OLAVO DA ROSA BELLOC

**Uma Arquitetura de Comunicação Escalável para
Sistemas de Visualização Imersivos**

São Paulo
2016

OLAVO DA ROSA BELLOC

**Uma Arquitetura de Comunicação Escalável para
Sistemas de Visualização Imersivos**

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para obtenção
do título de Doutor em Ciências.

São Paulo
2016

OLAVO DA ROSA BELLOC

**Uma Arquitetura de Comunicação Escalável para
Sistemas de Visualização Imersivos**

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para obtenção
do título de Doutor em Ciências.

Área de Concentração:

Sistemas Eletrônicos

Orientador:

Prof. Dr. Marcelo Knörich Zuffo

São Paulo
2016

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catálogo-na-publicação

Belloc, Olavo da Rosa

Uma Arquitetura de Comunicação Escalável para Sistemas de Visualização Imersivos / O. R. Belloc — versão corrigida — São Paulo, 2016.
108 p.

Tese (Doutorado) – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1. Realidade Virtual 2. Sistemas Distribuídos 3. OpenGL I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos. II.t.

Dedico este trabalho à Deus, à minha esposa Edilaine
e ao meu filho Daniel.

AGRADECIMENTOS

Ao Prof. Dr. Marcelo Knörich Zuffo, pelo apoio, disponibilização de recursos e de tempo para a realização desta pesquisa.

À minha esposa Edilaine, pelo apoio, paciência e compreensão ao longo de todo este trabalho.

À toda equipe do Núcleo de Realidade Virtual do LSI-USP e aos colegas de pós-graduação Marcio Cabral e Rodrigo Ferraz, pelas dicas e conselhos.

À Deus, porque todo o conhecimento pertence à Ele.

RESUMO

A complexidade dos sistemas de visualização imersivos pode variar tremendamente conforme a sua aplicação. Algumas ferramentas mais simples fazem uso de um único óculos de Realidade Virtual como infraestrutura de visualização. No entanto, aplicações mais complexas, como simuladores e outras ferramentas de treinamento, podem necessitar de uma infraestrutura distribuída, contendo diversos computadores e telas.

Alguns simuladores e outras aplicações de treinamento fazem uso frequente de periféricos sofisticados de interação, que reproduzem de maneira fiel os elementos encontrados no cenário real. Além disto, o espaço de treinamento pode ser compartilhado por dois ou mais usuários. Estes requisitos acabam por impor o uso de sistemas de visualização complexos e distribuídos, que visam cobrir de maneira quase completa o campo de visão destes usuários.

Por causa das características deste tipo de sistema, as aplicações desenvolvidas nestes cenários são inerentemente complexas, pois frequentemente consideram aspectos específicos da infraestrutura para realizar a distribuição e o sincronismo da cena virtual. Esta complexidade dificulta o desenvolvimento, a manutenção e a interoperabilidade destas ferramentas.

Este trabalho apresenta uma arquitetura de comunicação para promover o uso de sistemas imersivos de forma simples e transparente para as aplicações, viabilizando o uso de infraestruturas complexas e distribuídas. A arquitetura proposta utiliza o mecanismo de substituição do *driver* OpenGL para obter, de forma automática, a distribuição do aspecto gráfico das aplicações. Apesar deste conceito já ter sido discutido na literatura, esta proposta apresenta um conjunto de técnicas para contornar as limitações inerentes desta abordagem e obter ganhos de desempenho significativos, com resultados consistentes em um amplo conjunto de infraestruturas.

As técnicas apresentadas neste trabalho sugerem, entre outras coisas, o uso de recursos modernos do padrão OpenGL para reduzir o volume de comunicação entre CPU e GPU. Um dos recursos avaliados foi o uso de mecanismos de renderização indireta, onde a aplicação armazena os comandos de renderização na memória da placa gráfica. Juntamente com esta técnica, o trabalho também investigou o uso de um algoritmo de *culling* na própria GPU, o que permitiu que esta otimização fosse utilizada mesmo em sistemas com arranjos mais complexos de tela.

Os resultados obtidos mostram que a aplicação pode exibir o seu conteúdo em um conjunto amplo de sistemas imersivos, contendo mais resolução e mais geometria visível, sem deteriorar o seu desempenho. Os testes foram conduzidos em diferentes infraestruturas e com cenas de tamanhos variáveis. Nos casos mais complexos, as técnicas propostas podem reduzir em 86% o tempo médio de renderização, quando comparadas com as abordagens tradicionais.

Palavras-chave: Realidade Virtual. Sistemas Distribuídos. OpenGL.

ABSTRACT

The complexity of immersive visualization systems can vary tremendously depending on their application. Some simple tools might only require a conventional virtual reality goggle as a visualization infrastructure. However, more complex applications, such as simulators and other training tools, might require a distributed infrastructure, containing several computers and screens.

Some training applications and simulators invariably make use of physical peripherals for interaction, which are designed to faithfully reproduce the elements found in real scenarios. Furthermore, the training area may be shared by two or more users. These requirements usually impose the use of complex and distributed imaging systems, which are intended to cover almost the entire field of view of the users involved.

Because of the characteristics of this type of system, the applications developed for these infrastructures are inherently complex. They are required to consider specific aspects of the infrastructure itself to carry out the distribution and synchronization of the virtual scene. This complexity hampers the development, maintenance and interoperability of these tools.

This work presents a communication architecture to promote the use of immersive systems by allowing applications to use complex and distributed infrastructures in a simple and transparent way. The proposed architecture uses the approach of replacing the OpenGL driver to transparently achieve graphics distribution. Although this has already been discussed in the literature, this document presents a set of techniques to overcome the inherent limitations of this approach and ultimately achieve significant performance gains, with consistent results across a broad range of infrastructures.

The techniques presented here suggest, among other things, the use of modern features of the OpenGL standard to reduce the communication overhead between CPU and GPU. One of the features evaluated was the usage of indirect rendering, where the application stores all the rendering commands in the graphics card dedicated memory. Along with this feature, the work also investigated the use of a culling algorithm on the GPU itself, which allowed this optimization to be used even on systems containing screens with a more complex layout.

The results show that the application can render its content in a wide range of immersive systems, with higher resolution and more visible geometry, without degrading its performance. The tests were conducted at different infrastructures and scenes with variable sizes. In the more complex use cases, the proposed techniques can reduce by up to 86% the average rendering time, when compared to the traditional approaches.

Keywords: Virtual Reality. Distributed Systems. OpenGL.

LISTA DE FIGURAS

1	Instalação <i>The StarCAVE</i> , no EVL.	22
2	Instalação <i>Reality Deck</i> , na Universidade de Stony Brook.	23
3	Infraestrutura e aplicação na Caverna Digital da USP.	25
4	<i>Sort-First</i>	27
5	<i>Sort-Middle</i>	28
6	<i>Sort-Last</i>	29
7	Exemplo de falta de sincronismo.	30
8	Camadas de <i>software</i>	33
9	Interface uniforme de comunicação.	44
10	Interface de comunicação OpenGL.	45
11	Segmentos de um <i>frustum</i> convencional, estereoscópico e curvo.	47
12	Aumentando o aproveitamento dos estados no OpenGL.	53
13	Custo aproximado para mudança de estados - fora de escala.	56
14	Sequencia de <i>culling</i> em sistema imersivo.	58
15	<i>Culling</i> hierárquico vs paralelo.	59
16	Diagrama de comunicação no aglomerado gráfico.	63
17	Sincronismo realizado via <i>multicast</i> /UDP e TCP.	66
18	<i>Buffers</i> e <i>shaders</i> utilizados no modo <i>ubosub</i>	71
19	<i>Buffers</i> e <i>shaders</i> utilizados no modo <i>uborange</i>	73
20	<i>Buffers</i> e <i>shaders</i> utilizados no modo <i>indexedmdi</i>	76
21	Visão geral das infraestruturas de teste.	81
22	Modelo CAD da placa GeForce utilizado nos testes.	83
23	Percurso realizado pela câmera virtual.	84

24	Tempo médio de renderização (local).	86
25	Tempo médio de renderização (1 máquina remota).	88
26	Taxa de transferência na rede.	90
27	Tempo médio de renderização (3 máquinas remotas).	93
28	Tempo médio de renderização (9 máquinas remotas).	94
29	Tempo médio de renderização local (máquina atualizada).	97

LISTA DE TABELAS

1	Camada de <i>software</i> dos arcabouços.	38
2	Suporte à renderização paralela nos arcabouços.	39
3	Número de chamadas de renderização por quadro (local).	85
4	Tempo médio de renderização local (milissegundos).	87
5	Tempo médio de renderização com 1 máquina remota (milissegundos). . . .	89
6	Número de chamadas de renderização por quadro (3 máquinas remotas). .	92
7	Tempo médio de renderização com 3 máquinas remotas (milissegundos). . .	93
8	Tempo médio de renderização com 9 máquinas remotas (milissegundos). . .	95
9	Número de chamadas de renderização por quadro (9 máquinas remotas). .	96
10	Tempo médio de renderização local (máquina atualizada, milissegundos). .	97

LISTA DE ABREVIATURAS

API	<i>Application Programming Interface</i>
ARB	<i>Architecture Review Board</i>
CAVE	<i>CAVE Automatic Virtual Environment</i>
CPU	<i>Central Processing Unit</i>
CUDA	<i>Compute Unified Device Architecture</i>
EVL	<i>Electronic Visualization Laboratory</i>
GPU	<i>Graphics Processing Unit</i>
HMD	<i>Head Mounted Display</i>
HPC	<i>High Performance Computing</i>
LSI	Laboratório de Sistemas Integráveis
MPI	<i>Message Passing Interface</i>
OpenCL	<i>Open Computing Language</i>
OpenGL	<i>Open Graphics Library</i>
POR	Pipeline das Operações de Rasterização
RV	Realidade Virtual
SDK	<i>Software Development Kit</i>
SSBO	<i>Shader Storage Buffer Object</i>
TCP	<i>Transmission Control Protocol</i>
UBO	<i>Uniform Buffer Object</i>
UDP	<i>User Datagram Protocol</i>
UML	<i>Unified Modeling Language</i>
USP	Universidade de São Paulo
VBO	<i>Vertex Buffer Object</i>

SUMÁRIO

1	Introdução	14
1.1	Hipótese	16
1.2	Motivação	16
1.3	Objetivos	18
1.4	Contribuições	19
1.5	Estrutura do trabalho	19
2	Sistemas de visualização imersivos	21
2.1	Perspectiva histórica	21
2.1.1	A Caverna Digital da USP	24
2.2	Conceitos fundamentais	25
2.2.1	Renderização paralela	26
2.2.1.1	<i>Sort-First</i>	26
2.2.1.2	<i>Sort-Middle</i>	27
2.2.1.3	<i>Sort-Last</i>	28
2.2.2	Sincronismo	29
2.3	Síntese	31
3	Revisão da literatura	32
3.1	Camadas de <i>software</i>	33
3.1.1	Camada do <i>driver</i> gráfico	34
3.1.2	Camada do grafo de cena	36
3.1.3	Camada da aplicação	37
3.2	Renderização paralela	38

3.3	Análise e comparação	40
3.4	Síntese	41
4	Arquitetura de comunicação	42
4.1	Requisitos	42
4.2	Visão geral	43
4.3	Considerações sobre sistemas imersivos	47
4.4	Redução da taxa de transferência de dados	49
4.4.1	Consolidação de estados do OpenGL	50
4.4.2	Renderização indireta	54
4.4.3	Ordenação dos estados	55
4.5	Algoritmo de <i>frustum culling</i>	57
4.6	Síntese	60
5	Considerações sobre a implementação	62
5.1	Visão geral	62
5.2	<i>Driver</i> ClusterGL	64
5.2.1	Protocolo <i>multicast</i> e o sincronismo	65
5.2.2	Desafios na adaptação do <i>driver</i> OpenGL moderno	67
5.3	Aplicação gráfica para <i>benchmark</i>	68
5.3.1	Modo <i>ubosub</i>	70
5.3.2	Modo <i>uborange</i>	72
5.3.3	Modo <i>indexedmdi</i>	74
5.3.4	Modo <i>indexedmdi_unified</i>	78
5.4	Síntese	79
6	Resultados	80
6.1	Infraestrutura de testes	80

6.2	Modelo da cena virtual	82
6.3	Movimentação da câmera e amostragem	83
6.4	Análise dos resultados	85
6.4.1	Uma única máquina remota	88
6.4.2	Três máquinas remotas (<i>Powerwall</i>)	91
6.4.3	Nove máquinas remotas (<i>Caverna</i>)	94
6.4.4	Uma máquina local com especificações atualizadas	96
6.5	Síntese	98
7	Conclusão	100
7.1	Desafios pendentes	102
7.2	Outras abordagens e trabalhos futuros	103
	Referências	105

1 INTRODUÇÃO

Os sistemas de visualização imersivos são compostos por uma infraestrutura cuja principal função é inserir seus usuários em um ambiente virtual reproduzido por computadores. Estes sistemas podem ser usados em diversas aplicações, como simuladores, ferramentas de treinamento, avaliação de ergonomia, design, avaliação de protótipos de engenharia, entre outros.

A complexidade destas instalações pode variar de acordo com o seu propósito. Os sistemas que visam acomodar um número elevado de pessoas simultaneamente serão necessariamente maiores e conseqüentemente, precisarão de telas com grandes dimensões, imagens com altíssima resolução e contraste. Em contrapartida, sistemas que visam acomodar um único usuário podem apresentar requisitos mais simples, utilizando-se até mesmo de um óculos de Realidade Virtual como infraestrutura de visualização.

Neste trabalho, um sistema de visualização imersivo corresponde apenas aos elementos da infraestrutura que são responsáveis por renderizar e apresentar ao usuário as imagens do ambiente virtual. Além destes elementos de visualização, um sistema completo possui também uma infraestrutura computacional, rastreadores de posição, infraestrutura de áudio, controles para interação e outros periféricos auxiliares. Esta infraestrutura completa pode ser identificada como um Sistema de Realidade Virtual.

Além da infraestrutura de *hardware*, sistemas de Realidade Virtual (RV) utilizados para fins profissionais visam à apresentação de modelos tridimensionais complexos e de grande porte como, por exemplo, modelos completos de carros, aeronaves, plataformas de petróleo, turbinas hidroelétricas, entre outros. Para visualizar estes modelos de forma realista e interativa, é fundamental dispor de uma ferramenta de *software* que seja capaz de fazer uso adequado da capacidade computacional disponível.

Nos casos mais complexos, onde o sistema de visualização consiste em um número maior de telas - para manter uma resolução adequada neste cenário, muitas vezes se torna necessário fazer uso de um número elevado de computadores, pois a resolução propor-

cionada por um único computador não é suficiente para todo o sistema.

A alternativa mais comum nestes casos é fazer uso de um aglomerado de computadores convencionais (*cluster*) integrados através de uma rede de alta velocidade. Na época em que os primeiros sistemas de RV foram propostos, estes sistemas utilizavam supercomputadores gráficos como sua infraestrutura computacional (CRUZ-NEIRA et al., 1992; CRUZ-NEIRA; SANDIN; DEFANTI, 1993). Estes supercomputadores, fornecidos em sua maioria pela *Silicon Graphics* (SGI), possuíam uma arquitetura diferente dos computadores pessoais e apresentavam altos custos de aquisição e manutenção. Atualmente, os aglomerados de computadores convencionais podem apresentar desempenho superior a um supercomputador gráfico de custo equivalente (RAFFIN et al., 2006).

No entanto, ao utilizar um aglomerado de computadores na montagem de um sistema de RV, diversos problemas fundamentais precisam ser equacionados, entre eles, podemos mencionar: a distribuição da cena virtual entre os diversos computadores, a atualização desta cena durante a execução do programa e o sincronismo na apresentação das imagens renderizadas. Normalmente, cada uma das máquinas do aglomerado estará conectada à monitores ou projetores diferentes. Para manter uma coerência visual para o usuário do sistema, estas máquinas não devem apresentar inconsistência nas imagens exibidas, nem atrasos perceptíveis.

Todos estes problemas são recorrentes em um sistema de visualização imersivo. Para resolvê-los, diversas ferramentas ou bibliotecas de *software* já foram apresentadas no meio acadêmico e se tornaram soluções populares - utilizadas em diversas instalações de RV. Para se adotar uma das soluções existentes atualmente é necessário que toda a infraestrutura computacional seja previamente configurada - preparada para executar uma determinada aplicação. Normalmente, este procedimento é complexo e envolve a instalação de diversos programas, além de configurações específicas que devem ser efetuadas em cada computador do aglomerado.

No entanto, cada uma destas ferramentas de *software* funciona de forma diferente, o que as tornam incompatíveis entre si. Desta forma, para executar uma aplicação em uma infraestrutura de RV é necessário que esta infraestrutura seja preparada e configurada com a mesma biblioteca de *software* utilizada para desenvolver a aplicação. Frequentemente, existem incompatibilidades até mesmo entre versões diferentes de uma mesma solução.

Por causa destas incompatibilidades, não é trivial elaborar aplicações que sejam capazes de executar em sistemas distintos de RV. Em muitos casos, um longo intervalo de tempo precisa ser dedicado à adaptação do *software* da aplicação e à configuração da

infraestrutura computacional em que a mesma será executada.

Em uma tentativa de amenizar estes problemas, muitos grupos e instituições de pesquisa acabam adotando uma única biblioteca de *software* para o desenvolvimento de todas as suas aplicações. Desta forma, as suas instalações de RV e as instalações de seus parceiros permanecem preparadas para executar as aplicações desenvolvidas com a solução adotada.

Os problemas mencionados podem ser justificados pela falta de padronização nas soluções de *software* existentes. Atualmente, algumas destas soluções apresentam abordagens diferentes para o mesmo problema e, por sua vez, possuem vantagens e desvantagens distintas quando comparadas entre si. Estas diferenças fundamentais e conceituais tornam inviável a criação de uma plataforma de *software* que permita a interoperabilidade entre todas as soluções existentes.

Este trabalho apresenta uma discussão sobre os elementos presentes em um sistema de visualização imersivo e quais os principais conceitos envolvidos, e propõe uma arquitetura de comunicação que pode ser utilizada para o desenvolvimento de soluções em *software*. O trabalho apresenta os requisitos desta arquitetura de comunicação e alguns aspectos de sua implementação. Embora este trabalho não se proponha a estabelecer um padrão para o uso de sistemas de visualização imersivos, a arquitetura proposta define os elementos básicos e estabelece uma abordagem prática para os problemas mencionados.

1.1 Hipótese

Esta tese de doutorado está fundamentada na hipótese de que é possível estabelecer uma arquitetura de comunicação uniforme para sistemas de visualização imersivos, que garanta alto desempenho e promova a interoperabilidade, com poucas restrições ao escopo das aplicações e sem impor limitações ao tipo de infraestrutura que possa se beneficiar deste modelo.

1.2 Motivação

Recentemente, surgiram diversas iniciativas para reduzir o custo e a complexidade dos sistemas de visualização imersivos - a maioria das abordagens com foco no desenvolvimento de óculos de RV. Apesar do sucesso inquestionável destes projetos, existem casos em que sistemas de grande porte são indispensáveis e não podem ser substituídos por

outras alternativas. Alguns dos casos de uso mais proeminentes são simuladores práticos que envolvem dois ou mais indivíduos compartilhando o mesmo espaço, por exemplo, simuladores de voo, de tanques de guerra, de submarino e do passageiro de embarcações. Estas aplicações exigem a colaboração de diversos indivíduos no mesmo espaço físico - o que inviabiliza a utilização de óculos.

