaciona e como elas podem ser divididas em diferentes nós de um aglomerado.



Figura 2.6. Imagem adaptada do modelo canônico (MOLNAR, 1991)

De acordo com (MOLNAR, 1991) o problema da renderização distribuída se encontra na distribuição das primitivas geométricas pelos nós do aglomerado. Seu trabalho aponta três possibilidades para quando a distribuição das primitivas pode ocorrer: antes do processamento da geometria (*Sort-First*), entre o processamento da geometria e a rasterização (*Sort-Middle*) e após a rasterização (*Sort-Last*).

### 2.2.1. Sort-First

O *Sort-First* é o método onde a distribuição ocorre antes do processamento geométrico (Figura 2.7). Nele cada nó do aglomerado é responsável pelo processamento de geometrias e pela rasterização de uma região de tela.

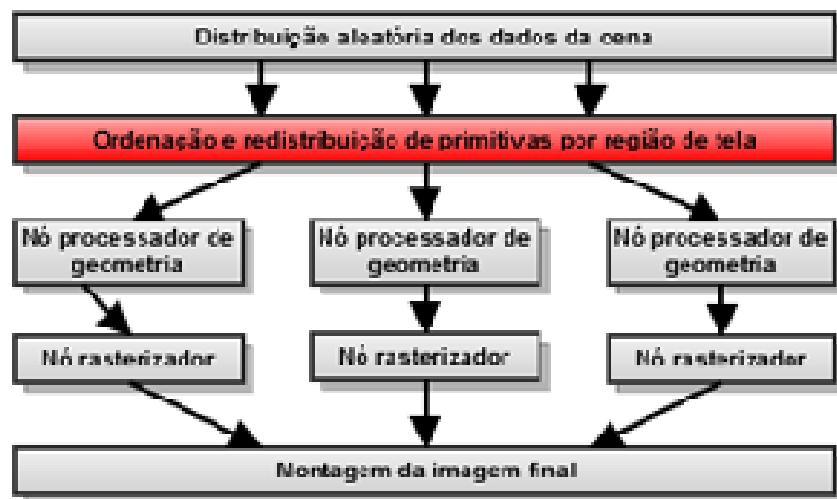


Figura 2.7. O esquema de trabalho no método *Sort-First* (Baseado em (MOLNAR, 1991))

Por exemplo, no *Sort-First*, um aglomerado com quatro nós teria a tela dividida em quatro regiões (Figura 2.8).

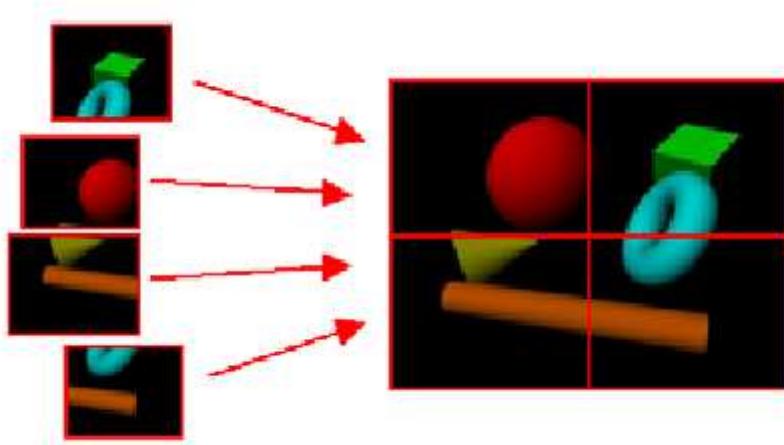


Figura 2.8. No *Sort-First* as primitivas são distribuídas por região de tela (SOARES, 2005)

Se cada nó do aglomerado é responsável por uma região de tela, saber para qual nó irá uma primitiva é o mesmo que saber em que região de tela a primitiva está. Após a rasterização, os resultados são enviados para uma camada que monta a imagem final.

A distribuição por região de tela, porém, faz com que cenas com muita animação causem uma sobrecarga no tráfego de rede. Por exemplo, se as primitivas mudarem constantemente de região de tela, ou seja, se houver muita animação na cena, haverá um tráfego intenso de dados entre os nós do aglomerado.

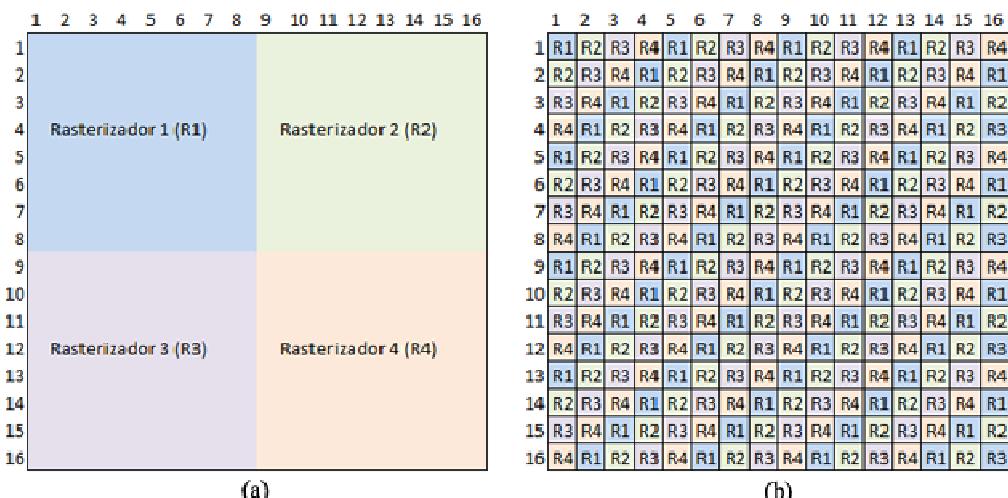
Existe também uma suscetibilidade ao desbalanceamento de carga. Geometrias complexas, que não são divisíveis entre regiões de tela, podem se acumular em um único nó, fazendo-o trabalhar em excesso. Em uma cena com muitas geometrias, onde há uma grande diversidade de tamanho e complexidade, essa disparidade de carga costuma ser acentuada.

Por fim, como o *Sort-First* já foi largamente estudado e implementado (Seção 3), ele não será usado neste trabalho.

## 2.2.2. Sort-Middle

O *Sort-Middle* é o método onde a distribuição ocorre entre o processamento geométrico e a rasterização (Figura 2.9). Nele os nós do aglomerado possuem duas categorias: os processadores de geometria, que transformam algebricamente as primitivas, e os rasterizadores, que recebem as primitivas já transformadas e rasterizam um trecho da tela.

No *Sort-Middle* existe uma camada que relaciona os processadores de geometria com os rasterizadores, roteando as primitivas de um para outro. Cada rasterizador recebe apenas algumas geometrias, rasterizando-as em fragmentos contíguos ou entrelaçados (pixels), de acordo com o modelo de divisão de tela escolhido (Figura 2.10).



**Figura 2.10. Os modelos de divisão de trecho de tela:** (a) Por fragmentos contíguos e (b) Por fragmentos entrelaçados (Baseado em (MOLNAR, 1991))

A existência de duas categorias de nós exige que se acesse as primitivas durante a execução do *pipeline gráfico*. Esse acesso não é eficiente em hardwares gráficos convencionais, sendo viável apenas em hardwares gráficos proprietários (SOARES, 2005).

Assim, como o *Sort-Middle* ainda não é viável em aglomerados de computadores pessoais, ele não será usado neste trabalho.

### 2.2.3. Sort-Last

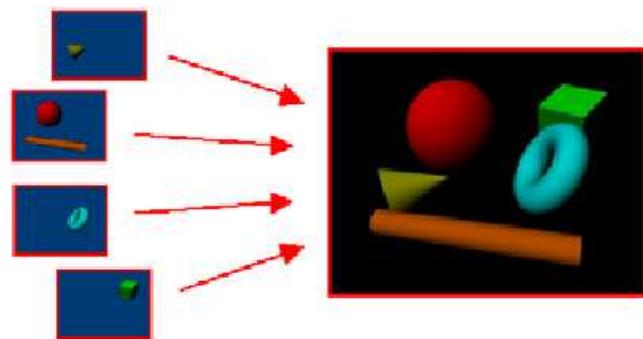
O *Sort-Last* é o método onde a distribuição ocorre após a rasterização (Figura 2.11). Nele cada nó do aglomerado é responsável pelo processamento da geometria e pela rasterização. Diferentemente do *Sort-First*, as primitivas são distribuídas segundo um critério qualquer, não por região de tela.



Figura 2.11. O esquema de trabalho do método Sort-Last (Baseado em (MOLNAR, 1991))

No *Sort-Last* existe uma camada que monta os resultados da rasterização em uma imagem final. Essa camada recebe o nome de compositora, devido à técnica usada na montagem da imagem final. A composição de imagens foi estudada em detalhes e é explicada na Seção 2.5.

Como dito anteriormente, a distribuição das primitivas não é por região de tela e, portanto, não há necessidade de redistribuir as primitivas. Contudo, os resultados transmitidos para a camada compositora, a cada quadro, possuem o tamanho da tela toda (Figura 2.12) e podem causar um tráfego intenso de dados na rede.



**Figura 2.12.** No *Sort-Last* as imagens transmitidas para a camada compositora possuem o tamanho da tela toda  
(SOARES, 2005)

Por fim, como o *Sort-Last* é relativamente pouco explorado em estudos acadêmicos e em ferramentas, e como sua implementação não vai de encontro a nenhuma das propostas iniciais, ele será usado neste trabalho.

## 2.3. Sincronismo

Os sistemas distribuídos em aglomerados de computadores são compostos por partes chamadas de nós. Um nó do sistema distribuído pode acessar e alterar dados que são compartilhados entre todos os nós. Para evitar que esses dados fiquem inconsistentes, é necessário que o acesso concorrente a eles seja controlado. No contexto deste trabalho, sincronismo é o nome do controle de acesso concorrente aos dados.

Apesar de existirem sistemas distribuídos com um certo nível de tolerância à inconsistência de dados, esse não costuma ser o caso dos sistemas de RV distribuídos. Os sistemas de RV distribuídos necessitam manter seus dados consistentes, pois um dado inconsistente pode ocasionar a falta de coerência visual (Figura 2.13).



**Figura 2.13.** Os sistemas de RV distribuídos precisam manter seus dados consistentes, caso contrário podem sofrer de falta de coerência visual (SOARES, 2005)

Os sistemas distribuídos que exigem um sincronismo rígido, como os sistemas distribuídos de RV, precisam ter cuidado para que não haja um aumento drástico de sua *latência*. A *latência* é o tempo de espera existente entre as partes de um sistema distribuído. Esse tempo de espera existe pois as partes devem ser impedidas de acessar

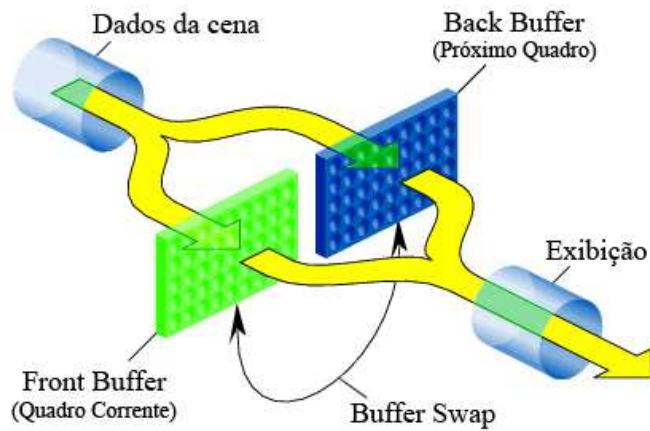
certos dados até que outras partes já o tenham acessado. O impedimento do acesso aos dados geralmente é implementado pelos *locks*. Os *locks* implementam o conceito de exclusão mútua, pois só permitem que os dados sejam acessados por uma parte de cada vez. Isso previne problemas clássicos de alteração de dados em paralelo, como a condição de corrida (Figura 2.14).



Figura 2.14. A condição de corrida é um problema clássico de alteração de dados em paralelo

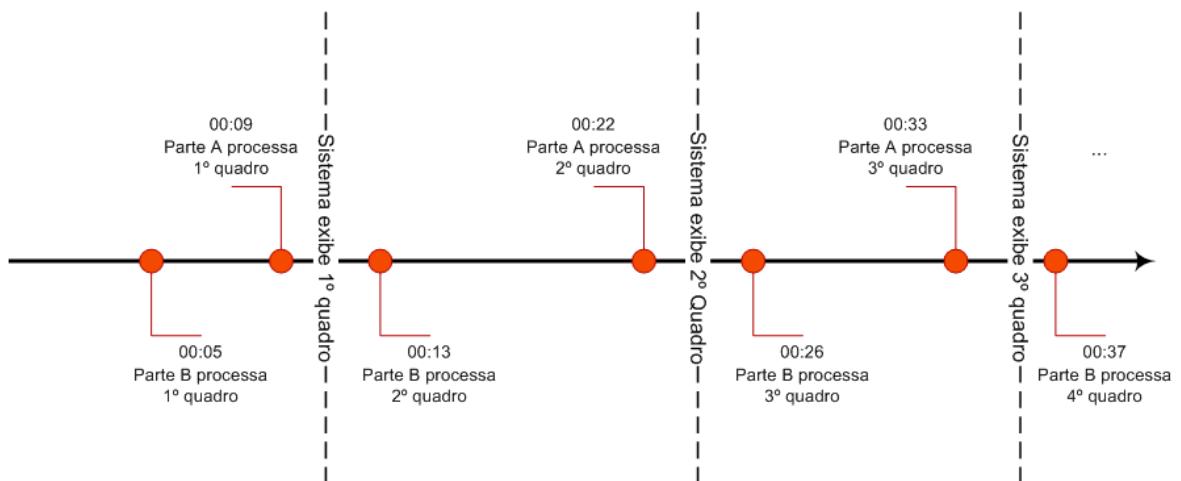
### 2.3.1. Frame-Lock

Nos sistemas de RV distribuídos, o sincronismo de quadros, ou *frame-lock*, é o mecanismo que garante que os nós não gerem quadros adiantados. Quando se usa a técnica de armazenamento múltiplo (*double* ou *quad buffering*), ele atua no momento da troca de quadro, em cada um dos nós. Isso é possível pois quando um nó termina a exibição do quadro corrente, um comando de troca de *buffers* (*swap-buffer*) muda o quadro em exibição para um dos quadros que estavam sendo processados em paralelo (Figura 2.14).



**Figura 2.15.** O swap-buffer muda a exibição entre o buffer frontal e um dos buffers traseiros (Baseado em (ECKEL, 1998))

O *frame-lock* impede que um nó troque o quadro em exibição enquanto os outros estiverem exibindo um quadro anterior. Em outras palavras, o *frame-lock* impede que um nó adiante a exibição de quadros em relação a outro (Figura 2.15).



**Figura 2.16.** O frame-lock faz que nenhum nó se adiante na exibição dos quadros em relação ao outro nó

### **2.3.2. Rate- Lock**

Nos sistemas de RV distribuídos, o sincronismo de taxa de atualização, ou *rate-lock*, é o mecanismo que garante uma *latência* máxima para o sistema. Isso ocorre pois a taxa de atualização de quadros de um sistema de RV distribuído depende da taxa de atualização de quadros dos seus nós. Assim, se um nó se atrasar o sistema inteiro pode atrasar.

O *rate-lock* age impondo um tempo limite para a atualização de quadro dos nós. Se um nó se atrasar além do limite permitido, o *rate-lock* o faz descartar seu quadro corrente, obrigando-o a processar o quadro seguinte imediatamente.

### **2.3.3. Data-Lock**

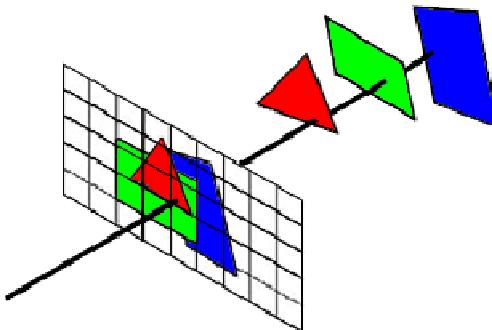
Nos sistemas de RV distribuídos, o sincronismo de primitivas, ou *data-lock*, é o mecanismo que garante que as primitivas compartilhadas são iguais em cada nó que as acessa. Para manter a coerência visual, qualquer mudança sofrida por uma primitiva compartilhada por dois ou mais nós precisa ser repercutida em todos eles. Por exemplo, se uma primitiva muda de posição em um nó, ela deve mudar de posição em todos os nós que também a usam.

O *data-lock* atua nas primitivas impedindo que elas sejam mudadas por um nó até que todas as mudanças realizadas por outro nó tenham sido propagadas para todos. Muitos estudos já foram feitos sobre o *data-lock*, já que essa é a principal causa de inconsistência de dados entre sistemas de RV colaborativos.

## 2.4. Z-Buffering

Na renderização é importante que se tome cuidado com a ordem que as primitivas são renderizadas. A renderização de primitivas na ordem incorreta pode fazer com que uma primitiva mais distante se sobreponha a uma mais próxima do observador, causando a perda da noção de profundidade. Na literatura, esse problema recebe o nome de problema de visibilidade.

Existem diversas maneiras de contornar o problema de visibilidade. A mais conhecida é o algoritmo do pintor. O algoritmo do pintor garante que as primitivas mais distantes serão renderizadas antes das mais próximas do observador (Figura 2.16). Contudo, apesar do algoritmo do pintor impedir a sobreposição de primitivas, ele não impede que os pixels que ocupam a mesma posição de tela sejam renderizados mais de uma vez. Como a renderização é uma operação custosa, evitar que um pixel seja renderizado mais de uma vez pode aumentar significativamente a performance de um sistema de RV.



**Figura 2.17.** A renderização das primitivas é realizada na ordem correta através do algoritmo do pintor (Baseado em (CONCI))

Uma alternativa ao algoritmo do pintor é a técnica do *z-buffering*. O *z-buffering* usa uma matriz de distância dos pixels (*z-buffer*). A cada tentativa de renderizar um pixel, testa-se a distância do pixel contra a distância armazenada na posição do pixel. Caso a distância do pixel a ser renderizado seja menor, ou seja, se o pixel estiver mais próximo do observador, ele é desenhado e sua distância é armazenada na matriz para aquela posição (Figura 2.17).

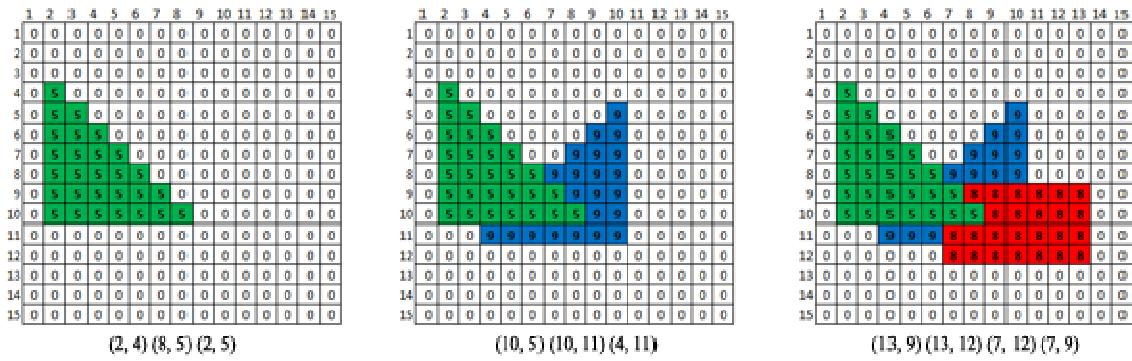
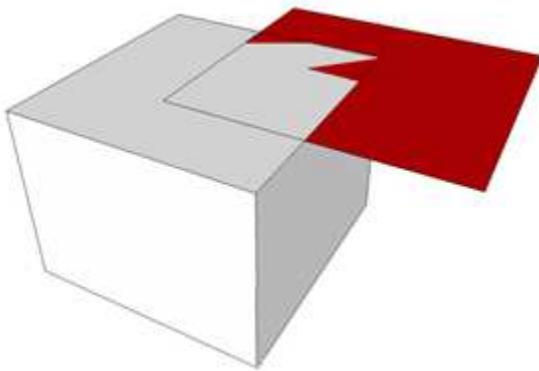


Figura 2.18. As distâncias dos pixels no *z-buffer*, após o desenho de três primitivas (Baseado em (CONCI))

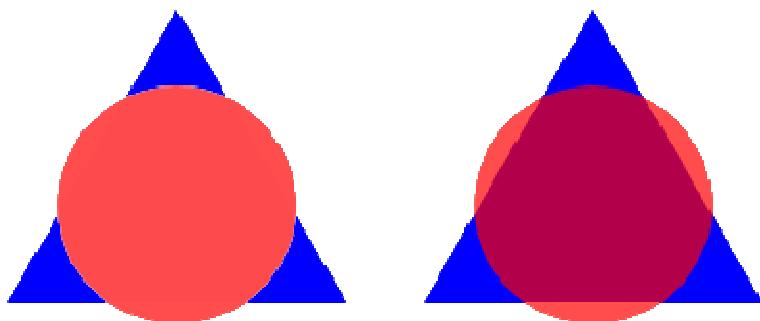
O *z-buffering* pode ser implementado em software, mas atualmente já se encontra disponível até mesmo nos hardwares gráficos mais simples. Na maioria dos hardwares, o *z-buffer* possui dimensões iguais a da tela e armazena valores de até 32-bits. No passado, quando os hardwares usavam *z-buffers* de 8 ou 16-bits, era comum haver erro de sobreposição de primitivas muito próximas (Figura 2.18), por causa da pouca representatividade numérica apresentada pelo *z-buffer*.



**Figura 2.19.** Nos *z-buffers* de 8 e 16-bits, partes das primitivas podiam se sobrepor indesejavelmente quando estavam muito próximas (*z-fighting*)

Um *z-buffer* com as dimensões de tela pode ocupar um tamanho expressivo na memória gráfica. Por exemplo, para uma tela de resolução 1650x1080, o *z-buffer* terá aproximadamente 54 megabytes. Por esse motivo, normalmente se aplica alguma técnica para manipulá-lo no próprio hardware gráfico, como o comando para limpar o *z-buffer*, sem precisar acessá-lo.

Como o *z-buffer* não permite que pixels sejam redesenhados, uma primitiva transparente pode não conseguir mesclar sua cor (*alpha blending*) com a cor de um pixel que seria desenhado atrás dela (Figura 2.20).



**Figura 2.20.** Círculo transparente desenhado sobre triangulo azul com e sem o *z-buffer* habilitado

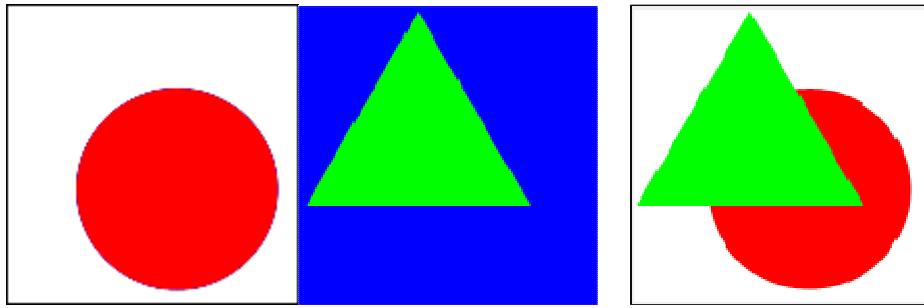
Para solucionar esse problema é necessário desenhar as primitivas não-transparentes antes de desenhar as primitivas transparentes. Tomando essa precaução é possível resolver a maioria dos problemas de transparência, à exceção de um: quando duas ou mais primitivas transparentes se sobrepõem.

## 2.5. Composição de Imagens

A composição de imagens apresenta duas principais abordagens: a composição por prioridade fixa e a composição por prioridade de pixels. Suas vantagens e desvantagens são expostas adiante, através da análise do trabalho de (MOLNAR, 1991).

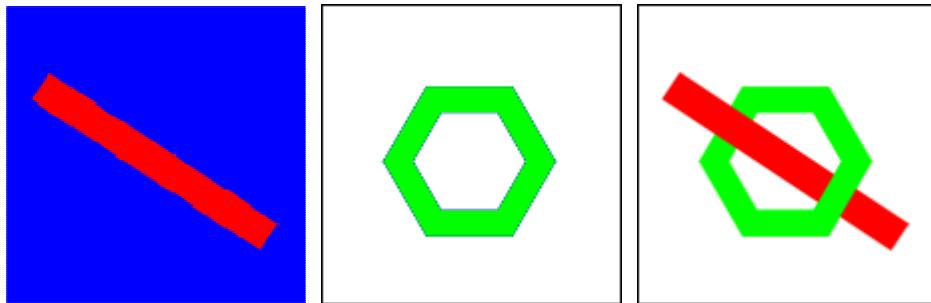
Na composição por prioridade fixa, o sistema atribui um valor de prioridade para cada uma das imagens usadas. Assim, durante a composição, as imagens de maior prioridade sobrepõem as de menor prioridade.

A técnica do *chroma key* usa essa abordagem. No *chroma key* existem duas imagens: a imagem frontal e a imagem de fundo. A imagem frontal possui partes com uma cor chave (*chroma key*) que será ignorada durante a composição com a imagem de fundo. Assim, a imagem de fundo será exibida atrás da imagem frontal, dando a impressão de um segundo plano. O efeito final pode ser observado na Figura 2.21.



**Figura 2.21.** A imagem de fundo e a imagem frontal, com cor chave azul, resultando na imagem final a direita

A implementação da composição por prioridade fixa é, geralmente, simples e pouco custosa. Contudo, essa abordagem apresenta a desvantagem de não ser possível realizar o enlace de figuras, como representado na Figura 2.22.



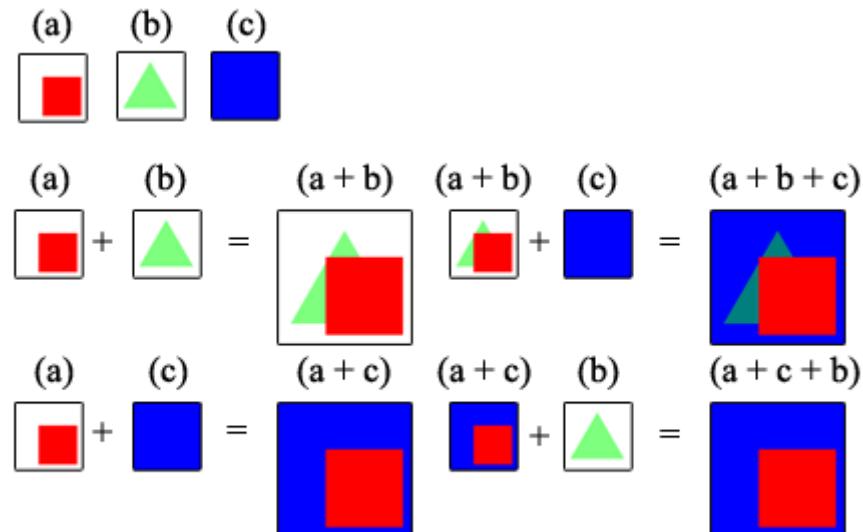
**Figura 2.22.** Na composição de imagens por prioridade fixa não é possível haver enlace de figuras

Já a composição por prioridade de pixels possibilita efeitos muito mais refinados do que a composição por prioridade fixa. Nessa abordagem, é atribuída uma prioridade para cada pixel das imagens geradas. O valor de prioridade de cada pixel é geralmente o valor da distância do observador para o pixel.

A composição por prioridade de pixels pode ser implementada usando o *z-buffer* (Seção 2.4). O *z-buffer* pode ser usado para obter as distâncias do observador de cada pixel. Assim, a imagem final pode ser composta através dos pixels de menor distância.

Uma implementação como essa, apesar de simples, permite haver o enlace de figuras da Figura 2.22.

Essa abordagem, no entanto, demanda o mesmo cuidado com primitivas transparentes que o *z-buffer*. De maneira análoga, durante a composição, as imagens que não contém primitivas transparentes devem ser usadas antes das imagens que contém primitivas transparentes. Por exemplo, a Figura 2.21 mostra três imagens (a), (b) e (c), com apenas (b) contendo uma primitiva transparente. Se as imagens forem compostas na ordem (a + b + c), a primitiva transparente não é exibida. Contudo, se as imagens forem compostas na ordem (a + c + b), sendo a imagem (b) usada por último, a primitiva transparente é exibida corretamente.



**Figura 2.23.** Ao compor imagens usando o *z-buffer* as imagens que contém transparência devem ser compostas por último

# 3. Trabalhos Relacionados

## 3.1. OpenSG

O OpenSG (OPENS) é um grafo de cena *open source*. Escrito em C++ e desenvolvido sobre a biblioteca OpenGL (OPENGL), seus objetivos principais podem ser descritos como: portabilidade, extensibilidade, suporte a *Multi-Threading* e suporte à aglomerados de computadores.

Atualmente o OpenSG funciona declaradamente nas plataformas Linux, MacOS, Solaris e Windows. Contudo, por ser baseado em ferramentas de padrão aberto, como C++ e OpenGL, ele é virtualmente executável em todas as plataformas suportadas por essas ferramentas.

A extensibilidade preconizada pela ferramenta não fica apenas no código aberto, mas provê mecanismos de extensão dinâmica em tempo de execução. Como a arquitetura do OpenSG é fortemente voltada para o uso de grafos de cenas (Seção 2.1), ele possibilita a adição dinâmica de código cliente, invocado inteligentemente através de um mecanismo chamado expedição dupla (*Double-Dispatch*). O *Double-Dispatch* permite que o código cliente seja invocado contextualmente, a partir de decisões que envolvem o tipo do percurso e o tipo do nó correntes.

Uma característica marcante do OpenSG é o suporte à *Multi-Threading*. Diferentemente de outros grafos de cena que dão suporte a essa funcionalidade, o OpenSG não se apoia sobre o uso de *locks* (Seção 2.3), mas faz uso de replicação de dados (*Multi-Buffering*) (REINERS, 2002). Para cumprir essa meta ele se apoia em duas técnicas

avançadas de programação: ponteiros inteligentes e reflexividade. Seu ponteiro inteligente (*Field Container Pointer*), cujo uso é reforçado em todas as classes da ferramenta, é o engenho responsável pelo sincronismo entre *Threads*. Através dele a aplicação não acessa diretamente um dado, mas uma cópia, impedindo que haja a criação de inconsistências por modificações concorrentes. Conjuntamente, o uso de reflexividade, que é a capacidade de uma classe conhecer detalhes sobre si mesma, permite que as informações do grafo sejam replicadas durante a criação de novos *buffers* (ex.: na criação de uma nova *Thread*). Na realidade, o suporte à *Multi-Threading* é construído sobre a teoria de que, nos grafos de cena, dados escalares são muito menos frequentes do que dados vetoriais (na razão estimada de 1 para 10). Assim, o OpenSG apenas replica os dados escalares, disponibilizando os dados vetoriais em memória compartilhada. Essa medida torna sustentável o uso de uma cópia do grafo por *Thread*, uma vez que a replicação inteira dos dados do grafo seria impraticável para a maioria dos sistemas.

O suporte a aglomerados é similar ao suporte a *Multi-Threads* pois em ambos os casos o processo computacional é compartilhado entre múltiplas unidades de trabalho. Dessa forma, muito é emprestado do mecanismo de sincronismo e da capacidade reflexiva dos objetos empregados no suporte à *Multi-Threading*. Por exemplo, no envio de parte do grafo pela rede, a reflexividade torna natural a serialização dos objetos em fluxos de dados. O modelo de operações do OpenSG dita que haja apenas um cliente para múltiplos servidores em seu aglomerado. Isso ocorre pois seu conceito de servidor está diretamente ligado ao de nó renderizador, enquanto que o cliente é aquele que realiza interface com o usuário e que detém os dados da cena, distribuindo-os. O OpenSG implementa dois dos métodos de distribuição vistos anteriormente: o *Sort-First* e o *Sort-Last* (Seção 2.2). O desenvolvimento de aplicações distribuídas com qualquer um destes métodos é quase

totalmente transparente. Isso acontece pois toda a separação de primitivas, balanceamento de carga, transferência de imagens e composição são gerenciados automaticamente pelo OpenSG.

## 3.2. Chromium (WireGL)

O Chromium (CHROMIUM) é uma ferramenta que substitui dinamicamente as bibliotecas do OpenGL. Sendo completamente não invasiva, ela pode transformar aplicações gráficas não distribuídas em aplicações distribuídas sem precisar recompilá-las. Isso é feito através de uma série de componentes, responsáveis não só pela interceptação das chamadas ao OpenGL, como também por um modelo inovador de processamento de *Streams*, um mecanismo de rastreamento de estado e de extensibilidade do OpenGL e um gerenciador de processos dinâmico. Além disso, o Chromium também é parte de uma pilha de tecnologias usadas para dar maior suporte à paralelização de sistemas gráficos.

O modelo de processamento de *Streams* é baseado no conceito *Stream Processing Unit*<sup>1</sup> (SPU). A SPU é uma estrutura de dados que intermedeia as chamadas ao OpenGL. Isso possibilita o controle transparente da maneira com que essas chamadas são executadas e distribuídas pelo aglomerado. Um exemplo de SPU é a Render SPU, que sintetiza as imagens a serem apresentadas. O Chromium vem embarcado com uma série de SPUs, que podem ser compostas para gerar a saída final.

O OpenGL possui uma máquina de estados complexa, onde são armazenadas informações de cor, textura, modo de renderização, etc. O mecanismo de rastreamento de

---

<sup>1</sup> O termo *Stream Processing Unit* pode ser traduzido como unidade de processamento de fluxo

estado possibilita que a máquina de estado do OpenGL seja transmitida e restaurada, em qualquer nó do aglomerado, através do menor número de chamadas possíveis. Isso é particularmente interessante em cenários onde as chamadas OpenGL devem ser paralelizadas entre vários renderizados, como no caso de um sistema com múltiplos dispositivos de exibição.

O mecanismo de extensão ao OpenGL é intrínseco ao próprio modelo de processamento de *Streams*. Com a possibilidade de criar novas SPUs além das embarcadas com o Chromium, o desenvolvedor adquire a capacidade de criar abstrações sobre a operação de renderização. Com isso ele pode modificar o *Pipeline* de geometrias, adicionar efeitos à imagem final, além de realizar a depuração das chamadas ao OpenGL (o que é de grande utilidade e difícil de ser feito por modos mais convencionais).

A última parte dessa ferramenta, conhecida por *Mothership*<sup>2</sup>, é responsável por gerenciar os processos envolvidos na paralelização da renderização. Este gerenciamento de nós é feito *On Demand*, permitindo que novos participantes ingressem tardiamente ao aglomerado. Outro aspecto importante da *Mothership* é a capacidade de ser configurada através de *Scripts Python*. A configuração através de Scripts permite que o Chromium se adapte dinamicamente às mudanças de ambiente, o que é comum nos *commodity clusters*.

A *Mothership* é bastante flexível, podendo ser configurada para usar os métodos clássicos de distribuição de renderização, como o *Sort-First* (Figura 3.1) e o *Sort-Last* (Figura 3.2), e também mesclá-los em combinações exclusivas ao Chromium (Figura 3.3).

---

<sup>2</sup> Mothership traduzido do inglês significa nave mãe

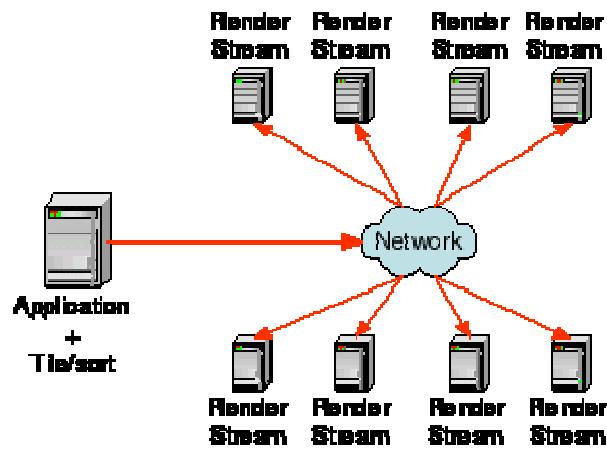


Figura 3.1. Mothership configurada para realizar o Sort-First (CHROMIUM)

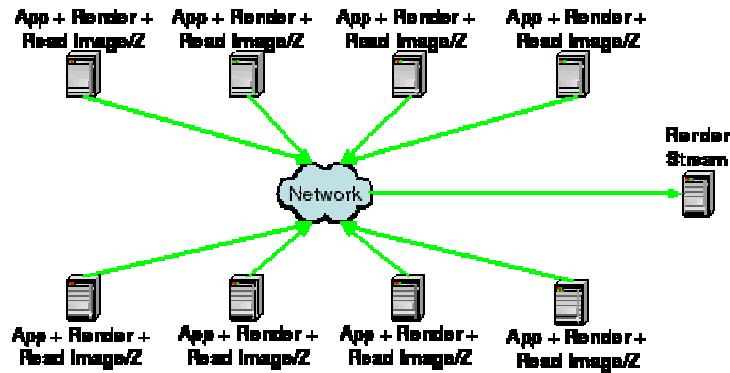


Figura 3.2. Mothership configurada para realizar o Sort-Last (CHROMIUM)

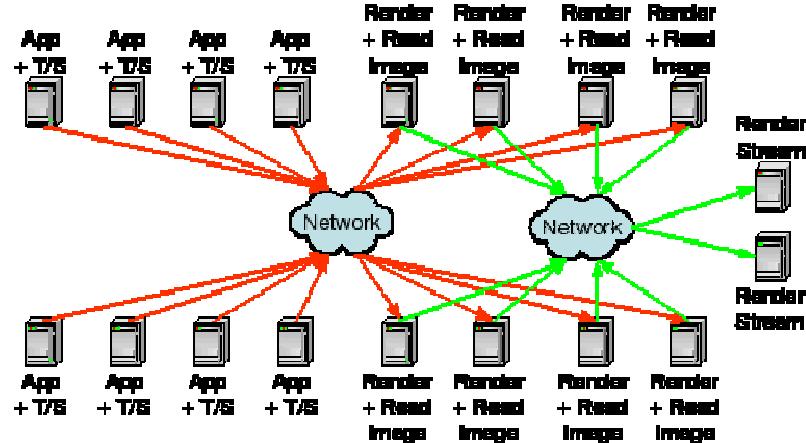


Figura 3.3. Configuração que combina características de Sort-First com Sort-Last de maneira única (CHROMIUM)

Finalmente, o Chromium faz parte de uma pilha de ferramentas que, conjugadas, permitem um suporte mais completo à paralelização de aplicações gráficas. As ferramentas comumente conjugadas ao Chromium são o DMX (DMX), que abstrai a distribuição das chamadas ao servidor de janelas X11, a PIXA [PIXA], uma API que determina um modelo para a composição de imagens, a MIDAS (MIDAS), que provê a separação da realização da renderização da entrega da imagem, dentre outras. Vale notar que todas as ferramentas dessa pilha são *open source*, assim como o próprio Chromium.

### 3.3. Syzygy

O Syzygy (SHAEFFER, 2000) é uma solução completa para a construção de aplicações de RV distribuídas. Seus principais objetivos são alta performance e gerenciamento fácil. Os pilares do Syzygy são quatro: uma camada de comunicação, um sistema operacional (SO) distribuído, um *framework* de E/S e seus *frameworks* de aplicação.

A camada de comunicação do Syzygy é uma abstração sobre *Sockets*, *Threads*, *Pipes* e outras funcionalidades de um sistema operacional. Ela também engloba a infraestrutura de troca de mensagens do Syzygy, que é responsável pela troca eficiente de informações em formato binário. Construída sobre bibliotecas famosas como CORBA (CORBA) e RPC, ela disponibiliza uma série de funções que tratam detalhes de integração de baixo nível, como aqueles ligados às particularidades de máquina (ex.: *Endianess*<sup>3</sup>).

---

<sup>3</sup> *Endianess* é o termo utilizado para se referir a ordem dos *bytes* utilizada na representação de um tipo de dado.

O SO incluso no Syzygy se chama Phleet. O Phleet disponibiliza inúmeras ferramentas de linha de comando que auxiliam no gerenciamento dos aglomerados do Syzygy. Estas ferramentas permitem a inicialização e parada dos nós, assim como o ingresso tardio de novos nós a um aglomerado. Outra característica interessante é a substituição dos arquivos de configuração por um banco de dados de parâmetros acessível por rede. Como Phleet é construído sobre a mesma camada de comunicação usada pelas aplicações, ele torna natural a criação de aglomerados mesmo a partir de sistemas heterogêneos.

O *framework* de E/S do Syzygy é responsável por compartilhar os dados dos dispositivos de entrada com o resto do sistema. Os principais componentes deste framework são as fontes de entrada, as *Sinks*<sup>4</sup> de entrada e os filtros. As fontes de entrada são responsáveis por produzir mensagens Syzygy a partir dos fluxos de dados recebidos da rede. Elas são usadas como interface entre os dados coletados dos dispositivos e o resto do sistema. As pias de entrada, por outro lado, absorvem dados e os lançam como fluxos na rede. Elas são responsáveis pela comunicação com as bibliotecas de captura. Fontes e pias podem ser conjugadas a fim de combinar dados e formar novos dispositivos virtuais. Os filtros podem ser conectados aos fluxos de dados a fim de modificá-los. Eles podem suavizar movimento de *Joysticks*, remover o pressionamento de certos botões ou mesmo redirecionar os dados para rotinas de calibragem.

Os dois *frameworks* de aplicações disponibilizados pelo Syzygy são o grafo de cena distribuído e o mestre/escravo. Eles diferem no tratamento da relação antagônica entre flexibilidade e eficiência. O primeiro visa atender cenários onde há muita animação ou

---

<sup>4</sup> O termo *Sink* não foi traduzido para pia, pois possui um significado particular no contexto da computação.

acesso à fontes de dados externos. Estes cenários sugerem uma maior flexibilização na maneira com que os dados são acessados e distribuídos. Assim, existe todo um cuidado no balanceamento da carga de trabalho e na manutenção do sincronismo entre os nós, o que pode vir ao custo de uma diminuição na eficiência. O segundo consiste em uma maneira de executar cópias de uma mesma aplicação em diversos nós do aglomerado. Uma cópia mestra se responsabiliza por coletar inputs de usuário, realizar computações e distribuir os resultados às outras cópias chamadas de escravas. Essa inflexibilidade no acesso e distribuição dos dados possibilita uma otimização nos protocolos de compartilhamento, como o uso de um mecanismo de compressão, o que aumenta a eficiência final.

Composto por essas cinco partes, o Syzygy provê uma API que atende a múltiplos cenários de distribuição, além de um mecanismo que unifica e simplifica o gerenciamento de aglomerados. Concluímos, dessa forma, que o objetivo principal do Syzygy é endereçar o problema do gerenciamento de complexidade dos grandes sistemas de RV.

### **3.4. FlowVR**

O FlowVR (RAFFIN, 2006) é um conjunto de ferramentas para construção de aplicações de RV distribuídas. Ele objetiva endereçar questões de engenharia de software e limitações de hardware comuns aos sistemas de RV complexos. Para tanto ele se apoia sobre três ideias principais: o modelo de fluxo de dados, uma arquitetura modularizada e um protocolo de comunicação de alta performance.

Diferindo de muitas ferramentas do gênero e se assemelhando a aplicações de visualização científica, o FlowVR adota um modelo de fluxo de dados. Neste modelo o

aglomerado se estrutura como um grafo orientado, tendo tarefas como nós e canais de comunicação FIFO como arestas. Este modelo é preferido ao uso de grafos de cena, pois possibilita a construção de aglomerados onde fica mais clara a dependência de dados entre os nós.

Outra peculiaridade do FlowVR é sua arquitetura modular. O módulo é o componente de arquitetura onde as tarefas do sistema são implementadas. O FlowVR disponibiliza uma biblioteca para o desenvolvimento de módulos, assim como já disponibiliza alguns módulos prontos. Os módulos não possuem conhecimento da existência ou podem ser comunicar diretamente com outro módulo, antes interagem com um *Daemon* residente no mesmo computador. O *Daemon*, que é outro componente arquitetural, é o responsável pelo envio de mensagens entre módulos (locais ou remotos). Cada nó do aglomerado do FlowVR deve possuir um *Daemon*, estabelecendo assim uma rede de comunicação. Tanto módulos quanto *Daemons* são configurados através de arquivos XML. Existe um módulo especial chamado *Controller* que realiza as tarefas gerenciais do aglomerado.

Dois módulos embarcados muito importantes no FlowVR são o visualizador e o renderizador. Os visualizadores são responsáveis pela criação e pela distribuição das primitivas geométricas. Já os renderizadores recebem as primitivas geométricas enviadas pelos visualizadores e as renderizam em imagens. A comunicação entre estes dois módulos é feita através de um protocolo de alta performance que explora o fato dos renderizadores serem completamente baseados em shaders para trafegar menos informações durante as trocas de mensagem. Os shaders são programas gráficos que podem especificar a aparência dos objetos a partir de poucos parâmetros. Dessa forma, apenas os dados necessários para invocar os shaders são trafegados, em oposição ao envio de toda primitiva e suas

informações e suas informações de renderização. Isso também faz com que o protocolo se desresponsabilize pelo gerenciamento da máquina de estados do OpenGL. Uma otimização adicional é o *Retained Mode* sob o qual os renderizadores trabalham. No *Retained Mode* apenas as atualizações das primitivas precisam ser enviadas pelos visualizadores, diminuindo ainda mais o número de informações trafegadas.

O FlowVR também provê ferramentas que facilitam o desenvolvimento de aplicações de RV. Ele disponibiliza o FlowVR-GLGraph, que é um visualizador da organização do aglomerado, o FlowVR-GLTrace, que é um rastreador de eventos e de mensagens trocadas entre os módulos e o FlowVR-VTK, que encapsula o visualizador VTK na forma de módulos do FlowVR.

### 3.5. jReality

A jReality (BRINKMANN, 2010) é uma biblioteca usada na criação de aplicações de RV em Java. Concebida originalmente como uma ferramenta para visualização matemática, seu crescimento a fez tomar um caminho mais abrangente que esse. Seu principal diferencial é a capacidade de fazer aplicações de RV executarem tanto em *desktops* quanto em ambientes virtuais (VEs) sem necessitar alteração de código. A jReality se destaca por seu suporte à diferentes renderizadores (*Backends*), seu mecanismo de abstração de entrada e saída (sistema de ferramentas) e algumas funcionalidades inéditas no âmbito das aplicações matemáticas alcançadas através da sua infraestrutura de neutralidade métrica.

Na base do suporte a diferentes renderizadores se encontra um grafo de cena. Implementando o padrão de projeto *Visitor* (GAMMA, 1995) a jReality é capaz de

alimentar diferentes renderizadores simultaneamente, fazendo-os produzir diferentes saídas. Dentre os renderizadores disponibilizados pela ferramenta, os mais notáveis são o renderizador baseado em OpenGL e o renderizador distribuído. Eles são responsáveis, respectivamente, pela exibição de uma cena em uma única tela ou em múltiplas telas. Além desses, a jReality oferece um renderizador que exporta suas imagens para PDF, um renderizador que realiza interface com o RenderMan (software da empresa Pixar) e um renderizador implementado puramente em software, especialmente útil para cenas onde existe muita transparência.

O sistema de ferramentas é o mecanismo que separa o significado de uma interação com o usuário do hardware de captura de entrada. Ele usa os conceitos de dispositivo virtual e ferramenta para correlacionar dinamicamente uma entrada a uma ação. Seu funcionamento pode ser descrito da seguinte forma: os dispositivos físicos de entrada (*Raw Devices*) disponibilizam matrizes de transformação. Deles compõe-se dispositivos virtuais (*Virtual Devices*), que disponibilizam matrizes resultantes da operação entre duas ou mais matrizes. Cada transformação, vinda de um *Raw* ou *Virtual Device*, é mapeada para uma posição de entrada (*Input Slot*). Uma determinada ferramenta que implementa uma ação (ex.: rotação, translação, escala, arraste...) mantém uma lista de *Input Slots* que são capazes de ativá-la. Uma vez ativada, a ferramenta realiza seu trabalho, recebendo o *Input Slot* que a ativou como parâmetro. Além das ferramentas que possibilitam a interação direta com as geometrias (*Point-Based Tools*) existem ferramentas que implementam os mecanismos de navegação pelo mundo virtual, como a classe *ShipNavigationTool*, que permite a navegação pelo mundo como que caminhando pelo chão, e a classe *FlyTool*, que possibilita uma movimentação similar ao voo.

A neutralidade métrica da jReality permite a representação do mundo virtual através de três espaços matemáticos: o euclidiano, o esférico e o elíptico. Para tanto, a infraestrutura desta biblioteca foi projetada para realizar o processamento geométrico (ex.: cálculo de interpolação de curvas, projeções, distâncias, ângulos, etc.) de maneira parametrizada. Essa flexibilização possibilitou o desenvolvimento de funcionalidades únicas à jReality, como o shading em tempo real e rastreamento posicional (*Tracking*) não euclidianos.

Por fim, a jReality é uma ferramenta de código fonte aberto, licenciada sob o modelo BSD, sendo a plataforma de software adotada pela universidade de Berlim e pelo laboratório VisorLab da Faculdade da Cidade de Nova York (*City College of New York*).

### 3.6. Comparativo

As ferramentas estudadas foram comparadas em cinco aspectos: meios de configuração, métodos de distribuição, suporte a diferentes dispositivos de entrada/saída próprios às aplicações de RV, mecanismos de sincronismo e disponibilidade de plataforma. Segue abaixo (Tabela 3.1) o resultado das comparações.

	<b>OpenSG</b>	<b>Chromium</b>	<b>Syzygy</b>	<b>FlowVR</b>	<b>jReality</b>	<b>XithCluster</b>
Configuração	Dependente da aplicação	<i>Scripts Python</i>	Bancos de dados de parâmetros	<i>Scripts Perl e arquivos XML</i>	Arquivos de texto	Arquivos de texto e linha de comando
Distribuição	<i>Sort-First e Sort-Last</i>	<i>Sort-First, Sort-Last e combinações</i>	<i>Sort-First e Sort-Last</i>	<i>Sort-First</i>	<i>Sort-First</i>	<i>Sort-Last</i>
Suporte à E/S para RV	Não	Não	Sim	Sim	Sim	Não
Sincronismo	<i>Multi-Buffering</i>	Manual <sup>5</sup>	<i>frame-lock</i>	<i>frame-lock e Data-Lock</i>	<i>Desconhecido</i>	<i>frame-lock</i>
Plataformas	Linux, Windows, MacOS e Solaris	Linux, Irix, Windows e MacOS	Linux, Irix, Windows e MacOS	Linux	Linux, Windows e MacOS	Linux, Windows e MacOS

**Tabela 3.1. Comparativo de funcionalidades entre as diferentes soluções analisadas**

<sup>5</sup>

O sincronismo no Chromium é feito programaticamente através de barreiras de sincronização (ANDREW, 2000)

# 4. Solução Proposta (XithCluster)

## 4.1. Introdução

Neste trabalho foi desenvolvida uma solução, chamada XithCluster, que permite a construção de aplicações de RV distribuídas. Escrita na linguagem Java, ela usa a biblioteca *xSockets* (XSOCKETS) e o grafo de cena Xith3D (XITH3D). O Xith3D é um grafo de cena *open source*, versátil e com grande quantidade de recursos, mas que não pode ser distribuído através de múltiplos computadores.

O XithCluster é extensível e pode ser personalizado para atender amplos cenários. Também provê uma camada de comunicação de alta performance, que minimiza a intensa transmissão de dados prevista no método *Sort-Last* (Seção 2.2.3). Ela está organizada em uma biblioteca de classes e duas aplicações independentes (Seções 4.7, 4.8 e 4.9).

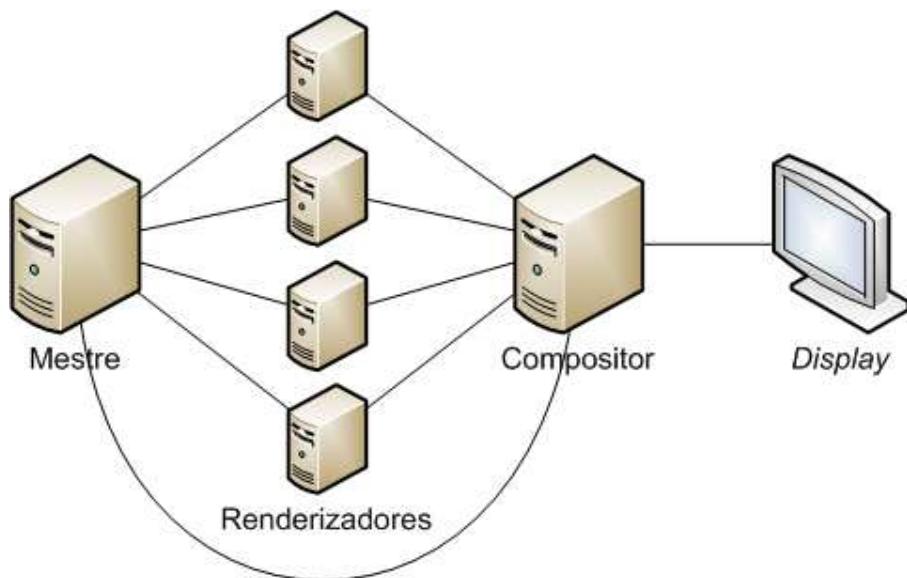
## 4.2. Componentes de Arquitetura

A arquitetura do XithCluster possui três componentes: o mestre, o renderizador e o compositor (Figura 4.1).

O componente mestre roda no nó do aglomerado que detém acesso às geometrias e aos outros recursos da cena. Ele captura das entradas do usuário e processa as simulações da aplicação. Ele é o responsável pelo gerenciamento de sessão, pela distribuição das primitivas (Seção 2.2.3) e pela manutenção da coerência visual (Seção 2.3.1).

Os componentes renderizadores rodam em um ou mais nós do aglomerado, responsáveis pela renderização das geometrias. O conjunto de geometrias sob sua responsabilidade é determinado ao início de uma sessão. Eles submetem as imagens renderizadas ao componente compositor. Antes de submetê-las eles podem, opcionalmente, comprimi-las.

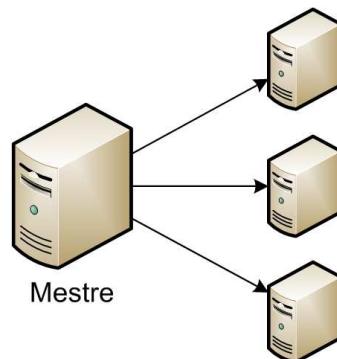
O componente compositor roda no nó do aglomerado responsável pela composição e a exibição da imagem final. A composição de imagens é baseada na implementação discutida no final da Seção 2.5, onde o *z-buffer* é usado na determinação da prioridade dos pixels.



**Figura 4.1. Os componentes de arquitetura do XithCluster**

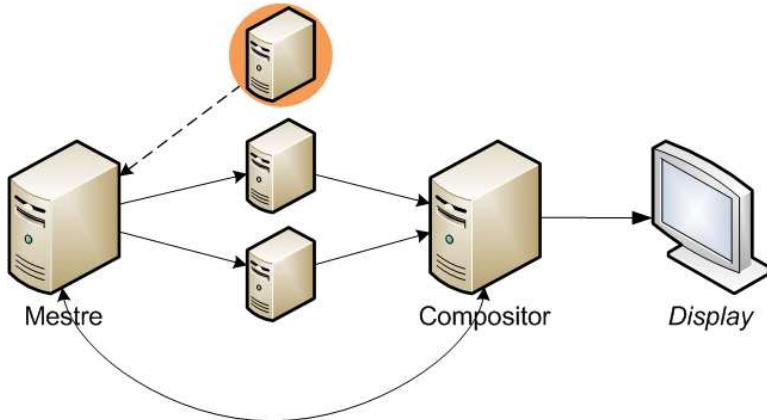
## 4.3. Sessão de Colaboração

A sessão de colaboração representa uma configuração dos nós do aglomerado que permita que ocorra a renderização distribuída. Ela é criada quando se tem pelo menos um componente de cada tipo rodando no aglomerado. Enquanto essa condição não for atingida, o XithCluster fica em estado ocioso e a renderização distribuída não é realizada (Figura 4.2). Quando um nó ingressa ou abandona o aglomerado, o XithCluster avalia os componentes presentes e cria ou destrói a sessão de colaboração.



**Figura 4.2. O XithCluster fica em estado ocioso quando não há pelo menos um componente de cada tipo rodando no aglomerado**

Uma nova sessão também é criada toda vez que um novo renderizador se conecta ao aglomerado, independente de já haver uma sessão funcionando. Isso ocorre para que o XithCluster redistribua as geometrias pelos nós do aglomerado. A essa funcionalidade dá-se o nome de ingresso tardio (Figura 4.2).



**Figura 4.3.** O XithCluster permite que um renderizador entre no aglomerado durante a renderização distribuída

## 4.4. Hierarquia de Nós do Grafo de Cena

O Xith3D dispõe de uma vasta hierarquia de classes de nós do grafo de cena. Apesar de todos os nós descendem de uma classe comum (Node), eles se ramificam em duas hierarquias distintas: os nós grupo e os nós folha. Os nós grupo são aqueles que podem possuir filhos e descendem da classe GroupNode (Figura 4.4). Os nós folha, como as geometrias, representam os pontos terminais da árvore e descendem da classe Leaf (Figura 4.5).

Como o Xith3D não provê nenhum suporte à serialização, foi necessária a análise cuidadosa de cada classe de nó para que ela pudesse ser distribuída. Infelizmente a quantidade de tipos de nós do Xith3D é suficientemente grande para que não seja viável analisá-los por completo dentro do escopo deste trabalho. Assim, o XithCluster decidiu suportar, inicialmente, apenas os nós grupo que representam transformações espaciais e os nós folha que representam as geometrias. No entanto, é possível dar suporte aos nós que

não foram suportados inicialmente através da extensão do mecanismo de serialização (Seção 4.3).