Em outros casos mais simples, embora o simulador contemple apenas um único usuário, estes simuladores precisam reproduzir de forma fiel o ambiente (*cockpit*) ocupado pelo treinando, incluindo os instrumentos, os controles e outros elementos de interface. Neste grupo de aplicações, podemos mencionar como exemplo, alguns simuladores de voo, simuladores de guindastes, simuladores de veículos e a operação de outros maquinários complexos.

O intuito de reproduzir fielmente, no simulador, o lugar que será habitado pelo usuário no ambiente real, é proporcionar a familiarização com os elementos de interface existentes, o que é extremamente importante para a transferência do conhecimento obtido durante o treinamento. Desta forma, sistemas de visualização imersivos de grande porte são indispensáveis para um conjunto muito amplo de aplicações. Sendo que grande parte destas aplicações são relevantes para segmentos estratégicos da indústria, do comércio e de defesa.

Outro aspecto importante é que a complexidade destes sistemas tende a aumentar com o aprimoramento dos seus requisitos. Atualmente, diversos laptops e periféricos portáteis, como celulares e *tablets*, têm sido produzidos com telas de resolução retina. Isto significa que, para uma distância convencional de uso, o usuário não é capaz de fazer distinção entre os diferentes pixels presentes na tela destes periféricos. Ou seja, dado que a tela é observada a uma determinada distância, a resolução espacial desta tela é maior que a acuidade média do olho humano.

O uso contínuo destes periféricos em nosso cotidiano nos torna habituados a este nível de detalhe e resolução. Desta forma, o compromisso com a qualidade no desenvolvimento de sistemas imersivos se torna, naturalmente, compelido a acompanhar a evolução destes periféricos, e prover telas não apenas com dimensões maiores, mas também com maior densidade de pixel.

O Laboratório de Sistemas Integráveis da Universidade de São Paulo (LSI-USP), onde desenvolvo minhas atividades de pesquisa, possui um sistema de visualização imersivo conhecido como Caverna Digital, que consiste em cinco telas de projeção, sendo quatro telas laterais e uma no piso. Este sistema, inaugurado em 2001, sofreu uma série de adap-

tações visando acompanhar a evolução tecnológica deste período. Em sua inauguração, a infraestrutura computacional era composta por um único supercomputador da SGI, e cada tela utilizava um único projetor com resolução de 1024 x 1024. Atualmente, a sua infraestrutura computacional é composta por 15 computadores (3 por tela) e 30 projetores com resolução 1280 x 720 (6 por tela). Este panorama histórico, apresentado em maior detalhe no próximo capítulo, ilustra que o avanço da tecnologia e o aumento dos requisitos de qualidade destes sistemas influenciam em sua complexidade.

Outro aspecto importante, é que somente neste período em que realizo minhas atividades de pesquisa, surgiram diversas oportunidades para instalar sistemas semelhantes em instituições parceiras e outras organizações. Diante deste cenário, que envolve um aumento dos requisitos de qualidade e alta complexidade, a principal motivação deste trabalho é propor meios e discutir técnicas para facilitar, simplificar e promover a interoperabilidade no uso de sistemas de visualização imersivos.

1.3 Objetivos

O objetivo principal deste trabalho é estabelecer uma arquitetura de comunicação para sistemas de visualização imersivos. A função desta arquitetura é identificar os principais componentes deste tipo de infraestrutura, estabelecer os requisitos de alto nível e propor um modelo de comunicação para estes componentes, visando garantir alto desempenho gráfico (taxa de atualização de quadros), baixa latência, escalabilidade em termos de resolução e em termos de complexidade dos modelos geométricos.

O objetivo desta proposta é iniciar uma discussão sobre os impactos positivos e negativos de se adotar uma arquitetura uniforme de comunicação para sistemas de visualização imersivos. O amadurecimento destes conceitos pode promover, em longo prazo, um debate no meio acadêmico sobre a viabilidade de se estabelecer uma implementação de referência para uma determinada arquitetura de comunicação - o que promoveria uma eventual padronização de uma interface de comunicação para este tipo de infraestrutura de visualização e, conseqüentemente, maior interoperabilidade.

Para alcançar o objetivo principal deste trabalho, é necessário atingir alguns objetivos específicos, mencionados a seguir:

- Realizar um estudo aprofundado das principais soluções de *software* existentes, procurando identificar o seus mecanismos fundamentais de funcionamento;

- Identificar os requisitos necessários para um sistema de visualização imersivo;
- Elaborar uma arquitetura que contenha um modelo de comunicação para estes sistemas e que satisfaça os requisitos enumerados anteriormente;
- Implementar uma ferramenta de *software* com base na arquitetura proposta, com o objetivo de verificar e testar o funcionamento do modelo de comunicação;
- Analisar os resultados obtidos, avaliando as vantagens e desvantagens da abordagem proposta pela arquitetura.

1.4 Contribuições

As contribuições deste trabalho visam influenciar a maneira como os sistemas de visualização imersivos são utilizados, procurando estabelecer métodos e meios para facilitar e simplificar o uso destas infraestruturas, que hoje são parte fundamental dos Sistemas de Realidade Virtual. Estas contribuições podem ser enumeradas da seguinte forma:

- Uma proposta de arquitetura de comunicação para o uso de sistemas de visualização imersivos;
- Protótipo de uma ferramenta de *software*, desenvolvida para verificar e validar o modelo de comunicação apresentado;

Além destas, o trabalho visa proporcionar uma discussão mais ampla, sobre como o meio acadêmico pode tratar de forma mais uniforme este tipo de infraestrutura e quais são as implicações deste tipo de abordagem.

1.5 Estrutura do trabalho

No capítulo 2 são apresentadas algumas características importantes de sistemas de visualização imersivos, junto com uma breve perspectiva histórica, alguns conceitos fundamentais sobre estas instalações e algumas classificações importantes para sistemas de renderização paralela, que é um recurso usado recorrentemente neste tipo de infraestrutura.

O capítulo 3 apresenta uma revisão da literatura, através da investigação das ferramentas e bibliotecas de *software* utilizadas atualmente no desenvolvimento de aplicações

para sistemas imersivos de RV. Este capítulo estabelece a fundamentação teórica para a proposta de uma arquitetura de comunicação.

A proposta da arquitetura de comunicação é apresentada no capítulo 4. Esta arquitetura identifica os principais elementos de um sistema de visualização imersivo, e estabelece meios para uniformizar o uso deste tipo de infraestrutura.

O capítulo 5 apresentará as considerações sobre a implementação da ferramenta de *software* utilizada para verificar o funcionamento do modelo de comunicação e identificar as suas vantagens e desvantagens.

Para avaliar o funcionamento desta arquitetura, o capítulo 6 apresentará e discutirá os resultados obtidos através de medidas na ferramenta de *software*.

O capítulo 7 apresentará as conclusões finais desta tese, mencionando as contribuições realizadas e as considerações a serem feitas em trabalhos futuros.

2 SISTEMAS DE VISUALIZAÇÃO IMERSIVOS

Este capítulo apresenta as principais características de um sistema de visualização imersivo, juntamente com uma breve perspectiva histórica, e alguns conceitos fundamentais para este tipo de instalação. A perspectiva histórica apresenta uma visão geral sobre a evolução destes sistemas imersivos, mostrando algumas das instalações mais relevantes. Uma parte considerável destas instalações foi realizada pelo EVL (*Electronic Visualization Laboratory*), da Universidade de Illinois em Chicago, que foi um dos laboratórios pioneiros na concepção deste tipo de sistema.

2.1 Perspectiva histórica

Uma das instalações mais icônicas de Realidade Virtual foi montada em 1992 no EVL, por Cruz-Neira (CRUZ-NEIRA et al., 1992; CRUZ-NEIRA; SANDIN; DEFANTI, 1993). Conhecida como CAVE (*Cave Automatic Virtual Environment*) ou Caverna, este ambiente imersivo contava com quatro telas (3 laterais e o piso). Os projetores usados eram fabricados com tubos de raios catódicos (CRT), e possuíam resolução de 1024 x 768 (aprox. 1,2 megapixel por tela). As telas tinham de 2 a 3 metros de largura (dependendo da instalação e do tamanho disponível) e formavam um cubo quase completo, cobrindo toda a visão periférica do usuário. Os projetores eram posicionados atrás das telas para evitar a formação de sombras do usuário sobre a imagem.

O sistema ainda possuía estereoscopia ativa, rastreadores de posição e áudio com múltiplos canais. Na época, o uso de supercomputadores da SGI (*Silicon Graphics*) era muito comum neste tipo de instalação, pois eram as únicas máquinas com capacidade de processamento e saídas de vídeo com resolução suficiente para este tipo de sistema. Apesar do uso de supercomputadores, o desempenho da primeira Caverna era limitado à aproximadamente 10 quadros/segundo (DEFANTI et al., 2009a). Apenas o preço deste supercomputador representou 85% de todo o investimento realizado.

Devido aos altos custos de aquisição e manutenção dos supercomputadores gráficos,

Figura 1: Instalação *The StarCAVE*, no EVL.

Fonte: DeFanti et al. (2009a)

no início da década do ano 2000, diversos centros de pesquisa começaram a propor alternativas que consistiam na utilização de um conjunto de computadores gráficos convencionais (aglomerados gráficos), conectados por uma rede de alta velocidade (RAFFIN et al., 2003, 2006) . Nesta solução, ao invés de se utilizar um supercomputador com memória compartilhada, um conjunto de computadores convencionais é usado para alimentar os diversos projetores de um sistema imersivo.

Neste mesmo período, impulsionadas pelo crescente mercado de jogos eletrônicos, as placas gráficas para computadores convencionais apresentaram uma melhora significativa de desempenho. A popularização desta tecnologia reduziu os seus custos de produção, incentivando ainda mais o uso de computadores convencionais na construção de sistemas imersivos.

Em 2008, com a tecnologia de aglomerados gráficos consolidada, o Instituto para Telecomunicações e Tecnologia da Informação da Califórnia (Calit2), juntamente com a EVL, construiu uma nova instalação imersiva (DEFANTI et al., 2009a). Nesta instalação, de nome *StarCAVE*, foram utilizadas 16 telas e 34 projetores com resolução de 1920 x 1080p (aproximadamente 68 megapixels no total). As telas são dispostas em torno do usuário, como mostra a figura 1. Para renderizar o mundo virtual e prover as saídas gráficas para todos os projetores foram utilizados 17 computadores com 2 placas gráficas em cada computador. Nesta instalação, o custo do aglomerado de computadores representou

Figura 2: Instalação *Reality Deck*, na Universidade de Stony Brook.

Fonte: Papadopoulos, Petkov e Kaufman (2013)

apenas 10% de todo o investimento (DEFANTI et al., 2009a).

No ano de 2009, a empresa americana Mechdyne Corporation construiu um sistema cúbico (DEFANTI et al., 2011) (incluindo o piso e o teto), onde cada uma das 6 telas é iluminada por 4 projetores com resolução 4096 x 2160 pixels, totalizando aprox. 190 megapixels. Este sistema utiliza 96 placas gráficas, provavelmente distribuídas em 24 computadores (não mencionado no artigo). Esta instalação foi montada primeiramente na Universidade do Estado de Iowa, nos Estados Unidos, e posteriormente transferida para KAUST (*King Abdullah University of Science & Technology*), na Arábia Saudita.

Apesar de muitos sistemas imersivos utilizarem projetores atrás das telas (*rear projection*), este tipo de sistema só pode ser montado em uma sala com grandes dimensões, isto porque o projetor precisa de uma distância mínima da tela para conseguir cobrir toda a sua superfície. Outro aspecto negativo dos projetores é o calor e a degradação de suas lâmpadas, que exigem mais refrigeração e atividades constantes de manutenção. Para contornar estes problemas, muitos sistemas utilizam painéis de LCD ou LED. Uma desvantagem destas alternativas são as bordas do monitor, que apesar de pequenas em alguns modelos, são perceptíveis ao usuário e podem prejudicar a imersão.

Diversos sistemas com LCDs são apresentados por DeFanti (DEFANTI et al., 2009b), incluindo configurações com 16 painéis em um plano (4x4) e também 21 painéis (3x7) distribuídos em um arranjo cilíndrico em torno do usuário. Atualmente, o sistema imersivo de Realidade Virtual com maior resolução foi instalado em 2011/2012 na Universidade de Stony Brook nos Estados Unidos (PAPADOPOULOS; PETKOV; KAUFMAN, 2013). Este último sistema possui 416 painéis de LCD com resolução de 2560 x 1440, totalizando aprox.

1,5 bilhão de pixels. Conforme ilustrado na figura 2, os painéis estão distribuídos em uma grande sala, podendo acomodar diversos usuários simultaneamente. Para renderizar o ambiente virtual, foi utilizado um aglomerado de 18 máquinas, sendo que cada máquina possui 4 placas gráficas. Nesta configuração, cada placa gráfica possui 6 saídas de vídeo, com cada computador gerando imagens para 24 monitores.

Esta breve perspectiva histórica demonstra que embora a resolução individual dos projetores e monitores tenha aumentado neste período, os sistemas imersivos têm utilizado um número cada vez maior destes elementos, buscando uma resolução compatível com a acuidade espacial do olho humano. Para produzir imagens com tamanha resolução e disponibilizá-las de forma interativa ao usuário, se torna imprescindível dispor de um recurso computacional equivalente. Os exemplos apresentados mostram que o número de computadores e placas gráficas utilizados nestes sistemas também aumentou.

2.1.1 A Caverna Digital da USP

A Caverna Digital da Universidade de São Paulo é uma instalação localizada no Laboratório de Sistemas Integráveis (LSI) da Escola Politécnica. Esta instalação foi inaugurada em 2001 (ZUFFO et al., 2001), e foi a primeira infraestrutura desta natureza a ser implantada na América Latina. Naquele momento, o sistema era composto por 5 telas (4 lados e piso), de 3 x 3 metros cada, 5 projetores Marquee 9500 LC (260 ANSI lumens) e um supercomputador gráfico SGI Onyx 3000 com Infinity Reality 3. Com esta configuração, as aplicações exibiam imagens com resolução de 1024 x 1024 (1 megapixel por tela) e estereoscopia em 96 Hz (48 Hz para cada olho).

Pouco tempo depois, o supercomputador da *Silicon Graphics* foi substituído por um aglomerado de 6 máquinas Dual-Pentium III Xeon 1 GHz, com placas gráficas Oxygen GVX1 Pro. Esta mudança de infraestrutura foi pioneira na época, e visava à redução dos custos de aquisição e manutenção através da implantação de computadores e placas convencionais. O sucesso deste caso de uso promoveu a elaboração de um curso no Siggraph (KACZMARSKI; ZUFFO, 2002), realizado juntamente com outros laboratórios parceiros.

Assim como em outras instituições, a Caverna Digital da USP está em processo contínuo de aperfeiçoamento. Atualmente, conforme ilustrado na figura 3, a sua infraestrutura é composta por 15 computadores (3 por tela) e 30 projetores (6 por tela) com resolução 1280 x 720 (2500 ANSI lumens) em 120 Hz. Por causa das características deste último sistema, um *software* foi desenvolvido especificamente para facilitar o processo de calibração e alinhamento dos projetores que iluminam uma mesma tela (TEUBL et al., 2012).

Figura 3: Infraestrutura e aplicação na Caverna Digital da USP.

Fonte: Autor

Estes fatores são evidências da complexidade deste tipo de infraestrutura e como esta complexidade tem aumentado com a disponibilidade de novos recursos tecnológicos. Embora seja possível construir um sistema imersivo mais simples com HMDs (*Head-Mounted Displays*), estes sistemas são limitados a um único usuário e seus elementos de interação são preferencialmente virtuais, o que prejudica a sua utilização em aplicações que priorizam o treinamento em grupo ou com instrumentos reais, como simuladores de avião (ARCHDEACON; IWAI; SWEET, 2012), passadiço de embarcações (RODRIGUES, 2010), guindastes (GARCÍA-FERNÁNDEZ et al., 2011), entre outros.

As instalações mencionadas exemplificam o alto número de computadores que se faz necessário utilizar na montagem de um sistema imersivo de alta resolução. Para usar estes recursos computacionais de forma eficiente, é necessário compreender quais são as técnicas existentes para dividir o trabalho de processamento entre os diversos computadores, quais os mecanismos de sincronismo que precisam ser aplicados e quais as vantagens e desvantagens de cada abordagem.

2.2 Conceitos fundamentais

Um ambiente virtual é composto por um conjunto de objetos, onde cada um destes objetos possui uma representação geométrica que define o formato de sua superfície. Estas representações geométricas são, normalmente, realizadas através de um conjunto de primitivas básicas como, por exemplo, triângulos. Sendo assim, um determinado objeto virtual pode ser representado por um amplo conjunto de primitivas que definem o formato de sua superfície. Estas primitivas, juntamente com as características do material e suas respectivas texturas, irão definir a aparência deste objeto no ambiente virtual.

O processo de renderização praticado pelas placas gráficas mais modernas é dividido em

diversos estágios de um *pipeline*. Este *pipeline* de renderização é responsável por processar a descrição geométrica do ambiente virtual e produzir uma (ou mais) imagem(ns) deste ambiente, considerando um determinado ponto de vista do observador. Para produzir imagens de resolução compatível com a infraestrutura do sistema de visualização imersivo, se torna necessário distribuir o processo de renderização nas diversas máquinas do aglomerado gráfico. Para garantir que os resultados obtidos por todos estes computadores estejam consistentes, é fundamental fazer uso de mecanismos para manter o sincronismo.

2.2.1 Renderização paralela

Em 1994, Steven Molnar e outros pesquisadores criaram uma classificação para as técnicas de renderização paralela (MOLNAR et al., 1994). Apesar de antiga, esta classificação é uma das mais aceitas e utilizadas até hoje. Em seu trabalho, Molnar propõe que as técnicas de renderização paralela sejam classificadas em três categorias: *Sort-First*, *Sort-Middle* e *Sort-Last*. Estas categorias foram estabelecidas com base em duas operações importantes executadas no *pipeline* de renderização: o processamento da geometria e a rasterização.

O processamento da geometria envolve, entre outras coisas, o processamento das propriedades dos vértices que compõem a geometria e a transformação de suas posições para o sistema de coordenadas da tela (*screen-space*). Já a rasterização corresponde ao processo que irá obter a coordenada destes vértices na tela e identificar quais fragmentos precisam ser pintados e a cor que deve ser utilizada para pintá-los. Embora o *pipeline* de renderização de uma placa gráfica moderna seja composto por diversos estágios de processamento, estas duas operações ainda podem ser identificadas no *pipeline* atual.

2.2.1.1 *Sort-First*

Na categoria *Sort-First*, a distribuição dos objetos para renderização paralela ocorre antes das duas principais etapas de renderização: o processamento das geometrias e a rasterização. Nesta categoria, após a distribuição dos objetos, não há mais comunicação entre os diferentes processadores gráficos, sendo que cada um deles executará todas as etapas de renderização para seus respectivos objetos.

Para viabilizar a implementação do *Sort-First*, a tela é dividida em regiões diferentes, onde cada região será de responsabilidade de um processador gráfico. Desta forma, um processador é capaz de executar todas as operações de renderização em uma região da tela, sem sofrer interferência de outros processadores. Nesta situação, a divisão dos objetos para a renderização é feita conforme a posição em que os mesmos serão exibidos na tela.

Figura 4: *Sort-First*.

Fonte: Molnar et al. (1994)

Para descobrir em qual posição estes objetos serão exibidos, é necessário realizar uma operação denominada por Molnar como pré-transformação.

A pré-transformação consiste em executar a primeira etapa de renderização (processamento das geometrias) apenas para 8 vértices específicos. Estes vértices pertencem a um cubo hipotético que envolve completamente o objeto (*bounding box*). Esta transformação simplificada permite identificar em qual(is) região(ões) da tela o mesmo será posicionado, ou seja, em qual(is) processador(es) gráfico(s) o objeto deverá ser renderizado.

Uma vantagem desta categoria é que há pouca ou nenhuma comunicação entre os processadores. Uma desvantagem desta categoria é que a complexidade da cena não será distribuída igualmente entre os diferentes processadores gráficos. Como cada processador é integralmente responsável por uma região da tela, dependendo do ponto de vista do usuário virtual, uma região da tela pode acomodar objetos mais complexos do que outras, o que sobrecarregaria um determinado processador.

2.2.1.2 *Sort-Middle*

A categoria *Sort-Middle* considera os casos onde os objetos são redistribuídos entre as duas etapas de renderização. Primeiramente, os objetos são distribuídos arbitrariamente entre os processadores gráficos. Esta primeira distribuição pode considerar a complexidade destes objetos, procurando dividir igualmente a complexidade da cena entre os diferentes processadores.

Após esta distribuição, os processadores gráficos executam apenas a primeira operação

Figura 5: *Sort-Middle*.

Fonte: Molnar et al. (1994)

de renderização: o processamento das geometrias. Em seguida, os objetos (já em coordenadas da tela) são redistribuídos, sendo transferidos para outros processadores para a execução da segunda operação de renderização: a rasterização. Como a rasterização considera os pixels da tela, nesta redistribuição, cada processador gráfico é responsável por uma região da tela, e este receberá somente os objetos que serão exibidos em sua região, de forma semelhante ao *Sort-First*.

Uma vantagem desta categoria é a melhor distribuição da complexidade da cena, pois a operação que envolve a transformação das geometrias pode ser executada em qualquer processador gráfico. No entanto, existe a desvantagem da comunicação entre os processadores, pois para executar a segunda etapa de renderização, os objetos precisam ser transferidos para outros processadores. Neste caso, as informações que precisam ser transferidas correspondem às geometrias dos objetos, porém já transformadas para as coordenadas da tela.

2.2.1.3 *Sort-Last*

A categoria *Sort-Last* considera os casos em que os objetos serão redistribuídos após as duas etapas de renderização. Primeiramente, os objetos são distribuídos arbitrariamente entre os processadores gráficos. Nesta distribuição, pode-se considerar a complexidade dos objetos, procurando dividir igualmente a complexidade da cena. No entanto, ao contrário do *Sort-Middle*, os processadores executam as duas operações de renderização.

Neste caso, após a renderização, cada processador vai obter como resultado um con-

Figura 6: *Sort-Last*.

Fonte: Molnar et al. (1994)

junto de fragmentos correspondente à rasterização dos seus respectivos objetos, oriundos da primeira distribuição. No entanto, para exibir o resultado final, os fragmentos obtidos por cada processador precisam ser transferidos para os elementos que irão compor a imagem final. Molnar identificou estes elementos como sendo os Processadores de Composição.

Os Processadores de Composição serão responsáveis por obter os fragmentos obtidos por cada processador gráfico, e juntá-los na composição da imagem final a ser apresentada na tela. Uma vantagem desta abordagem é a ótima distribuição da complexidade da cena, pois ambas as operações de renderização podem ser igualmente distribuídas para todos os processadores gráficos. No entanto, existe a desvantagem da comunicação com os processadores de composição. Neste caso, as informações transferidas correspondem aos pixels gerados após a renderização dos objetos. Dependendo da resolução da imagem final, o volume de informação a ser transferido pode ser muito grande, deteriorando o desempenho.

2.2.2 Sincronismo

Outro aspecto que precisa ser considerado no desenvolvimento de aplicações distribuídas de visualização é o sincronismo (RAFFIN et al., 2006; SOARES et al., 2005). Ao utilizar diversos computadores para apresentar um determinado conteúdo, é fundamental garantir que a imagem apresentada por todo o aglomerado esteja sincronizada.

O sincronismo destas imagens deve ser garantido em diferentes níveis, conhecidos

Figura 7: Exemplo de falta de sincronismo.

Fonte: Soares et al. (2005)

como: *Frame-lock*, *Swap-lock* e *Data-lock*. O *Frame-lock* é o sincronismo da varredura dos pixels em todos os sinais de vídeo (*pixel scanning*) e garante que a taxa de atualização dos projetores ou monitores estarão sincronizadas. Esta sincronização é fundamental para aplicações que utilizam a tecnologia de estereoscopia ativa (RAFFIN et al., 2006) - em outras ocasiões, a ausência deste sincronismo pode causar a visualização de pequenos artefatos entre diferentes computadores (*screen-tearing*).

O *Swap-lock* é o sincronismo da atualização dos quadros renderizados pela aplicação. Todos os computadores do aglomerado devem atualizar os quadros renderizados ao mesmo tempo. Desta forma, caso alguns computadores sejam mais rápidos que os outros, esses deverão aguardar todo o aglomerado para apresentar os resultados da renderização simultaneamente. Este sincronismo deve ser usado em qualquer circunstância - algumas placas gráficas oferecem recursos para garantir este sincronismo via *hardware*, no entanto, como este recurso não é padronizado, muitas aplicações utilizam um sincronismo via *software*.

O *Data-lock* deve garantir que as informações da cena virtual estejam sincronizadas entre todos os computadores, antes de se iniciar o processo de renderização. Este sincronismo impõe que todas as alterações realizadas na cena virtual sejam transferidas para todas as máquinas, antes de se iniciar a renderização.

A figura 7 ilustra uma situação onde dois computadores (parte superior) estão sincronizados, porém outros dois computadores (parte inferior) não estão sincronizados. A ausência do *Data-lock* poderia provocar o resultado ilustrado. Neste exemplo, embora a imagem tenha sido apresentada simultaneamente (*Frame-lock* e *Swap-lock*), as informações usadas para renderizar a cena não estavam sincronizadas (*Data-lock*), colocando o personagem em posições diferentes em cada uma das máquinas.

2.3 Síntese

Este capítulo apresentou uma breve perspectiva histórica sobre alguns sistemas imersivos em Realidade Virtual. Esta perspectiva ilustra que, embora a capacidade de processamento das placas gráficas tenha evoluído de forma significativa, juntamente com a resolução e a qualidade dos painéis e projetores, os requisitos destes sistemas imersivos também foram elevados, exigindo um número cada vez maior de equipamentos e, consequentemente, aumentando a complexidade da infraestrutura.

Além disto, o capítulo também apresentou conceitos importantes sobre renderização paralela e sincronismo, elementos usados de forma recorrente em qualquer infraestrutura distribuída de visualização.

3 REVISÃO DA LITERATURA

Este capítulo apresenta um estudo sobre os arcabouços de *software* elaborados para facilitar o desenvolvimento de aplicações em sistemas de Realidade Virtual com aglomerados de computadores. Estes arcabouços permitem que aplicações sejam executadas em um conjunto de máquinas, paralelizando o processo de renderização e apresentando o resultado de forma sincronizada em um conjunto de projetores ou monitores.

Este estudo foi realizado utilizando somente artigos publicados a partir do ano 2000. Esta restrição foi considerada, pois somente a partir deste ano é que foi consolidado o uso de aglomerados de computadores para a construção de sistemas imersivos. Antes deste período, era muito comum se fazer uso de supercomputadores gráficos com memória compartilhada ou outros periféricos de *hardware* desenvolvidos especificamente para distribuir a renderização.

Neste estudo, também foram desconsiderados todos os trabalhos que não foram elaborados para tempo real, ou seja, ferramentas que distribuem o processo de renderização visando à criação de vídeos. Além disto, as ferramentas específicas para renderização volumétrica também não foram incluídas. Apesar da importância destas aplicações na visualização de dados científicos, a renderização de volumes faz uso de técnicas específicas desta área, o que as coloca em uma categoria distinta de aplicações, que não será considerada nesta revisão.

Desta forma, os arcabouços de *software* considerados neste estudo são: Chromium (HUMPHREYS et al., 2002), Net Juggler (ALLARD et al., 2002; BIERBAUM et al., 2005), Syzygy (SCHAEFFER; GOUDESEUNE, 2003), OpenSG (ROTH; VOSS; REINERS, 2004), Multipipe SDK (BHANIRAMKA; ROBERT; EILEMANN, 2005), Broadcast GL (ILMONEN; REUNANEN; KONTIO, 2005), FlowVR Render (ALLARD; RAFFIN, 2005; ARCILA et al., 2006), Equalizer (EILEMANN; MAKHINYA; PAJAROLA, 2009), Cluster GL (NEAL; HUNKIN; MCGREGOR, 2011), CGLX (DOERR; KUESTER, 2011) e TechViz (VERHILLE et al., 2014). Em função do número de arcabouços revisados, este documento não apresentará uma avaliação detalhada de cada ferramenta individualmente. Entretanto, com o objetivo de esclarecer

Figura 8: Camadas de *software*.

Fonte: Autor

os seus meios de funcionamento e identificar suas semelhanças, este trabalho estabeleceu uma classificação para os arcabouços estudados. Esta classificação apresenta categorias conforme a camada de *software* em que atua uma determinada ferramenta.

Desta forma, estes arcabouços serão apresentados e comparados conforme dois aspectos: em qual camada de *software* é feita a distribuição e o sincronismo da cena virtual e, quais são as categorias suportadas de renderização paralela, conforme a classificação de Molnar apresentada no capítulo 2. Estes aspectos irão auxiliar na compreensão sobre o funcionamento destas ferramentas e também sobre as vantagens e desvantagens de cada abordagem.

3.1 Camadas de *software*

Os *softwares* de Realidade Virtual podem ser divididos em diversas camadas, cada uma destas camadas realizando um conjunto específico de operações. Quando consideramos apenas o aspecto gráfico destas ferramentas, podemos dividir qualquer aplicação moderna de RV em no mínimo três camadas: *driver* gráfico (com OpenGL ou Direct3D), grafo de cena (motor gráfico) e aplicação, conforme ilustrado na figura 8.

O *driver* gráfico é a camada responsável por enviar as geometrias, as texturas e outras informações necessárias para renderização na placa gráfica do computador. A interface com esta camada de *software* é representada pela API (*Application Programming Interface*) do OpenGL ou Direct3D. Estas API's, padronizadas e bem estabelecidas, permitem que qualquer aplicação possa usar os recursos disponíveis em uma placa gráfica, independente do seu modelo ou fabricante.

A camada do grafo de cena (motor gráfico) contém recursos e efeitos que auxiliam a criação de uma aplicação gráfica. Esta camada de *software* exerce função semelhante ao de um Motor de Jogos - apresentando uma interface mais fácil e amigável para o

desenvolvedor da ferramenta, aumentando a produtividade e consequentemente, reduzindo o tempo de desenvolvimento. Esta camada utiliza a camada inferior para se comunicar com a placa gráfica.

A camada da aplicação contém todo o *software* desenvolvido para um propósito específico. Este *software* possui recursos que visam resolver os problemas de um determinado domínio. Normalmente, o desenvolvimento desta camada faz uso de uma biblioteca de grafo de cena, sendo cada vez mais incomum aplicações que utilizam diretamente os recursos disponíveis no OpenGL ou Direct3D.

A observação de um sistema gráfico de RV em camadas é importante para compreender a forma de atuação dos arcabouços estudados. Alguns destes arcabouços efetuam a distribuição diretamente no *driver* gráfico. Outros são utilizados como uma biblioteca de grafo de cena (motor gráfico). E por último, alguns arcabouços oferecem recursos apenas para auxiliar na distribuição e sincronismo da cena virtual, sendo de responsabilidade da aplicação determinar quais informações precisam ser distribuídas.

Estes aspectos são apresentados em maior detalhe nas próximas seções, juntamente com suas respectivas vantagens e desvantagens.

3.1.1 Camada do *driver* gráfico

Entre os arcabouços estudados, aqueles que atuam na camada do *driver* gráfico são: Chromium (HUMPHREYS et al., 2002), Broadcast GL (ILMONEN; REUNANEN; KONTIO, 2005), Cluster GL (NEAL; HUNKIN; MCGREGOR, 2011) e TechViz (VERHILLE et al., 2014). O mecanismo de operação destes arcabouços é muito semelhante e funciona através da substituição do *driver* gráfico OpenGL original, por outro, capaz de interceptar as chamadas de função desta API. Nas alternativas estudadas, nenhuma ferramenta foi desenvolvida para o Direct3D, da Microsoft. Este fato pode ser justificado, pois o OpenGL funciona em diversas plataformas e sistemas operacionais.

Ao substituir o *driver* original, as ferramentas interceptam as chamadas de função efetuadas pela aplicação, ou pela biblioteca do grafo de cena, e enviam as instruções OpenGL pela rede para os outros computadores do aglomerado. Nesta solução, as instruções OpenGL são distribuídas automaticamente para os diversos computadores do sistema.

Uma das principais vantagens desta abordagem é o fato que a aplicação de RV não precisa ser modificada. Por se tratar de uma substituição do *driver* gráfico, uma ferramenta que originalmente foi desenvolvida para um desktop convencional pode ser executada em

um aglomerado de computadores e exibir suas imagens em um sistema imersivo, somente através da substituição do *driver* gráfico instalado no computador. Obviamente, esta substituição só será efetiva caso a aplicação em questão tenha sido desenvolvida originalmente utilizando OpenGL, de forma direta ou indiretamente.

Atualmente, o OpenGL suporta dois modos de renderização: o modo imediato (*immediate mode*) e o modo retido (*retained mode*). No modo imediato, os vértices das geometrias, juntamente com suas propriedades, não podem ser gravados na memória da placa gráfica - todas estas informações precisavam ser reenviadas para a placa gráfica a cada quadro renderizado (*frame*). Neste modo, a única informação que pode permanecer na memória da placa são as texturas.

Já no modo retido, a memória da placa gráfica pode ser utilizada para salvar os vértices das geometrias, juntamente com suas propriedades e outras informações. Desta forma, a aplicação não precisa enviar as mesmas informações a cada renderização, apenas modificar as informações que precisam ser atualizadas.

A partir da versão 3 do OpenGL (2008), o modo imediato de renderização foi oficialmente declarado como obsoleto na especificação. Com o aumento da memória disponível nas placas gráficas modernas, muitas aplicações passaram a fazer uso do modo retido, obtendo ganhos consideráveis de desempenho, uma vez que grande parte das informações necessárias à renderização não precisam ser transferidas repetidas vezes para a placa gráfica.

Embora este último modo favoreça os arcabouços que operam na camada do *driver*, as informações que antes eram transferidas para a placa gráfica através de uma interface *PCI Express*, agora serão enviadas pela rede para os outros computadores da infraestrutura. Dependendo da aplicação, a quantidade de informação transferida ainda pode ser muito elevada, deteriorando o seu desempenho.

Outro aspecto negativo é o fato que muitas aplicações são incompatíveis com esta abordagem. As aplicações originalmente desenvolvidas para desktop, com suporte apenas a uma única tela, frequentemente efetuam otimizações para aliviar o processamento da placa gráfica - por exemplo, não enviando para renderização as geometrias que estão fora do campo de visão do usuário (*frustum culling*).

Nestes casos, para apresentar o conteúdo em um ambiente imersivo com diversas telas (e.g. uma Caverna), a substituição do *driver* gráfico não é suficiente, uma vez que o *driver* OpenGL não receberá da aplicação nenhum objeto para ser exibido nas telas laterais.

Para contornar estes problemas, é comum desabilitar algumas opções de otimização disponíveis nas aplicações como, por exemplo, o *frustum culling* (ZIELINSKI et al., 2014). Obviamente, ao abrir mão destas otimizações, a aplicação perderá desempenho para priorizar a compatibilidade com a infraestrutura em questão. No entanto, outras alternativas podem ser utilizadas caso o código da aplicação esteja disponível.

3.1.2 Camada do grafo de cena

As soluções que atuam na camada do grafo de cena são: Net Juggler (ALLARD et al., 2002), Syzygy (SCHAEFFER; GOUDESEUNE, 2003), OpenSG (ROTH; VOSS; REINERS, 2004) e FlowVR (ALLARD; RAFFIN, 2005). Nesta camada, as soluções devem ser usadas de forma semelhante a um motor de jogos. Estas ferramentas oferecem recursos para organizar os objetos da cena, realizar animações, aplicar efeitos gráficos (*shaders*), entre outros.

Com exceção do Net Juggler, todas as outras alternativas desta categoria funcionam de forma semelhante. Os objetos virtuais carregados pela aplicação são distribuídos automaticamente para todos os computadores do aglomerado. Durante a execução do programa, apenas as alterações realizadas na cena são enviadas a todas as máquinas, mantendo a cena virtual sincronizada e reduzindo a transferência de informações pela rede.

Nesta abordagem, a aplicação de RV executa em um único computador do *cluster* (*master*). As outras máquinas (*slaves*) executam um programa específico, disponibilizado pelo próprio arcabouço. Este programa se comunica com a aplicação de RV e aguarda receber os objetos da cena virtual, além de outras instruções para efetuar a renderização. Durante a execução da aplicação, qualquer alteração realizada na cena virtual será automaticamente enviada aos outros computadores.

O Net Juggler, ao contrário das outras alternativas, possui uma abordagem simplificada para o problema da distribuição. Nesta solução, a mesma aplicação de RV executa simultaneamente em todos os computadores do aglomerado. Para manter as alterações do usuário sincronizadas em todas as máquinas, os eventos dos periféricos de entrada como teclado, mouse e outros, são enviados e reproduzidos em todos os computadores. Nesta abordagem, não há distribuição de trabalho, todas as máquinas executarão as mesmas operações a partir das solicitações enviadas pelos periféricos de entrada.

A vantagem dos arcabouços desta categoria é que uma aplicação pode ser distribuída de forma eficiente em diversas máquinas, fazendo uso apropriado dos recursos da rede. Em contrapartida, a aplicação é obrigada a adotar uma destas ferramentas como biblioteca

gráfica, ao invés de um motor de jogos convencional (para desktop). Este último aspecto pode ser considerado uma desvantagem, pois os motores de jogos tradicionais possuem um mercado mais amplo, e são constantemente atualizados com novos efeitos, recursos e otimizações de desempenho. Além disto, caso se torne necessário adaptar uma aplicação convencional, já desenvolvida para desktop, será necessário reprogramar toda a parte gráfica desta aplicação utilizando um dos arcabouços mencionados.