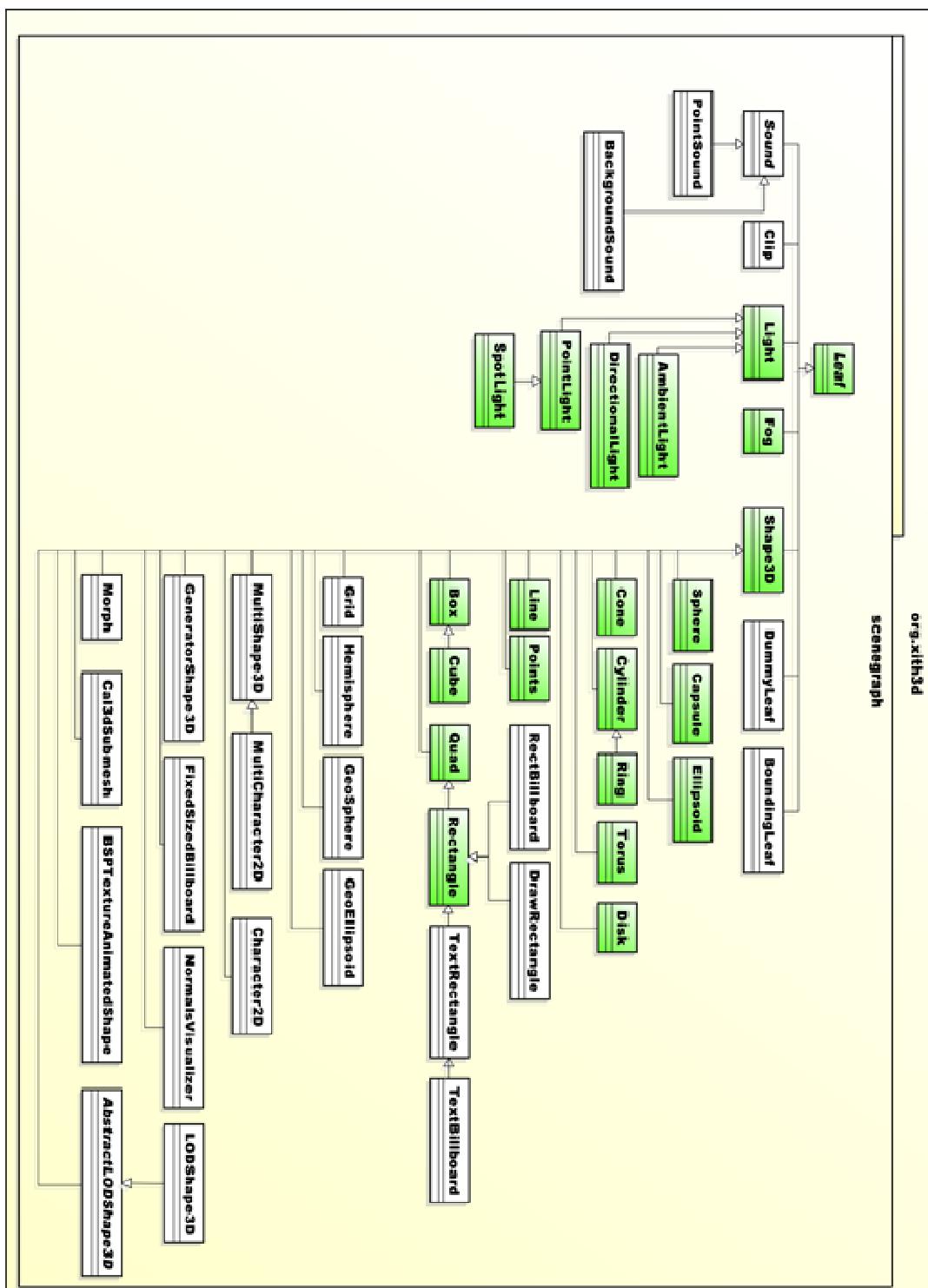


Figura 4.4. A hierarquia completa de nós folha do Xith3D, com os nós suportados pelo XithCluster pintados de verde

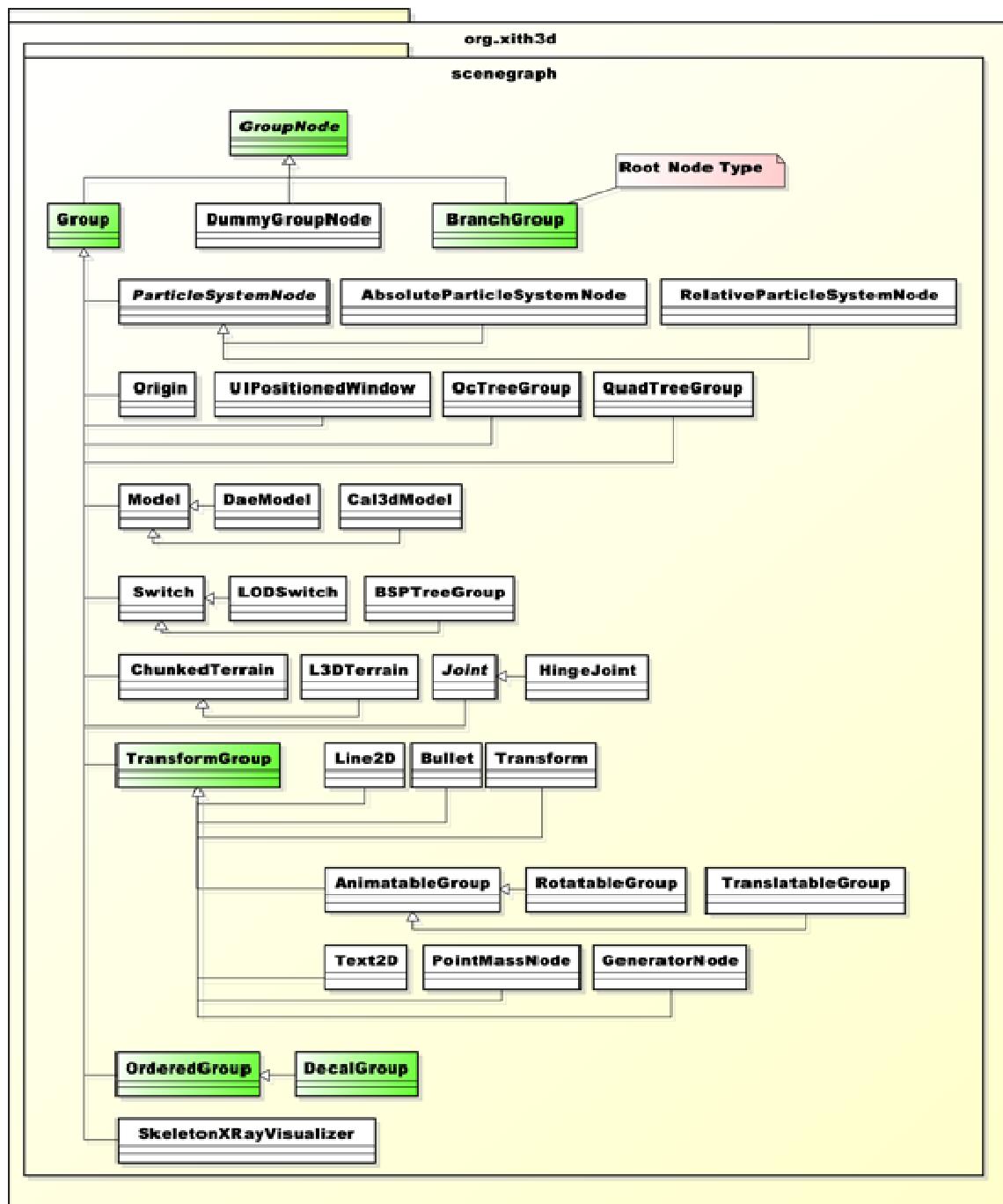


Figura 4.5. A hierarquia completa de nós grupo do Xith3D, com os nós suportados pelo XithCluster pintados de verde

O suporte à grande parte dos nós do Xith3D foi garantido sem a necessidade de estender suas classes. Contudo, os nós de geometria precisaram ser estendidos, pois seus parâmetros de construção<sup>6</sup> não são mantidos pelo Xith3D. O Xith3D não mantém os parâmetros de construção das geometrias pois mantém a malha de vértices, construída com eles através do processo de *tesselation*. Contudo, como a malha de vértices ocupa muito mais espaço que os parâmetros de construção, optou-se por usar os últimos durante a serialização.

## 4.5. Serialização de Dados

A serialização é um dos mecanismos mais importantes do XithCluster. Ela é responsável pela comunicação de alta performance, pois codifica os nós do grafo de cena em bytes usando menor número de informações possível. Ela usa as classes de manipulação de fluxo de dados DataInputStream e DataOutputStream para manipular os fluxos de bytes com segurança. Essas classes, pertencentes ao *framework* da linguagem Java, dispõem de métodos para a leitura e escrita de tipos primitivos (inteiros, decimais, booleanos, etc.) de forma independente de plataforma.

Na realidade, a serialização codifica em byte as estruturas de dados mais importantes que compõem um nó, e não o nó propriamente dito. A decodificação reconstrói um nó a partir das mesmas informações que foram usadas na codificação. Por exemplo, uma esfera pode ser codificada e decodificada a partir das informações de raio, posição e cor.

---

<sup>6</sup> Parâmetros utilizados como argumentos do construtor da classe.

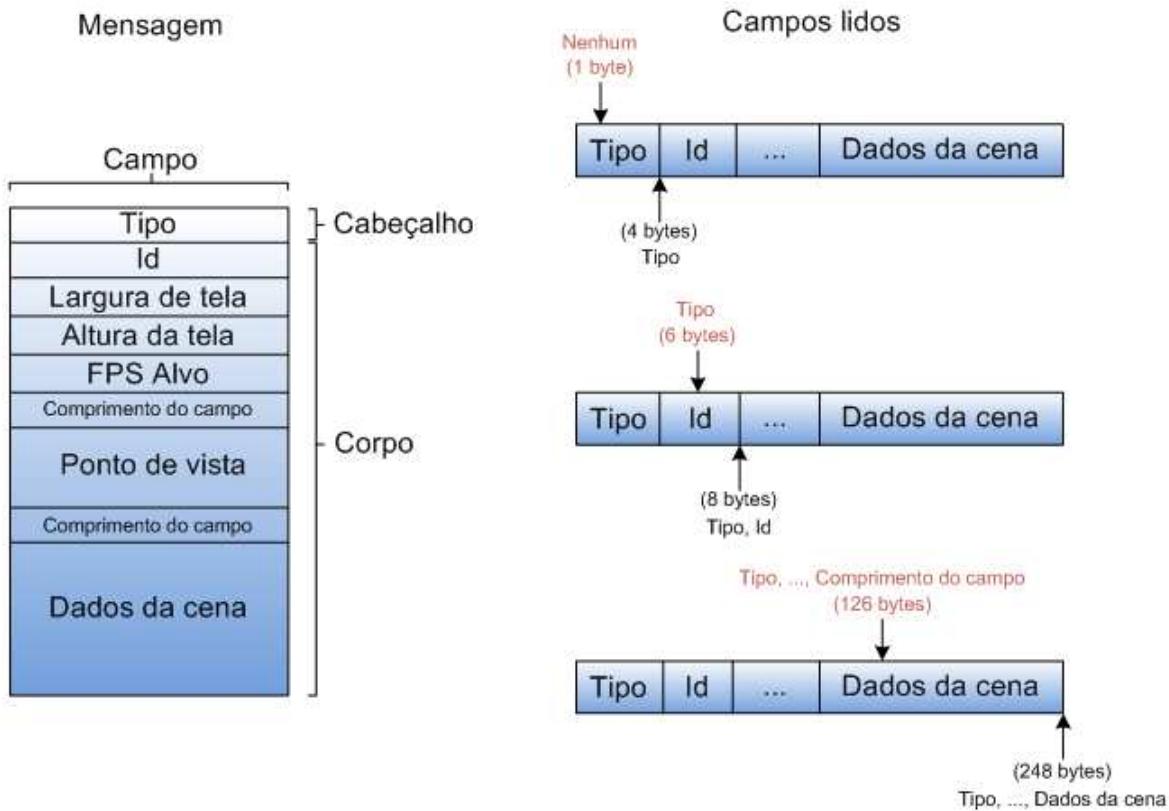
O mecanismo de serialização já provê meios para codificar e decodificar dezenas de estruturas de dados do Xith3D. Sendo assim, para dar suporte à serialização de novos tipos de nó, basta decidir quais as informações mais importantes do tipo de nó não suportado.

## 4.6. Camada de Comunicação

A camada de comunicação desenvolvida é baseada na troca de dados através de *sockets*. Ela usa a biblioteca xSockets, uma biblioteca orientada à conexões que, por sua vez, usa as novas classes de *E/S* da linguagem Java para acessar dados em memória com alto desempenho.

Ela é construída em dois níveis, que realizam tarefas distintas, porém complementares. Esses níveis são responsáveis por receber, processar e enviar mensagens de acordo com o contrato estabelecido pelo protocolo de comunicação descrito na Seção 4.6.1.

No nível inferior encontram-se as classes MessageBroker (HOHPE, 2003). Essas classes lêem as mensagens, garantindo que elas que não sejam processadas pela aplicação enquanto não tiverem sido completamente lidas. Isso é feito através do uso de *buffers* de leitura, durante a leitura dos dados do *socket*. Desta forma, as MessageBroker lêem os dados de maneira transacional, descartando uma mensagem se ela não for recebida por completo (Figura 4.6).



**Figura 4.6.** A leitura transacional determina que uma mensagem só seja processada caso todos os seus campos sejam lidos

No nível superior encontram-se as classes NetworkManager. Elas processam as mensagens lidas e enviam mensagens novas. As NetworkManager garantem que as mensagens sejam processadas na ordem em que foram enviadas, pois as ordena pelo horário de envio. Isso é possível pois como elas também são responsáveis por enviar as mensagens, elas as enviam juntamente com o horário do relógio global, sincronizado entre os nós do aglomerado a cada novo quadro.

Ao processar uma mensagem, as classes SceneManager são invocadas, pois elas detêm acesso ao grafo de cena e podem atualizá-lo com segurança. Essa indireção no acesso ao grafo de cena existe pois o Xith3D não é *thread-safe* e só pode ser acessado entre a renderização dos quadros.

#### **4.6.1. Protocolo de Comunicação**

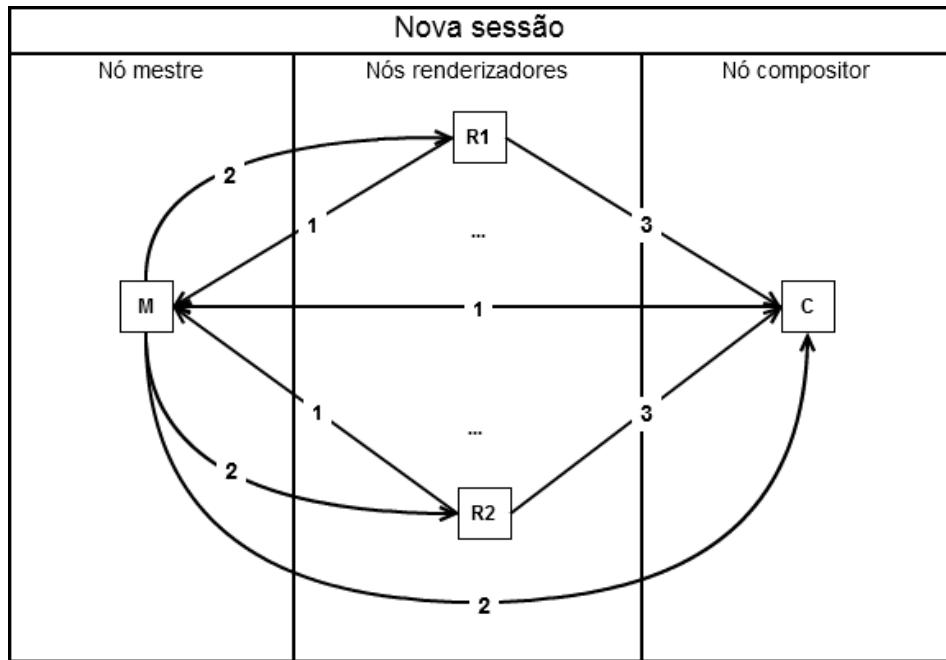
O protocolo de comunicação é a regra que rege a troca de mensagens entre os componentes da solução. Ele descreve a ordem e o conteúdo das mensagens trocadas entre os componentes do XithCluster. No modelo OSI (TANENBAUM, 2003), ele se situa na camada de aplicação. Ele usa os protocolos TCP/IP como protocolos de transporte e de rede. A escolha pela família TCP/IP é justificada pelo uso de conexões *full duplex*<sup>7</sup>.

São apresentados abaixo os diagramas de troca de mensagens durante a criação de uma nova sessão (Figura 4.7) e durante a renderização de um novo quadro (Figura 4.8). Nesses diagramas os componentes estão separados em raias e são identificados pelas iniciais M (de Mestre), R (de Renderizador) e C (de Compositor). Uma seta não pontilhada indica a troca de uma mensagem obrigatória, enquanto que uma seta pontilhada indica uma troca de mensagem opcional. O número nas setas indica a ordem em que as mensagens devem ser trocadas.

---

<sup>7</sup>

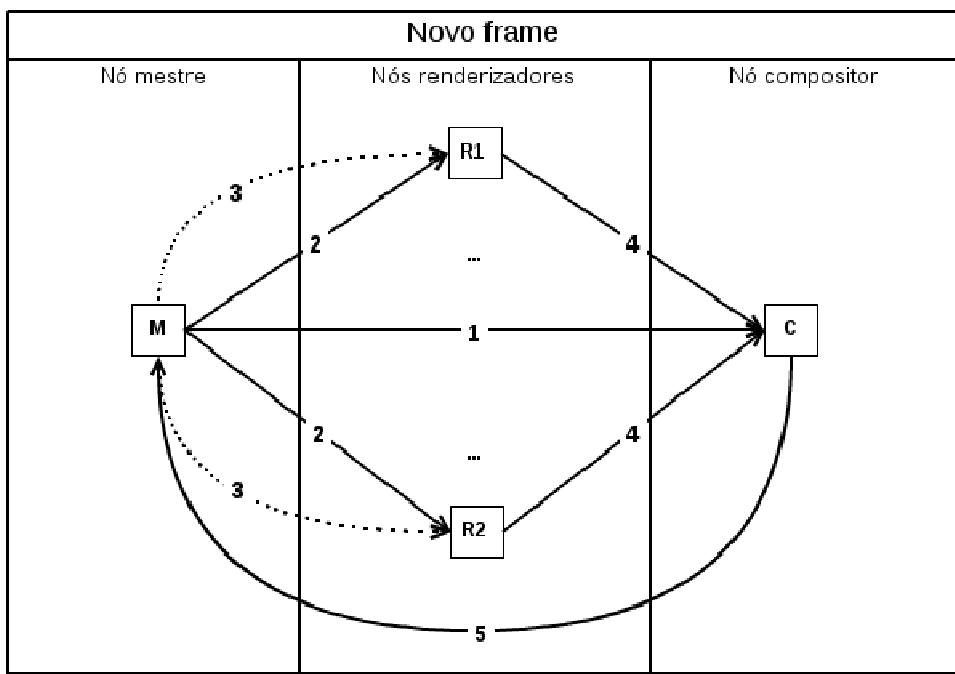
Uma conexão *full duplex*, em contraste com uma conexão *half duplex*, permite comunicações em ambas direções.



**Figura 4.7.** O diagrama ilustra a troca de mensagens durante a criação de uma nova sessão

Ordem	Descrição	Emissor	Receptor	Conteúdo
1	Solicitação de ingresso na sessão	Nós renderizadores e nó compositor	Nó mestre	Flag SYN do protocolo TCP
2	Divisão das primitivas e ajustes de aplicação	Nó mestre	Nós renderizadores e nó compositor	Dimensões de tela, taxa de atualização de quadros, ponto de vista, fontes de luz e geometrias da cena
3	Ordem de composição	Nós renderizadores	Nó compositor	Prioridade da imagem no processo de composição de geometrias transparentes

**Tabela 4.1.** Descrição das mensagens enumeradas no diagrama de nova sessão



**Figura 4.8.** O diagrama ilustra a troca de mensagens durante a sinalização de um novo quadro

Ordem	Descrição	Emissor	Receptor	Conteúdo
1	Solicitação de quantidade de quadros a descartar	Nó mestre	Nó compositor	Quantidade de quadros a descartar
2	Sinal de início de novo quadro	Nó mestre	Nós renderizadores	Nº. do quadro e horário correntes
3	Atualização de cena	Nó mestre	Nós renderizadores	Alterações nos nós
4	Imagen rasterizada	Nós renderizadores	Nó compositor	Nº. do quadro corrente, imagem e método de compressão
5	Sinal de fim de quadro	Nó compositor	Nó mestre	Nº. do quadro corrente

**Tabela 4.2.** Descrição das mensagens enumeradas no diagrama de novo quadro

## 4.7. Extensão ao Xith3D

O Xith3D é uma biblioteca que, além de prover uma implementação de grafo de cena, faz interface com importantes subsistemas de uma aplicação de RV. O Xith3D disponibiliza diversos pontos de extensão, principalmente através de *Listeners*<sup>8</sup> e *Callbacks*<sup>9</sup>. O XithCluster, contudo, usa não só esses pontos de extensão, mas, em alguns casos, trabalha intimamente com estruturas de dados do Xith3D através da API de reflexão da linguagem Java.

O Xith3D é fácil de usar, pois precisa de pouco código para criar uma aplicação de RV funcional. Sua API faz bom uso do encapsulamento preconizado pelo paradigma OO. Durante o desenvolvimento do XithCluster foi mantida a mesma preocupação com o encapsulamento e a facilidade de uso do Xith3D.

Através da classe SampleApplication é permitido ao usuário do XithCluster escolher entre a renderização normal ou a renderização distribuída. Essa classe torna transparente o conhecimento dos detalhes da distribuição da renderização (como código de rede, serialização, sincronismo, etc.), o que reduz a curva de aprendizado da solução e facilita o monitoramento de erros.

### 4.7.1 Distribuição da Renderização

---

<sup>8</sup> Listener é um nome alternativo para o padrão *Observer* (GAMMA, 1995).

<sup>9</sup> Na computação, Callback é o nome de uma técnica em que se passa uma função como argumento para outra função.

O primeiro ponto de extensão explorado pelo XithCluster é a classe RenderLoop. Essa classe implementa o laço principal da aplicação, onde são feitas chamadas às funções de atualização do grafo de cena, leitura dos dispositivos de E/S e, o mais importante, renderização. A classe DistributedRenderLoop, que estende a classe RenderLoop, intercepta essas chamadas e as distribui pelos nós do aglomerado. Ela também implementa o mecanismo *frame-lock* (Seção 2.3.1), fazendo que as chamadas aos nós do aglomerado aconteçam em ordem. O Código 4.1 apresenta, em pseudocódigo, uma versão simplificada do algoritmo implementado pela classe DistributedRenderLoop.

```
loopIteration(delta_time) {
    if hasNoSession() return;
    if canStartANewFrame() {
        for each renderer in renderers {
            startNewFrame(renderer)
        }
    }
    processInput(delta_time)
    updateSceneGraph(delta_time)
    sendSceneUpdates()
}
```

**Código 4.1** Uma versão simplificada do algoritmo implementado pela classe **DistributedRenderLoop**

## 4.7.2 Atualização de Estado dos Nós do Grafo de Cena

O segundo ponto de extensão explorado pelo XithCluster é a interface SceneGraphModificationListener. Através dela é possível observar as alterações que ocorrem nos nós do grafo de cena. A classe UpdateManager, que implementa a interface SceneGraphModificationListener, registra as alterações no grafo de cena através de objetos

PendingUpdate. Os PendingUpdate permitem que essas alterações possam ser repetidas em outro nó grafo. Por exemplo, ao rotacionar um nó do grafo em 45° no eixo X, a classe UpdateManager cria um PendingUpdate capaz de repetir essa mesma rotação em um outro nó do grafo.

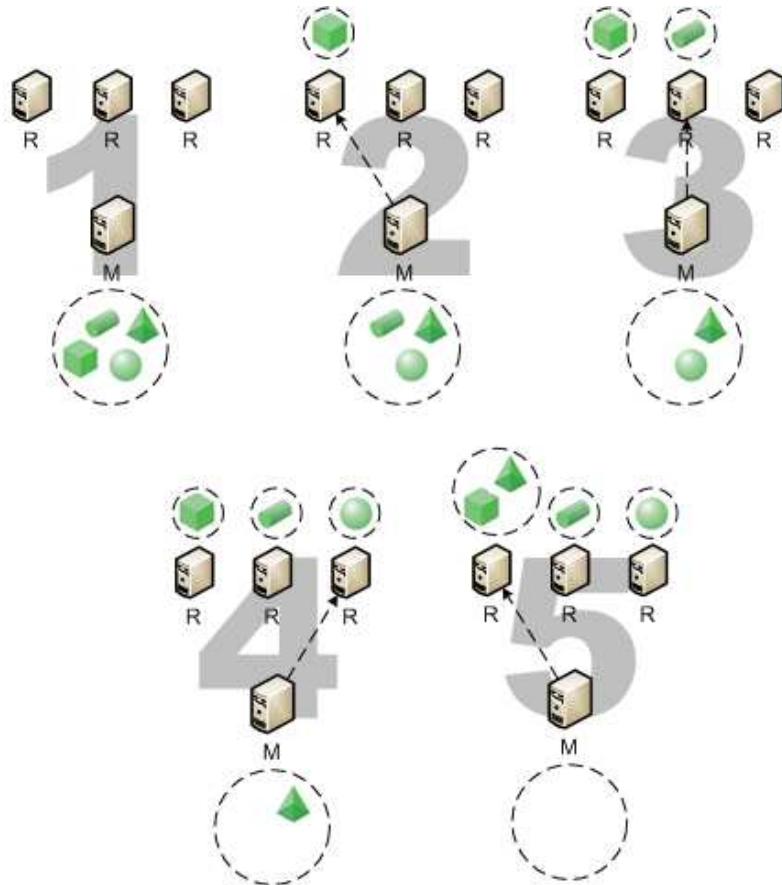
Ao final do laço principal da aplicação (Código 4.1), um PendingUpdate é enviado ao renderizador que detêm uma cópia do nó alterado. Por exemplo, se o renderizador A detém uma cópia do nó do grafo B, o PendingUpdate criado para o nó do grafo B seja enviado para o renderizador A.

Ao receber um PendingUpdate, um renderizador o executa na sua cópia local do nó do grafo referenciado pelo PendingUpdate. Dessa forma, as alterações ocorridas no mestre são repercutidas para os renderizadores, e a integridade dos dados da cena é mantida.

### 4.7.3 Distribuição da Geometria e Replicação dos Nós do Grafo de Cena

O terceiro ponto de extensão explorado pelo XithCluster é a interface TraversalCallback. Essa interface permite visitar os nós do grafo de cena recursivamente. A visitação dos nós do grafo é crucial para a implementação de dois mecanismos: a distribuição de primitivas e a replicação dos nós do grafo.

A distribuição de primitivas é realizada no momento da criação de uma sessão de colaboração (Seção 4.3). O XithCluster permite que a estratégia de distribuição que será usada seja definida programaticamente. Uma estratégia de distribuição deve garantir que cada geometria da cena seja destinada a um, e somente um, renderizador. O XithCluster provê uma estratégia padrão de distribuição, que usa o algoritmo *Round Robin* para distribuir as geometrias de maneira balanceada. Essa estratégia reúne todas as geometrias da cena e, uma a uma, vai distribuindo-as a um renderizador (Figura 4.9). O XithCluster possibilita o desenvolvimento de novas estratégias de distribuição através da interface DistributionStrategy.



**Figura 4.9.** A estratégia padrão de distribuição garante que todas as geometrias da cena sejam distribuídas pelos renderizadores

A replicação dos nós do grafo é feita no momento da serialização (Seção 4.5). Durante a serialização, cada geometria a ser serializada é enviada para a classe NodePathReplicator, que replica o caminho da geometria. O NodePathReplicado devolve, então, o caminho da geometria, para que ele, e não a geometria, seja serializado e enviado para um renderizador (Figura 4.10). Isso é necessário por causa da herança de atributos que pode existir entre os nós do grafo (Seção 2.1). Se um geometria for enviada sozinha, ela pode ficar inconsistente, já que algumas de suas informações podem estar armazenadas em seus nós ancestrais.

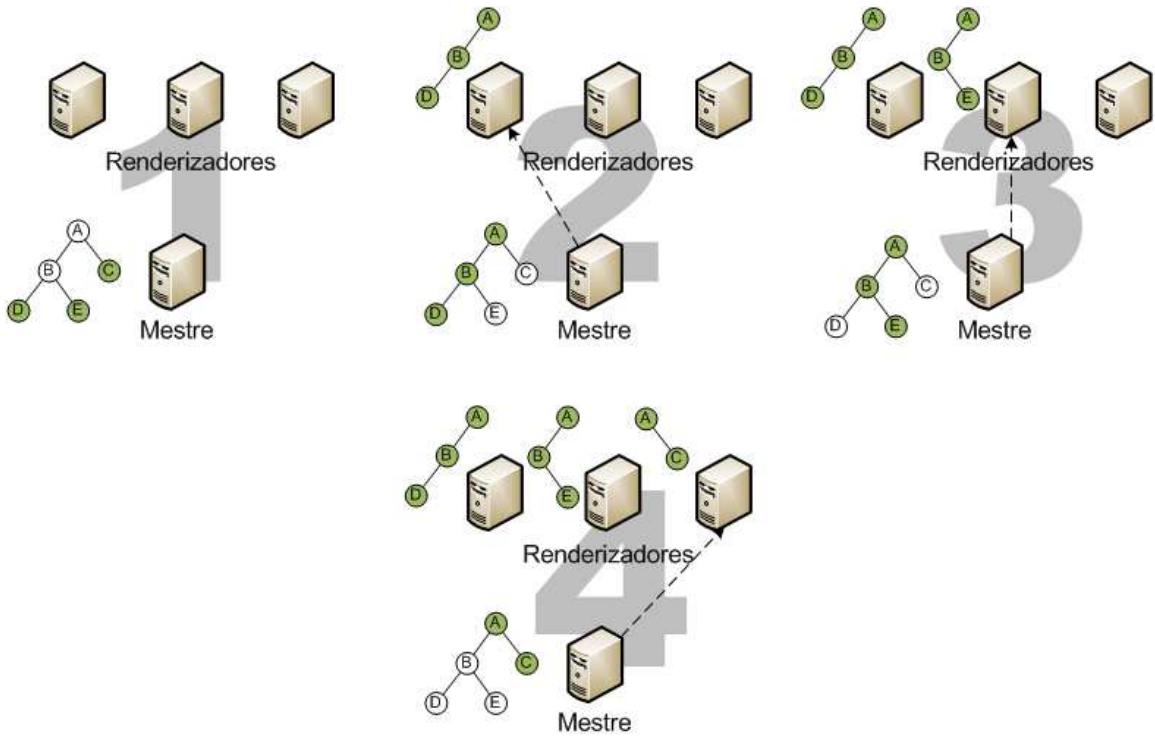


Figura 4.10. Cada geometria tem seu caminho replicado e enviado para um renderizador

## 4.8. Aplicação Renderizadora

O componente renderizador é implementado pela aplicação *rendererApp*. Essa aplicação deve rodar em pelo menos um nó para que o XithCluster inicie, e em inúmeros nós para que se distribua efetivamente o processo de renderização. Cada instância da aplicação pode ser configurada individualmente através de parâmetros de linha de comando ou de arquivo de configuração (Tabela 4.4).

A *rendererApp* renderiza as geometrias recebidas do mestre durante a criação da sessão de colaboração. O momento da renderização é controlado pelo sinal de novo quadro transmitido pelo componente Mestre periodicamente.

Para que sejam enviados os dados das imagens ao componente Compositor, os *buffers* de cor, transparência e profundidade são lidos da memória gráfica a cada quadro renderizado. Isso acarreta em uma perda significativa de performance (Seção 5.3.2).

A *rendererApp* pode ser configurada para realizar, opcionalmente, a compressão dos dados da imagem. Assim, o nome do método de compressão usado é transmitido junto com os dados da imagem para que o compositor seja capaz de descomprimi-los corretamente. Sugere-se avaliar o custo da execução do algoritmo de compressão, para que a taxa de geração de quadros não seja comprometida.

Por fim, a *rendererApp* repercute as alterações nas geometrias que estão sob sua responsabilidade, de acordo com as atualizações de estado dos nós do grafo, como explicado na Seção 4.7.2.

Propriedade	Linha de comando	Arquivo texto	Valores possíveis	Valor padrão
Endereço do nó mestre	--hostname	masterNode.hostname	Qualquer	localhost
Porta do nó mestre	--port	masterNode.port	[0-65532]	10000
Método de compressão	--compression.method	compression.method	[PNG   NONE]	NONE
Ordem na composição	--order	composition.order	Número inteiro	0

Tabela 4.4. Parâmetros de configuração da aplicação *rendererApp*

## 4.9. Aplicação Compositora

O componente compositor é implementado pela aplicação *composerApp*. É necessário executar essa aplicação em um nó do aglomerado para que o XithCluster inicie. De maneira análoga à *rendererApp*, a *composerApp* pode ser configurada por parâmetros de linha de comando e por arquivo de propriedades (Tabela 4.5).

A *composerApp* recebe os *buffers* de cor, transparência e profundidade enviados pelos renderizadores para compor a imagem final a ser exibida. Note que ela só inicia a composição da imagem final após receber todas as imagens dos renderizadores.

A composição de imagens usa o *z-buffer* junto com a ordem de composição a fim de atenuar o problema de transparência explicado no final da Seção 2.5. No entanto, para que isso funcione é necessário isolar as primitivas transparentes em um único renderizador, configurando sua ordem de composição como a última (Figuras 4.11 e 4.12).

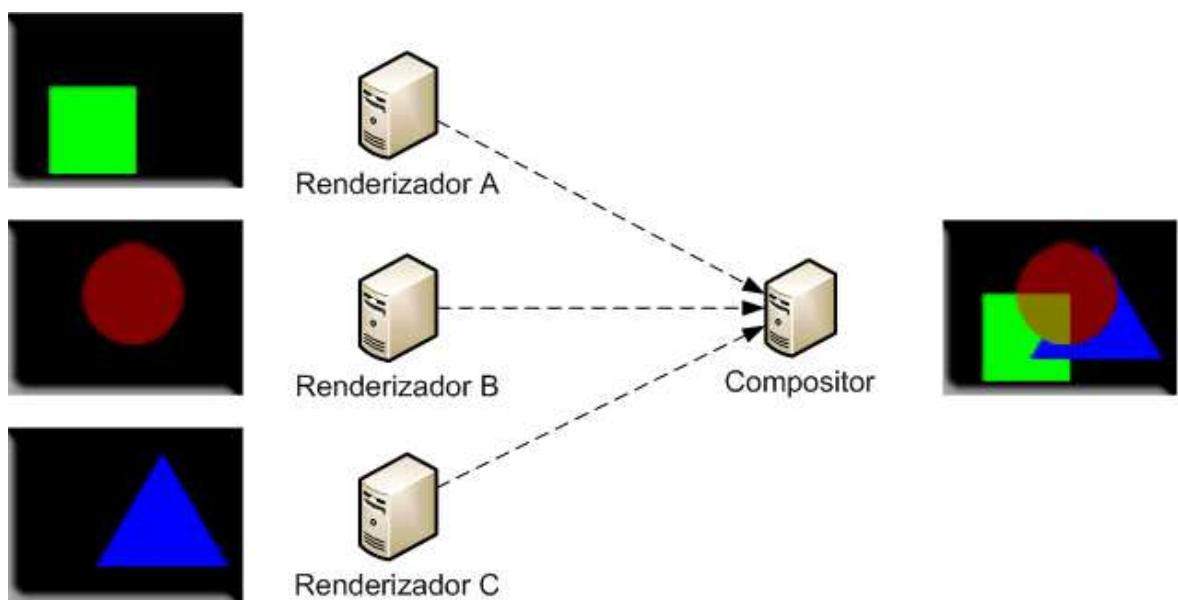


Figura 4.11. Composição com erro pois a ordem de composição do renderizador com a primitiva transparente não foi a última

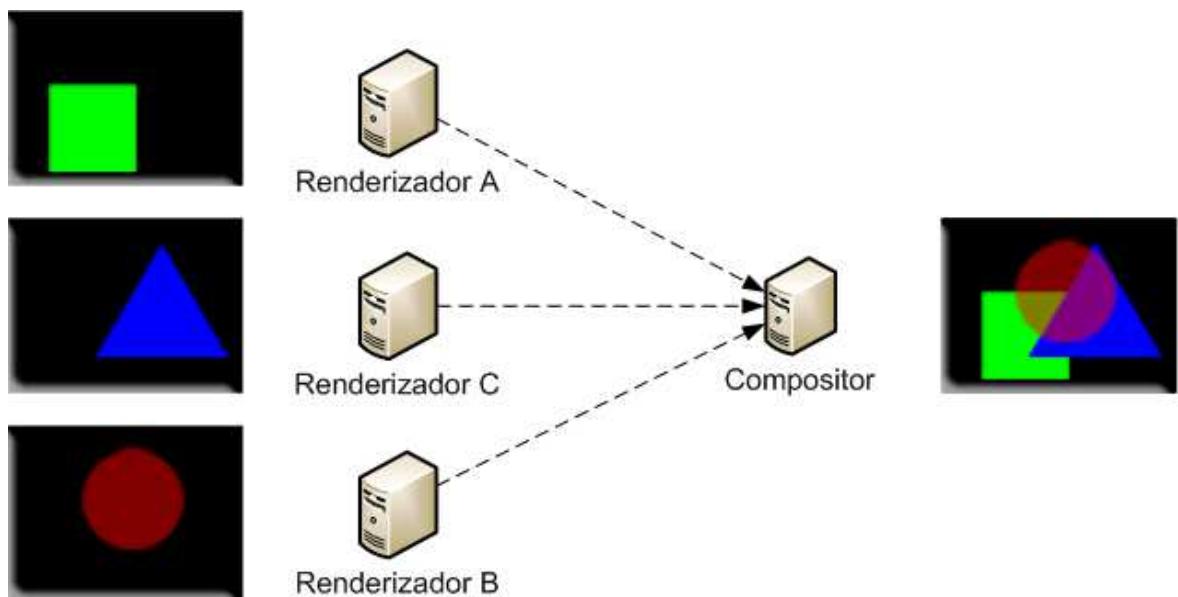


Figura 4.12. Composição sem erro pois a ordem de composição do renderizador com a primitiva transparente foi usada a última

A *composerApp* também realiza a descompressão dos dados da imagem, caso tenha sido enviado o nome do método de compressão junto com os dados da imagem.

Propriedade	Linha de comando	Arquivo texto	Valores possíveis	Valor padrão
Endereço do nó mestre	--hostname	masterNode.hostname	Qualquer	localhost
Porta do nó mestre	--port	masterNode.port	[0-65532]	20000
Razão de quadros alvo	--framerate	target.frameRate	Número de quadros por segundos	120

Tabela 4.5. Parâmetros de configuração da aplicação *composerApp*

## 4.10. Dependências de Plataforma

Ao desenvolver uma aplicação com o XithCluster, faz-se necessário o uso de bibliotecas de terceiros que requerem a instalação de código binário dependente de plataforma. Apesar do XithCluster disponibilizar o código binário para três plataformas: Linux, Windows e MacOS, ela foi testada apenas nas plataformas Linux (Ubuntu 10.10) e Windows (Versões Vista e 7).

## **5. Experimentos e Resultados**

Essa Seção apresenta a análise dos resultados dos testes de escalabilidade realizados sobre a implementação desenvolvida. Escalabilidade de carga é a habilidade que um sistema tem de executar de maneira estável em diferentes níveis de carga (BONDI, 2000). Os resultados obtidos constatam os impactos do aumento de carga sobre certas características do XithCluster, dando subsídio para as conclusões apresentadas na Seção 6.

Note que os testes de carga foram realizado em uma única máquina. Não foi possível realizá-los em um ambiente distribuído por não haver um aglomerado adequado disponível. No entanto, a validade dos testes não foi comprometida, pois eles são de natureza comparativa e foram executados criteriosamente sob as mesmas condições.

### **5.1. Métricas do teste**

Em cada um dos três teste realizados, uma cena diferente foi renderizada. Cada cena renderizada teve um nível de carga diferente, medido na forma de número total de polígonos: 100, 1.000 e 10.000 polígonos. Cada teste foi configurado para terminar assim que se atingisse 2000 quadros computados. Os polígonos exibidos não foram texturizados e durante toda a execução dos testes permaneceram completamente visíveis (sem oclusão). Os testes foram executados na resolução 1024x768 e com taxa de atualização fixada em 80 quadros por segundo.

## 5.2. Coleta de dados

A coleta de dados foi feita através da ferramenta de JVisualVM (Figura 5.1). O JVisualVM é um *profiler* gratuito, de licenciamento irrestrito que é distribuído juntamente com kit de desenvolvimento da linguagem Java (JDK).

O JVisualVM fornece dados na forma de *tempo total de execução* (em milissegundos) e *número de invocações*, para cada método que é invocado durante a execução da aplicação. Contudo, na análise realizada, esses dados foram transformados em *tempos médios de execução* (tempo total de execução / número de invocações).

Os métodos invocados foram agrupados em operações de nível mais alto. O tempo de execução das operações de nível mais alto foi obtido através da soma dos tempos médios de execução dos métodos agrupados. As operações escolhidas foram aquelas que poderiam ser influenciadas pelo aumento de carga.

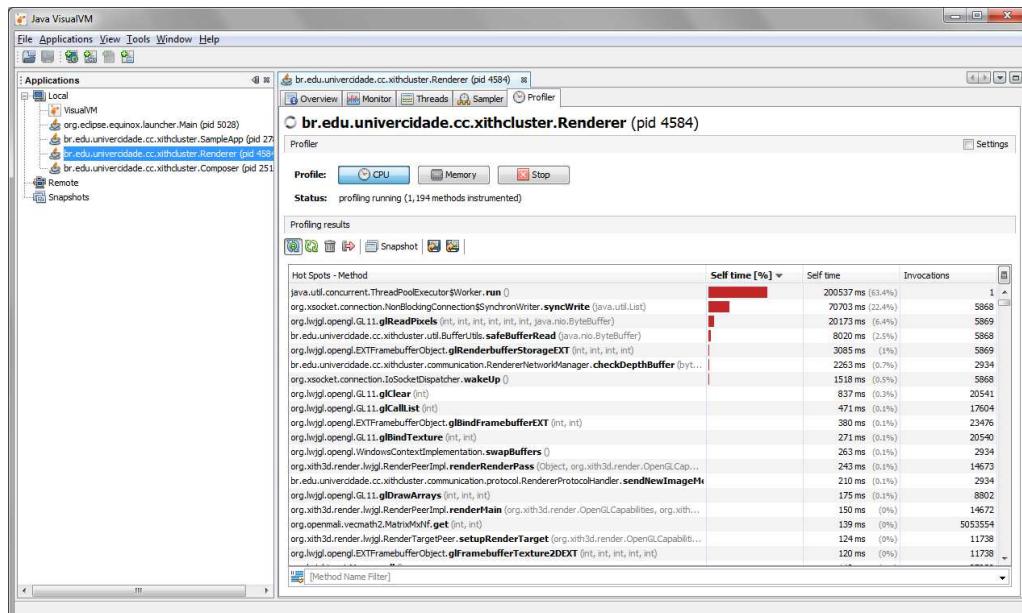


Figura 5.1. A coleta de dados feita através da ferramenta JVisualVM

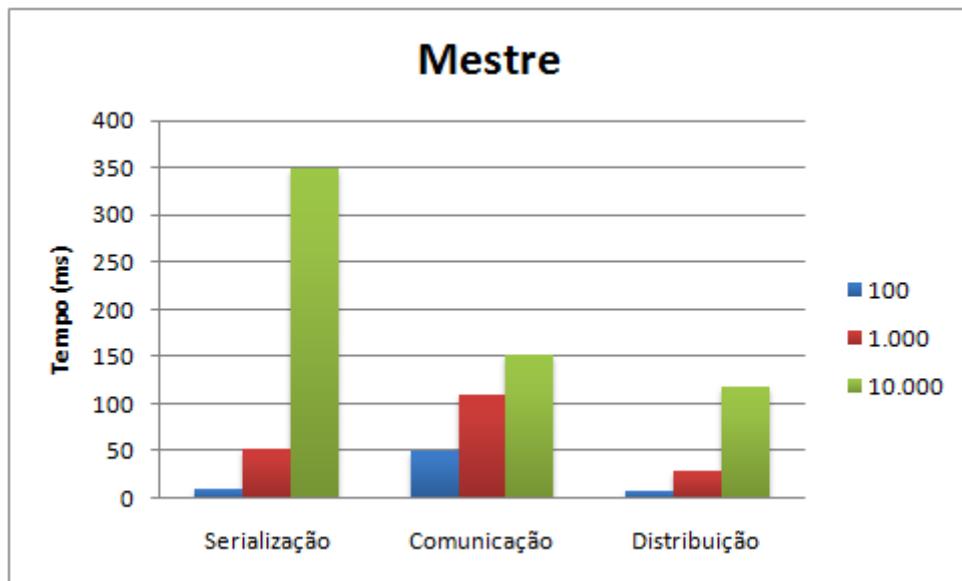
## 5.3. Análise dos resultados

Os gráficos de barra a seguir, mostram os tempos de execução, em milissegundos, para cada componente do XithCluster. Em cada gráfico, as operações foram separadas de acordo com níveis de carga. As legendas de cor relacionam um tempo de execução com um determinado nível de carga. Por exemplo: Na Figura 5.2, a barra verde acima de "Serialização" representa o tempo de execução da operação "Serialização" com o nível de carga em 10.000 polígonos.

### 5.3.1. Mestre

As operações mais importantes do componente Mestre são: serialização, comunicação e distribuição. A operação de serialização consiste no conjunto de métodos que codificam os nós do grafo a fim enviá-los pela rede (Seção 4.5). A operação de comunicação consiste no conjunto de métodos que recebem e enviam mensagens pela rede (Seção 4.6). Por fim, a operação de distribuição consiste no método que realiza a distribuição das geometrias, que é selecionado durante a configuração do XithCluster e que, neste caso particular, é o método *distribute(..)* da classe RoundRobinDistribution.

A partir da análise do gráfico conclui-se que todas as operações são influenciadas pelo aumento da carga (Figura 5.2). Nota-se, também, uma maior influência na operação de serialização. Essa maior influência na operação de serialização é atribuída à necessidade de alocação de grandes áreas de memória dinâmica, principalmente na criação dos vetores de *byte* usados para o armazenamento dos dados codificados.



**Figura 5.2. Tempo médio de execução das operações do mestre**

### 5.3.2. Renderizador

As operações mais importantes do componente Renderizador são: renderização, leitura da memória gráfica e deserialização. A operação de renderização consiste nas chamadas às funções da biblioteca gráfica OpenGL, feitas internamente pelo Xith3D. A operação de leitura da memória gráfica consiste nas chamadas à biblioteca gráfica OpenGL que possibilitam a leitura da memória gráfica. A operação de deserialização consiste no conjunto de métodos que recria os nós do grafo, decodificando os dados enviados pela operação de serialização (Seção 5.3.1).

A partir da análise do gráfico conclui-se que todas as operações são influenciadas pelo aumento da carga (Figura 5.3). Pode-se notar, também, que o aumento do tempo de execução da operação de deserialização é proporcional ao aumento do tempo de execução de sua operação inversa, a serialização, realizada pelo componente Mestre (Seção 5.3.1). Essa noção, no entanto, é prejudicada pelas operações de renderização e leitura da

memória gráfica, pois elas possuem um tempo de execução muito maior que a operação de deserialização.

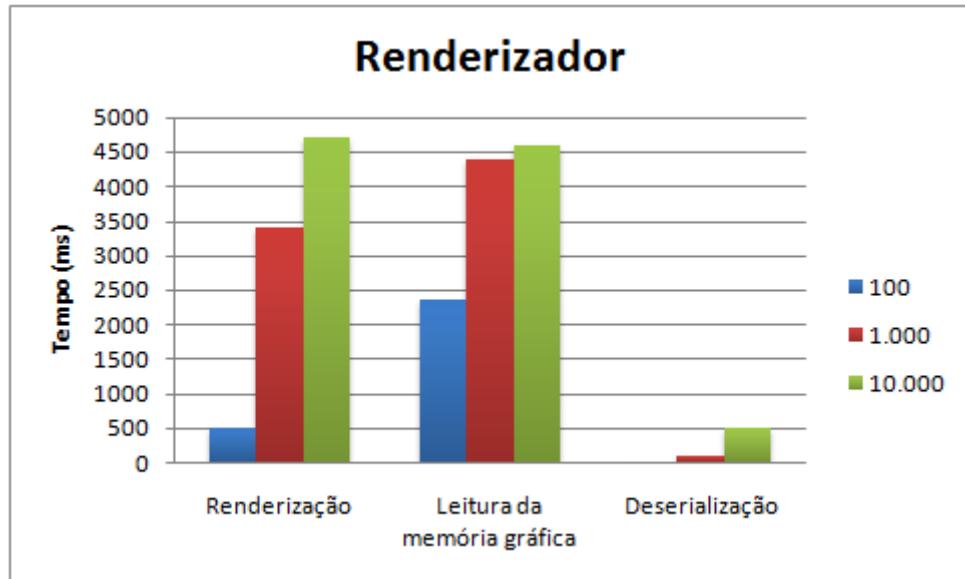


Figura 5.3. Tempo médio de execução das operações do renderizador

### 5.3.3. Composer

As operações mais importantes do componente Composer são: escrita na memória gráfica, composição e comunicação. A operação de escrita na memória gráfica consiste na chamada à API gráfica do Java que permite definir os pixels da tela. A operação de composição consiste no método que realiza a composição de imagens, que é selecionado durante a configuração do XithCluster e que, neste caso particular, é o método *compose(..)* da classe ZBufferAndImageOrderComposition. Por fim, a operação de comunicação consiste no conjunto de métodos que recebem e enviam mensagens pela rede (Seção 4.6).

A partir da análise do gráfico conclui-se que as operações do compositor não são influenciadas diretamente pelo aumento da carga (Figura 5.4). Nota-se, entretanto, que há uma relação entre o desempenho das operações e a resolução de tela, por causa do aumento do tamanho dos *buffers* de cor e de profundidade.

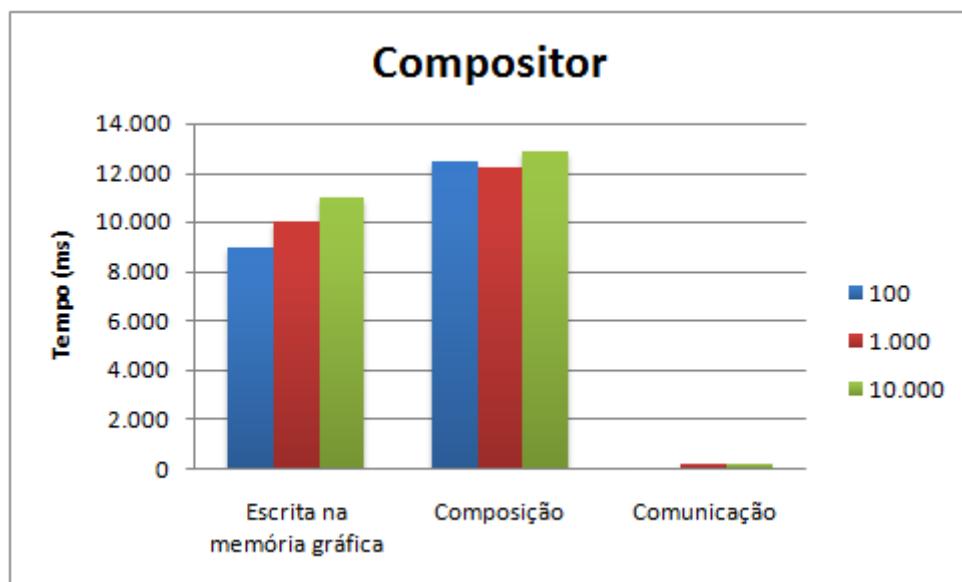


Figura 5.4. Tempo médio de execução das operações do compositor

## 6. Conclusões

Este trabalho demonstra que é possível distribuir o processo de renderização através do método *Sort-Last*. A implementação foi considerada bem sucedida, pois atendeu à proposta inicial de maneira escalável, obtendo ganhos mensuráveis em performance.

A escolha pela linguagem Java mostrou-se útil no suporte a múltiplas plataformas. Adicionalmente, constatou-se que o uso de ferramentas gratuitas não representou empecilho algum no desenvolvimento da solução, e permitiu com que se mantivesse o baixo custo proposto inicialmente.

Através da análise dos gráficos (Seção 5), concluiu-se que a baixa razão de quadros evidencia que a implementação do *Sort-Last* ainda é significativamente lenta, mesmo diante da evolução do hardware gráfico. Notou-se que a leitura e a escrita da memória gráfica (Figuras 5.2 e 5.3) ainda apresentam um empecilho para o envio das imagens renderizadas pela rede. Acredita-se, no entanto, que o uso de shaders (ROST, 2006) e de extensões do OpenGL<sup>10</sup> podem otimizar as operações em memória gráfica, melhorando a qualidade da solução. Sendo assim, sugere-se que qualquer continuação deste trabalho procure atender, primeiramente, à questão das operações em memória.

---

<sup>10</sup> Um exemplo de uso é a extensão Pixel Buffer Object (PBO), que permite um acesso mais rápido de buffers em memória gráfica (AHN, 2007).

## **6.1. Contribuições**

Este trabalho realiza algumas pequenas contribuições ao desenvolvimento de sistemas de RV distribuídos. Muitas funcionalidades desenvolvidas não foram baseadas em nenhuma implementação estudada. Por exemplo, a atenuação de problemas com transparências, o mecanismo de *frame-lock*, o ingresso tardio, etc.

O uso da linguagem Java também pode ser considerado uma contribuição, tendo em vista suas facilidades em comparação a linguagens nativas: gerenciamento de memória automático, reflexividade, segurança, etc.

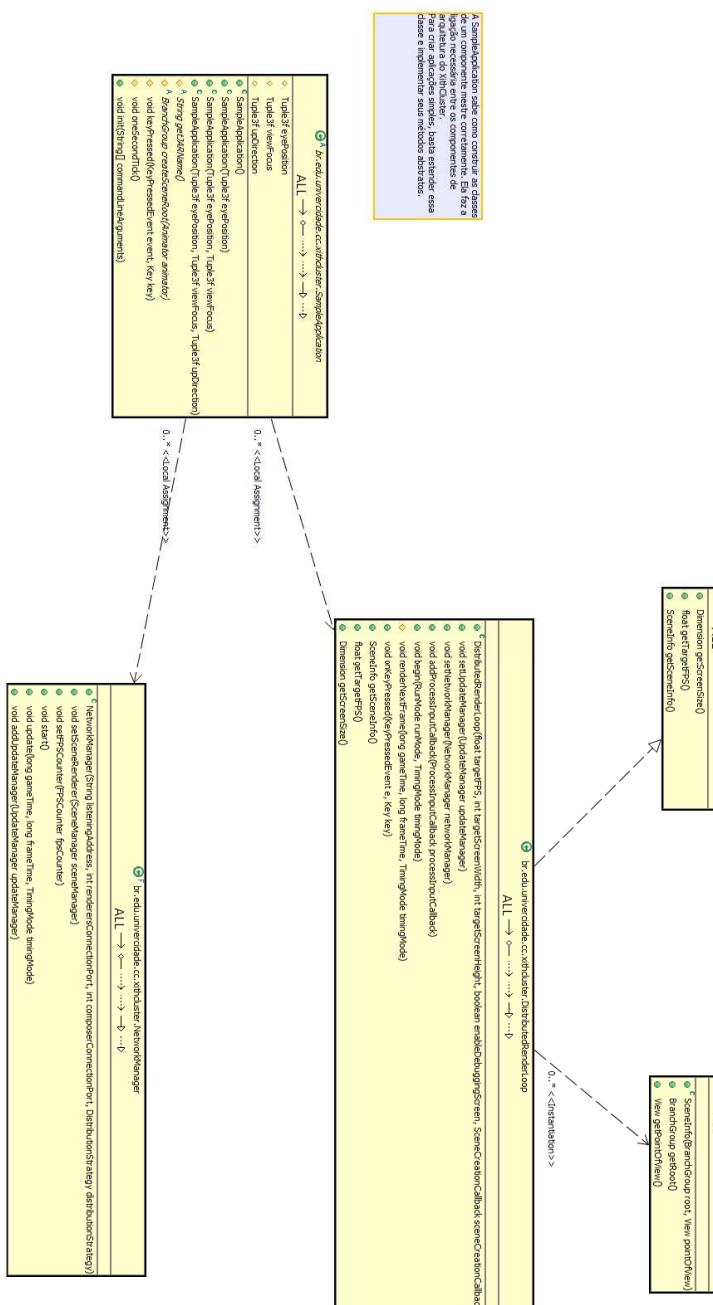
## **6.2. Trabalhos Futuros**

Abaixo encontram-se algumas sugestões para a continuação deste trabalho:

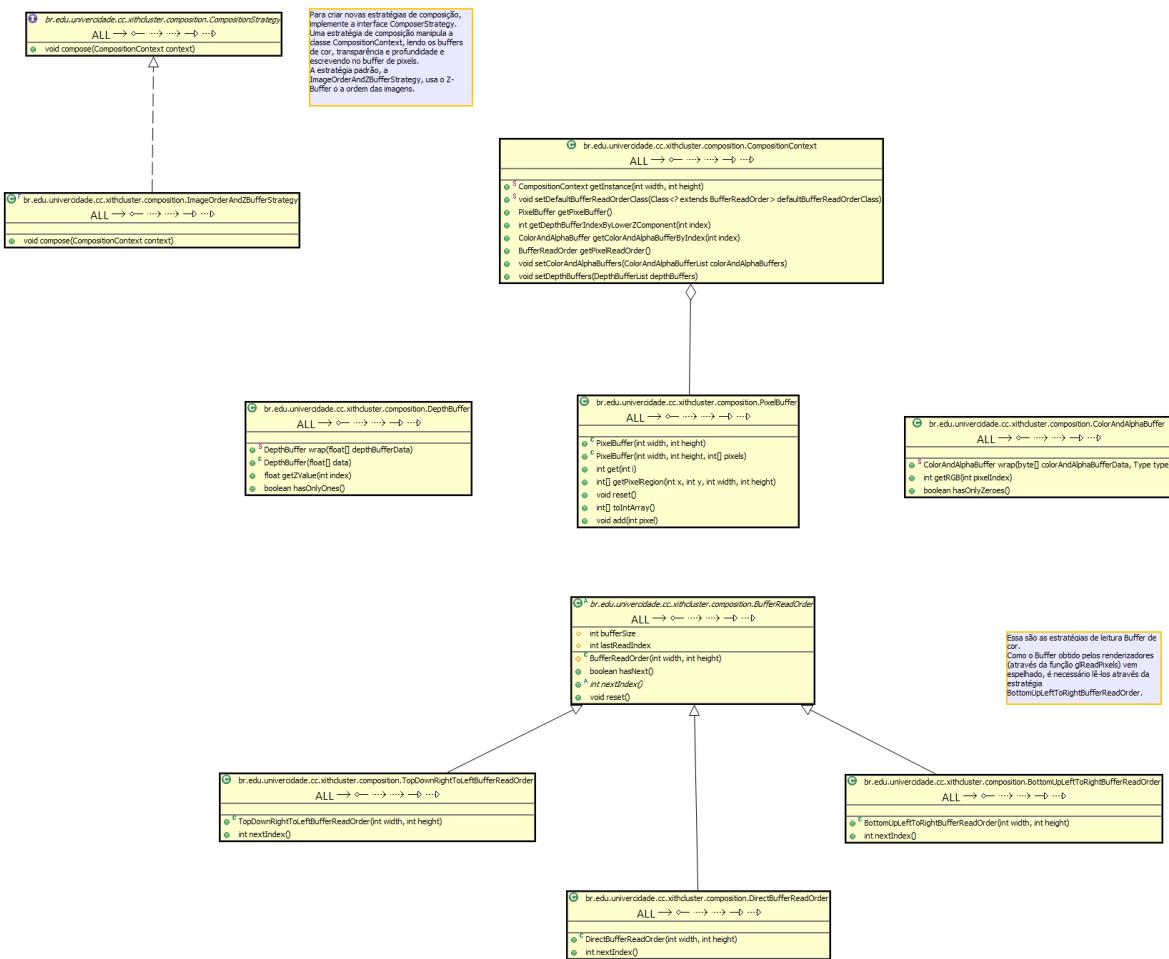
1. Otimizar a leitura e a escrita da memória gráfica.
2. Dar suporte a distribuição de shaders.
3. Dar suporte a novos métodos de compressão para imagens.
4. Dar suporte a novos nós da hierarquia do Xith3D.
5. Dar suporte a renderização com múltiplas passagens (ex.: skybox).
6. Dar suporte ao paralelismo de tarefas (ex.: simulações físicas, inteligência artificial, etc.).