3.1.3 Camada da aplicação

Os arcabouços que atuam na camada da aplicação são: Syzygy (SCHAEFFER; GOUDESEUNE, 2003), OpenGL Multipipe SDK (BHANIRAMKA; ROBERT; EILEMANN, 2005), Equalizer (EILEMANN; MAKHINYA; PAJAROLA, 2009) e CGLX (DOERR; KUESTER, 2011). O Syzygy também foi incluído nesta categoria, pois a ferramenta possui dois modos de operação.

Nesta camada, os arcabouços disponibilizam recursos para auxiliar diretamente a execução e o sincronismo das aplicações gráficas em um ambiente distribuído. No entanto, estes arcabouços não desempenham funções de um grafo de cena, nem funções semelhantes a um motor de jogos. Neste caso, as aplicações podem escolher um motor de jogos convencional, e utilizar um dos arcabouços desta categoria para auxiliar na distribuição das informações entre os diversos computadores e sincronizar a apresentação das imagens.

De certa forma, as ferramentas presentes nesta categoria exercem função semelhante à biblioteca MPI (PACHECO, 1996) (*Message Passing Interface*). No entanto, as mesmas foram elaboradas com recursos específicos para aplicações gráficas, facilitando a criação de sistemas imersivos de alta resolução.

Na abordagem utilizada por estes arcabouços, a mesma aplicação é executada simultaneamente em todos os computadores do aglomerado. Neste caso, a aplicação deve utilizar os recursos disponibilizados no arcabouço para manter a sua execução sincronizada. A principal vantagem desta abordagem é que a aplicação pode escolher a biblioteca gráfica (motor de jogos) mais apropriada para o domínio do problema que se deseja resolver, e ainda distribuir de forma eficiente o seu processamento nos diversos computadores disponíveis.

Apesar destas vantagens, soluções desenvolvidas nesta abordagem são mais complexas e exigem maior esforço de programação. Isto ocorre pois é necessário integrar de forma apropriada duas ferramentas distintas: a biblioteca gráfica e o arcabouço de distribuição.

Além disto, a aplicação é responsável por determinar como será realizado o sincronismo entre as diversas máquinas, ou seja, quais informações precisarão ser transferidas pela rede.

Ao contrário da camada de grafo de cena, onde o sincronismo e a distribuição são realizados automaticamente - nesta categoria, a aplicação deve ser planejada e desenvolvida para considerar quais atributos da cena são alterados em cada quadro (*frame*) e, se necessário, atualizar estas informações nas outras máquinas do aglomerando, usando os recursos disponíveis em cada arcabouço.

Tabela 1: Camada de *software* dos arcabouços.

Arcabouços	Camada de <i>software</i>		
	<i>Driver</i> gráfico	Grafo de cena	Aplicação
Chromium	x		
Broadcast GL	x		
Cluster GL	x		
TechViz	x		
Net Juggler		x	
Syzygy		x	x
OpenSG		x	
Flow VR Render		x	
OpenGL Multipipe SDK			x
Equalizer			x
CGLX			x

A tabela 1 apresenta um resumo das camadas de *software* em que atuam os arcabouços estudados. Com exceção do Syzygy, que possui dois modos de operação, todas as outras alternativas são bem definidas com relação a esta classificação. Desta forma, as vantagens e desvantagens de cada ferramenta podem ser inferidas a partir das camadas em que atuam.

3.2 Renderização paralela

Os arcabouços estudados também foram avaliados com relação aos tipos de renderização paralela suportados segundo a classificação de Molnar. Conforme apresentado anteriormente, a classificação de Molnar estabelece três categorias para paralelizar o processo de renderização: *Sort-First*, *Sort-Middle* e *Sort-Last*. Entretanto, nos arcabouços estudados, não há nenhuma alternativa que paralelize a renderização conforme a categoria *Sort-Middle*.

Na categoria *Sort-Middle*, os objetos da cena 3D são redistribuídos entre as duas principais etapas do processo de renderização: o processamento da geometria e a rasterização. Ou seja, nesta categoria, o processamento da geometria poderia ocorrer em um computador e a rasterização em outro. A ausência do *Sort-Middle* nos arcabouços estudados pode ser justificada pelo surgimento e aprimoramento das placas gráficas modernas. Na arquitetura das placas gráficas mais recentes, o processo de renderização é executado em um *pipeline* onde grande parte das informações necessárias à renderização é mantida dentro da própria placa gráfica.

Sendo assim, dividir este trabalho em dois computadores exigiria interromper o processamento do *pipeline*, retirar as informações da memória da placa gráfica, transferi-las para outro computador e reiniciar o *pipeline*. Além do excesso de comunicação, mudanças repentinas no *pipeline* de renderização prejudicam o aproveitamento do *cache* na placa gráfica, deteriorando ainda mais o desempenho.

Desta forma, os arcabouços estudados apresentam suporte apenas às categorias *Sort-First* e *Sort-Last*. Em qualquer uma destas abordagens, todo o *pipeline* de renderização é executado em um único processador gráfico. No caso do *Sort-First*, a tela é dividida em regiões, sendo que cada computador é responsável por renderizar todos os objetos que forem exibidos em sua região. No caso do *Sort-Last*, os objetos da cena são divididos uniformemente para todos os computadores, independente de sua região na tela. Neste último caso, ao final da renderização, os computadores precisam transferir as imagens obtidas para compor o resultado final.

Tabela 2: Suporte à renderização paralela nos arcabouços.

Arcabouços	Categorias	
	<i>Sort-First</i>	<i>Sort-Last</i>
Chromium	x	x
Broadcast GL	x	
Cluster GL	x	
TechViz	x	
Net Juggler	x	
Syzygy	x	
OpenSG	x	x
Flow VR Render	x	
OpenGL Multipipe SDK	x	x
Equalizer	x	x
CGLX	x	

A tabela 2 apresenta um resumo dos recursos de renderização paralela disponíveis

nos arcabouços estudados. Nesta tabela, é possível observar que ambas as alternativas de *Sort-First* e *Sort-Last* podem ser implementadas independente da camada de *software* em que estes arcabouços atuam.

Esta última tabela também nos permite afirmar que, enquanto o *Sort-First* é suportado por todos os arcabouços estudados, são poucas as alternativas que apresentam suporte ao *Sort-Last*. Isto pode ser parcialmente justificado pela complexidade de implementação desta última categoria. Enquanto o *Sort-First* pode ser facilmente implementado através de um ajuste na área renderizada em cada computador (*frustum*), o *Sort-Last* envolve uma divisão dos objetos conforme a sua complexidade geométrica, além de uma etapa de transferência e composição das imagens obtidas em cada máquina.

3.3 Análise e comparação

Conforme apresentado, as ferramentas que atuam na camada do *driver* gráfico exigem pouca ou nenhuma alteração no código da aplicação, o que facilita a adoção desta abordagem. No entanto, conforme avaliações de desempenho realizadas por Staadt (STAADT et al., 2003) e Allard (ALLARD; RAFFIN, 2005), alternativas que atuam em outras camadas de *software* podem apresentar desempenho consideravelmente superior.

Neste caso, os arcabouços que atuam na camada do *driver* gráfico se mostram interessantes para situações onde a aplicação de RV já foi desenvolvida e não é possível adaptá-la para um sistema imersivo, seja por motivos econômicos ou pela falta de disponibilidade do código fonte (nos casos de aplicações comerciais). Mesmo nestas situações, as aplicações podem se mostrar incompatíveis com esta abordagem, por exemplo, por otimizações no *culling* da cena virtual (*frustum culling*).

Os outros arcabouços podem ser usados quando a aplicação de RV ainda será desenvolvida ou sua adaptação se mostrar viável. Nestes casos, é importante determinar se a aplicação que será desenvolvida requer o uso de recursos sofisticados disponíveis apenas em um determinado motor de jogos. Quando a aplicação exige que uma biblioteca gráfica específica seja utilizada, será necessário usar os arcabouços que atuam na camada da aplicação. No entanto, é importante considerar que nestas situações, o planejamento e o desenvolvimento da ferramenta em questão se tornarão mais complexos.

Caso os recursos gráficos disponíveis nos arcabouços que atuam na camada do grafo de cena sejam suficientes para a aplicação a ser desenvolvida, a utilização de uma destas alternativas poderá simplificar o seu desenvolvimento.

Outro aspecto avaliado nos arcabouços foram os recursos disponíveis de renderização paralela. Como muitos sistemas imersivos utilizam diversas telas, seja com monitores ou projetores, cada computador do aglomerado será utilizado para prover saída gráfica para parte destas telas. Nesta situação, o sistema já está naturalmente organizado para uma distribuição *Sort-First*, onde cada computador é responsável por renderizar os objetos que serão exibidos em sua tela.

No entanto, existem algumas configurações onde o número de máquinas disponíveis é maior que o número de projetores ou monitores. Nestas situações, apenas parte dos computadores estarão projetando imagens diretamente na tela. Para fazer uso dos recursos computacionais restantes, se torna interessante utilizar um arcabouço com suporte à técnica *Sort-Last*. Com este recurso, embora alguns computadores não estejam exibindo imagens diretamente na tela, estes poderão participar do processo de renderização enviando suas imagens para composição em uma máquina que, por sua vez, esteja conectada a um projetor ou monitor.

3.4 Síntese

Neste capítulo, foram apresentados diversos arcabouços de *software* utilizados no desenvolvimento de aplicações para sistemas imersivos de RV. Estes arcabouços foram classificados conforme a camada de *software* em que atuam e os recursos suportados de renderização paralela. As categorias estabelecidas para as camadas de *software* foram: Camada do *driver* gráfico, camada do grafo de cena e camada da aplicação. Para identificar os recursos de renderização paralela foi utilizada a classificação de Molnar (MOLNAR et al., 1994), da qual apenas duas categorias foram aplicadas: *Sort-First* e *Sort-Last*.

A classificação conforme estas características esclarece o funcionamento destas ferramentas e auxilia na identificação das vantagens e desvantagens de cada abordagem. Conforme analisado, cada grupo de arcabouços pode se mostrar mais adequado dependendo das características da aplicação e das características da infraestrutura do sistema de visualização.

Esta revisão evidencia a complexidade do atual cenário de sistemas de visualização imersivos, e a dificuldade em promover a interoperabilidade de aplicações nos diferentes tipos de infraestrutura.

4 ARQUITETURA DE COMUNICAÇÃO

Neste capítulo será apresentada uma arquitetura de comunicação para sistemas de visualização imersivos. Esta arquitetura identifica os principais elementos conceituais presentes neste tipo de instalação e propõe um mecanismo de comunicação uniforme, visando abranger um amplo conjunto de variações presentes nas atuais infraestruturas de visualização.

A infraestrutura utilizada em sistemas imersivos pode apresentar um número arbitrário de computadores, projetores ou monitores. Além disto, as telas podem estar dispostas em diversas configurações como, por exemplo, planar, cilíndrica, retangular, entre outras. O objetivo desta proposta é facilitar o desenvolvimento de aplicações e uniformizar o acesso a este tipo de infraestrutura, simplificando o cenário atual de sistemas imersivos e promovendo a interoperabilidade das aplicações.

4.1 Requisitos

Para estabelecer a arquitetura proposta, foram considerados diversos requisitos que são recorrentes em aplicações de visualização interativa (tempo real) e também requisitos para facilitar e uniformizar o uso deste tipo de recurso. Os requisitos de alto nível que foram considerados estão enumerados abaixo:

- Simplificar o uso de sistemas de visualização imersivos;
- Promover a interoperabilidade de aplicações entre diferentes tipos de infraestrutura. Dependendo dos recursos disponíveis, a aplicação poderá apresentar variações de desempenho, porém, com resultados consistentes;
- Ser flexível com relação ao tipo de aplicação. Embora o modelo da arquitetura possa favorecer determinados tipos de ferramenta, o objetivo é reduzir estas distorções e estabelecer um mecanismo que possa ser utilizado por um amplo conjunto de aplicações, reduzindo o impacto nas suas características de funcionamento;

- Promover alto desempenho e escalabilidade. A arquitetura deve oferecer e priorizar mecanismos para promover alto desempenho, de forma independente da infraestrutura;
- Promover baixa latência. Com igual importância, o sistema deve oferecer e priorizar mecanismos para promover baixa latência, independentemente da infraestrutura utilizada.

4.2 Visão geral

A proposta da arquitetura está fundamentada na ideia de realizar uma separação conceitual entre os dois principais elementos de um sistema de visualização imersivo: a aplicação e a infraestrutura de visualização. Esta separação nos permite identificar a infraestrutura como sendo um conjunto de recursos cuja função é exibir (renderizar) o ambiente virtual para seus usuários. Desta forma, a única função a ser desempenhada por esta infraestrutura é exibir o conteúdo de forma interativa. Todas as outras responsabilidades podem ser atribuídas à aplicação - entre outras coisas, a aplicação deve construir o ambiente virtual e modificá-lo conforme a ocorrência de eventos em tempo de execução.

O propósito desta separação é isolar a aplicação das características da infraestrutura de visualização, impedindo que a complexidade da infraestrutura influencie no desenvolvimento e na execução da aplicação. Embora em menores proporções, esta separação também evita que a complexidade inerente à aplicação influencie no funcionamento da infraestrutura.

Nesta proposta, a separação entre aplicação e infraestrutura é promovida pela criação de uma interface bem definida de comunicação. Esta interface deve suportar um amplo conjunto de variações da infraestrutura física, visando criar um meio uniforme através do qual a aplicação possa construir e atualizar o ambiente virtual.

Conforme ilustrado na figura 9, o propósito de se estabelecer uma interface bem definida de comunicação é criar um padrão de uso para sistemas imersivos, que seja semelhante ao que existe atualmente para monitores ou projetores. Ou seja, um meio e um protocolo que permitem que qualquer computador possa exibir imagens em um determinado dispositivo.

Muitas ferramentas de RV disponíveis atualmente, principalmente aquelas que atuam nas camadas do grafo de cena e da aplicação, possuem recursos para auxiliar a aplicação a distribuir o seu processamento na infraestrutura computacional de visualização. Nestas

Figura 9: Interface uniforme de comunicação.

Fonte: Autor

situações, as aplicações visam utilizar de forma eficiente os recursos disponíveis nesta infraestrutura para executar tarefas que vão além do processo de renderização, como a realização de cálculos relacionados à física do ambiente virtual. A separação de conceitos proposta nesta arquitetura desencoraja esta abordagem, visando proteger a aplicação da complexidade presente na infraestrutura de visualização.

Desta forma, caso a aplicação necessite de mais recursos computacionais para executar seu processamento, a arquitetura proposta encoraja o uso de uma infraestrutura independente - onde a distribuição de processamento poderá usar recursos elaborados especificamente para HPC (*High Performance Computing*). Mesmo nos casos em que a aplicação possua demanda para cálculos vetoriais - o que supostamente seria beneficiado pela arquitetura da GPU - já existem soluções em *hardware* elaboradas especificamente para estes cenários, que oferecem as vantagens da GPU sem o *pipeline* de renderização como, por exemplo, a Nvidia Tesla¹.

Esta abordagem promove a separação das infraestruturas, o que facilita o desenvolvimento e a interoperabilidade de aplicações. O planejamento e o dimensionamento da infraestrutura do sistema imersivo devem considerar apenas os requisitos de renderização - o mesmo conceito deve ser aplicado ao dimensionar os recursos computacionais necessários aos cálculos realizados pelas aplicações. Além disto, dispondo de uma interface bem definida de comunicação, se torna possível, por exemplo, executar uma aplicação em um *tablet* ou laptop, porém mostrando suas imagens em um sistema imersivo com um número arbitrário de telas.

Nesta arquitetura, a interface de comunicação deve disponibilizar os recursos sufi-

¹<http://www.nvidia.com/object/tesla-servers.html>. Último acesso: 5 de outubro de 2016.

Figura 10: Interface de comunicação OpenGL.

Fonte: Autor

cientes para a exibição do ambiente virtual, porém isolando dos detalhes do sistema de visualização. Esta separação permite que uma mesma ferramenta exiba imagens em um monitor convencional ou em um sistema imersivo arbitrário. Sendo assim, características da infraestrutura de renderização, como aspectos de renderização paralela e sincronismo, não irão afetar o desenvolvimento da ferramenta.

Para definir esta interface de comunicação, é importante notar que a simples transferência de pixels entre a aplicação e o sistema de visualização não é adequada por inúmeros motivos - motivos que contrariam até mesmo os requisitos fundamentais estabelecidos para a arquitetura. Para transferir os pixels, é necessário que a aplicação conheça não apenas a resolução do sistema de visualização, mas também a geometria e a disposição das telas - o que invalida requisitos fundamentais como a interoperabilidade das aplicações. Além disto, sistemas imersivos podem dispor de telas com resolução arbitrariamente grande, o que tornaria inviável a transmissão de um volume tão elevado de informações a cada quadro, principalmente ao considerar que este volume pode dobrar nos casos em que houver estereoscopia.

Conforme ilustrado na figura 10, a arquitetura apresentada neste trabalho propõe que esta interface de comunicação seja implementada na camada do próprio *driver* OpenGL. Nos casos de sistemas de visualização distribuídos, as chamadas de função OpenGL e seus respectivos argumentos seriam transferidos pela rede de forma transparente para a aplicação. A solução proposta nesta arquitetura é semelhante às ferramentas de RV que atuam na camada do *driver* gráfico, discutidas no capítulo 3. Nestas soluções, o *driver* gráfico OpenGL é substituído com o intuito de viabilizar o uso de aplicações convencionais em sistemas imersivos, sem a necessidade de adaptar estas ferramentas.

Sendo assim, a proposta desta arquitetura é utilizar o OpenGL como uma ferramenta para validar a hipótese original do trabalho, de que é possível estabelecer uma interface de comunicação flexível, escalável e de alto desempenho entre uma determinada aplicação e a sua infraestrutura de visualização. Embora o padrão OpenGL não tenha sido originalmente elaborado para atender aos requisitos destes casos de uso, entende-se que o mesmo possuiria muitos recursos em comum com um padrão hipotético, elaborado especificamente para o uso de sistemas imersivos. Além disto, as abstrações e os conceitos propostos pelo OpenGL têm sido utilizados por um vasto conjunto de aplicações nas últimas décadas, o que serve de fundamento sólido para a validação da interface de comunicação proposta.

Os aspectos negativos desta abordagem, que envolve a distribuição da aplicação através do *driver* OpenGL, são o baixo desempenho e a incompatibilidade com algumas aplicações já elaboradas (STAADT et al., 2003; ILMONEN; REUNANEN; KONTIO, 2005; NEAL; HUNKIN; MCGREGOR, 2011). O problema do desempenho é evidenciado em aplicações que, durante a sua execução, transferem muitas informações para a placa gráfica de forma recorrente. Nestes casos, este volume de informações que normalmente é transferido no barramento *PCI Express*, passa a ser enviado pela rede *Ethernet*, o que deteriora de forma considerável o seu desempenho.

Com relação à incompatibilidade, muitas ferramentas gráficas utilizam as características físicas do sistema de visualização para otimizar a exibição da cena, aplicando algoritmos de *frustum culling* aos objetos virtuais. Nestes casos, a aplicação não envia para o *driver* OpenGL os objetos que não estão no campo de visão do usuário. Sendo assim, aplicações que foram elaboradas para uma única tela possuem otimizações que as tornam incompatíveis com outros sistemas imersivos. Nestas situações, é necessário adaptar o código destas aplicações, mesmo após a substituição do *driver* OpenGL.

Apesar dos aspectos negativos já discutidos na literatura, a arquitetura proposta neste trabalho visa revisitar esta abordagem através do uso de técnicas e recursos modernos do padrão OpenGL. O uso de recursos modernos deste padrão visa contornar os problemas frequentemente mencionados na literatura e oferecer os fundamentos para viabilizar uma interface de comunicação uniforme que minimize os impactos no desempenho da aplicação e permita que o conteúdo da mesma seja exibido em um amplo conjunto de infraestruturas de visualização.

Figura 11: Segmentos de um *frustum* convencional, estereoscópico e curvo.

Fonte: Autor

4.3 Considerações sobre sistemas imersivos

O OpenGL foi planejado e desenvolvido considerando um *pipeline* de renderização que compreende apenas um único ponto de vista da cena virtual. Este ponto de vista consiste em um volume de visualização, no qual todos os objetos presentes neste volume serão projetados na superfície da tela. Após o processo de projeção, os objetos são rasterizados para a criação da imagem final. Desta forma, o *pipeline* de renderização do OpenGL produz uma imagem contendo apenas os objetos presentes no ponto de vista especificado.

Caso seja necessário produzir dois pontos de vista distintos de uma mesma cena virtual, as aplicações precisam executar dois passos de renderização - um para cada ponto de vista. Este aspecto se torna especialmente importante ao considerar o uso do OpenGL em sistemas imersivos.

Sistemas imersivos invariavelmente precisam de dois ou mais pontos de vista. Por exemplo, um sistema com uma única máquina e um HMD precisará renderizar, no mínimo, dois pontos de vista estereoscópicos. Atualmente, o único mecanismo disponível no OpenGL para fazer uso adequado do HMD é recorrer à dois passos de renderização. Como ambos os passos serão executados sequencialmente, um após o outro, esta abordagem implica em um impacto significativo no desempenho da aplicação. Em sistemas imersivos arbitrários, com diversos pontos de vista, o número de renderizações irá causar uma queda ainda maior no desempenho.

Conforme ilustrado na figura 11, uma tela com geometria arbitrária pode ser interpretada como um conjunto de segmentos planos. Estes segmentos podem representar, por exemplo, os planos de projeção que compõem os diversos pontos de vista de uma determinada infraestrutura. Sendo assim, para aumentar o desempenho na renderização destes diversos pontos de vista, é comum paralelizar o processo de renderização fazendo uso de um aglomerado de computadores, com diversas placas gráficas. Através desta infraestrutura, cada placa gráfica é responsável pela renderização de um determinado segmento de

plano. Este processo ocorre simultaneamente em todas as placas, promovendo ganho de desempenho considerável quando comparado com a abordagem sequencial de uma única GPU.

No entanto, para viabilizar o uso desta infraestrutura distribuída através do OpenGL, conforme proposto pela arquitetura, se torna necessário adaptar o seu respectivo *driver*. Esta adaptação deve permitir que a infraestrutura imersiva seja usada de forma transparente. Neste caso, a infraestrutura pode ser interpretada como sendo uma única GPU virtual, capaz de renderizar diversos pontos de vista simultaneamente.

Uma possível abordagem para implementar este *driver* seria replicar todos os comandos OpenGL para todas as máquinas do sistema. Esta replicação manterá uma cópia de *buffers* e texturas nas GPU's de todas as máquinas. Esta disponibilidade de recursos é importante para a maioria das aplicações. Em muitos casos, a câmera virtual pode girar em torno do seu próprio eixo, fazendo com que em um curto intervalo de tempo, todas as máquinas tenham que exibir grande parte das geometrias presentes na cena. Caso uma determinada geometria não estivesse disponível para a GPU no momento da exibição, a mesma precisaria ser transferida rapidamente. Sendo assim, embora outras abordagens sem replicação possam ser adotadas, estas devem ser consideradas com cautela, podendo impactar consideravelmente o desempenho da aplicação.

Mesmo utilizando a replicação dos comandos, algumas funções do OpenGL necessitam de um tratamento especial ao serem recebidas por estas máquinas. Por exemplo, uma aplicação gráfica precisa especificar o ponto de vista do usuário (*modelview matrix*) e o seu respectivo volume de visualização (*frustum*). Estas informações devem ser utilizadas normalmente quando a aplicação for executada em um computador convencional. Porém, em um sistema imersivo (com *driver* OpenGL adaptado), tanto o ponto de vista como o volume de visualização precisam ser corrigidos. Cada máquina precisa adaptar as informações enviadas pela aplicação, visando considerar a posição e o volume correspondentes ao seu respectivo segmento de plano na infraestrutura.

Estas adaptações já são realizadas nas bibliotecas que trabalham na camada do *driver* gráfico (NEAL; HUNKIN; MCGREGOR, 2011; ILMONEN; REUNANEN; KONTIO, 2005). No entanto, a arquitetura proposta apresenta, nas seções seguintes, soluções para contornar os problemas mencionados de desempenho e de incompatibilidade com alguns sistemas.

4.4 Redução da taxa de transferência de dados

Conforme mencionado anteriormente, o OpenGL possui dois modos de renderização: o modo imediato (*immediate-mode*) e o modo retido (*retained-mode*). No modo imediato, a geometria de todos os objetos é armazenada na memória RAM do computador e deve ser transferida para a GPU durante a renderização de cada quadro. Neste modo, apenas as texturas podem permanecer na memória da placa gráfica. Recentemente, com o aumento da memória disponível nestes periféricos, o padrão do OpenGL tornou obsoleto o modo imediato, mantendo apenas suporte ao modo retido.

Neste último modo de renderização, *buffers* de propósito genérico podem ser alocados diretamente na placa gráfica. Desta forma, a aplicação pode enviar as texturas e as geometrias para a GPU uma única vez, e renderizá-las diversas vezes fazendo referência às regiões de memória preenchidas anteriormente.

Estas regiões de memória na placa gráfica são reservadas através de um recurso da API do OpenGL conhecido como *Buffer Object*. As propriedades dos vértices de cada geometria são armazenados em VBOs (*Vertex Buffer Objects*), enquanto que os índices destes vértices são armazenados em IBOs (*Index Buffer Objects*). A aplicação pode reservar um número arbitrário destes espaços, cada um deles com tamanho diferente. Durante o processo de renderização, os programas de *shader*, que são executados diretamente na GPU, acessam o conteúdo destes *buffers* através de variáveis de entrada e saída.

No entanto, mesmo no modo retido, onde grande parte das informações da cena já está presente na memória da GPU, a aplicação ainda precisa configurar parâmetros específicos de cada geometria ao solicitar a sua renderização. Entre os possíveis parâmetros utilizados nesta configuração, podemos mencionar as matrizes de transformação do objeto, as matrizes de *skinning* (animação) e as propriedades do seu material. Muitas aplicações utilizam o recurso de variáveis uniformes do shader (*uniform variables*) para configurar estes parâmetros. Entretanto, ao utilizar as variáveis uniformes de maneira convencional, estas informações precisam ser enviadas para a placa gráfica antes da renderização de cada geometria.

Para evitar que a transferência destas informações sobrecarregue a rede, principalmente em cenas mais complexas, a sugestão é novamente utilizar os recursos do OpenGL que favorecem a permanência destes dados na memória da placa gráfica. No caso das variáveis uniformes, a aplicação pode utilizar *buffers* como UBOs (*Uniform Buffer Objects*) ou SSBOs (*Shader Storage Buffer Objects*), mantendo na GPU as informações destas

variáveis.

Sendo assim, para solicitar a renderização de uma determinada geometria, a aplicação só precisa alterar o estado corrente do OpenGL através da ativação dos seus respectivos *buffers*. Isto é necessário pois o OpenGL funciona de forma semelhante a uma máquina de estados. Quando uma operação de renderização é solicitada pela CPU, esta operação é executada conforme o estado corrente do OpenGL.

Entre os parâmetros que compõem o estado do OpenGL, podemos mencionar: os *buffers* de vértices e índices ativos (*buffers* de onde serão extraídos os vértices da geometria), os *buffers* contendo as variáveis uniformes, o conjunto de texturas ativas (de onde serão extraídas a cor e outras informações relativas à aparência do objeto) e os programas de *shader* ativos em cada estágio do *pipeline* (contendo os cálculos de iluminação e outras operações a serem realizadas neste objeto).

4.4.1 Consolidação de estados do OpenGL

Após manter a maioria das informações da cena virtual já na memória da placa gráfica, o perfil de interação da aplicação com a GPU mudará drasticamente, sendo que a aplicação passará a maior parte do tempo alterando estados do OpenGL ao invés de enviando informações para a placa.

No entanto, para alterar o estado corrente, a aplicação precisa invocar ao menos uma função da API do OpenGL. No caso de uma infraestrutura distribuída, esta função, juntamente com os seus argumentos, precisará ser enviada aos outros computadores através da rede. Em cenários com milhares ou até centenas de milhares de objetos, estas alterações de estado podem facilmente sobrecarregar a rede, uma vez que a aplicação precisa repeti-las em todos os quadros renderizados.

Para otimizar a troca de estados no OpenGL é importante compreender alguns elementos destes estados e o seu impacto no *pipeline* de renderização. Para solicitar que um determinado objeto seja renderizado, três atributos do estado do OpenGL são importantes no contexto destas otimizações: os programas de *shader* ativos na placa gráfica, os *buffers* mapeados para estes programas e as texturas mapeadas.

Embora o OpenGL permita que um número arbitrário de *shaders*, texturas e *buffers* sejam criados pela aplicação, apenas um número limitado destes recursos pode ser utilizado em cada solicitação de renderização. No caso do *shader*, por exemplo, apenas um único programa deve estar ativo em cada estágio do *pipeline*. Este programa é respon-

sável por obter o conteúdo dos *buffers* e das texturas e utilizar estas informações para determinar como este objeto será exibido na tela. Os *buffers* e as texturas, que residem na memória da placa gráfica, são mapeados pela aplicação para variáveis de entrada e saída nos programas de *shader*. Como o número destas variáveis de entrada/saída é limitado pela implementação do OpenGL, apenas um número restrito de *buffers* e texturas podem ser utilizados simultaneamente.

O uso convencional do OpenGL pode sugerir que cada objeto tenha diversos *buffers*: um para armazenar as coordenadas dos vértices, outro para armazenar os vetores normais e outro para guardar as coordenadas de textura, por exemplo. Desta forma, cada objeto precisa que todos os seus respectivos *buffers* estejam mapeados para o programa de *shader* no momento da sua renderização. Este conjunto de *buffers* mapeados compõem o estado do OpenGL, e precisa ser alterado para renderizar outra geometria. Para reduzir o número de alterações necessárias, esta arquitetura propõe que diversos objetos sejam alocados em um único *buffer*.

Desta forma, diversos objetos podem ter as propriedades dos seus vértices armazenados no mesmo VBO. O mesmo processo pode ser usado também para os índices dos vértices no IBO. Ao consolidar diversas geometrias nos mesmos *buffers*, um mesmo estado do OpenGL pode ser usado para renderizar diversas geometrias.

Além do conteúdo do VBO e do IBO, a aplicação também pode consolidar os *buffers* das variáveis uniformes (UBO ou SSBO). Neste caso, outros parâmetros necessários para renderizar as geometrias, como as matrizes de transformação e as propriedades do material, seriam todos armazenados no mesmo *buffer*. Entretanto, ao realizar esta consolidação, se faz necessário estabelecer um mecanismo para associar cada geometria com o seu respectivo conjunto de parâmetros dentro do UBO.

O OpenGL dispõe de diversas alternativas para realizar esta associação. Um dos mecanismos mais eficientes faz uso de um atributo de vértice especial, conhecido como atributo de instância (*instanced attribute*²). Este atributo pode ser armazenado dentro do VBO, porém, ao invés de possuir um valor para cada vértice da geometria, o mesmo contém apenas um valor para cada instância da geometria. Sendo assim, a variável de entrada do *shader* associada com este atributo obterá um único valor para toda a geometria. Este valor pode ser usado, por exemplo, como um *offset* dentro do UBO, para obter os seus respectivos parâmetros de renderização.

²https://www.opengl.org/wiki/Vertex_Specification#Instanced_arrays. Último acesso: 5 de outubro de 2016.

Outra sugestão semelhante para reduzir o número de estados é efetuar o agrupamento de texturas. Neste caso, ao invés de colocar diversas imagens em uma única textura do OpenGL, o que exigiria a adaptação de todas as coordenadas de textura, a arquitetura propõe que seja utilizado um recurso do OpenGL conhecido como Array de Texturas (*Texture Array*). Um único *array* de textura do OpenGL pode acomodar um número relativamente grande de imagens. A vantagem do *array* em relação a uma única textura é que todas as imagens do *array* podem ser mapeadas para o programa de *shader* utilizando uma única variável, ou seja, aumentando o número de imagens disponíveis para o programa simultaneamente, e conseqüentemente, permitindo que vários objetos sejam renderizados sem precisar efetuar alterações no estado do OpenGL.

Uma das restrições para se utilizar um *array* de texturas é que todas as imagens de um *array* devem possuir o mesmo formato e as mesmas dimensões, ou seja, duas imagens com dimensões diferentes não podem ser armazenadas no mesmo *array*. Para contornar esta limitação e aumentar a taxa de agrupamento, a arquitetura propõe que as imagens tenham suas dimensões adaptadas para um conjunto predefinido de resoluções, por exemplo, organizadas em potência de 2. Desta forma, a aplicação poderia alocar um número restrito de *arrays*, com as seguintes resoluções: 128 x 128, 256 x 256, 512 x 512 e 1024 x 1024.

Além desta técnica, o ARB (*Architecture Review Board*) do OpenGL aprovou em 2013 uma extensão cuja finalidade é atenuar o custo de alteração destes estados de textura. Ao utilizar esta extensão, cujo nome é *ARB_bindless_texture*³, o programa de *shader* pode acessar qualquer textura presente na placa gráfica através do seu endereço na memória (*handle*). Neste caso, o *shader* não está mais limitado a um conjunto fixo de texturas previamente mapeadas. Embora esta extensão ainda não tenha sido incorporada à nenhuma versão do OpenGL, a mesma já está disponível nas GPU's mais modernas de grande parte dos fabricantes.

A figura 12 ilustra a diferença entre o uso convencional dos estados do OpenGL (esquerda) e o sugerido nesta arquitetura (direita). No uso convencional, quando a renderização de um objeto *A* é solicitada, apenas as informações deste objeto são mapeadas para o programa de *shader*. Ao utilizar as técnicas apresentadas, entretanto, um único estado pode acomodar um elevado número de objetos, com *buffers* maiores e com diversas texturas.

Embora nesta última abordagem, o número de texturas e *buffers* mapeados ao *shader*

³https://www.opengl.org/registry/specs/ARB/bindless_texture.txt. Último acesso: 5 de outubro de 2016.

Figura 12: Aumentando o aproveitamento dos estados no OpenGL.

Fonte: Autor

seja maior, não há nenhuma relação entre o número de elementos que podem ser acessados pelo *shader* e o seu consumo de memória ou deterioração de desempenho. O que ocorre é um maior aproveitamento das variáveis presentes no estado do OpenGL. Naturalmente, como este *shader* será utilizado para renderizar um número maior de objetos, o mesmo deve ter todas as técnicas necessárias de iluminação consolidadas em um único programa (*uber-shader*⁴).

Após a consolidação destes estados, a extensão *ARB_base_instance*⁵ do OpenGL pode ajudar uma aplicação a solicitar a renderização de uma determinada geometria, utilizando os recursos de atributos de instância (*instanced attributes*). Esta extensão disponibiliza a função *glDrawElementsInstancedBaseVertex*. Esta função permite que a aplicação solicite a renderização de uma única geometria, cujas informações estão armazenadas em um *buffer* contendo outras geometrias. Esta solicitação, ao usar o recurso de atributos de instância, já estabelece a associação da geometria renderizada com as suas respectivas propriedades, armazenadas em um UBO.

Outra extensão que pode ser utilizada pela aplicação é a *ARB_draw_elements_base_vertex*⁶. Esta extensão disponibiliza a função *glMultiDrawElementsBaseVertex*, que permite que diversas geometrias sejam renderizadas com uma única solicitação. Apesar de ren-

⁴Termo normalmente utilizado para programas de *shader* que acomodam diversas técnicas de renderização.

⁵https://www.opengl.org/registry/specs/ARB/base_instance.txt. Último acesso: 5 de outubro de 2016.

⁶https://www.opengl.org/registry/specs/ARB/draw_elements_base_vertex.txt. Último acesso: 5 de outubro de 2016.

derizar diversas geometrias, esta última função não permite que sejam utilizados os atributos de instância (*instanced attributes*), ou seja, este mecanismo não permite que seja estabelecido a associação entre uma determinada geometria e as suas respectivas propriedades. Desta forma, todas as geometrias de uma mesma solicitação serão renderizadas com o mesmo conjunto de propriedades.

4.4.2 Renderização indireta

O mecanismo de renderização indireta permite que diversas geometrias sejam renderizadas através de uma única solicitação, porém fazendo uso do recurso de atributos de instância. Anteriormente, a aplicação precisava escolher entre exibir diversas geometrias sem o recurso de atributos de instância, ou exibir uma única geometria utilizando este tipo de atributo.

A renderização indireta permite que ambos os recursos sejam utilizados simultaneamente, onde uma única solicitação pode efetuar a renderização de diversas geometrias, sendo que cada geometria poderá utilizar um conjunto diferente de propriedades (como matrizes e materiais), sem que seja necessário alterar qualquer estado do OpenGL, simplesmente fazendo uso do recurso de atributos de instância em cada uma destas geometrias.

Este recurso está disponível a partir da versão 4.3 do padrão, ou através da extensão *ARB_multi_draw_indirect*⁷. Uma das funções disponibilizadas por esta extensão é a *glMultiDrawElementsIndirect*. Ao utilizar esta função, ao invés de passar um número muito elevado de argumentos, o padrão OpenGL optou por introduzir um novo tipo de *buffer*. Este *buffer* deve ser criado e preenchido pela aplicação, e deve conter os parâmetros necessários para a renderização de um determinado conjunto de geometrias. Sendo assim, esta função é utilizada apenas para solicitar à GPU o processamento dos comandos de renderização presentes neste *buffer*.

Nesta abordagem, a sobrecarga de comunicação entre a aplicação e a GPU é reduzida drasticamente, uma vez que este *buffer* com comandos de renderização pode ser preenchido uma única vez no início do programa e utilizado repetidas vezes a partir de então, o que beneficiaria o desempenho destas aplicações, principalmente em ambientes distribuídos.

⁷https://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt. Último acesso: 5 de outubro de 2016.

4.4.3 Ordenação dos estados

Apesar das técnicas para reduzir as mudanças de estado do OpenGL, é inevitável que algumas alterações sejam mandatórias - nem todos os objetos da cena serão renderizados com uma única solicitação da CPU. Sendo assim, é importante estabelecer uma metodologia para realizar estas mudanças de forma eficiente.