# Apêndice A

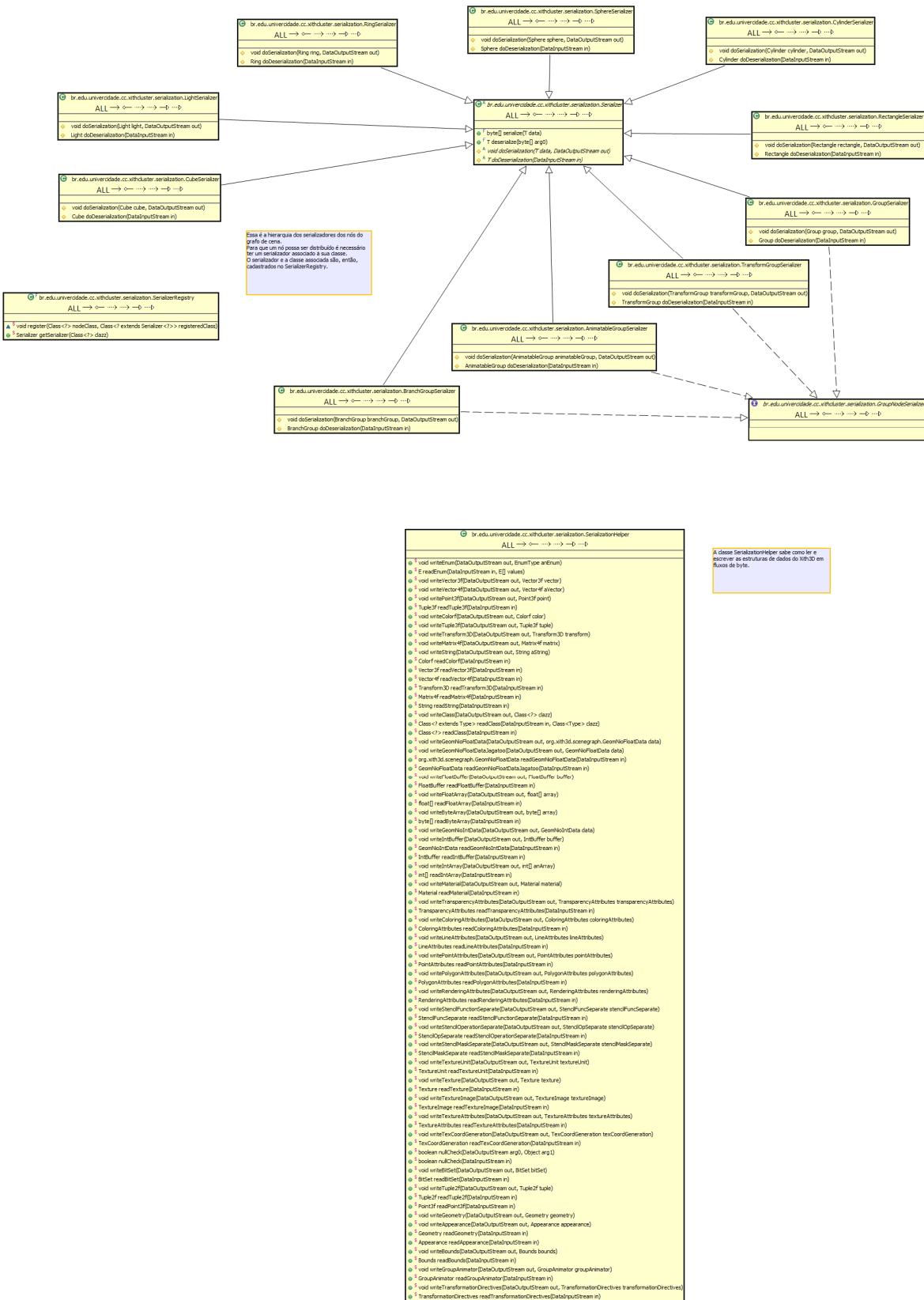
Este apêndice apresenta 7 diagramas UML da solução: 4 de classe e 3 de sequência.



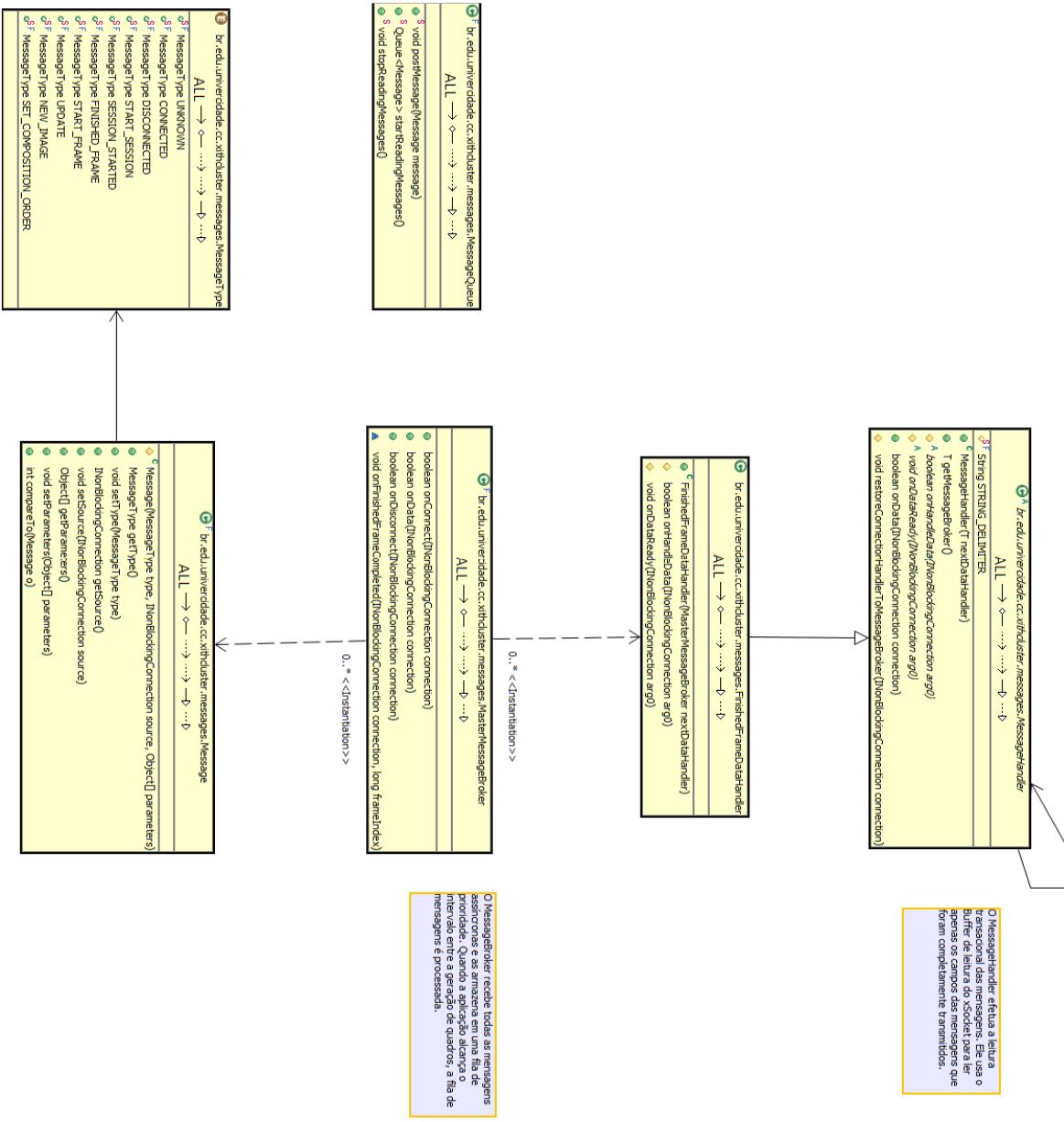
**Figura I - Diagrama de classes geral**



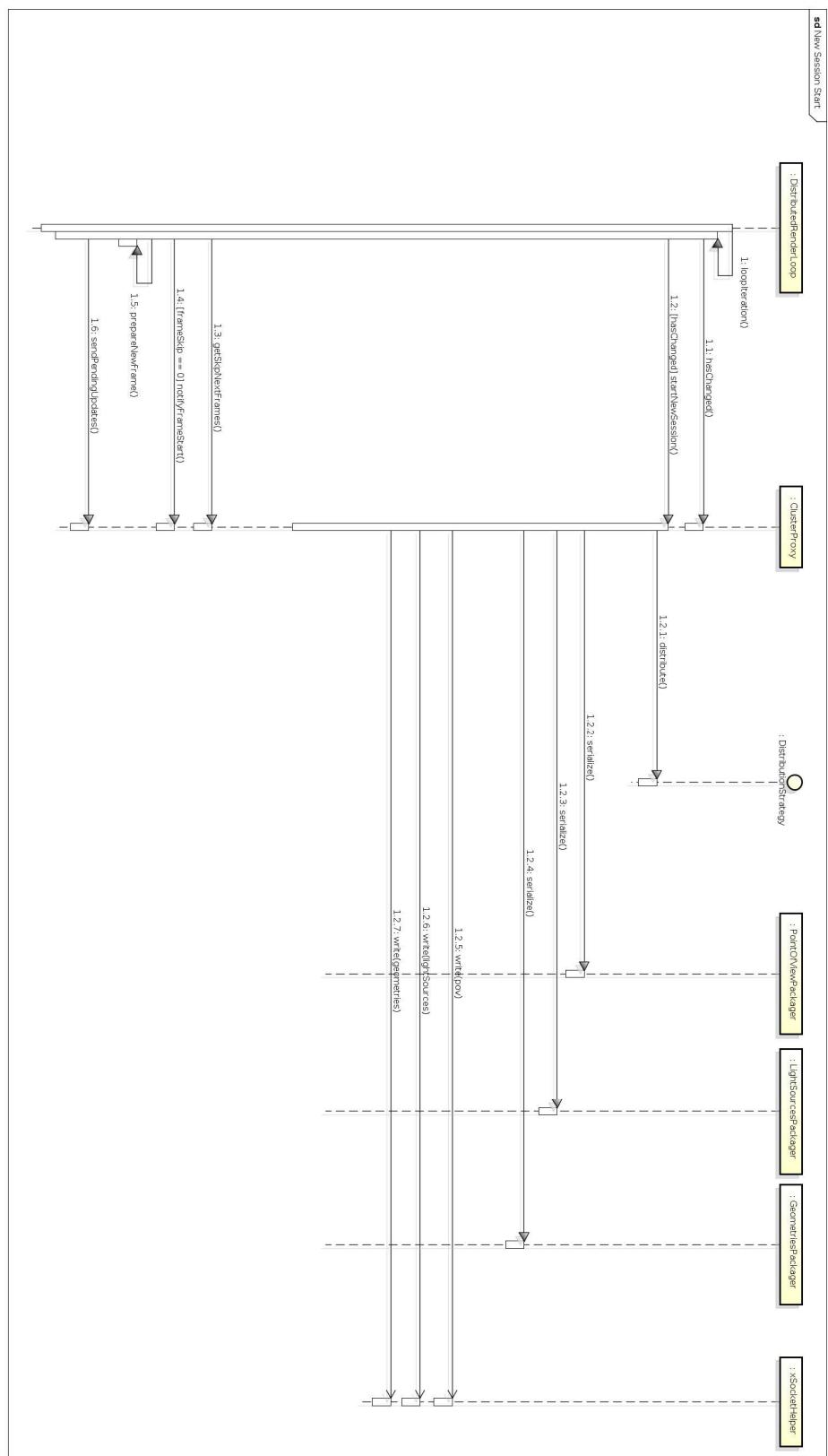
**Figura II - Diagrama de classes do subsistema de composição**



**Figura III - Diagrama de classes do subsistema de serialização**



#### **Figura IV - Diagrama de classe do subsistema de mensagens**



**Figura V - Diagrama de atividades da criação de uma nova sessão**

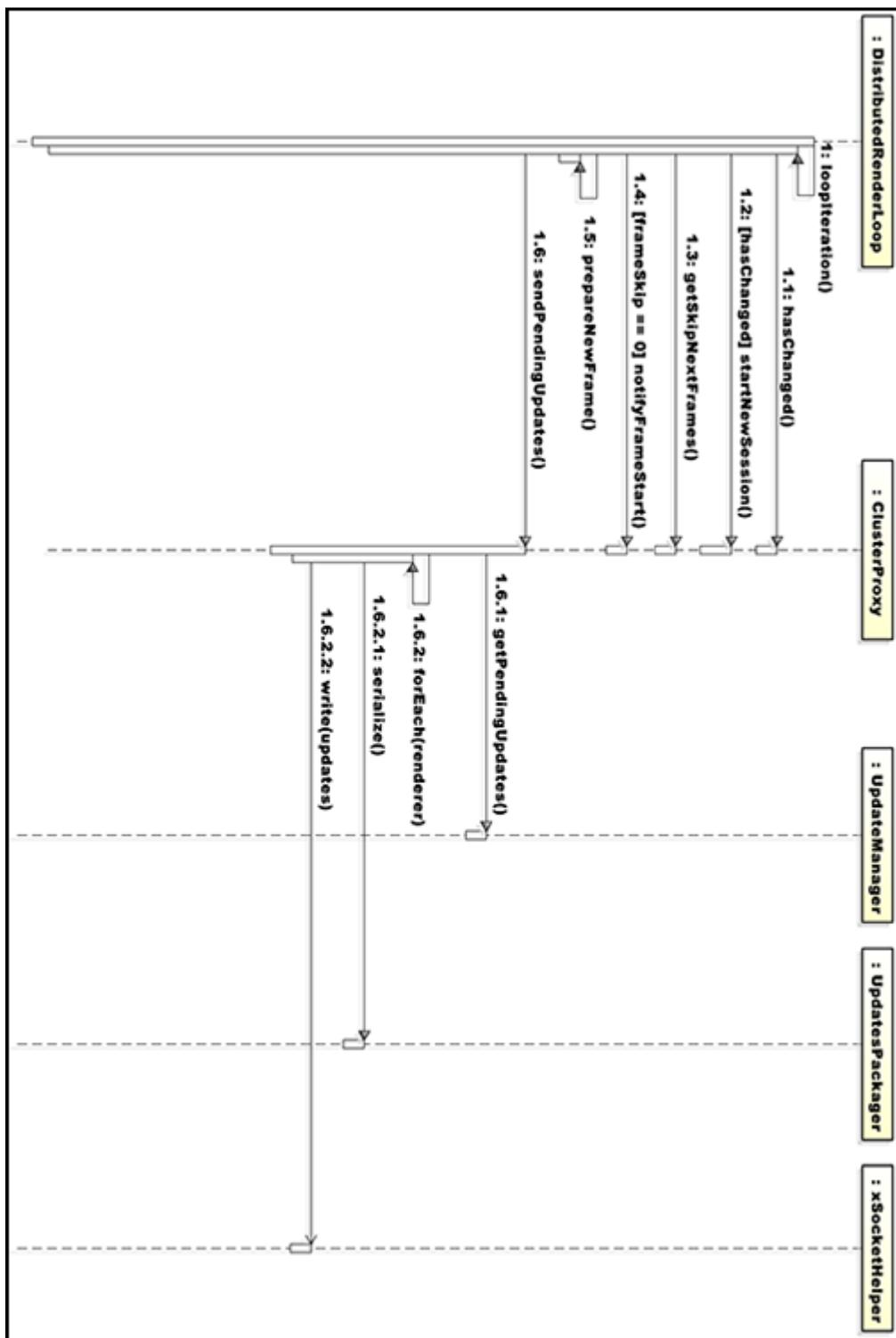


Figura VI - Diagrama de atividades do envio de atualizações

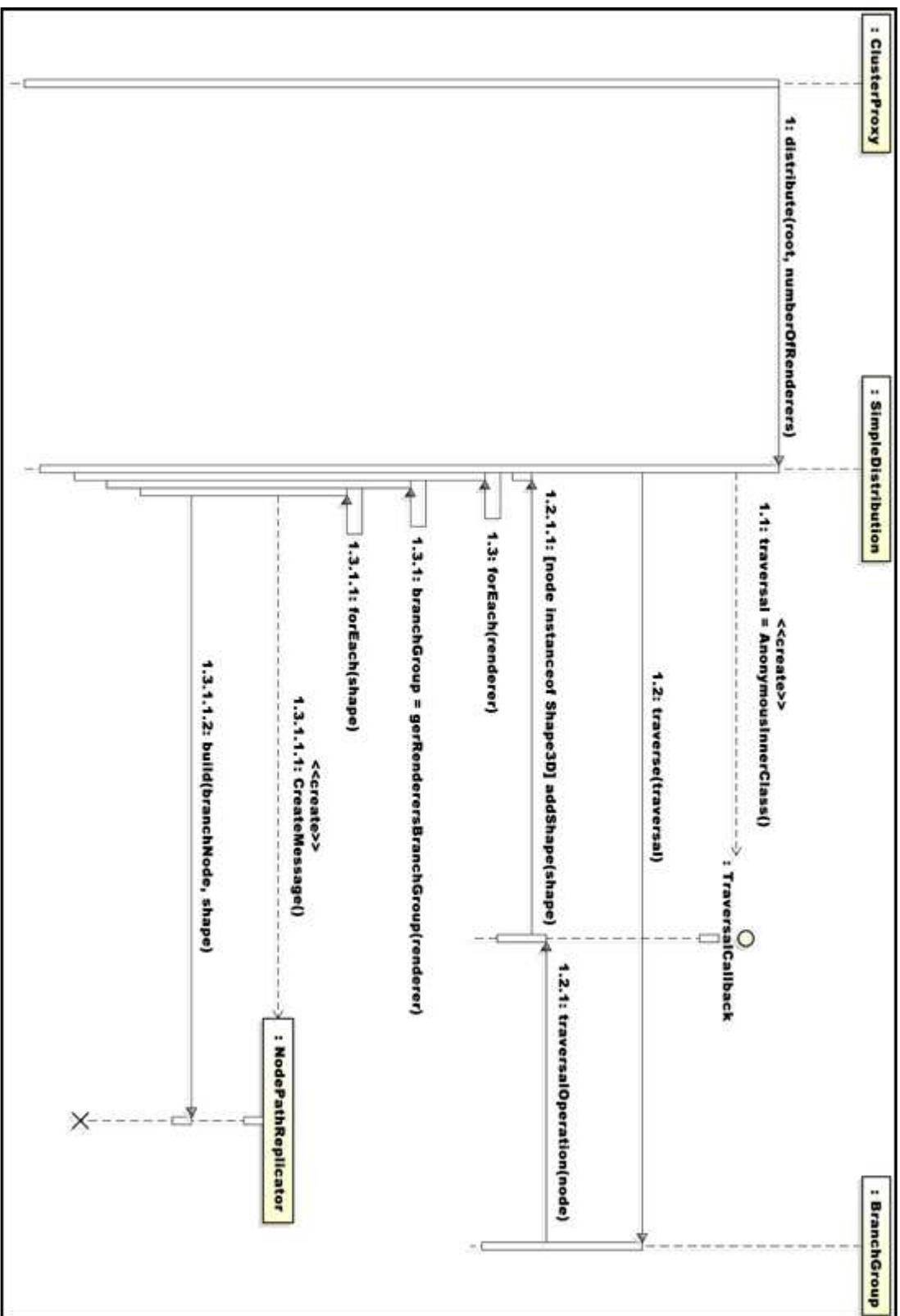


Figura VII - Diagrama de atividades da estratégia de distribuição simples (classe SimpleDistribution)

# **Apêndice B**

## **Manual de Uso do XithCluster**

### **Pré-requisitos de Software**

### **Pré-requisitos de Hardware**

#### **i. Obtendo o XithCluster**

- i.i. A partir do código fonte
  - i.i.i. Usando o Subclipse
  - i.i.ii. Usando outro cliente SVN

#### **ii. A partir das bibliotecas compiladas**

- ii.i. Downloads do site do projeto no googlecode

#### **iii. Criando uma aplicação mestra**

- iii.i. Criando um novo projeto no eclipse
  - iii.i.i. Configurando o caminho de construção do projeto
- iii.ii. Copiando o arquivo de propriedades
- iii.iii. Implementando a aplicação
  - iii.iii.i. Estendendo a classe SampleApplication
  - iii.iii.ii. Criando uma cena
  - iii.iii.iii. Criando o ponto de entrada da aplicação
  - iii.iii.iv. Código completo da aplicação de exemplo

#### **iv. Rodando a aplicação mestre**

- iv.i. Configurando o caminho de biblioteca (Library Path) do projeto
- iv.ii. Usando os parâmetros de linha de comando
- iv.iii. Aumentando a memória disponível para a aplicação

#### **v. Rodando a solução XithCluster 36**

- v.i. Obtendo as aplicações da solução (rendererApp e composerApp)
- v.ii. Rodando as aplicações da solução

## **Pré-requisitos de Software**

- Linux, Windows ou MacOS.
- OpenGL 1.1 (ou superior)
- Oracle JDK 1.6 (ou superior).
- Eclipse 3.6 (ou superior).
- Subclipse 1.6.x (ou outro cliente SVN).

## **Pré-requisitos de Hardware**

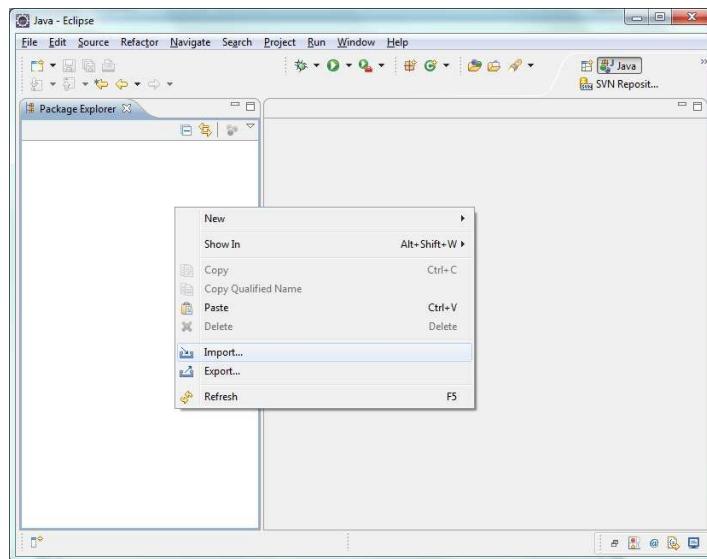
- Aceleração gráfica por Hardware.

# i. Obtendo o XithCluster

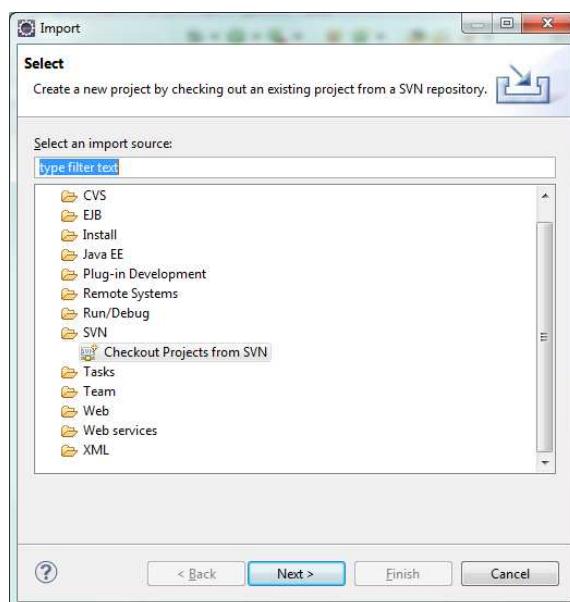
## i.i. A partir do código fonte

### i.i.i. Usando o Subclipse

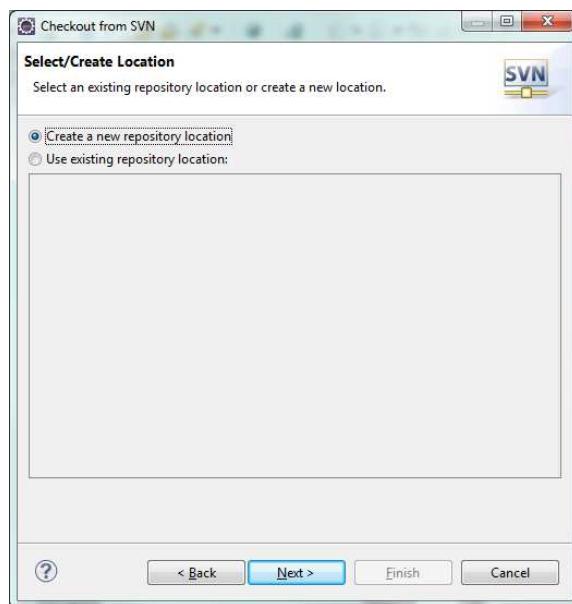
Com o botão direito abra o menu de contexto e selecione a opção "Import...".



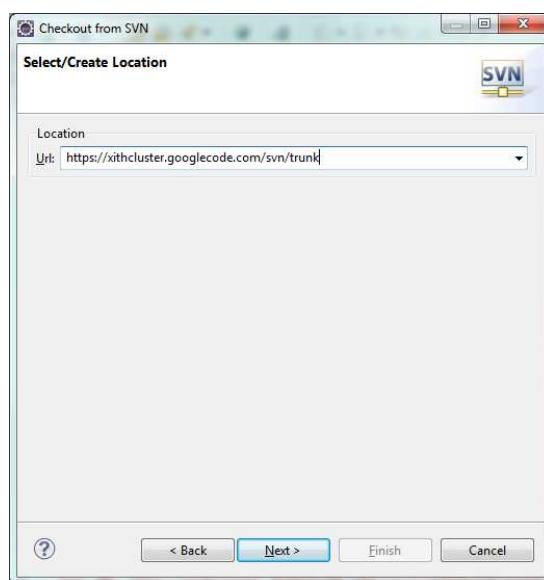
Expanda as opções da pasta SVN, selecione "Checkout Projects from SVN" e aperte o botão "Next".



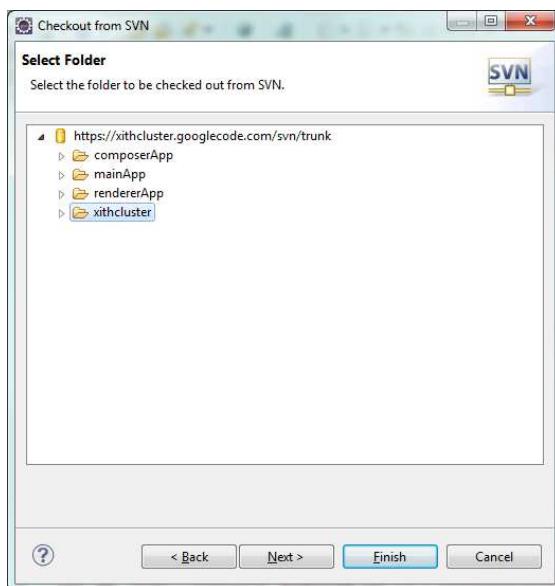
Selecione a opção "Create a new repository location" e aperte o botão "Next"



Informe a Url " https://xithcluster.googlecode.com/svn/trunk" e aperte o botão "Next".



Selecione apenas o projeto "xithcluster" e aperte o botão "Next".



Confime os dados abaixo, especialmente o campo "Project Name" e o checkbox "Check out HEAD revision", e aperte o botão "Finish".



### **i.i.ii. Usando outro cliente SVN**

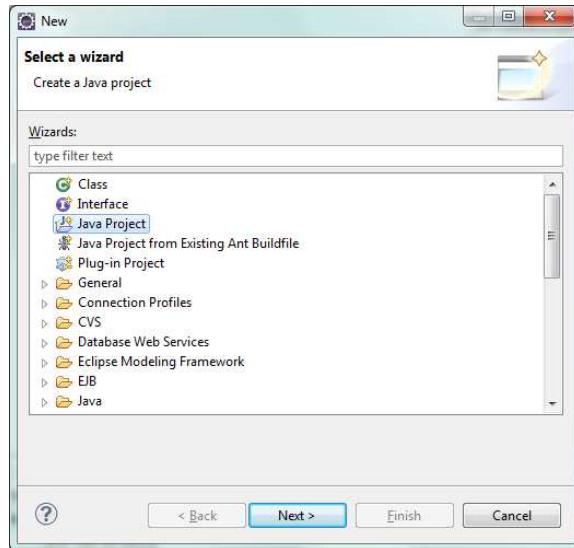
Ao usar outros clientes SVN, você deverá informar a seguinte URL de repositório:

**svn checkout https://xithcluster.googlecode.com/svn/trunk/ xithcluster**

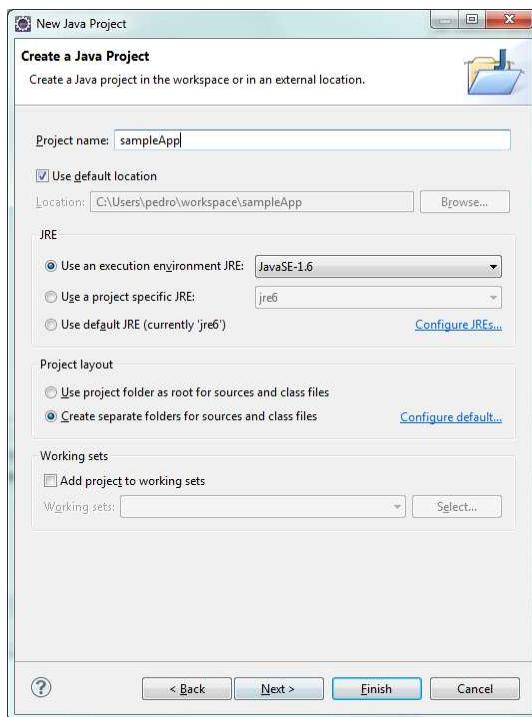
## iii. Criando uma aplicação mestra

### iii.i. Criando um novo projeto no eclipse

Selecione a opção de menu "File > New > Other...". Na janela que se abre selecione "Java Project" e aperte o botão "Next".