Cada mudança de estado necessita que uma função ou rotina OpenGL seja invocada contendo os novos atributos do estado desejado. No contexto desta arquitetura, repetidas chamadas para alteração de estado irão comprometer o desempenho da aplicação por dois motivos: Estes comandos podem trafegar pela rede, dependendo das características da infraestrutura de visualização, e estes comandos possuem um custo associado à sua execução por parte da GPU. Ou seja, qualquer alteração no estado do OpenGL interfere no *pipeline* de renderização da GPU, afetando de forma negativa o seu desempenho.

Isto ocorre porque alterações no estado corrente do OpenGL exigem que o *driver* execute um processo de validação destas modificações para verificar se o estado atual permanece consistente. No entanto, grande parte destas validações só será executada quando o processo de renderização for solicitado, interferindo no seu desempenho. Nestes casos, a validação não é solicitada anteriormente pois o *driver* OpenGL não possui meios para distinguir se a aplicação continuará realizando alterações nas variáveis de estado ou não. Sendo assim, realizar uma validação antes do processo de renderização poderia implicar em múltiplas validações, sem necessidade. O impacto destas alterações no desempenho da GPU irá depender do tipo de atributo que está sendo alterado e da quantidade destas alterações.

Desta forma, supondo que uma cena necessite de um amplo conjunto de estados distintos para ser renderizada, cada um destes estados pode afetar o desempenho da GPU de forma diferente, dependendo de quais mudanças este determinado estado irá realizar na GPU. Determinar a sequência de estados com custo mínimo, ou menor perda de desempenho, se caracteriza como um problema N-P completo, ou seja, todas as combinações precisariam ser testadas.

Estes fatores de desempenho estão diretamente relacionados com as características do *hardware* da placa gráfica e também com a implementação do seu respectivo *driver* OpenGL. No entanto, existem alguns atributos que normalmente possuem maior influência sobre o desempenho da GPU. Apesar deste fato já ser amplamente discutido na comunidade de jogos, apresentações técnicas realizadas recentemente apresentam estatísticas que são comuns e uniformes para diversos fabricantes de placas gráficas (EVERITT et al.,

Figura 13: Custo aproximado para mudança de estados - fora de escala.

Fonte: Everitt e McDonald (2014)

2014; EVERITT; MCDONALD, 2014).

A figura 13 apresenta uma estimativa grosseira dos custos para a troca de atributos nos estados do OpenGL em uma placa gráfica moderna. Os números à direita do gráfico representam quantas alterações por segundo é possível realizar nestes atributos. O POR (*Pipeline* das Operações de Rasterização) compreende todas as propriedades que influenciam a rasterização (*blend*, *depth test*, *stencil test*, entre outras).

O atributo de Layout das geometrias envolve alterações no layout dos dados armazenados no *buffer* de geometria (VBO). Toda vez que um *buffer* de geometria com formato diferente for utilizado, este atributo precisa ser atualizado. O *Buffer* de variáveis contém os valores das variáveis dos programas de *shader* (UBO). O atributo de Variáveis individuais corresponde à atualização de uma única variável no programa de *shader*, sem a utilização de *buffers*.

Este gráfico sugere um critério geral para a ordenação das mudanças no estado do OpenGL. Esta ordenação deve evitar alterações frequentes em atributos de custo mais elevado, priorizando alterações nos atributos de custo mais baixo. Por exemplo, dado dois *buffers* de geometria *A* e *B* e as texturas 1 e 2. Caso tenhamos uma cena que contenha todas as combinações *A_1*, *A_2*, *B_1* e *B_2* - uma das maneiras de se renderizar esta cena de forma otimizada, seria agrupar por texturas ao invés de geometria, por exemplo, na seguinte ordem: $A_1 \rightarrow B_1 \rightarrow A_2 \rightarrow B_2$.

Apesar desta ordenação não promover, necessariamente, uma redução na quantidade de dados transferidos entre a aplicação e a GPU, esta proposta considera o fato de que as técnicas anteriores já promoveram esta redução. Sendo assim, a ordenação dos estados

deve priorizar a ordem sugerida na figura 13, visando manter o desempenho da aplicação ao executar em uma infraestrutura local. Esta sugestão, portanto, deve causar pouco impacto em uma aplicação distribuída, onde o desempenho é fortemente influenciado pela rede.

4.5 Algoritmo de *frustum culling*

Atualmente, o algoritmo de *frustum culling* é uma otimização importante realizada por quase todas as aplicações gráficas de tempo real. O objetivo desta otimização é acelerar o processo de renderização, enviando para a placa gráfica apenas os modelos geométricos que se encontram dentro do volume de visualização do usuário.

Na proposta apresentada por esta arquitetura, onde qualquer aplicação deve ser capaz de exibir imagens em diversos sistemas de visualização através da própria API OpenGL, esta otimização é uma das práticas mais comuns que tornam as aplicações incompatíveis com alguns sistemas. Isto ocorre porque para efetuar o *frustum culling* é necessário conhecer a geometria do volume de visualização, ou seja, a aplicação precisa conhecer detalhes do sistema, como a posição e a orientação das telas.

Para atender a todos os requisitos estabelecidos para esta arquitetura, a aplicação não deve conhecer detalhes sobre o sistema de visualização. No entanto, o *frustum culling* é uma otimização importante, e não pode ser descartada. Para contornar este problema, a arquitetura propõe que o *frustum culling* seja executado na própria GPU, utilizando o *Compute Shader*, que é um programa de *shader* para computação paralela disponível no padrão OpenGL.

Para utilizar o recurso de *Compute Shader*, a aplicação deve submeter um determinado programa para a placa gráfica. A vantagem desta abordagem é que o programa que realizará o cálculo do *frustum culling* será transferido para a GPU através do *driver* OpenGL. Desta forma, dependendo do *driver*, este poderá adaptar o programa de *culling* para que o algoritmo seja efetuado conforme o sistema de visualização que está sendo utilizado. Sendo assim, a aplicação seria responsável por transferir o programa de *shader* para a GPU e solicitar a sua execução, sendo que o *driver* OpenGL teria acesso a este programa e poderia adaptá-lo para a infraestrutura corrente.

Uma maneira prática para implementar esta proposta seria submeter um programa de *culling* cujo algoritmo esteja parametrizado em função do *frustum*, através de um conjunto de variáveis. Neste caso, ao submeter o programa, a aplicação deve enviar

Figura 14: Sequencia de *culling* em sistema imersivo.

Fonte: Autor

também os valores associados a estas variáveis. No entanto, um *driver* OpenGL para sistemas imersivos poderia ignorar os valores submetidos pela aplicação e substituí-los com as dimensões adequadas da infraestrutura em uso. Nesta abordagem, a aplicação funcionaria tanto com um *driver* convencional, que aceitaria os valores enviados, como também com um *driver* OpenGL adaptado para um sistema imersivo.

A vantagem de se utilizar um programa de *Compute Shader* ao invés de um recurso de computação paralela, como o OpenCL ou CUDA, é que o primeiro permite que o programa tenha acesso completo ao contexto OpenGL. Ou seja, o programa pode ler e escrever os resultados em um *buffer* na própria placa gráfica, e este *buffer* poderá ser usado no *pipeline* de renderização. Sendo assim, o algoritmo de *frustum culling* poderia criar uma lista contendo todos os objetos visíveis em um *buffer* na memória da própria placa. Isto aumentaria o desempenho e facilitaria o desenvolvimento de aplicações interoperáveis.

A figura 14 demonstra a sequencia de *culling* em um sistema imersivo com diversos computadores distribuídos. Nesta sequencia, a aplicação primeiro submete o programa de *culling*, que é repassado para todas as máquinas. Em seguida, a aplicação envia os parâmetros do *frustum*, que neste caso são ignorados por se tratar de um sistema imersivo - cada computador do aglomerado gráfico já contém as informações do seu próprio *frustum*. Logo após, a aplicação solicita a execução do programa. Durante a sua execução, o programa consulta a matriz de transformação (MT) e o volume delimitador (*bounding box*, BB) de cada objeto (armazenados em *buffers* na placa gráfica). O resultado consiste em uma lista dos objetos visíveis (OV), que será colocada em outro *buffer*, também localizado

Figura 15: *Culling* hierárquico vs paralelo.

Fonte: Autor

na placa gráfica de cada máquina.

Conforme ilustrado na figura 15, a cena virtual é normalmente organizada de forma hierárquica para facilitar a animação e o posicionamento relativo dos objetos. Nesta organização, cada nó na hierarquia da cena possui um volume delimitador (*bounding volume*) que compreende o espaço ocupado pelo próprio nó, mais o espaço ocupado por todos os seus filhos. Tradicionalmente, o algoritmo de *culling* percorre esta organização de cima para baixo, utilizando a hierarquia para acelerar a verificação dos objetos visíveis. Ou seja, caso a verificação do volume delimitador do nó *T2*, por exemplo, indicasse que o mesmo está fora da região visível do usuário, o algoritmo não precisaria percorrer os nós filhos de *T2* (*A* e *T3*), pois o seu volume delimitador já contém estes elementos.

Este algoritmo tradicional, no entanto, não é adequado para ser executado na GPU. O recurso de *Compute Shader* disponível no OpenGL exige que ao solicitar a execução de um determinado programa, todas as tarefas deste programa sejam especificadas de antemão, no momento da sua solicitação. Isto para que estas tarefas sejam distribuídas pela GPU de maneira uniforme para todos os seus núcleos de processamento. Não é possível, por exemplo, criar uma nova tarefa durante a execução do *Compute Shader*.

Este fato inviabiliza o uso do algoritmo tradicional, exclusivamente hierárquico, pois ao utilizá-lo, o número de objetos a serem verificados não é conhecido previamente. Desta forma, o algoritmo proposto por esta arquitetura consiste em percorrer os objetos da cena para consolidar suas novas matrizes de transformação e seus respectivos volumes delimitadores, e então atualizá-los nos *buffers* da placa gráfica. Em seguida, o programa de *culling* deve ser submetido para a GPU para verificar quais as geometrias da cena

que permanecerão visíveis. Nesta abordagem, todos os objetos da cena serão verificados, porém de forma contundentemente paralela.

Caso o número de objetos na cena seja muito elevado, o algoritmo de *culling* pode agrupar estes objetos por proximidade, reduzindo o número de verificações a serem efetuadas. Outro aspecto importante é que um número elevado de objetos também pode exceder a quantidade de memória disponível nas placas gráficas do sistema. Embora este último problema não seja específico da arquitetura apresentada, a aplicação pode utilizar técnicas de paginação para remover dos *buffers* os objetos que estão distantes do usuário, mantendo na memória gráfica apenas os objetos mais próximos.

Desta forma, embora a arquitetura recomende que a aplicação faça o *culling* na GPU, outras otimizações ainda podem permanecer na CPU. Para cenas muito grandes, por exemplo, a aplicação pode fazer uso de modelos com diferentes níveis de detalhe (*levels of detail*) e usar técnicas de paginação para manter na memória da placa apenas os elementos do cenário que estiverem próximos do usuário.

4.6 Síntese

Este capítulo apresentou uma arquitetura de comunicação para sistemas de visualização imersivos, que utiliza o OpenGL como mecanismo fundamental de interface. O capítulo estabeleceu os principais requisitos a serem atingidos pela arquitetura, como: uniformidade e compatibilidade entre diferentes infraestruturas, alto desempenho e baixa latência. Ao invés de promover uma determinada biblioteca de *software* ou ferramenta específica, a arquitetura estabelece um conjunto de técnicas que podem ser utilizadas por qualquer aplicação moderna elaborada em OpenGL.

Entre as técnicas apresentadas neste capítulo, podemos mencionar algumas iniciativas para reduzir a sobrecarga de comunicação entre a aplicação e a GPU, a consolidação dos estados do OpenGL, através do agrupamento de *buffers* e texturas, e o uso do algoritmo de *frustum culling* em sistemas imersivos. Além de promover o desempenho, as técnicas apresentadas visam tornar a aplicação independente das características da infraestrutura de visualização, reduzindo a sua complexidade e a tornando interoperável entre diferentes sistemas.

Na abordagem proposta, uma mesma aplicação poderá ser utilizada para exibir imagens em um sistema simples ou imersivo, através da substituição do seu *driver* OpenGL. Este novo *driver* é adaptado para capturar e processar as chamadas de função deste

padrão e enviá-las na rede para os outros computadores do aglomerado gráfico, alterando algumas solicitações e os seus respectivos argumentos, quando necessário.

5 CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO

Este capítulo apresenta as considerações feitas durante a implementação de uma infraestrutura de *software* para validar os conceitos e as técnicas apresentadas no capítulo anterior. Esta infraestrutura de *software* foi utilizada como uma plataforma de testes (*test bed*), onde diferentes técnicas de renderização foram testadas com diferentes infraestruturas imersivas de visualização.

O propósito destes testes é verificar qual a contribuição das técnicas sugeridas, no desempenho de aplicações distribuídas. Para realizar estas verificações, a implementação considerada neste capítulo realizou diversas alterações em ferramentas gratuitas, disponíveis em código aberto, como o *driver* ClusterGL¹ e também a aplicação *gl_cadscene_rendertechniques*², que consiste em uma aplicação exemplo da Nvidia sobre como utilizar alguns recursos modernos disponíveis nas versões mais recentes do OpenGL.

5.1 Visão geral

A plataforma de teste desenvolvida visa verificar as vantagens e desvantagens das soluções propostas, em diferentes tipos de infraestruturas imersivas, entre elas: uma Caverna e um *Powerwall*. O objetivo é verificar infraestruturas com telas de geometria diferente e com um número variável de computadores.

Para acomodar estas necessidades, a implementação foi efetuada conforme ilustrado na figura 16, onde uma determinada aplicação OpenGL é executada em uma máquina distinta, contendo apenas uma conexão *Ethernet* com o restante da infraestrutura. Neste arranjo, a comunicação com as máquinas do aglomerado gráfico é realizada através da replicação dos comandos OpenGL utilizando o protocolo UDP *multicast*. Este protocolo

¹<https://github.com/bieh/clustergl>. Último acesso: 5 de outubro de 2016.

²https://github.com/nvpro-samples/gl_cadscene_rendertechniques. Último acesso: 5 de outubro de 2016.

Figura 16: Diagrama de comunicação no aglomerado gráfico.

Fonte: Autor

reduz a utilização da rede, pois os pacotes são enviados para todos os computadores simultaneamente, semelhante a um *broadcast*, porém com um conjunto restrito de máquinas.

Na implementação realizada, o número de computadores utilizados na infraestrutura pode variar, sendo que as particularidades de cada computador são obtidas através de um arquivo de configuração. Este arquivo contém informações que definem, entre outras coisas, o formato do *frustum* e o volume de visualização de cada máquina.

Nesta última figura, o *driver* OpenGL já foi substituído para considerar a infraestrutura imersiva. Os comandos referentes às chamadas OpenGL são transferidos pela rede para os programas de renderização executando nos computadores do aglomerado, responsáveis pela exibição dos resultados.

Estes programas, conhecidos como *daemons*, rodam permanentemente nos computadores e são responsáveis por receber as chamadas OpenGL efetuadas pela aplicação. Este *daemon* cria sua própria janela de exibição e realiza as alterações necessárias nas chamadas recebidas de OpenGL, visando corrigi-las conforme as informações disponíveis no seu respectivo arquivo de configuração.

5.2 *Driver ClusterGL*

Para a implementação do *driver* OpenGL neste trabalho, foi utilizado o projeto de código aberto ClusterGL³. Este projeto disponibiliza tanto o *driver* responsável por capturar e enviar as chamadas OpenGL, como também a aplicação *daemon* responsável por receber as chamadas da rede e realizar a exibição.

O ClusterGL possui algumas vantagens quando comparado com outras alternativas. Segundo o trabalho acadêmico apresentado por Neil (NEAL; HUNKIN; MCGREGOR, 2011), o projeto possui otimizações que produzem ganhos de desempenho consideráveis. Entre as otimizações mais importantes, podemos mencionar: compressão/descompressão do *stream* OpenGL, transferência com protocolo UDP *multicast* e uma técnica de codificação em delta dos comandos OpenGL (*delta encoding*).

Das otimizações presentes, a proposta de codificação em delta é a mais inovadora, não estando presente em nenhum dos outros trabalhos estudados. Esta técnica se beneficia do fato que aplicações OpenGL apresentam muito pouca ou nenhuma variação de comandos entre quadros consecutivos de renderização. Ou seja, os comandos usados para realizar a exibição do quadro no instante de tempo $N + 1$, normalmente, são muito semelhantes aos comandos utilizados no instante de tempo N .

A codificação em delta mantém uma cópia dos comandos utilizados no quadro anterior, tanto no *driver* como também no *daemon* de exibição. Antes de enviar os comandos pela rede, o *driver* compara as solicitações recebidas pela aplicação com os comandos do quadro anterior - quando houver semelhanças, ao invés de enviar as solicitações recebidas, o *driver* os substituirá por um único comando contendo apenas o número de requisições que o *daemon* deve repetir com base no histórico de comandos do quadro anterior.

Desta forma, o *driver* envia na rede apenas os comandos que são diferentes dos transmitidos anteriormente. Antes de enviá-los, os mesmos serão compactados e em seguida transmitidos com protocolo *multicast*.

Apesar destas vantagens, o projeto do *ClusterGL* contempla apenas funções do padrão OpenGL versão 2.1, lançado oficialmente em julho de 2006. No entanto, para verificar o funcionamento das técnicas propostas no capítulo 4, se torna necessário utilizar uma versão mais recente, como a versão 4.3 ou superior.

A versão 4.3 foi lançada em agosto de 2012. Atualmente, um amplo conjunto de placas gráficas já apresenta suporte a este padrão, no entanto, para viabilizar o uso destes

³<https://github.com/bieh/clustergl>. Último acesso: 5 de outubro de 2016.

recursos mais modernos foi necessário atualizar a implementação do ClusterGL. Para reduzir o esforço de programação e permitir que a atualização fosse efetuada no escopo deste trabalho, apenas um subconjunto do total de recursos foi implementado - considerando apenas os itens mais importantes, pertinentes aos testes que foram realizados.

Entre os recursos adicionados ao projeto, podemos mencionar: *Compute shader*, *Multi-draw indirect*, *Shader storage buffer object* e *Direct state access*. O *Compute shader* permite utilizar a placa gráfica para realizar cálculos de propósito genérico, de forma independente do *pipeline* de renderização. O recurso de *Multi-draw indirect* se refere ao mecanismo de renderização indireta, onde os comandos de renderização são armazenados diretamente em *buffers* na memória da placa gráfica. Por sua vez, o *Shader storage buffer object* é um tipo de *buffer* semelhante ao UBO, porém que permite a alocação de regiões com tamanhos maiores.

O *Direct state access* permite que as propriedades, associadas ao estado do OpenGL, sejam alteradas de forma mais eficiente e conveniente. Este último, foi o único recurso adicionado como uma extensão à versão 4.3 do padrão, pois o mesmo só foi integrado ao *core* do OpenGL a partir da versão 4.5.

5.2.1 Protocolo *multicast* e o sincronismo

Apesar do trabalho acadêmico do ClusterGL ter utilizado o protocolo *multicast* durante o seu desenvolvimento e avaliação, a parte do código que contempla o uso deste protocolo foi retirada do *branch* principal do projeto, no Github, em função de problemas na sua implementação.