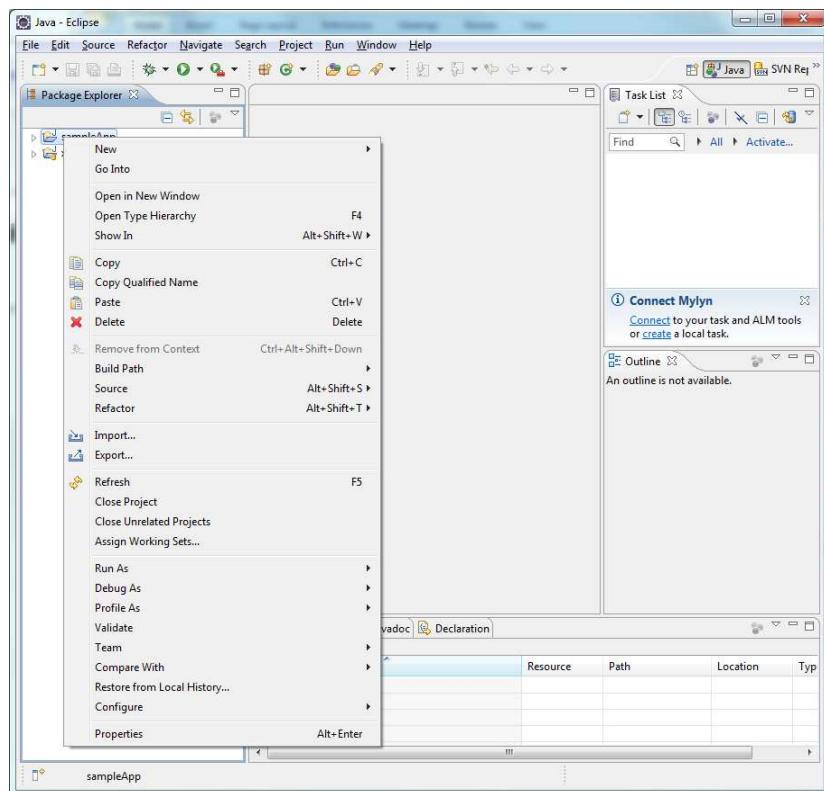


No campo "Project Name" digite "sampleApp" e aperte o botão "Finish". Certifique-se se o ambiente de execução Java está na versão 1.6, como abaixo:

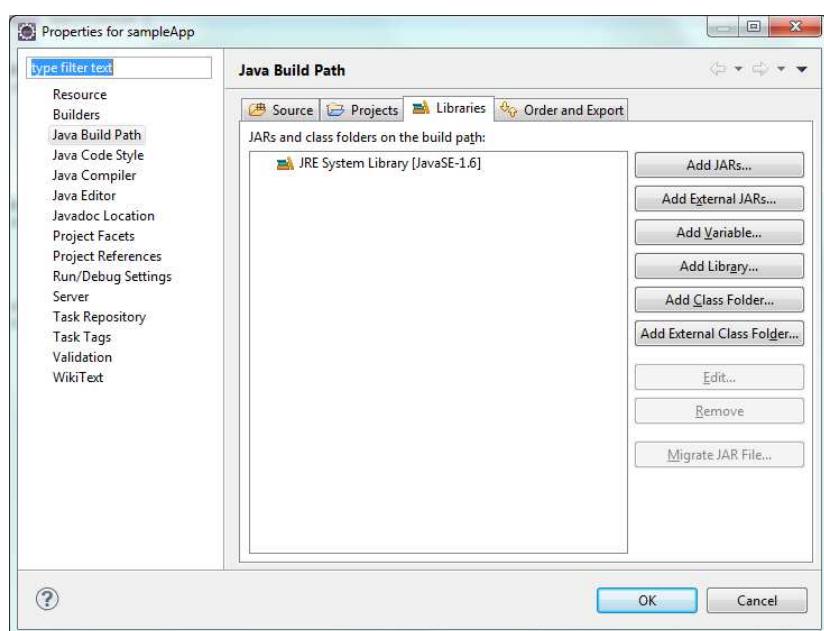


### iii.i.i. Configurando o caminho de construção (Build Path) do projeto

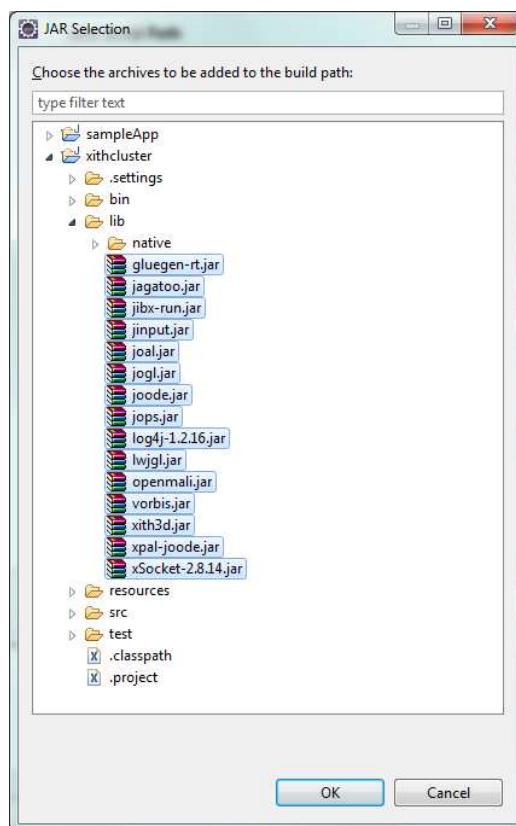
Com o botão direito do mouse pressionado sobre seu projeto, abra o menu e então selecione a opção "Properties".



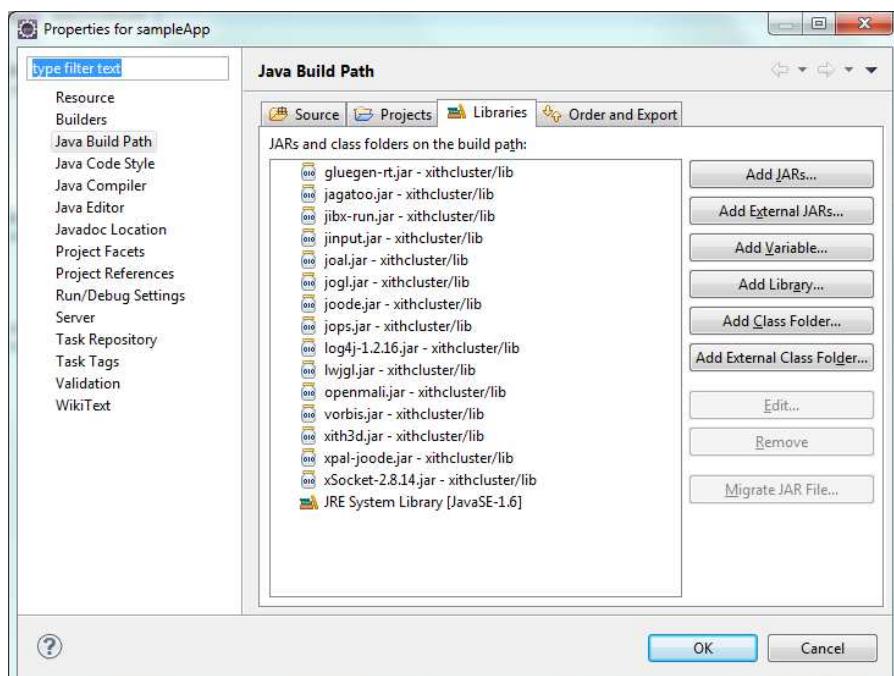
Selecione a opção "Java Build Path" na parte esquerda da janela. Navegue para a aba "Libraries" e então aperte o botão "Add JARs...".



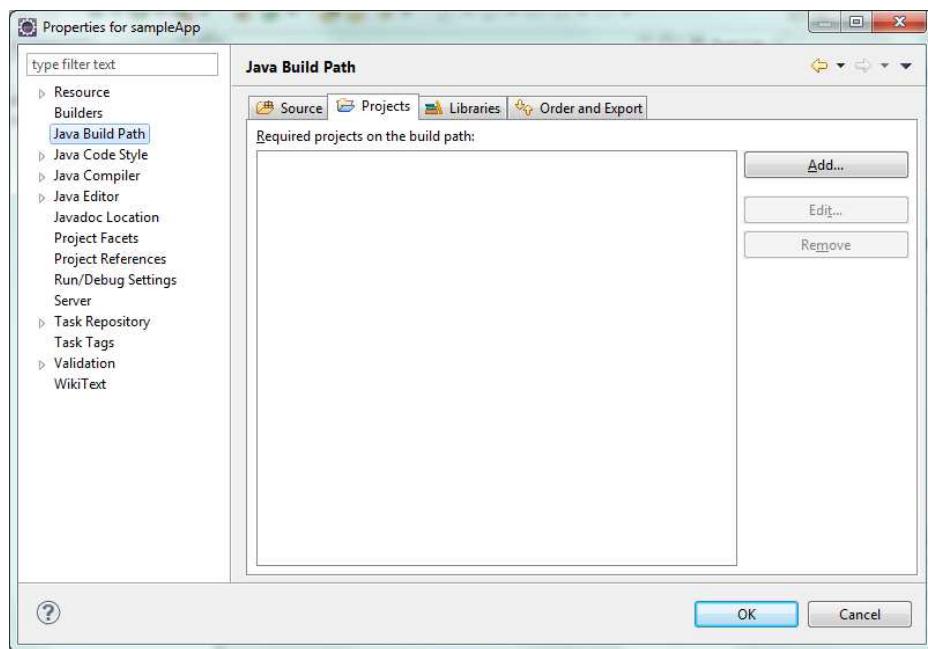
Explore a pasta "xithcluster/lib" e a expanda como ilustrado abaixo. Selecione todos os arquivos JAR e aperte o botão "Ok".



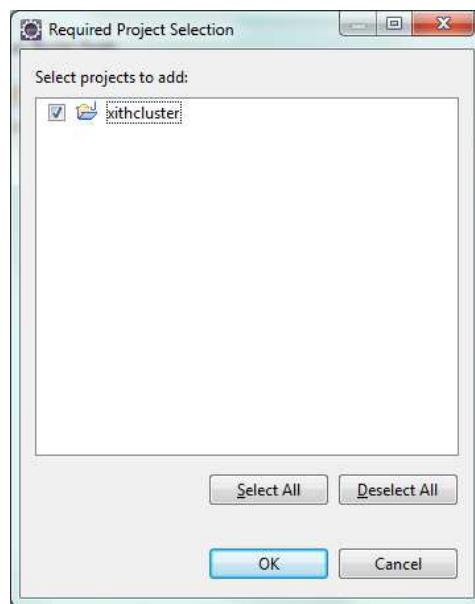
A aba "Libraries" deverá ficar como abaixo.



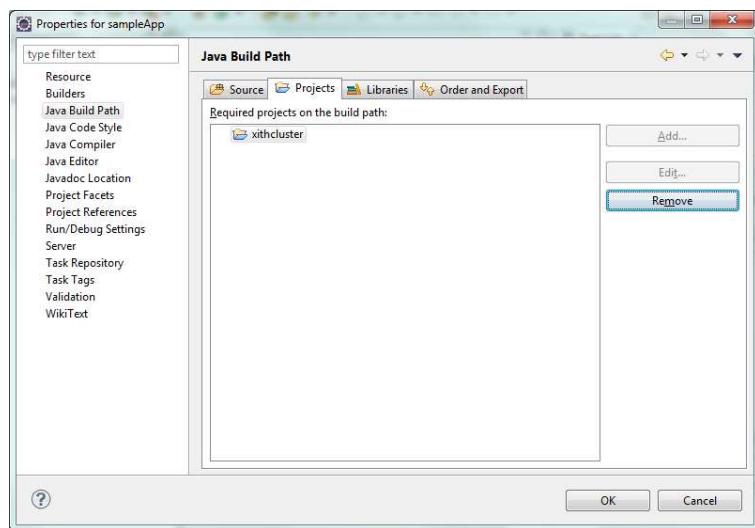
Navegue para a aba "Projects" e aperte o botão "Add...".



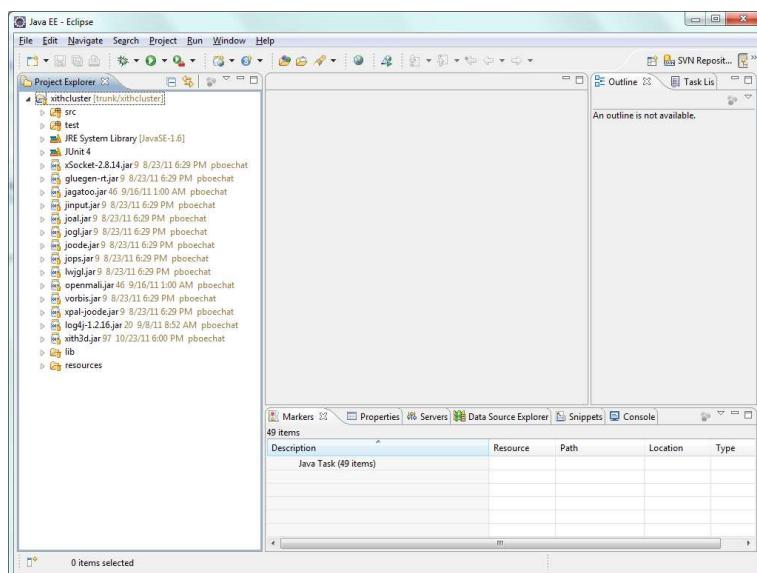
Marque o checkbox "xithcluster".



A aba "Projects" deverá ficar como abaixo.

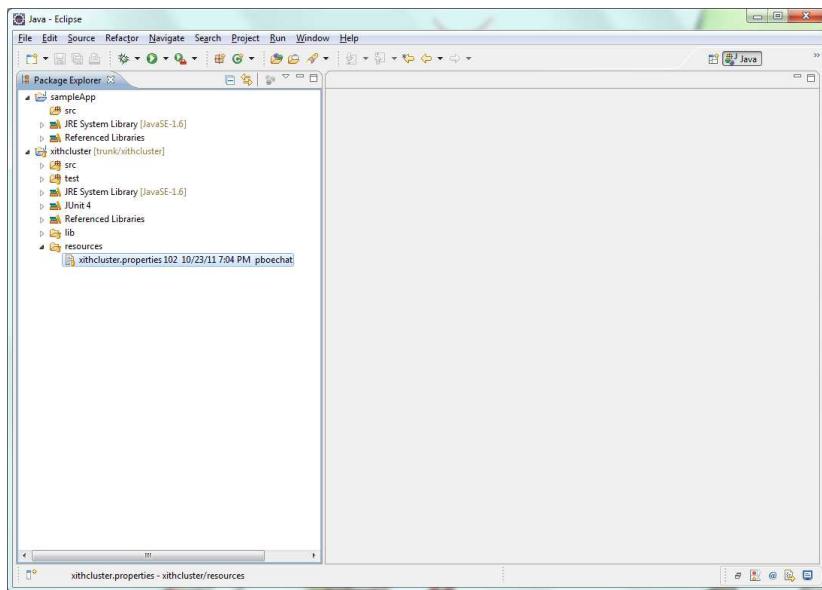


Confirme suas alterações apertando o botão "Ok" e seu projeto deverá ficar como o abaixo.

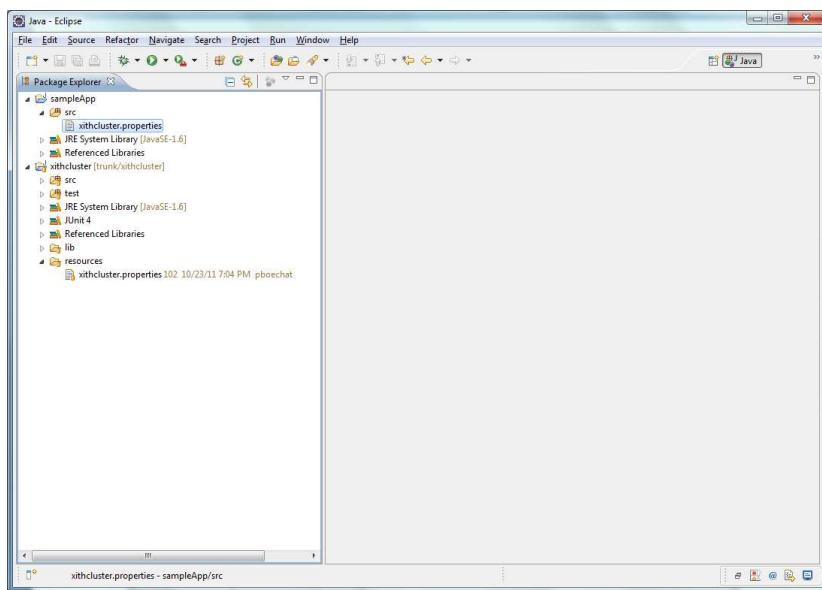


### iii.ii. Copiando o arquivo de propriedades

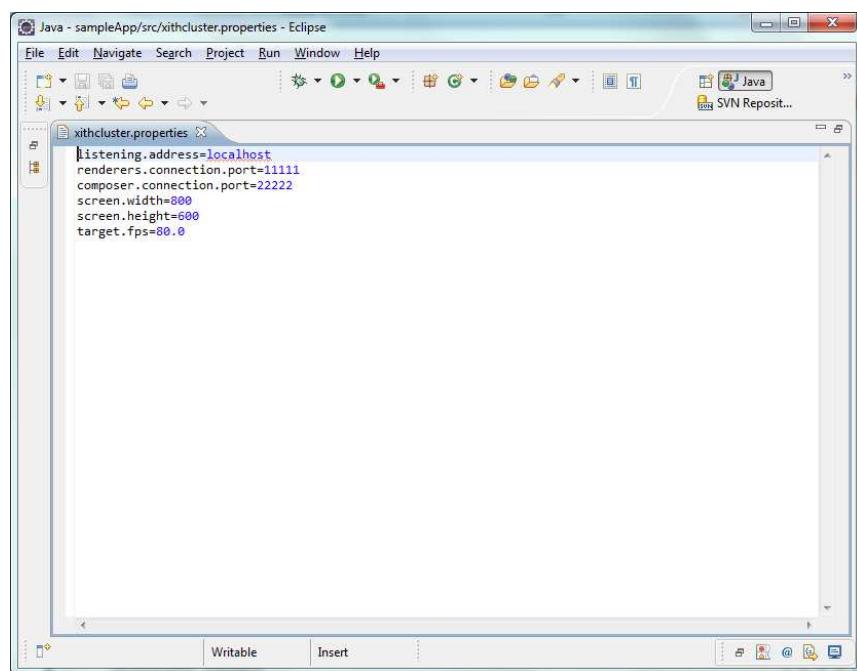
Explore o diretório "xithcluster/resources" e copie o arquivo "xithcluster.properties".



Expanda a pasta "src" de seu projeto e cole o arquivo copiado.



Ao abrir esse arquivo, você poderá alterar as configurações da sua aplicação.

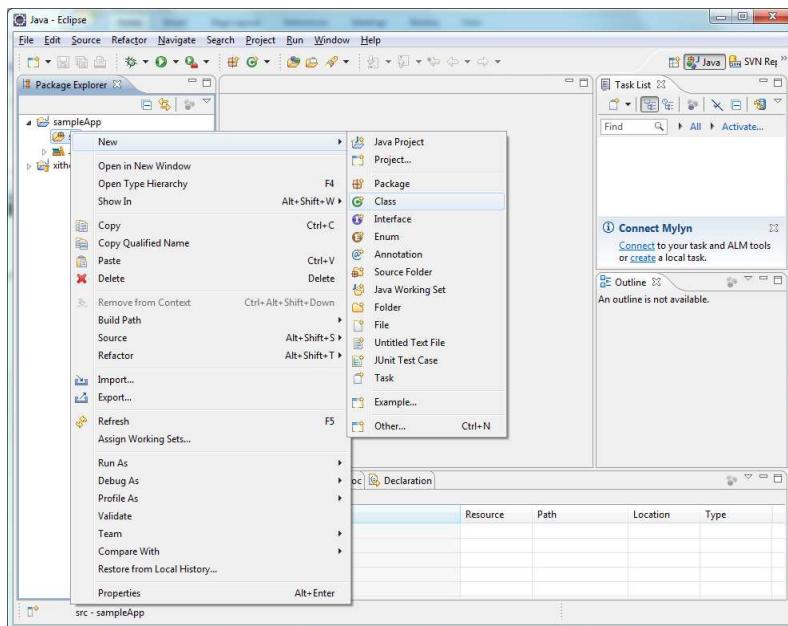


Nota: No item iv.ii é apresentada uma outra maneira de configurar sua aplicação.

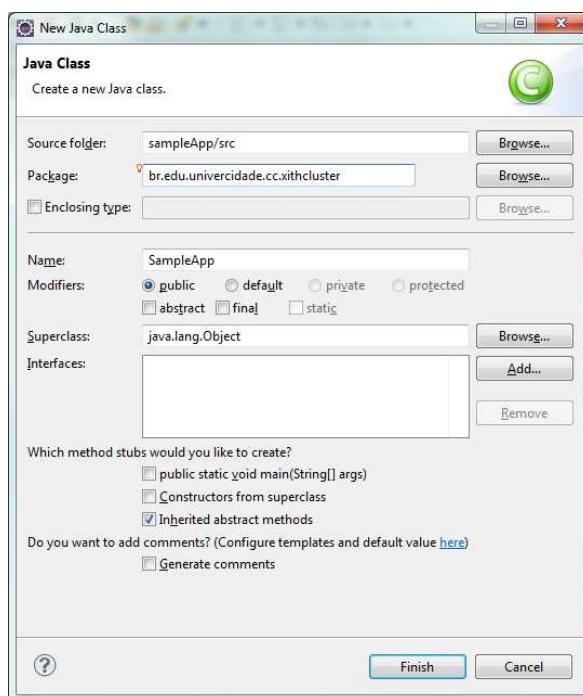
### iii.iii. Implementando a aplicação

#### iii.iii.i. Estendendo a classe Application

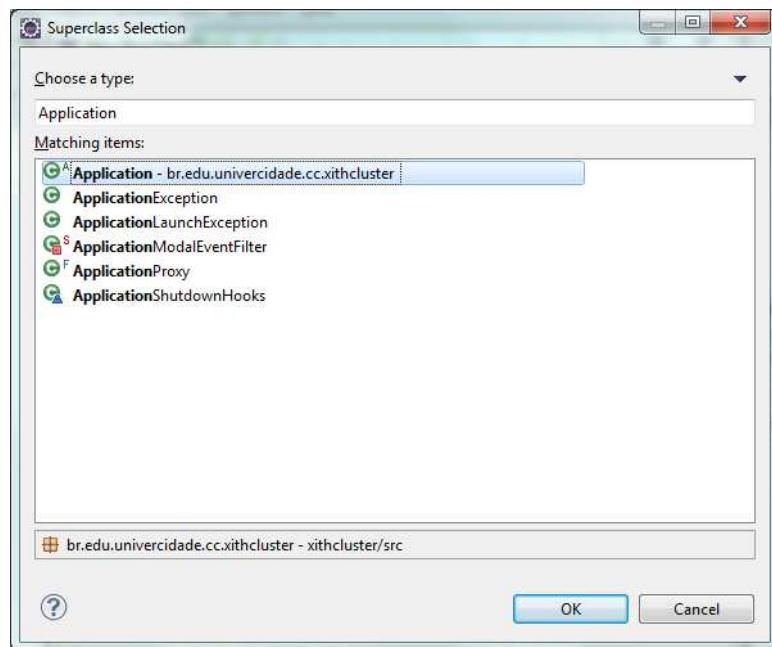
Com o botão direito do mouse pressionado sobre a pasta "src", abra o menu e então selecione a opção "New > Class".



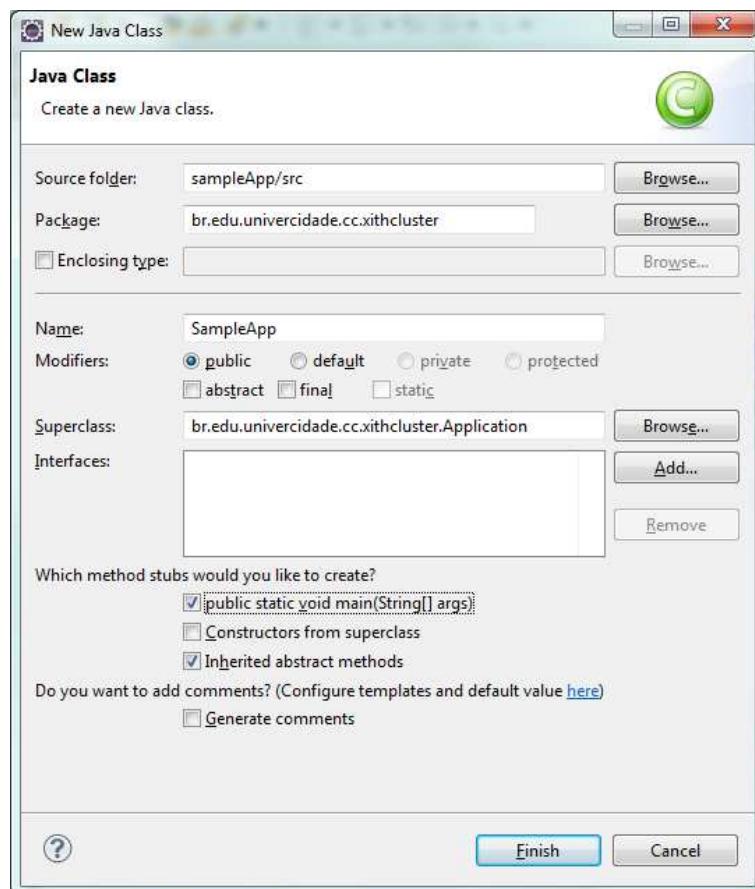
Defina o campo "Package" com o valor "br.edu.univercidade.cc.xithcluster" e o campo "Name" com o valor "SampleApp".



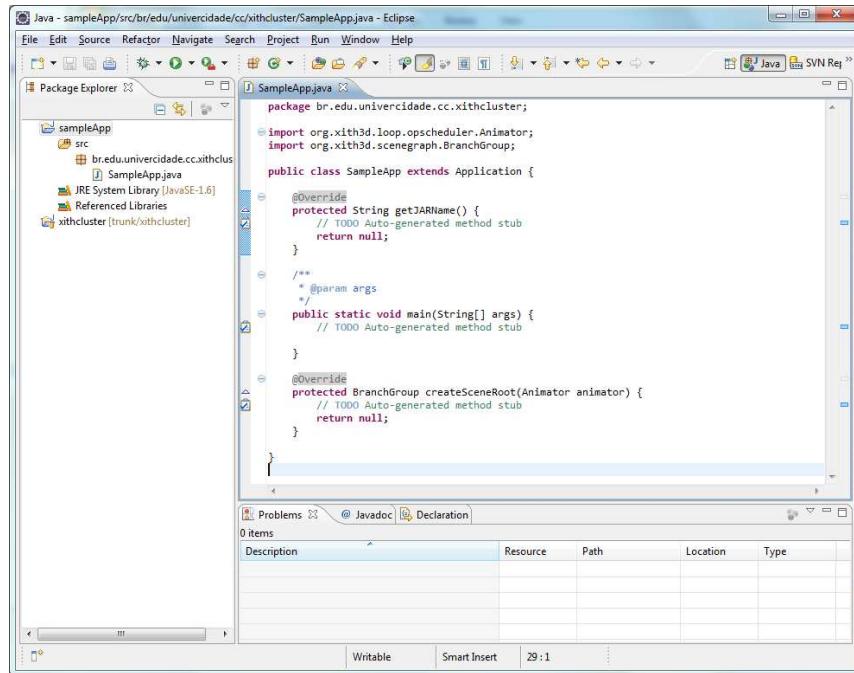
Aperte o botão "Browse..." e digite "Application" no campo "Choose a type". Selecione a opção "Application - br.edu.univercidade.cc.xithcluster" como mostra abaixo.



Marque o checkbox "public static void main(String[] args)" e aperte o botão "Finish".



A classe criada pelo Eclipse deverá conter os "imports" e métodos abaixo.



The screenshot shows the Eclipse Java IDE interface. The title bar reads "Java - sampleApp/src/br/edu/univercidade/cc/xithcluster/SampleApp.java - Eclipse". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, Help. The toolbar has various icons for file operations. The left sidebar is the "Package Explorer" showing a project named "sampleApp" with a "src" folder containing "SampleApp.java". Other items include "JRE System Library [JavaSE-1.6]" and "Referenced Libraries". The main editor window displays the following Java code:

```
package br.edu.univercidade.cc.xithcluster;
import org.xith3d.loop.opscheduler.Animator;
import org.xith3d.scenegraph.BranchGroup;
public class SampleApp extends Application {
    @Override
    protected String getJARName() {
        // TODO Auto-generated method stub
        return null;
    }
    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
    @Override
    protected BranchGroup createSceneRoot(Animator animator) {
        // TODO Auto-generated method stub
        return null;
    }
}
```

The bottom right corner of the editor shows "29:1". Below the editor is the "Problems" view, which is currently empty. The status bar at the bottom indicates "Writable", "Smart Insert", and "29:1".

### iii.iii.ii. Criando uma cena

A lógica de criação da cena deve ser implementada pelo método "*protected BranchGroup createSceneRoot(Animator animator)*". Ele determina que o desenvolvedor retorne um objeto não nulo do tipo *BranchGroup*, que representa a raiz da cena.

Detalhar o uso dos objetos do Xith3D não está dentro do escopo deste manual. Para saber como usá-las, recorra à documentação do Xith3D (<http://xith.org/downloads/docs/xin/xin-3rd-edition.pdf>). O conhecimento de como criar nós, animações, texturas, etc. pode ser facilmente contextualizado ao XithCluster.

Para facilitar a prototipação rápida de aplicações, a classe *SceneUtils* possibilita a criação de algumas formas geométricas através de uma interface simples, ocultando alguns detalhes do Xith3D.

```

Java - sampleApp/src/br/edu/univercidade/cc/xithcluster/SampleApp.java - Eclipse
File Edit Source Refactor Navigate Project Run Window Help
SampleApp.java
package br.edu.univercidade.cc.xithcluster;
import org.openmali.vecmath2.Colorf;
public class SampleApp extends Application {
    public SampleApp(Tuple4f eyePosition, Tuple3f viewFocus) {
        super(eyePosition, viewFocus);
    }
    @Override
    protected String getJARName() {
        // O nome do JAR da sua aplicação
        return "sampleApp.jar";
    }
    /**
     * @param args
     */
    public static void main(String[] args) {
        // Instanciando e iniciando a aplicação de exemplo
        SampleApp sampleApp = new SampleApp(new Tuple4f(0.0f, 0.0f, 3.0f), // posição do observador
                                            new Tuple3f(0.0f, 0.0f, 0.0f)); // ponto focal
        sampleApp.init(args);
    }
    @Override
    protected BranchGroup createSceneRoot(Animator animator) {
        // Cria o nó raiz
        BranchGroup root = new BranchGroup();
        // Agrupa todos os geometrias estáticas da cena em um único nó (boa prática)
        Group staticGeoms = new Group();
        staticGeoms.setName("staticGeometries");
        root.addChild(staticGeoms);
        // Adiciona um cubo
        SceneUtils.addCube(staticGeoms, // nó pai
                           "cube", // nome
                           0.75f, // tamanho do lado
                           new Tuple3f(0.0f, 0.3f, -0.3f), // translação
                           new Tuple3f(-35.0f, 20.0f, 0.0f), // escala
                           Colorf.RED); // cor
        // Adiciona uma esfera
        SceneUtils.addSphere(staticGeoms, // nó pai
                             "sphere", // nome
                             0.55f, // tamanho do lado
                             new Tuple3f(-0.3f, -0.3f, -0.5f), // translação
                             Colorf.BLUE); // cor
        // Adiciona uma luz direcional
        SceneUtils.addDirectionalLight(root, // nó pai
                                       "luz-direcional", // nome
                                       Colorf.WHITE, // cor
                                       Vector3f.NEGATIVE_Z_AXIS); // direção
        // Retorna o nó raiz
        return root;
    }
}

```

### iii.iii.iii. Criando o ponto de entrada da aplicação

O breve método "main" do código anterior pode ser dividido em duas partes: a criação da classe e a chamada ao método *init*.

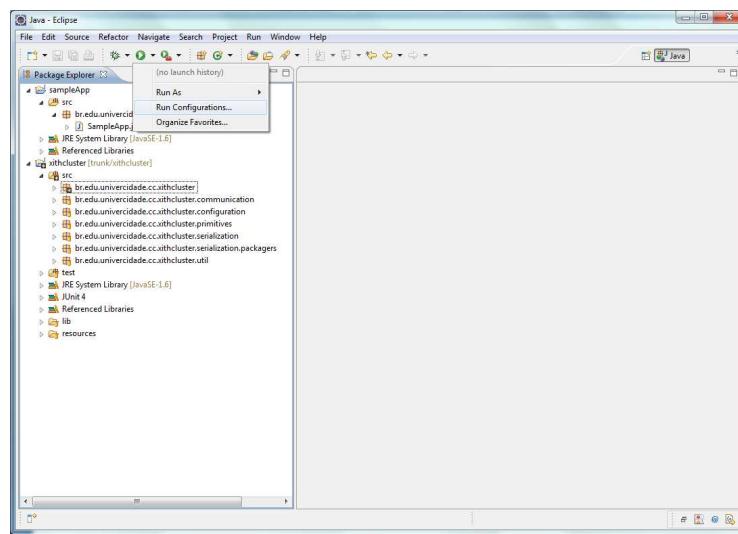
Como a classe *Application* não possui um construtor padrão (sem argumentos) a classe implementada no exemplo foi obrigada a declarar um dos construtores de sua classe pai. O construtor declarado no exemplo tem a particularidade de iniciar o ponto de vista da cena através dos dois parâmetros passados para ele.

A chamada ao método *init*, declarado na classe *Application*, recebe os argumentos de entrada da aplicação. Esse método tem que ser chamado obrigatoriamente, já que é ele que realiza toda a inicialização da aplicação: abre janelas, cria o grafo de cena, inicia conexões de rede, etc. A passagem correta dos argumentos de entrada é fundamental para que os parâmetros de linha de comando (Item iv.ii) sejam lidos.

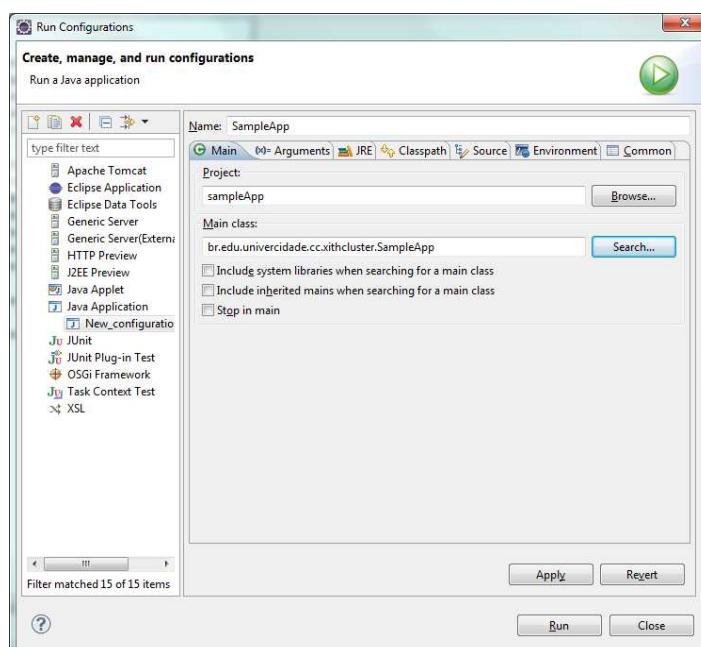
## iv. Rodando a aplicação mestra

### iv.i. Configurando o caminho de biblioteca (Library Path) do projeto

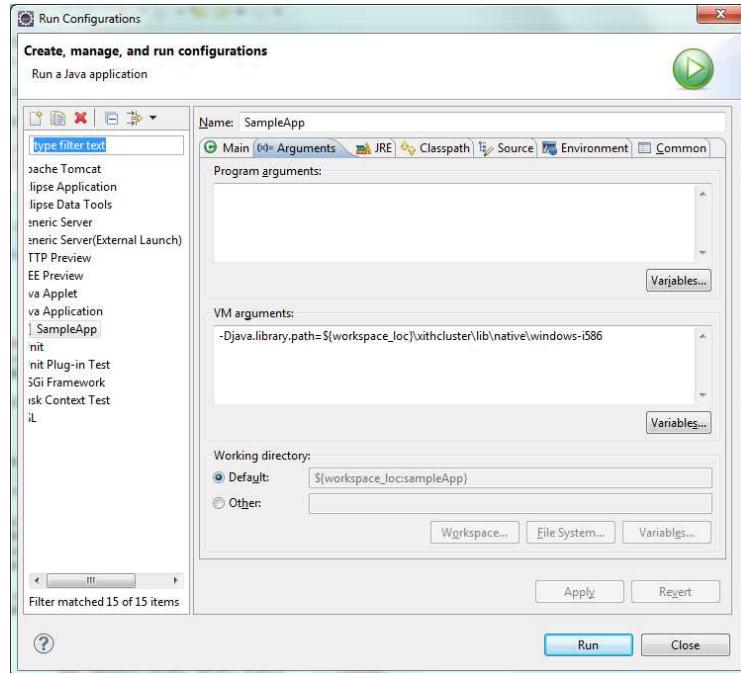
Aperte a seta ao lado do botão "Run As.." (botão verde com o símbolo *play*), abra o menu e então selecione a opção "Run Configurations...".



Aperte duas vezes o botão esquerdo do mouse sobre a opção "Java Application" (ao lado esquerdo). Defina o valor do campo "Name" como "SampleApp", o valor do campo "Main class" como "br.edu.univercidade.cc.xithcluster.SampleApp" e aperte o botão "Apply".



Navegue para a aba "Arguments" e defina o valor da área de texto "VM arguments" como "-Djava.library.path=\${workspace\_loc}\xithcluster\lib\native\windows-i586".



Nota: Onde se lê "windows-i586" deverá ser escrito o nome da sua plataforma. Os valores possíveis são:

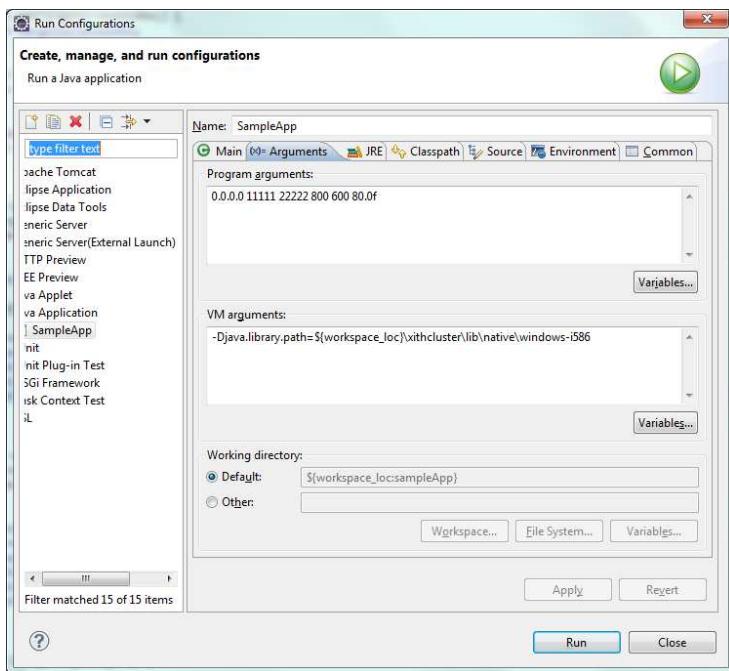
"windows-i586" (para Windows 32 e 64 bits), "linux-amd64" (para Linux 64 bits) e "macosx" (para MacOS X).

Por exemplo, se a sua plataforma for linux 64 bits, o valor de sua área de texto "VM Arguments" deverá ler:

-Djava.library.path=\${workspace\_loc}\xithcluster\lib\native\linux-amd64

## iv.ii. Usando os parâmetros de linha de comando

Para configurar os parâmetros de linha de comando, ainda na aba "Arguments", digite os valores "0.0.0.0", "11111", "22222", "800", "600" e "80.0f" na área de texto "Program arguments".



Nota 1: Os valores discriminados acima são os valores padrão de configuração. Em ordem, eles significam o seguinte:

1º parâmetro = endereço IP = 0.0.0.0 (aceita qualquer literal)

2º parâmetro = porta de conexão dos renderizadores = 11111 (aceita um número inteiro entre 1 e 65536)

3º parâmetro = porta de conexão dos compositores (aceita um número inteiro entre 1 e 65536)

4º parâmetro = largura de tela desejada (aceita qualquer número inteiro)

5º parâmetro = altura de tela desejada (aceita qualquer número inteiro)

6º parâmetro = número de quadros por segundo desejado (aceita um número decimal com um ponto separando a parte inteira da fracionária)

Caso algum parâmetro esteja faltando ou não obedeça aos valores válidos, a aplicação exibirá uma mensagem de erro na saída padrão apontando o parâmetro que está faltando ou está inválido, além da forma correta que a aplicação deve ser rodada usando parâmetros de linha de comando.

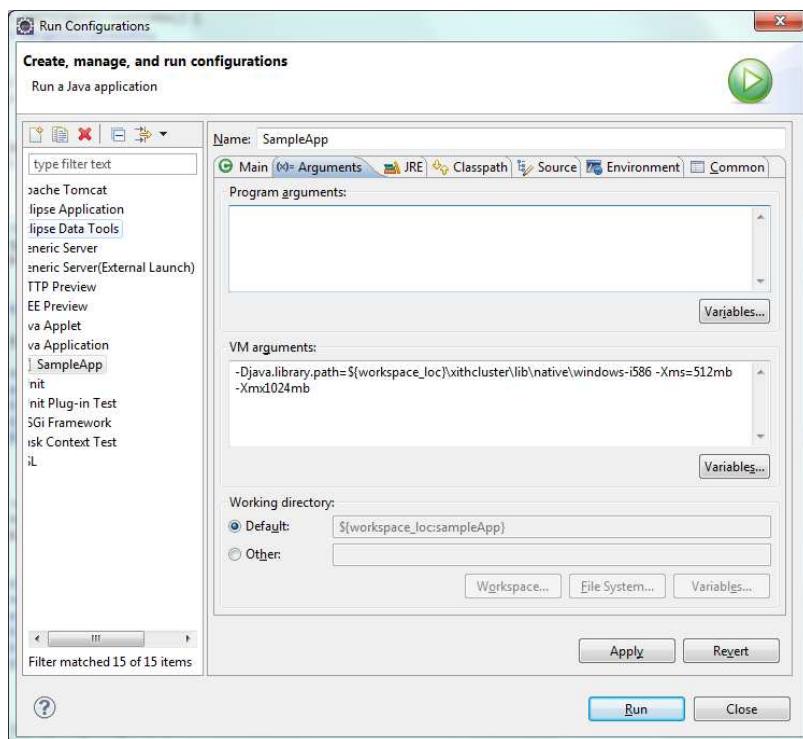
Por exemplo, ao informar um valor inválido para a porta de conexão dos renderizadores, a seguinte mensagem de erro será exibida:

Nota 2: No Item 3.2 este Item foi apresentado como alternativa ao uso do arquivo de propriedades.

### iv.iii. Aumentando a memória disponível para a aplicação

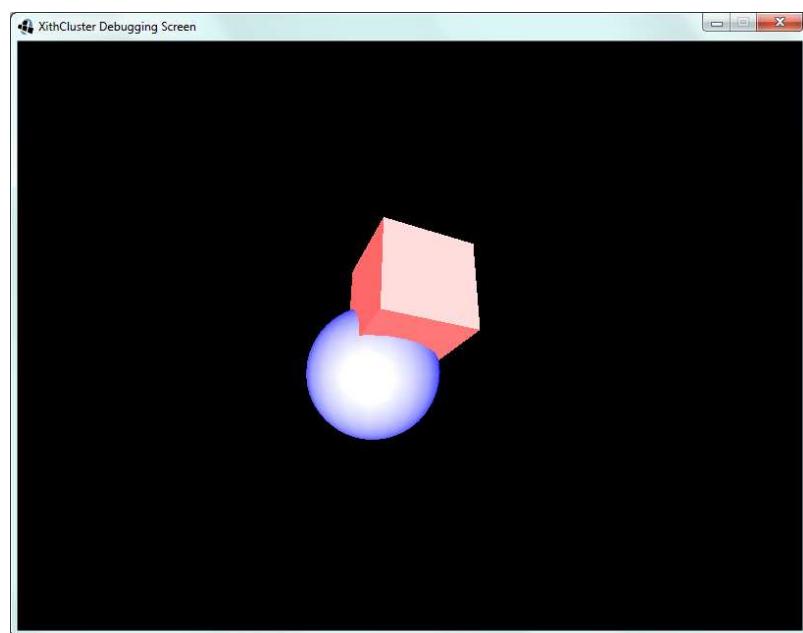
Por se tratar de uma aplicação Java, as configurações que dizem respeito a quantidade de memória disponível para a aplicação são configuradas no níveis da máquina virtual (JVM).

Na aba "Arguments", digite ao fim da área de texto "VM arguments" os seguintes parâmetros: "-Xms=512mb -Xmx=1024mb".



Nota: Através da escrita dos parâmetros acima a JVM alocará a quantidade mínima de 512 megabytes (mb) e máxima de 1024 megabytes para a aplicação.

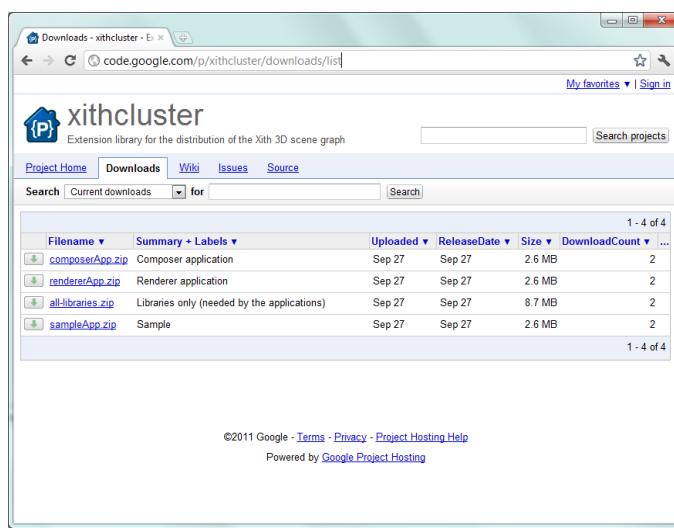
A figura abaixo ilustra a aplicação de exemplo executando corretamente.



# v. Rodando a solução XithCluster

## v.i. Obtendo as aplicações da solução (rendererApp e composerApp)

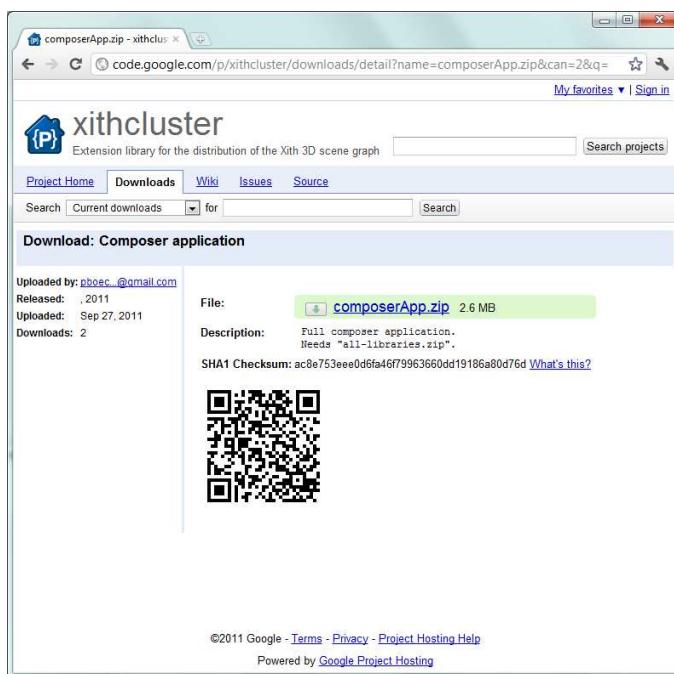
Visite o endereço "http://code.google.com/p/xithcluster/downloads/list" e em seguida entre no link "composerApp.zip".



The screenshot shows a list of four files under the 'Downloads' tab:

Filename	Summary + Labels	Uploaded	ReleaseDate	Size	DownloadCount
<a href="#">composerApp.zip</a>	Composer application	Sep 27	Sep 27	2.6 MB	2
<a href="#">rendererApp.zip</a>	Renderer application	Sep 27	Sep 27	2.6 MB	2
<a href="#">all-libraries.zip</a>	Libraries only (needed by the applications)	Sep 27	Sep 27	8.7 MB	2
<a href="#">sampleApp.zip</a>	Sample	Sep 27	Sep 27	2.6 MB	2

Entre no link "composerApp.zip" novamente para descarregar a aplicação compositora.



The screenshot shows the details for the "composerApp.zip" file:

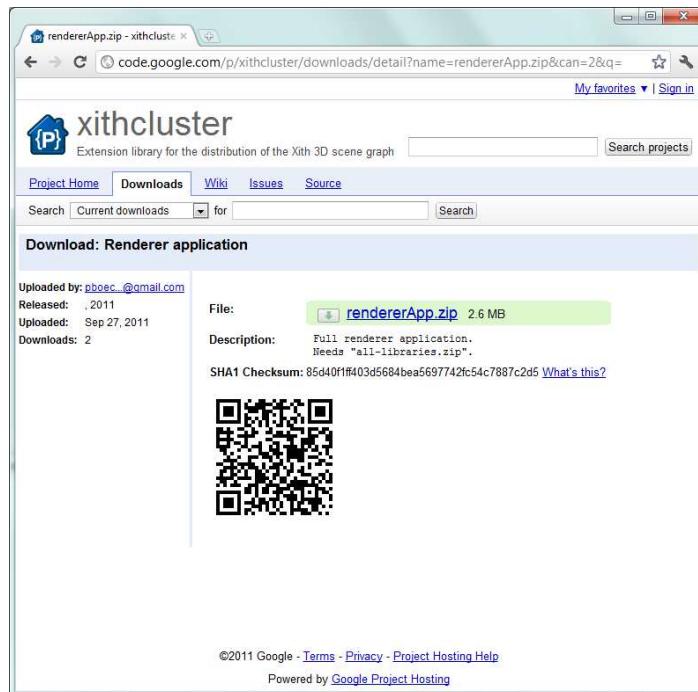
Uploaded by: [phoec\\_@gmail.com](#)  
Released: .2011  
Uploaded: Sep 27, 2011  
Downloads: 2

File: [composerApp.zip](#) 2.6 MB

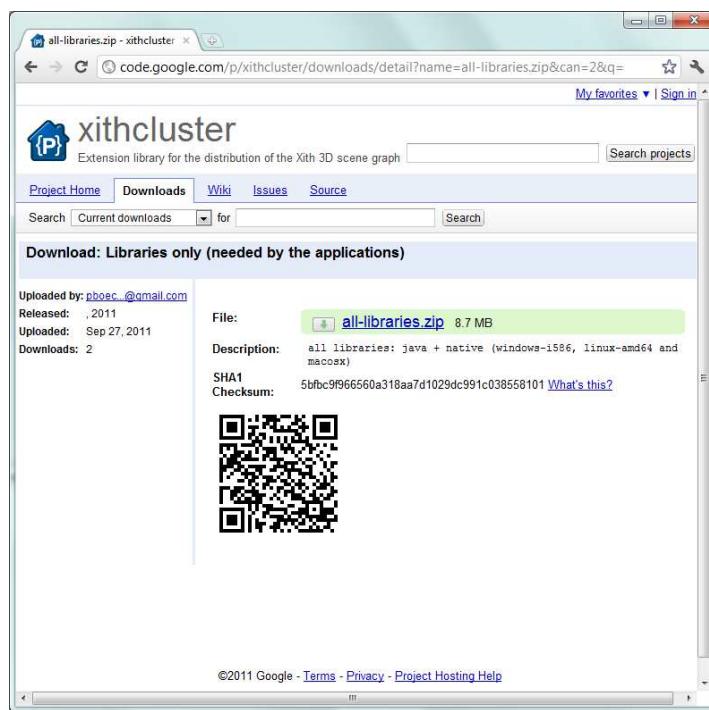
Description: Full composer application.  
Needs "all-libraries.zip".  
SHA1 Checksum: ac8e753eee0d6fa46f79963660dd19186a80d76d [What's this?](#)

QR code:

Volte para a tela anterior e visite o link "rendererApp.zip". Entre no link "rendererApp.zip" novamente para descarregar a aplicação renderizadora.



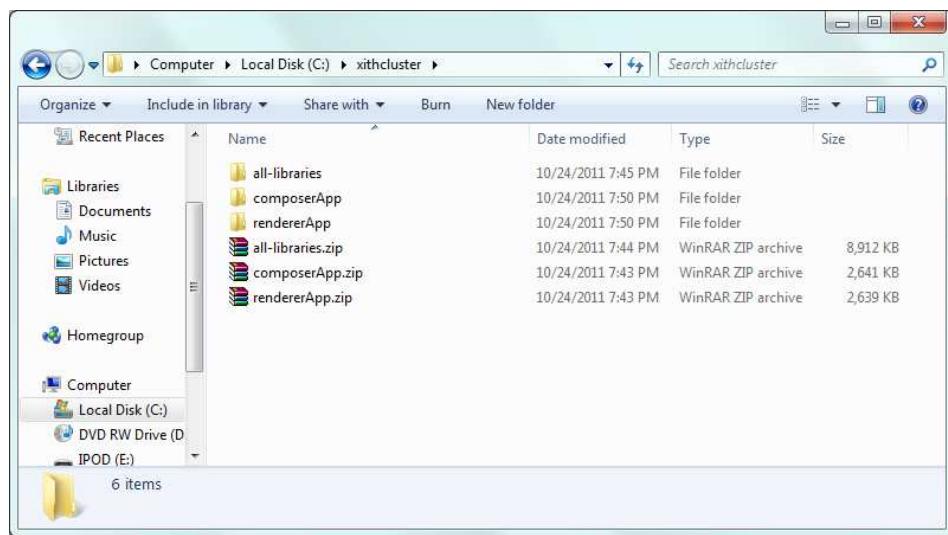
Enfim, volte para a tela anterior e entre no link "all-libraries.zip". Entre no link "all-libraries.zip" novamente para descarregar todas as dependências das aplicações renderizadora e compositora.



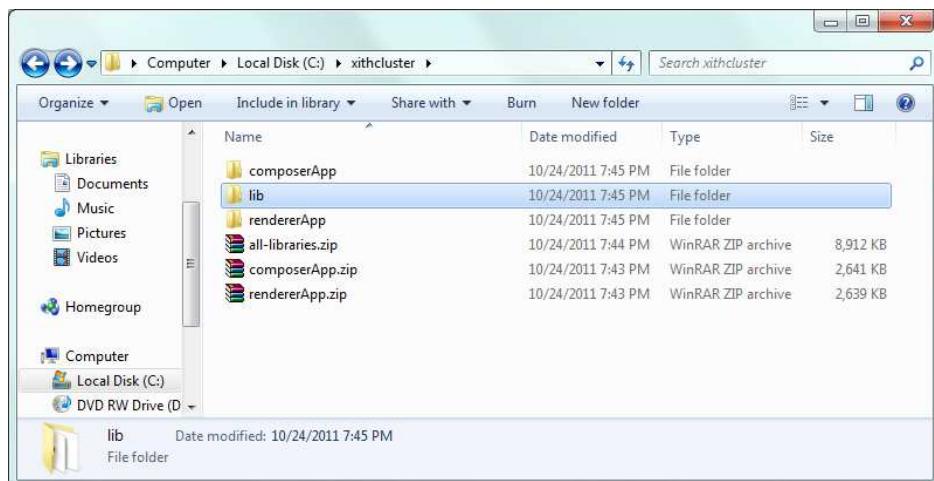
## v.ii. Rodando as aplicações da solução

Nota: Antes de rodar essas aplicações é necessário rodar a aplicação mestre, como por exemplo a aplicação de exemplo (ver Itens 3 e 4).

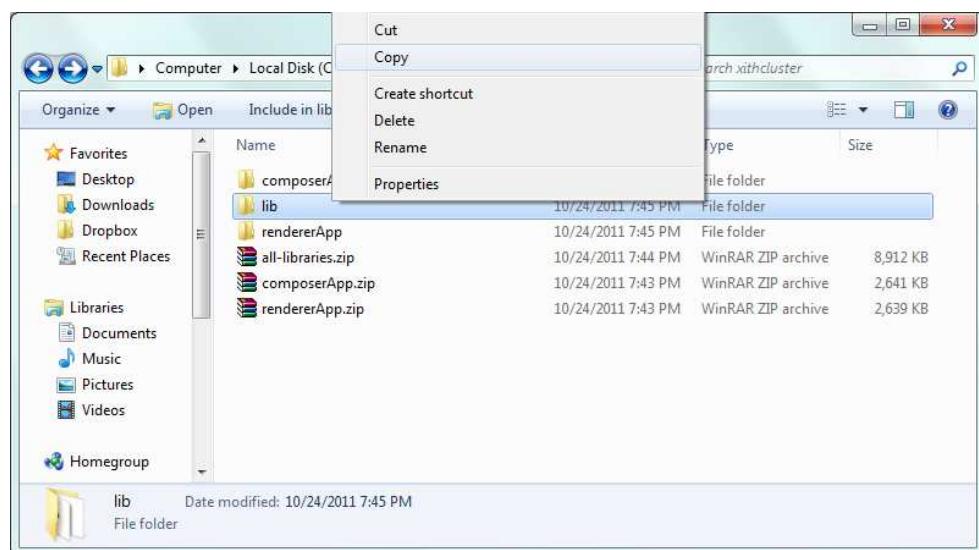
Explore a pasta onde foi efetuada a descarga das aplicações e das dependências. Descomprima cada arquivo .ZIP em uma pasta de mesmo nome.



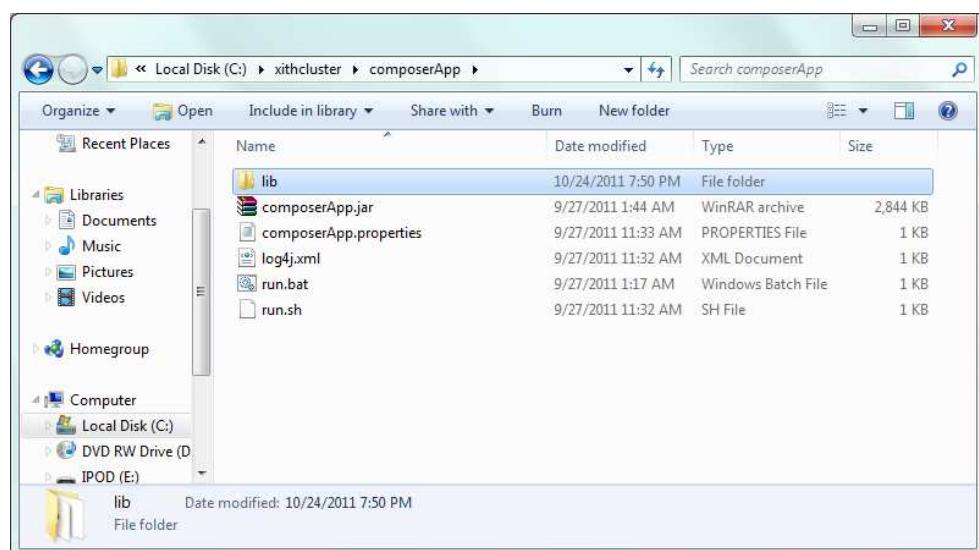
Renomeie a pasta "all-libraries" para "lib".