Por conta disto, um novo mecanismo de transmissão baseado em *multicast* foi desenvolvido no escopo deste trabalho. Como o *multicast* utiliza o protocolo UDP (*User Datagram Protocol*), o mesmo não oferece nenhuma garantia com relação à entrega dos pacotes enviados, nem com relação à ordem com que os pacotes serão entregues, quando entregues.

Para contornar estes problemas, a abordagem utilizada neste trabalho considerou uma implementação simples, onde todo pacote contém um identificador único, permitindo que o receptor tenha condições de aferir se o mesmo perdeu algum pacote ou se o recebeu fora de ordem. Ao constatar esta situação, o receptor poderia, em tese, solicitar que o pacote perdido fosse enviado novamente.

Apesar disto, diversos testes realizados com o protocolo *multicast* mostraram que, em

Figura 17: Sincronismo realizado via *multicast*/UDP e TCP.

Fonte: Autor

uma comunicação na rede local, contendo apenas um único switch de rede e tendo todos os equipamentos envolvidos funcionando adequadamente, a perda de pacotes só ocorre por um motivo: no caso do destinatário não conseguir receber os pacotes a uma taxa maior do que a taxa praticada pelo remetente. Neste caso, o *buffer* presente no destinatário irá encher, e os pacotes recebidos a partir deste momento serão descartados.

Para garantir que a perda de pacotes não influenciasse os resultados obtidos, a implementação realizada neste trabalho utiliza um *thread* dedicado para a recepção de dados da rede, no aplicativo *daemon* do ClusterGL. Após a implementação deste recurso, não foi constatado nenhum erro, devido à perda de pacotes, durante todo o período de testes.

Além da comunicação *multicast*/UDP, o *driver* também mantém uma comunicação individual com cada máquina através do protocolo TCP (*Transmission Control Protocol*). Esta comunicação TCP é utilizada somente para a realização do sincronismo, garantindo que todos os computadores venham a apresentar os resultados obtidos simultaneamente.

Sendo assim, a comunicação entre o *driver* e as máquinas ocorre de forma semelhante ao ilustrado na figura 17. Os comandos referentes às chamadas OpenGL são enviados via *multicast*/UDP, permitindo que todas as máquinas recebam os mesmos pacotes de maneira eficiente. Cada máquina processa estes comandos e realiza a renderização conforme solicitado. Este processo de renderização, no entanto, é efetuado em um *buffer* que não está sendo exibido: o *back buffer*.

Ao finalizar a renderização, cada máquina participando deste processo informará o *driver* da aplicação, através do protocolo TCP. Apenas quando todas as máquinas tiverem

concluído, o *driver* solicitará a exibição dos resultados via *multicast*. Sendo assim, todas as máquinas realizarão a mudança dos *buffers* visíveis (*swap*), de forma praticamente simultânea.

5.2.2 Desafios na adaptação do *driver* OpenGL moderno

O *driver* ClusterGL, desenvolvido originalmente para versão 2.1 do padrão OpenGL, foi adaptado para a versão 4.3 visando utilizar os recursos modernos deste padrão para viabilizar o uso de aplicações complexas em sistemas distribuídos.

No entanto, a partir da versão 3.0, o OpenGL tornou obsoleto o uso do *pipeline* fixo de renderização (*fixed-function pipeline*). A partir deste momento, as aplicações são responsáveis por determinar todos os programas (*shaders*) que serão executados em todos os estágios do *pipeline* de renderização. Os programas fixos, que antes da versão 3.1 estavam disponíveis por padrão, agora foram removidos.

A partir do momento que a aplicação se torna responsável por configurar o seu *pipeline*, a mesma deve determinar não apenas os *shaders* que serão utilizados, mas também os valores e os *buffers* que serão usados para prover informações aos *shaders*. Sendo assim, cada aplicação possui liberdade para organizar o seu *pipeline* e as suas respectivas informações de forma diferente, criando a possibilidade para que otimizações específicas sejam aplicadas em cada caso.

De uma forma geral, ao trabalhar com o *pipeline* fixo de renderização, existiam funções específicas para controlar cada aspecto do funcionamento deste *pipeline*. No entanto, a partir da versão 3.1, estas funções específicas foram removidas em favor de um esquema de configuração mais flexível, que envolve o uso de *shaders* customizados e *buffers* de propósito genérico.

Esta flexibilidade, no entanto, prejudica o estabelecimento de um *driver* único capaz de corrigir e distribuir um amplo conjunto de aplicações gráficas. Entre diversos aspectos que precisam ser corrigidos pelo *driver*, duas matrizes contêm informações importantes para viabilizar o uso da aplicação em um sistema imersivo: a matriz que determina o ponto de vista do usuário (*view matrix*) e a matriz de projeção (*projection matrix*).

No padrão obsoleto do OpenGL, existiam funções específicas para realizar a mudança destas matrizes, o que facilitava a adaptação destas informações por *drivers* como o ClusterGL. Nesta nova versão, no entanto, o *driver* não possui meios para identificar em qual *buffer* a aplicação está armazenando estas informações. Além disto, não basta determinar

apenas o *buffer*, mas também o *offset* e o formato adotado para armazenar estas matrizes.

Embora estes avanços tenham agravado o desenvolvimento de *drivers* com esta finalidade, as versões antigas do padrão também apresentavam obstáculos consideráveis, o que no entanto, não impediu a adoção destas soluções - inclusive na esfera comercial, como é o caso do produto oferecido pela TechViz⁴.

Por exemplo, alguns obstáculos presentes desde as versões mais antigas do OpenGL, e que permanecem atualmente, é o uso de texturas para salvar os resultados intermediários do processo de renderização (*render targets*). Ao criar estas texturas e utilizá-las como destino do processo de renderização, é desejável que a resolução destas imagens sejam iguais ao do *buffer* principal da aplicação. Ambas as resoluções são mantidas pela aplicação, porém, ao substituir o *driver* para exibir o seu conteúdo em um sistema imersivo de alta resolução, as dimensões deste *buffer* principal serão modificadas pelo *driver*. Sendo assim, seria necessário realizar estas alterações nas texturas relevantes, mantendo-as sincronizadas.

Outro exemplo de agravante é o caso da aplicação solicitar a leitura de todo ou parte de um *buffer*, cujo conteúdo está armazenado na placa gráfica. Em um sistema distribuído, este *buffer* estará replicado em diversos computadores do sistema, sendo que cada computador terá, potencialmente, um conteúdo diferente para a região solicitada.

Estes são apenas alguns exemplos de agravantes que precisam ser contornados pelo *driver* do sistema distribuído. Normalmente, a solução adotada é específica para cada caso, considerando aspectos particulares de cada aplicação. Apesar deste cenário aparentemente desfavorável, algumas soluções já obtiveram êxito com esta abordagem, conforme mencionado no capítulo 2. Em muitos casos, estas soluções funcionam com adaptações diferentes para cada conjunto de ferramentas.

5.3 Aplicação gráfica para *benchmark*

Após a implementação do *driver* OpenGL, é necessário estabelecer uma aplicação gráfica para a realização de medidas de desempenho. Um dos requisitos desta aplicação é que mesma seja capaz de empregar diferentes técnicas de renderização para uma mesma cena virtual. Este recurso permitirá comparar de forma objetiva as diferentes abordagens apresentadas no capítulo anterior.

Para cumprir com este propósito foi adotada uma aplicação exemplo desenvolvida pela

⁴<http://www.techviz.net/techviz-x1>. Último acesso: 5 de outubro de 2016.

Nvidia, e disponibilizada gratuitamente com o seu respectivo código fonte. A aplicação, cujo nome é *gl_cadscene_rendertechniques*, pode ser obtida no repositório de exemplos da Nvidia, no Github⁵.

A proposta de adotar uma aplicação desenvolvida por terceiros ilustra a possibilidade de se usar uma ferramenta arbitrária. Embora diversas alterações tenham sido realizadas nesta ferramenta, o propósito destas modificações foi aumentar a cobertura dos testes realizados, não se mostrando necessário implementar qualquer ajuste para torná-la compatível com o novo *driver* OpenGL.

A aplicação *gl_cadscene_rendertechniques* foi elaborada pela Nvidia para testar o desempenho de diferentes formas de renderização. Estas alternativas de renderização produzem o mesmo resultado final, ou seja, a mesma imagem, porém fazendo uso de recursos distintos do OpenGL. Apesar da aplicação apresentar 6 (seis) técnicas diferentes de renderização, grande parte destas técnicas utiliza extensões do OpenGL que são específicas da Nvidia, por exemplo, a extensão *NV_command_list*⁶, e não podem ser utilizadas com placas gráficas de outros fabricantes.

Como o objetivo deste trabalho prevê a cobertura de um conjunto amplo de infraestruturas de visualização, extensões específicas de determinados fabricantes não foram avaliadas. Desta forma, apenas os modos de renderização que utilizam extensões disponíveis a todos os fabricantes foram testados. Entre os modos adotados pela aplicação, os seguintes foram considerados: *ubosub*, *uborange* e *indexedmdi*. Além destes três, um novo modo de renderização foi desenvolvido no escopo deste trabalho e incorporado à aplicação, o *indexedmdi_unified*. Este último foi implementado para verificar o desempenho de alguns aspectos que ainda não haviam sido considerados pela aplicação original.

Além deste modo de renderização, também foram implementados dois algoritmos de *frustum culling*, ambos inseridos na mesma ferramenta. Um dos algoritmos é executado na CPU e foi implementado de forma tradicional, utilizando os volumes delimitadores (*bounding volumes*) dos objetos de forma hierárquica. O segundo algoritmo é executado na GPU e foi elaborado utilizando o recurso de *Compute Shader* disponível na versão 4.3 do OpenGL. Conforme discutido na arquitetura de comunicação e ilustrado nas figuras 14 e 15, este último algoritmo foi implementado para verificar os volumes delimitadores de forma massivamente paralela e escrever os resultados em um buffer disponível na própria placa gráfica.

⁵<https://github.com/nvpro-samples>. Último acesso: 5 de outubro de 2016.

⁶https://www.opengl.org/registry/specs/NV/command_list.txt. Último acesso: 5 de outubro de 2016.

Todos estes modos de renderização usam recursos relativamente modernos do OpenGL e se limitam ao modo retido de operação (*retained mode*), onde todas as geometrias permanecem na memória da GPU. Além disto, ao renderizar uma sequencia de geometrias, todos os modos utilizados ordenam as mudanças de estados do OpenGL procurando minimizar o custo associado a estas alterações.

Nesta aplicação, a cena virtual é carregada de um arquivo no formato CSF, sendo que cada geometria contém a sua matriz de transformação (para posicioná-la no espaço) e as características do seu respectivo material. No arquivo CSF, as geometrias são organizadas de forma hierárquica e podem ser divididas em subpartes. Esta divisão em subpartes é realizada através de conjuntos diferentes de índices, porém que referenciam os mesmos vértices da malha poligonal da geometria. Este mecanismo é usado rotineiramente em computação gráfica para atribuir materiais diferentes para partes distintas de uma mesma malha poligonal. No caso desta aplicação, no entanto, os materiais não possuem texturas, apenas informação de cor.

Os detalhes específicos sobre o funcionamento destes modos de renderização serão apresentados nas seções seguintes. Nestas seções, também será comentado como estes modos devem se comportar em um ambiente distribuído, e como o algoritmo de *frustum culling* é utilizado em cada caso. Estas alternativas visam verificar a contribuição no desempenho, dos aspectos mais relevantes da arquitetura, como a redução da sobrecarga de comunicação entre a aplicação e a GPU, e o algoritmo de *culling*.

5.3.1 Modo *ubosub*

O modo *ubosub* é talvez a alternativa moderna mais tradicional de renderização. Além de fazer uso de recursos OpenGL com maior disponibilidade nos equipamentos de mercado, é de fácil implementação e possui grande compatibilidade com dispositivos móveis, sendo por exemplo, compatível até mesmo com o OpenGL ES 2.0.

Neste modo, cada geometria da cena possui um *buffer*, na placa gráfica, para armazenar os vértices de sua malha poligonal e outro *buffer* para armazenar os seus respectivos conjuntos de índices. As matrizes e as informações dos materiais permanecem na memória principal e são enviadas para a GPU apenas no momento da renderização.

Nesta implementação, existe um *buffer* na placa gráfica para manter apenas as informações do material vigente, e outro *buffer* para manter a matriz de transformação vigente. Antes de solicitar a renderização de uma determinada subparte da geometria, realiza-se

Figura 18: *Buffers* e *shaders* utilizados no modo *ubosub*.

Fonte: Autor

uma verificação para determinar se a matriz de transformação ou o material desta subparte são diferentes daqueles que estão vigentes na GPU. Caso se mostre necessário, a aplicação atualiza o conteúdo destes *buffers* antes de iniciar a renderização desta sub-região da geometria.

É importante notar que, no *pipeline* de renderização do OpenGL, os valores das variáveis presentes nos *shaders* serão obtidos de apenas um único *buffer*. Sendo assim, embora a aplicação possa alocar um número arbitrário de regiões na memória da GPU, a mesma deve selecionar qual será o *buffer* vigente para uma determinada variável do *shader*. Esta seleção permanecerá válida até a próxima mudança, e os valores desta variável serão retirados apenas deste *buffer*, durante todo o processo de renderização.

Conforme apresentado na figura 18, todos os objetos da cena estão armazenados em um conjunto de *buffers* na placa gráfica contendo seus respectivos vértices e índices. No momento da renderização de uma determinada geometria, o programa irá selecionar o buffer de vértices desta geometria e selecionar o seu *buffer* de índices. Após esta seleção, o mesmo irá percorrer todas as subpartes desta geometria e para cada uma delas irá, se necessário, atualizar o conteúdo dos *buffers* contendo as informações do material e da matriz de transformação, para somente então solicitar a sua renderização. Nesta solicitação, é especificado que apenas uma sub-região do *buffer* de índices deve ser usada - a sub-região correspondente à subparte da geometria que está sendo renderizada.

Para representar este processo de configuração, a imagem 18 utiliza setas tracejadas para indicar mapeamentos fixos entre variáveis do *shader* e *buffers*, e setas pontilhadas para indicar mapeamentos dinâmicos, onde a aplicação precisa alterar, para cada geometria, o *buffer* utilizado em cada variável.

Os tamanhos das regiões reservadas para o material e para a matriz são pequenos, em torno de 64 bytes cada uma. No entanto, quando muitos objetos distintos estiverem presentes na cena, será necessário atualizar o conteúdo destas regiões diversas vezes para cada quadro renderizado, podendo prejudicar o desempenho da aplicação.

Neste modo de operação, a alternativa mais vantajosa para executar o *frustum culling* é através do algoritmo tradicional, efetuado na CPU. Uma vez que todas as matrizes permanecem na memória principal, a aplicação pode verificar na própria CPU quais são os objetos que permanecem visíveis para um determinado ponto de vista do usuário, e realizar somente a renderização destas geometrias, o que reduz o número de matrizes de transformação e materiais enviados para a placa gráfica.

Aplicações que fazem uso deste modo normalmente não realizam *culling* na GPU, pois seria necessário enviar todas as matrizes e outras informações da cena para a placa gráfica, submeter um programa de *Computer Shader* para realizar o *culling* e, em seguida, ler todos os resultados de volta da GPU para a memória principal, para só então verificar quais geometrias precisam ser renderizadas. Esta abordagem penaliza severamente o desempenho da aplicação pois envolve a transferência de um grande volume de informações tanto para escrita como também leitura na placa gráfica.

Conforme discutido no capítulo 4, as técnicas de *frustum culling* que operam na CPU são invariavelmente incompatíveis com sistemas distribuídos através do *driver* OpenGL. Para tornar estas aplicações funcionais nestes cenários, se torna necessário desabilitar o algoritmo de *culling*, fazendo com que todas as geometrias da cena sejam sempre enviadas através do *driver*.

5.3.2 Modo *uborange*

O modo *uborange* é semelhante ao anterior, onde todas as geometrias da cena permanecem em *buffers* individuais, contendo tanto a sua malha poligonal como também os seus conjuntos de índices. No entanto, neste modo, todas as matrizes de transformação e as informações dos materiais também são armazenadas na memória da placa gráfica.

Entretanto, considerando que o espaço necessário para armazenar uma única matriz de transformação e o material de uma determinada subparte é pequeno, em torno de 128 bytes no total, a acomodação destas informações para todos os objetos da cena implicaria na criação de um número elevado de *buffers*, cada um com tamanho muito pequeno. Isto poderia sobrecarregar desnecessariamente o *driver* com tarefas de gerenciamento de

Figura 19: *Buffers* e *shaders* utilizados no modo *uborange*.

Fonte: Autor

memória.

Para contornar este problema, esta implementação utiliza uma única região para armazenar todas as matrizes de transformação, de todas as geometrias presentes na cena. O mesmo processo é repetido para armazenar as informações dos materiais. Consequentemente, apenas dois *buffers* com tamanhos maiores podem ser utilizados, por exemplo.

Desta forma, para renderizar uma determinada geometria, o programa irá selecionar os seus respectivos *buffers* de vértices e índices, percorrer todas as subpartes desta geometria e para cada uma delas irá selecionar as regiões apropriadas dos *buffers* contendo as informações do seu material e da sua matriz de transformação. Para viabilizar esta implementação, ao invés de associar um *buffer* completo com uma determinada variável de *shader*, é preciso associar apenas uma sub-região deste *buffer* com a variável, o que requer um recurso do OpenGL/ES disponível apenas a partir da versão 3.

Conforme ilustrado na figura 19, a dinâmica de funcionamento é semelhante ao modo *ubosub*. No entanto, algumas conexões com as variáveis do *shader*, que antes eram estáticas e representadas por setas tracejadas, agora passam a ser dinâmicas e são representadas por setas pontilhadas, como as demais conexões. Sendo assim, apesar de ser necessário configurar dois novos mapeamentos para cada subparte de geometria, este mapeamento é realizado de forma mais eficiente do que enviar todas as matrizes e materiais para a placa gráfica a cada quadro de renderização - principalmente nos casos em que estes comandos são distribuídos na rede.

Apesar das matrizes já estarem presentes na memória da GPU, a CPU ainda é responsável por configurar os mapeamentos de forma adequada e solicitar as tarefas de renderização. Por conta disto, no caso do algoritmo de *culling*, a CPU ainda deve verificar

quais são os objetos que estão visíveis para o usuário, para somente então renderizá-los. No entanto, como as matrizes de transformação estão armazenadas na memória da placa gráfica, a aplicação precisaria obter estas matrizes da GPU para determinar a posição dos seus respectivos objetos.

Para evitar a leitura desta informação, o algoritmo de *culling*, ao operar neste modo, mantém uma cópia das matrizes na memória principal da máquina. Sendo assim, as matrizes são armazenadas de maneira duplicada, consumindo mais memória para obter desempenho. Desta forma, o algoritmo de *culling* utiliza somente as matrizes presentes na memória principal. Porém, após determinar a necessidade de exibir uma geometria específica, a mesma será renderizada com a matriz armazenada na memória da placa gráfica, sendo fundamental manter estas duas informações permanentemente sincronizadas.

Apesar de reduzir significativamente a quantidade de dados enviados para a GPU em cada quadro, este modo ainda padece da incompatibilidade com alguns sistemas imersivos, devido à utilização do algoritmo de *culling* na CPU. Aplicações que usam este modo de renderização ainda precisarão desabilitar este algoritmo ao serem utilizadas em infraestruturas imersivas.