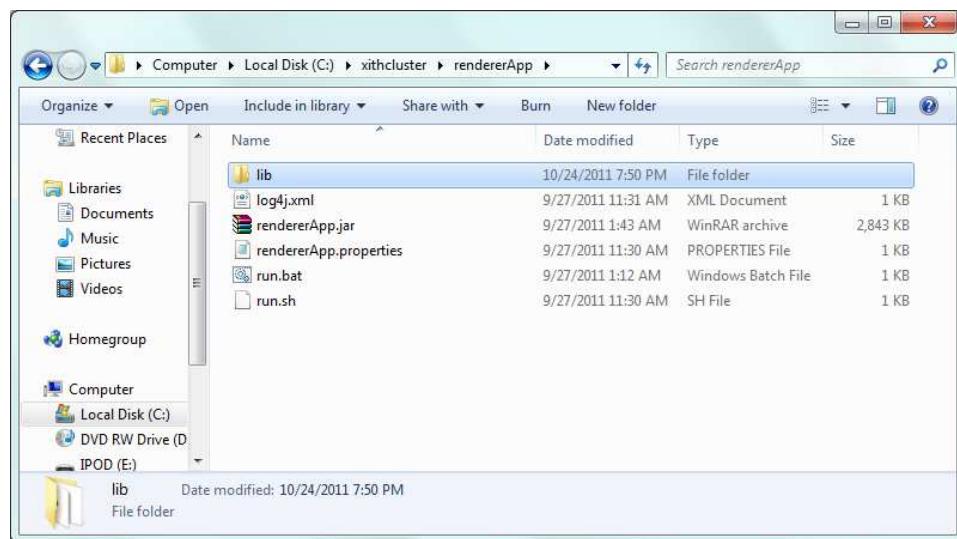
Copie a pasta "lib".



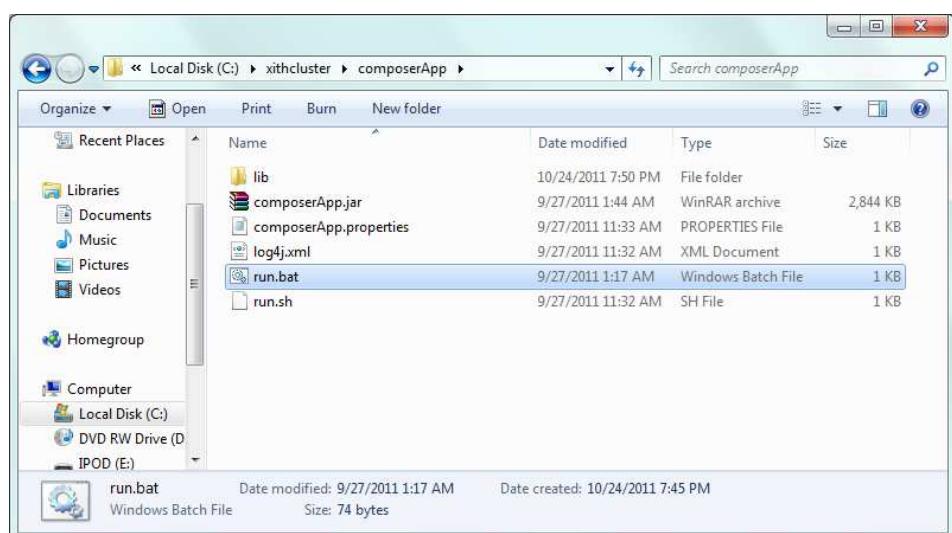
Coloque a pasta "lib" dentro da pasta "composerApp".



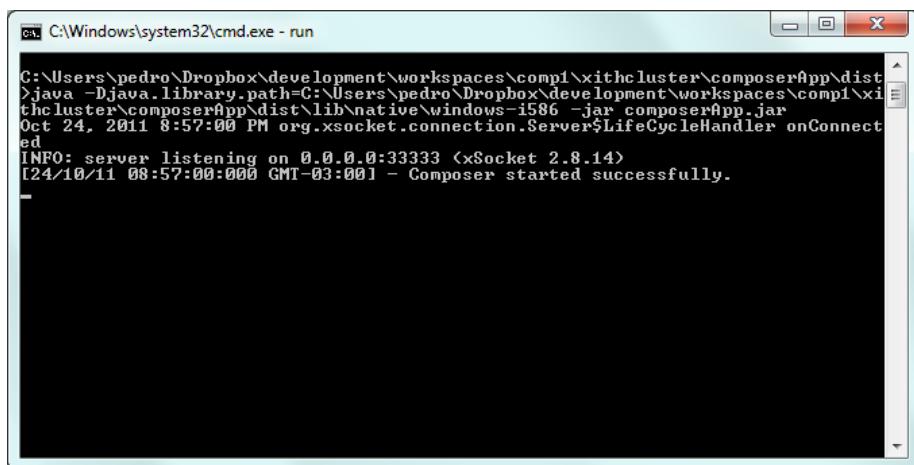
Cole também a pasta "lib" dentro da pasta "rendererApp".



Explore novamente a pasta "composerApp" e execute o arquivo "run.bat".

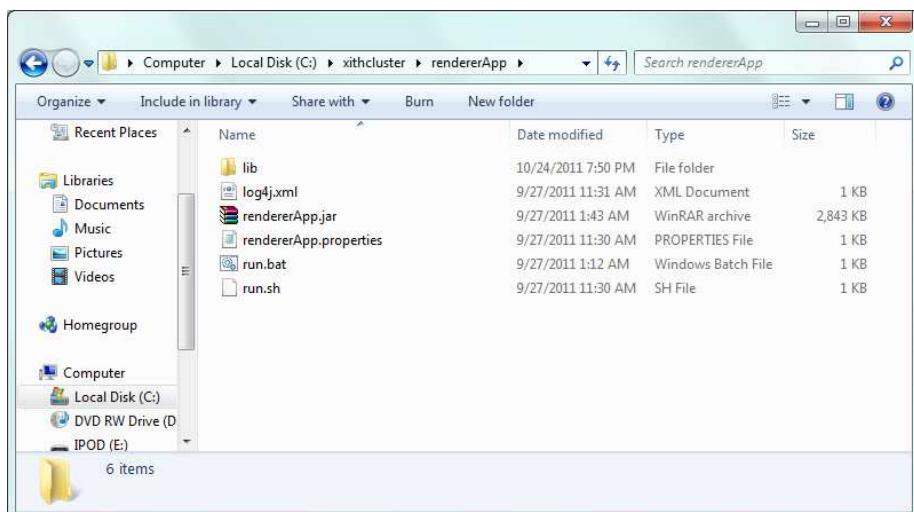


A mensagem no console indica que a aplicação compositora está executando com sucesso.

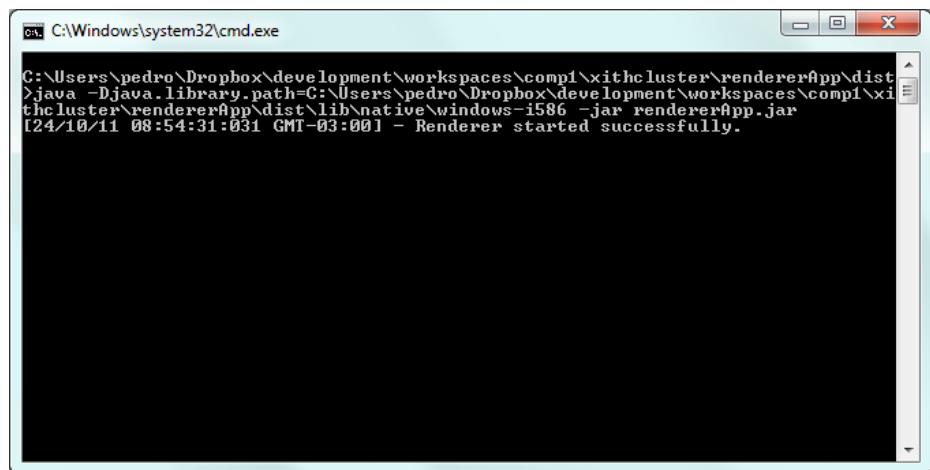


```
C:\Windows\system32\cmd.exe - run
C:\Users\pedro\Dropbox\development\workspaces\comp1\xithcluster\composerApp\dist>java -Djava.library.path=C:\Users\pedro\Dropbox\development\workspaces\comp1\xithcluster\composerApp\dist\lib\native\windows-i586 -jar composerApp.jar
Oct 24, 2011 8:57:00 PM org.xsocket.connection.Server$LifeCycleHandler onConnect
ed
INFO: server listening on 0.0.0.0:33333 (xSocket 2.8.14)
[24/10/11 08:57:00:000 GMT-03:00] - Composer started successfully.
```

Explore a pasta "rendererApp" e execute o arquivo "run.bat".

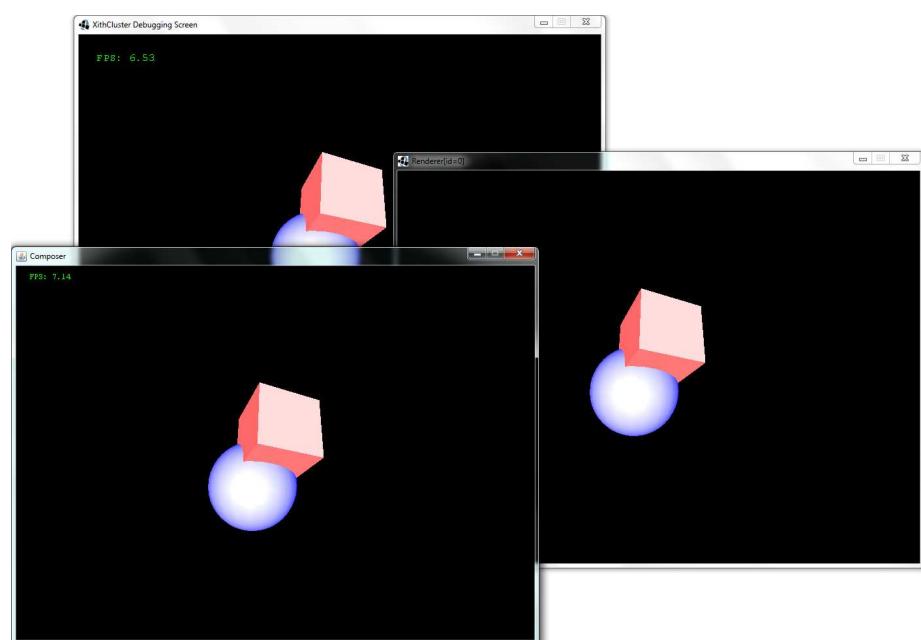


A mensagem no console indica que a aplicação renderizadora está executando com sucesso.



```
C:\Users\pedro\Dropbox\development\workspaces\compi\xithcluster\rendererApp\dist>java -Djava.library.path=C:\Users\pedro\Dropbox\development\workspaces\compi\xithcluster\rendererApp\dist\lib\native\windows-i586 -jar rendererApp.jar [24/10/11 08:54:31:031 GMT-03:00] - Renderer started successfully.
```

Por fim, a três janelas serão exibidas, como na ilustração abaixo.



# Apêndice C

Segue abaixo o artigo publicado no VIII Workshop de Realidade Aumentada e Virtual (WRVA) em Novembro de 2011.

## XithCluster: Uma Biblioteca de Grafo de Cena Distribuído

*Pedro Boechat de Almeida Germano*

*UniverCidade – Centro Universitário da  
Cidade*

*pboechat@gmail.com*

*Luciano Pereira Soares*

*Luciano Pereira Soares*

*PUC-RJ – Pontifícia Universidade Católica  
do Rio de Janeiro*

*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, 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, ela realiza 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 o mecanismo de sincronismo frame-lock.*

### 1 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, o uso dos *clusters* tornou-se 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.

A primeira sessão apresenta a introdução e motivação desta pesquisa, embasando o leitor em alguns tópicos abordados por ela. A sessão 2 relaciona alguns trabalhos relacionados que foram analisados para a criação da solução apresentada. A sessão 3 discorre sobre a arquitetura dessa solução, explicitando as ideias mais importantes por trás de sua elaboração. Por fim, a sessão 4 apresenta a conclusão e as considerações finais deste trabalho.

## 2 Trabalhos Relacionados

### 2.1 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.

### 2.2 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].

### 2.3 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 RV 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.

## 3 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*).

### 3.1.1 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.

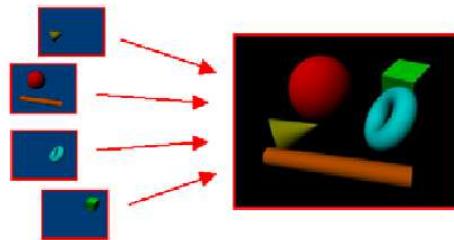


Figura 1. Redistribuição por Sort-Last [2]

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.

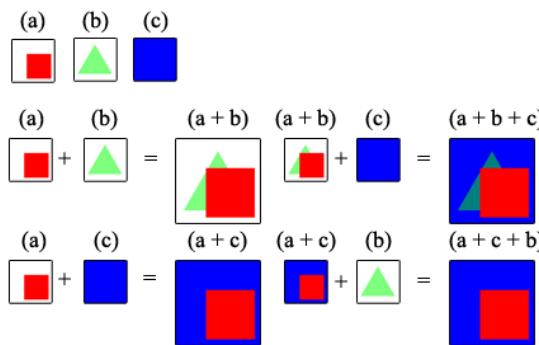
### 3.2 Composição de Imagens

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 geralmente 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 das composições influenciando imagem final**

## 4 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 disponibilizados.

O sistema é compreendido por uma biblioteca de classes (JAR) 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.

### 4.1 Componentes de Arquitetura

A arquitetura do XithCluster apresenta três componentes: mestre (Master), renderizadores (Renderers) e compositor (Composer). A interação entre eles está representada em um modelo simples

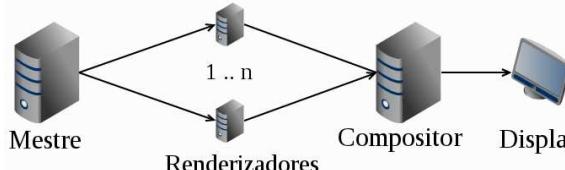
na [Figura 3](#).

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 pré-estabelecidos (ver Figura 4 abaixo) 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*).



**Figura 3. Modelo arquitetural simplificado**

## 4.2 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.

## 4.3 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 instânciação desnecessária de objetos (*Garbage Collection*).

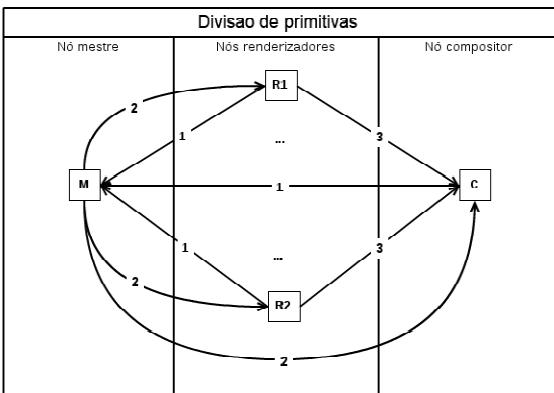
A camada de comunicação está presente em todos os componentes de arquitetura da solução e em todos eles atua de forma transparente. Sua responsabilidade é intermediar a troca de mensagens entre os componentes da arquitetura, reforçando o uso do protocolo desenvolvido. Ela atua tanto sincronismo da troca 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].

### 4.3.1 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. O pontilhamento da seta significa que a mensagem é não mandatória, mas pode ocorrer em determinadas circunstâncias. Todas as setas foram enumeradas de acordo com a ordem em que as mensagens devem ser trocadas para a execução correta do cenário.



**Figura 4.** Troca de mensagens durante a distribuição de primitivas

Nº	Descrição	Emissor	Rept.	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 dos rótulos das mensagens

do diagrama de divisão de primitivas

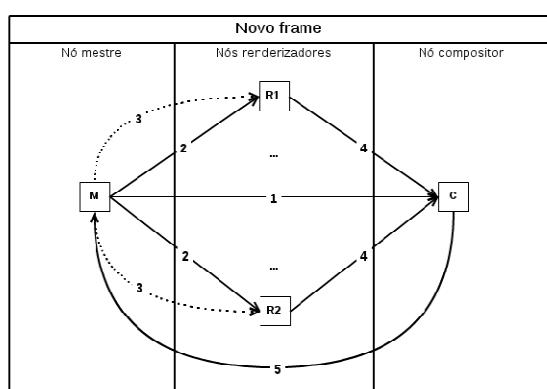
Nº	Descrição	Emissor	Rept.	Conteúdo
1	Solicitação da quantidade quadros a descartar	M	C	Quantidade de quadros
2	Sinal de novo quadro	M	R e C	Número do quadro corrente
3	Atualizações de cena (opcional)	M	R	Descrição das últimas alterações na cena pertinentes ao nó
4	Imagen rasterizada	R	C	buffers de cor,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

**Tabela 2.** Legenda dos rótulos das mensagens do diagrama de renderização de novo frame

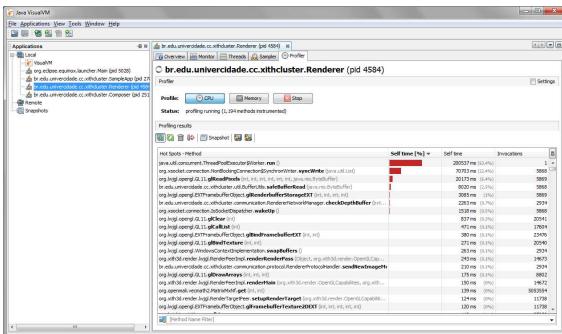
## 5 Resultados

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 5). Ela é de uso gratuito e vem embutida juntamente com o kit de desenvolvimento java (JDK) da Oracle.

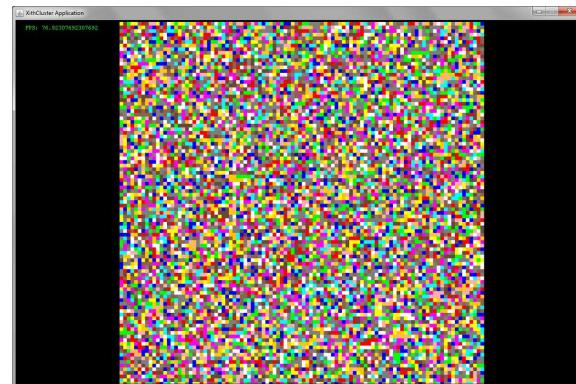
Para a implementação desta avaliação foram necessárias três aplicações de teste. As aplicações visam representar diferentes cenários de dados: um com cem (Figura 6), outro com mil (Figura 7) e outro dez mil (Figura 8) polígonos.



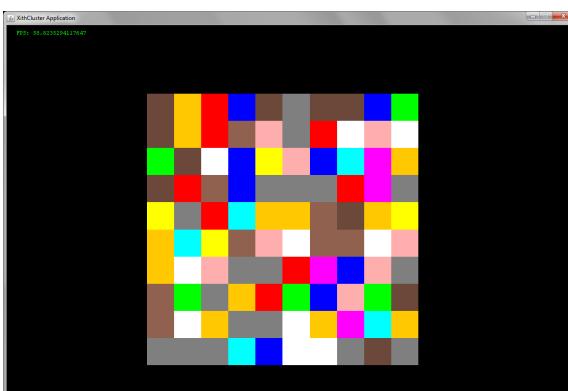
**Figura 5.** Troca de mensagens durante renderização de novo frame



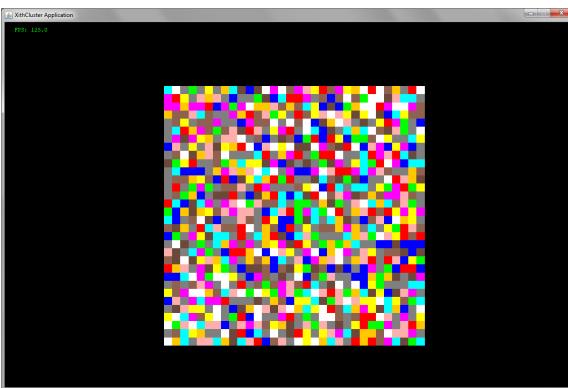
**Figura 5:** Interface gráfica da ferramenta jvisualvm no modo profiler



**Figura 8:** A imagem gerada pelo teste com dez mil polígonos renderizados simultaneamente

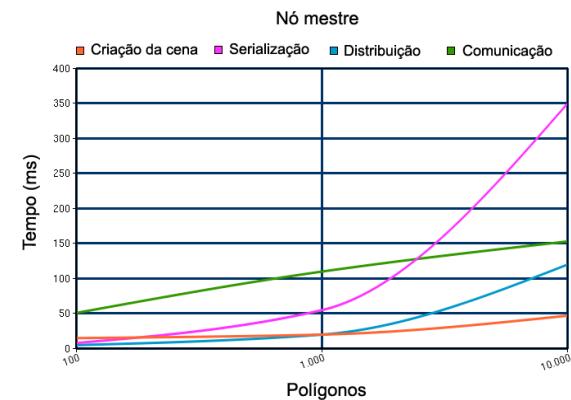


**Figura 6:** Imagem gerada pelo teste com cem mil polígonos renderizados simultaneamente

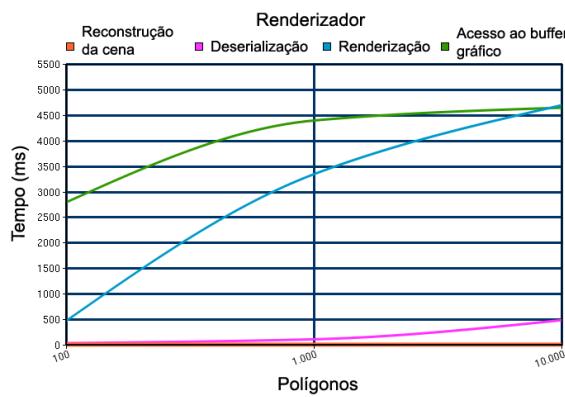


**Figura 7:** Imagem gerada pelo teste com mil polígonos renderizados simultaneamente

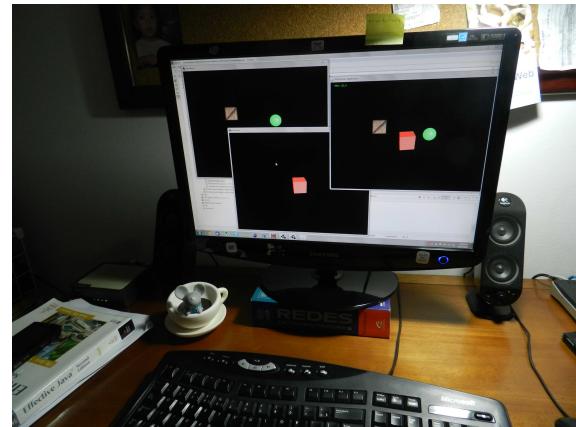
Os gráficos abaixo apresentam o tempo das operações mais importantes em cada um dos cenários anteriores. Os testes procedidos tiveram como parâmetro de equidade a geração de mil quadros. Os testes foram realizados em uma única máquina e com apenas uma instância de renderizador executando por vez.



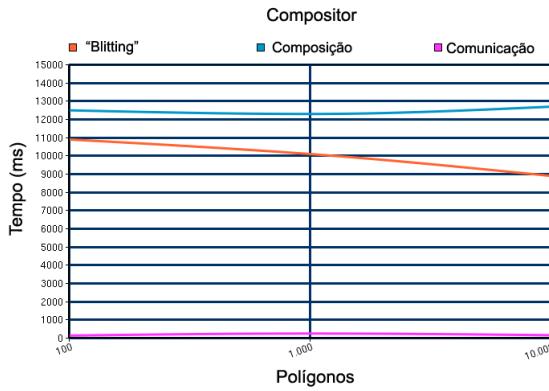
**Figura 9:** É possível observar a forte relação entre a operação de serialização e o número de polígonos da cena



**Figura 10:** A dependência observada entre a deserialização e o número de polígonos é baixa



**Figura 12:** Uma aplicação implementada com o XithCluster executando na plataforma Windows



**Figura 11:** A observação sobre o compositor aponta a inexistência de uma relação entre performance e o número de polígonos

## 6 Conclusões

O XithCluster provê meios para a construção de aplicações de RV distribuídas em Java sendo então naturalmente multiplataforma (Figura 12). 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.

## 7 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/vmvisualvm/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 *Proceeding of The 1st Ibero-American Symposium in Computer Graphics* (Guimarães , Portugal , Julho, 2002).

# Referências

AHN, H. S., 2007, "OpenGL Pixel Buffer Object". Personal Homepage. Disponível em: <[http://www.songho.ca/opengl/gl\\_pbo.html](http://www.songho.ca/opengl/gl_pbo.html)>. Acesso em: 7 dez. 2011.

ALCORN, B. RANDALL, F., 2002, "Parallel Image Composing API", *IEEE Visualization Conference*, Boston, Massachusetts, 2002. Disponível em: <[https://computing.llnl.gov/vis/images/pdf/IEEE\\_Viz\\_2002\\_rjf.pdf](https://computing.llnl.gov/vis/images/pdf/IEEE_Viz_2002_rjf.pdf)>. Acesso em: 13 jun. 2011.

ANDREWS, G. R., 2000, *Foundations of Multithreaded, Parallel, and Distributed Programming*. 1 ed. Addison Wesley.

BONDI, B. A., 2000, "Characteristics of Scalability and Their Impact on Performance". *Proceedings of the 2nd International Workshop on Software and Performance*, Ontario, Canada, pg. 195-203.

BRINKMANN, P. GUNN, C. WEIBMANN, S., 2010, "jReality: Interactive Audiovisual Applications Across Virtual Environments".

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

CLARK, R., 1976, "Hierarchical Geometric Models for Visible Surface Algorithms". *Communications of the ACM*, Califórnia, v. 10 (Out).

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

CONCI, A. *Material didático da cadeira Computação Gráfica I*. Disponível em: <<http://www.ic.uff.br/~aconci/z-buffer.pdf>>. Acesso em 13. jun. 2011.

CORBA - Common Object Request Broker Architecture Standard Webpage. Disponível em: <<http://www.corba.org/>>. Acesso em: 13 jun. 2011.

DMX – Distributed Multihead X Project Homepage. Disponível em: <<http://dmx.sourceforge.net/>>. Acesso em: 13 jun. 2011.

DUFF, T., 1985, “Compositing 3D Rendered Images”. *Computer Graphics (Proceedings of SIGGRAPH '85)*, v. 19, n. 3 (Jul).

DROSDEK, A., 2002, *Estrutura de Dados e Algoritmos em C++*. 1 ed. Tomson Learning.

EBERY, D., 2006, *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. 2 ed. Morgan Kauffman.

ECKEL, G. KEMPF, R. WENNERBERG, L., 1998, *OpenGL Optimizer Programmer's Guide: An Open API for Large-Model Visualization*. Silicon Graphics.

FOLEY, J. DAM, A. V. FEINER, S. HUGHES, J., 1996, *Computer Graphics: Principles and Practice*. 1 ed. Addison Wesley.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J., 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 ed. Addison Wesley.

HOHPE, G. WOOLF, B., 2003, *Enterprise Integration Patterns*. Addison-Wesley.

*JAGaToo Homepage*. Disponível em: <<http://sourceforge.net/projects/jagatoo/>>. Acesso em: 13 de jun. 2011.

*The Java Tutorials: Concurrency*. Disponível em:

<<http://download.oracle.com/javase/tutorial/essential/concurrency/index.html>>.

Acesso em: 13 jun. 2011.

*Jinput Project Homepage*. Disponível em: <<http://java.net/projects/jinput/>>. Acesso em: 13 jun. 2011.

*JOGL Project Homepage*. Disponível em: <<http://java.net/projects/jogl/>>. Acesso em: 13 jun. 2011.

*JOODE Homepage*. Disponível em: <<http://joode.sourceforge.net/>>. Acesso em 13 jun. 2011.

*JOPS – Java Open Particle System Homepage*. Disponível em:

<<http://www.ohloh.net/p/jops>>. Acesso em: 13 jun. 2011.

*LWJGL: Light Weight Java Game Library Homepage*. Disponível em: <<http://lwjgl.org/>>. Acesso em: 13 jun. 2011.

*MIDAS: Remote Rendering Services Project Homepage*. Disponível em:

<<https://computing.llnl.gov/vis/midas.shtml>>. Acesso em: 13 jun. 2011.

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

*OpenGL Homepage*. Disponível em: <<http://www.opengl.org/>>. Acesso em 13 jun. 2011.

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

RAFFIN, B. ARCILA, T. ALLARD, J. MÉNIER, C. BOYER, E., 2006, “FlowVR: A Framework For Distributed Virtual Reality Applications”. *Journees de l'AFRV*, Rocquencourt, França.

REINERS, D., 2002, *OpenSG: A Scene Graph System for Flexible and Efficient Real-time Rendering for Virtual and Augmented Reality Applications*. Tese de D.Sc., Universidade de Darmstadt, Alemanha.

ROST, J. R., 2006, *OpenGL Shading Language*, 2 ed., Addison Wesley.

SCHAEFFER, B. GOUDSEUNE, C., 2003, Syzygy: “Native PC Cluster RV”. *IEEE Virtual Reality Conference 2003*, Los Angeles, Califórnia, EUA.

SCHMIDT, D. C. STAL, M. ROHNERT, H. BUSCHMANN, F., 2000, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, v.2, Wiley & Sons, Nova Iorque, EUA.

*SimpleSceneGraph Project Homepage*. Disponível em: <<http://code.google.com/p/simplescenegraph/>>. Acesso em 13 jun. 2011.

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.

SOARES, L. P. RAFFIN, B. JORGE, A. J., 2008, “PC Clusters for Virtual Reality”, *The International Journal of Virtual Reality*.

STANEKER, D., 2005, *Hardware-assisted Occlusion Culling for Scene Graph Systems*. Tese de D.Sc., Universidade de Tubingen, Alemanha.

TANENBAUM, A., 2003, *Redes de Computadores*. 4 ed. Campus.

WAND, M., 2004, *Point-based multi-resolution rendering*. Tese de D.Sc., Universidade de Tubingen, Alemanha.

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

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