5.3.3 Modo *indexedmdi*

Este modo de renderização é substancialmente distinto dos anteriores e utiliza recursos modernos do OpenGL que só foram incorporados ao *core* do padrão a partir da versão 4.3. A principal diferença existente entre este modo e os anteriores é o uso do mecanismo de renderização indireta, onde as tarefas de renderização são processadas diretamente pela GPU, através de comandos colocados na memória da placa gráfica. A aplicação é responsável por criar e preencher este *buffer*, sendo que o mecanismo utilizado para preenchê-lo pode ser através da transferência direta do seu conteúdo, ou de forma programática, realizando a geração deste conteúdo através de um *Compute Shader*, por exemplo.

Após o preenchimento deste *buffer*, a aplicação deve solicitar que o mesmo seja processado pela GPU. Cada um destes comandos realiza a renderização de uma única geometria ou subparte de geometria. Sendo assim, através de uma única solicitação da CPU, diversos comandos podem ser processados, realizando a renderização de diversos elementos de forma indireta.

A listagem 1 apresenta o formato do comando de renderização. As informações presentes nesta estrutura são: **count**, que representa o número de índices usados nesta ge-

ometria ou subparte, `instanceCount`, o número de instâncias que devem ser renderizadas, `firstIndex`, que contém o *offset* para o primeiro índice desta geometria no *buffer* ativo de índices, `baseVertex`, que contém o *offset* do primeiro vértice no *buffer* de vértices e o `baseInstance`, com o *offset* dos atributos de instância em seu respectivo *buffer*.

Listagem 1: Formato do comando de renderização.

```
struct DrawCommand {
    uint    count;
    uint    instanceCount;
    uint    firstIndex;
    uint    baseVertex;
    uint    baseInstance;
};
```

Para solicitar a renderização, um número arbitrário destes comandos pode ser usado, todos eles colocados sequencialmente na memória da placa para uma mesma solicitação. Um aspecto importante é que estes comandos não possuem informação a respeito de quais *buffers* devem ser usados durante a renderização, mas apenas informações de *offset* nos *buffers* vigentes. Sendo assim, antes de solicitar a interpretação destes comandos, a aplicação ainda deve preparar o estado corrente do OpenGL, selecionando qual *shader* deverá ser usado e quais são os *buffers* vigentes, que proverão as informações necessárias para cada variável do *shader* selecionado.

Na implementação do projeto *gl_cadscene_rendertechniques*, todos os comandos de renderização, de todas as geometrias presentes na cena, são colocados em um único *buffer* na placa gráfica. Porém, como cada geometria utiliza uma região diferente de memória para armazenar os seus respectivos vértices e índices, a aplicação não pode solicitar o processamento de todos estes comandos simultaneamente, pois os *buffers* vigentes precisam ser alterados entre geometrias diferentes.

Desta forma, para renderizar uma determinada geometria, o programa irá selecionar (ativar) os seus respectivos *buffers* de vértices e índices, e solicitar o processamento dos comandos de renderização presentes em uma sub-região do *buffer* de comandos - sub-região que contém somente os comandos das subpartes desta geometria. Esta abordagem reduz o número de chamadas OpenGL e, conseqüentemente, a sobrecarga de comunicação, pois ao invés de solicitar a renderização de cada subparte individualmente, a aplicação realiza uma única solicitação, que compreende todas as subpartes.

Além disto, o comando associado com cada subparte precisa de meios para informar ao *shader* o material e a matriz de transformação que devem ser utilizados por cada subparte. Ao contrário do praticado pelo modo *uborange*, não é possível alterar os *buffers*

Figura 20: *Buffers* e *shaders* utilizados no modo *indexedmdi*.

Fonte: Autor

ativos (vigentes) enquanto a GPU processa uma sequência de comandos de renderização.

No entanto, conforme mencionado no capítulo 4, o mecanismo de renderização indireta pode utilizar o recurso de atributos de instância para fazer esta associação. Neste modo, este recurso é utilizado através do parâmetro **baseInstance** da estrutura do comando. Este atributo é utilizado para identificar o material e a matriz de transformação que devem ser utilizados em cada caso. Como existem duas informações que precisam ser identificadas através de uma única variável, esta identificação é realizada de forma indireta. O atributo **baseInstance** contém um *offset* para um *buffer* adicional, sendo que este *buffer* contém apenas dois índices (apontadores): um índice para o material e outro para a matriz de transformação.

De forma semelhante ao modo *uborange*, esta implementação utiliza uma única região de memória para armazenar todas as matrizes de transformação, e outra região para todos os materiais da cena. Conforme ilustrado na figura 20, este novo *buffer* possui, para cada subparte de geometria, dois índices (apontadores) que identificam em qual posição do *buffer* de materiais estão as informações desta subparte, e em qual posição do *buffer* de matrizes está a sua matriz de transformação.

Apesar da maior complexidade desta abordagem, a mesma permite que a comunicação entre a CPU e a GPU seja reduzida drasticamente, aumentando o desempenho. Ao invés de solicitar a renderização de cada subparte individualmente, a aplicação solicitará apenas a renderização de cada geometria, sendo que o processamento de cada subparte desta

geometria será realizado integralmente na placa gráfica.

Além disto, outra vantagem deste modo está na realização do algoritmo de *frustum culling*. Uma vez que todas as matrizes de transformação e todos os comandos de renderização estão armazenados na placa gráfica, se torna viável executar este algoritmo na GPU, sem a necessidade de aumentar as informações trocadas entre CPU/GPU e prejudicar o desempenho.

A implementação deste algoritmo utiliza um *Computer Shader* para ler dois *buffers* armazenados na placa gráfica: um deles contendo as matrizes de transformação e o outro contendo os volumes delimitadores (*bouding volumes*), de todas as geometrias. Este programa processa estas informações considerando o ponto de vista do observador virtual e identifica, para cada subparte de geometria, se a mesma está visível ou não. O resultado desta verificação é armazenado no *buffer* contendo os próprios comandos de renderização de cada subparte, mais especificamente no atributo `instanceCount` de cada comando. Este atributo assume o valor 1 (um) caso a subparte esteja visível, ou 0 (zero) caso contrário.

Desta forma, mesmo quando o objeto não estiver visível, o comando de renderização permanecerá na memória da placa gráfica, porém com o valor 0 (zero) associado ao número de instâncias solicitadas. Quando a GPU interpretar este comando, o mesmo será ignorado conforme previsto no padrão OpenGL.

De acordo com o que foi apresentado no capítulo 4, esta técnica de *frustum culling* é compatível com os sistemas imersivos cuja distribuição ocorre através do *driver* OpenGL, pois a mesma é executada na GPU e, portanto, não precisa ser desabilitada. Nestas circunstâncias, cada máquina presente na infraestrutura executará o *Compute Shader* considerando o seu próprio ponto de vista da cena. Consequentemente, este programa de *shader* produzirá resultados diferentes em cada computador - cada máquina terá a sua cópia do *buffer* de comandos preenchida de forma distinta, considerando apenas as geometrias visíveis para o seu ponto de vista.

Embora cada computador produza uma versão diferente do *buffer* de comandos, todas as cópias deste *buffer* possuirão exatamente o mesmo número de comandos, porém com variações diferentes do atributo `instanceCount`. Este fato é importante para viabilizar o funcionamento desta solução pois, para solicitar o processamento deste *buffer*, a aplicação precisa informar o número de comandos presentes - sendo este número igual em todas as máquinas, o comando pode ser simplesmente repassado pelo *driver*, sem a necessidade de qualquer alteração.

5.3.4 Modo *indexedmdi_unified*

O modo *indexedmdi_unified* possui o mesmo mecanismo de operação do modo *indexedmdi*, porém com algumas alterações para reduzir ainda mais a comunicação entre CPU e GPU. Neste modo, todas as geometrias da cena foram consolidadas em um mesmo conjunto de *buffers* visando maximizar o número de comandos processados pela placa gráfica, sem precisar de intervenção da aplicação.

Para efetuar esta consolidação, todos os vértices de todas as geometrias da cena são colocados em um único *buffer*. Este mesmo processo é repetido para os índices dos vértices, para os materiais, para as matrizes de transformação e para os índices (apontadores) de materiais e matrizes - cinco *buffers* para todas as geometrias da cena. Após consolidar as informações das geometrias em um mesmo conjunto de *buffers*, se torna possível processar todos os comandos de renderização através de uma única solicitação. Isto ocorre porque não é mais necessário alterar os *buffers* vigentes entre diferentes geometrias, uma vez que todas as geometrias utilizam os mesmos *buffers*.

Assim como no modo anterior, a aplicação cria uma única região contendo todos os comandos de renderização, de todas as geometrias da cena. Porém, ao contrário do modo anterior, onde a aplicação precisa solicitar cada geometria individualmente, neste modo, toda a cena virtual é renderizada com uma única requisição. Com esta requisição, a GPU irá processar todos os comandos presentes neste *buffer*.

Este modo de renderização foi implementado para investigar a contribuição de técnicas que procuram consolidar geometrias diferentes nas mesmas regiões de memória, reduzindo o número de alterações no estado corrente do OpenGL e reduzindo a comunicação CPU/GPU com o objetivo de aumentar o desempenho. Embora a consolidação de todos os objetos da cena possa se mostrar inviável em aplicações práticas, este caso de uso nos permite aferir qual a contribuição desta abordagem no desempenho da aplicação.

De forma semelhante ao modo anterior, este também utiliza o algoritmo de *frustum culling* na GPU, sendo compatível com infraestruturas imersivas de renderização que utilizam o *driver* OpenGL como mecanismo de distribuição. É importante notar, no entanto, que embora a renderização da cena exija uma única solicitação por parte da aplicação, a placa gráfica processará um *buffer* contendo a mesma quantidade de comandos.

5.4 Síntese

Este capítulo apresentou as ferramentas de *software* que foram adotadas para a realização dos testes deste trabalho. Estes testes permitiram uma avaliação quantitativa a respeito dos resultados proporcionados pelas técnicas discutidas no capítulo 4. As ferramentas escolhidas são gratuitas e de código aberto, o que viabilizou a realização de diversas alterações, visando atualizar os recursos disponíveis e ampliar a cobertura dos testes.

Os recursos desenvolvidos neste capítulo permitem que o conteúdo de uma determinada aplicação OpenGL seja exibido em uma infraestrutura distribuída de visualização, através da substituição do seu respectivo *driver*. Para a implementação deste *driver* foi adotado o projeto ClusterGL, cujo código foi expandido e adaptado neste trabalho, para acomodar diversas rotinas pertencentes à versão 4.3 do padrão.

A aplicação adotada como caso de uso deste *driver* foi a ferramenta *gl_cadscene_rendertechniques*, desenvolvida originalmente pela Nvidia. Esta ferramenta foi considerada por utilizar diversas técnicas de renderização, usando recursos de versões diferentes do OpenGL. Os seguintes modos de renderização da aplicação foram considerados nos testes: *ubosub*, *uborange*, *indexedmdi* e *indexedmdi_unified*. Entre as modificações realizadas nesta aplicação, podemos mencionar: a introdução de algoritmos de *frustum culling*, tanto em CPU como em GPU, e a criação do modo de renderização *indexedmdi_unified*.

6 RESULTADOS

Este capítulo apresenta os casos de uso elaborados para analisar o funcionamento da arquitetura de comunicação. Os testes visam verificar a contribuição de algumas das técnicas discutidas neste trabalho no desempenho de aplicações de visualização em um sistema convencional e em sistemas imersivos.

Os testes foram realizados utilizando as ferramentas de *software* apresentadas no capítulo anterior e a infraestrutura computacional disponível no laboratório Caverna Digital, da Escola Politécnica da USP. Conforme apresentado na implementação, o *software* fará uso dos sistemas imersivos mediante a troca do *driver* OpenGL. Os testes serão realizados com diferentes técnicas de renderização e com cenas de diferentes complexidades geométricas, para analisar o comportamento da aplicação em cada um destes cenários.

6.1 Infraestrutura de testes

Os testes foram realizados em 4 sistemas diferentes: um computador convencional com *driver* OpenGL original, para referência, um sistema plano contendo um único computador remoto, um sistema plano (*powerwall*) com 3 computadores remotos e um sistema de 3 telas, uma frontal e duas laterais, com 9 computadores remotos. Estes últimos sistemas, que mencionam o uso de computadores remotos, fazem uso destas máquinas exclusivamente para exibição, e são utilizadas de forma transparente pelo *driver* OpenGL. Nestes casos, a aplicação é executada em seu próprio computador, porém com imagens exibidas nos seus respectivos sistemas.

A figura 21 apresenta uma visão geral das infraestruturas que serão utilizadas com o *driver* adaptado. Nesta ilustração, os elementos representados com a letra M são as máquinas e os representados com a letra P são os projetores. A aplicação executa em seu próprio computador (M0), sendo o *driver* OpenGL responsável por transferir as informações necessárias às outras máquinas do sistema.

Figura 21: Visão geral das infraestruturas de teste.

Fonte: Autor

Nestas infraestruturas, cada computador está conectado a dois projetores: O primeiro sistema utiliza uma única máquina com dois projetores. O segundo sistema utiliza três máquinas e seis projetores, e o terceiro usa nove computadores e dezoito projetores, sendo seis projetores em cada tela. Para facilitar a comparação dos resultados, todas as máquinas são iguais, com a mesma configuração e placa gráfica.

Os computadores possuem processador Intel Xeon W3690 3,4GHz, com 6 núcleos e 12 threads, 12MB de cache, 24GB de memória RAM e uma placa gráfica Quadro 5000, com 2,5GB de memória dedicada. Cada máquina está conectada a dois projetores DepthQ HDs3D1, ambos com resolução de 1280 x 720. Todos os sistemas funcionam com projeção por trás da tela (*rear projection*), onde a imagem de cada projetor possui uma margem de sobreposição com os projetores vizinhos, para viabilizar a fusão dos resultados (*edge blending*).

Em todos os testes, as máquinas foram conectadas em um mesmo switch de rede, modelo 3Com Baseline 2824, que opera a 1Gbps (*duplex*) e cujo método de repasse de pacotes consiste em armazenar para somente depois encaminhar. As máquinas executam o sistema operacional Linux, distribuição Ubuntu Desktop LTS versão 16.04.1, com driver da Nvidia versão 367.57.

Os *benchmarks* foram efetuados com a aplicação gráfica sem estereoscopia, com o objetivo de realizar uma melhor estimativa do desempenho de cada modo de renderização. O impacto da estereoscopia no desempenho depende da infraestrutura adotada, se ativa

ou passiva, e quantos pontos de vista cada máquina irá renderizar.

No caso mais comum, cada máquina renderiza os dois pontos de vista (esquerdo e direito). Sendo assim, a cena virtual precisa ser submetida duas vezes para a GPU, aumentando consideravelmente o tempo de renderização. No entanto, algumas placas gráficas mais modernas, como a arquitetura Pascal da Nvidia, já possuem recursos para renderizar dois pontos de vista (estereoscópicos) simultaneamente, submetendo a cena virtual apenas uma única vez à GPU (*Single Pass Stereo*¹).

Desta forma, para aferir o desempenho de cada modo, evitando que os resultados obtidos estejam sujeitos a estas variações específicas de estereoscopia, a aplicação funciona renderizando um único ponto de vista por quadro, que é de fato, o caso de muitos sistemas imersivos, como apresentado no capítulo 2.

6.2 Modelo da cena virtual

O modelo utilizado na composição da cena virtual é um modelo de CAD, contendo uma placa gráfica GeForce da Nvidia, disponibilizado juntamente com o projeto *gl_cadscene_renderingtechniques*. Este modelo foi adotado por possuir características semelhantes aos encontrados neste segmento, uma vez que o arquivo disponibilizado foi extraído de uma aplicação real, não sendo portanto, um modelo sintético elaborado apenas para estes testes.

Para utilizar cenas virtuais com diferentes níveis de complexidade geométrica, o modelo da placa é replicado diversas vezes. Com o objetivo de se criar uma cena semelhante a um caso de uso, as placas são distribuídas em torno do usuário, simulando a presença de outros objetos na cena. Esta distribuição é realizada de tal forma que o usuário sempre terá uma placa em sua frente e, quando configurado, um conjunto de cópias distribuídas igualmente em todas as outras direções: frente, trás, esquerda e direita.

Durante a replicação deste modelo, as geometrias se comportam como se pertencessem à objetos diferentes, sendo que a aplicação não faz uso do fato que os modelos são iguais - não há nenhuma otimização neste aspecto. Todas as geometrias permanecem replicadas na memória da placa gráfica, procurando reproduzir com mais fidelidade cenas de maior complexidade geométrica. Para os testes realizados neste trabalho, utilizou-se cenas com 1, 27, 64 e 125 cópias da placa.

A figura 22 apresenta o modelo CAD da placa utilizada. A imagem da direita ilustra o

¹<https://developer.nvidia.com/vrworks>. Último acesso: 5 de outubro de 2016.

Figura 22: Modelo CAD da placa GeForce utilizado nos testes.

Fonte: Autor (modelo 3D da Nvidia)

mesmo objeto, porém com as geometrias expandidas e os componentes internos à mostra. Uma única instância deste objeto contém 218 mil vértices, divididos em 110 geometrias. Cada uma destas geometrias contém o seu próprio conjunto de vértices, que é dividido em subpartes através dos seus respectivos índices. Esta divisão é realizada para permitir a aplicação de materiais e matrizes de transformação diferentes em cada subparte.

O número total de subpartes do modelo é de 22.514, ou seja, em média 205 subpartes por geometria. Além disto, o modelo ainda contém 66 materiais e 5.004 matrizes de transformação. Sendo assim, para os casos com 27, 64 e 125 cópias da placa, a cena virtual terá 5,4 milhões, 13,9 milhões e 27,2 milhões de vértices, respectivamente.

6.3 Movimentação da câmera e amostragem

Para melhor representar o uso de uma ferramenta convencional, a câmera da cena virtual se movimenta durante a realização do *benchmark*, seguindo um percurso previamente determinado pela aplicação. O percurso realizado pela câmera pode ser observado na figura 23.

Nesta figura, a linha azul representa o caminho que é percorrido pela câmera. As esferas vermelhas representam as posições adotadas para os modelos com as placas gráficas da Nvidia. Embora o exemplo ilustrado contenha 27 cópias deste modelo, as cenas com um número diferente de cópias possuem uma distribuição semelhante, com os modelos posicionados em torno da origem, separados por uma distância igual em todos os eixos.

A posição inicial da câmera é na origem da cena, no centro dos modelos. A câmera demora 1 minuto para percorrer o caminho e retornar à posição inicial. Durante a realização do *benchmark*, o mesmo percurso é repetido 10 vezes na mesma cena, com a

Figura 23: Percurso realizado pela câmera virtual.

Fonte: Autor

mesma técnica de renderização. A amostragem do tempo médio de renderização, medida utilizada para comparar os diferentes testes, é realizada a cada 2 segundos. Sendo assim, são colhidas 300 amostras para cada *benchmark*.

Conforme pode ser observado na figura 23, o percurso estabelecido contém posições em que a câmera virtual enxergará porções consideravelmente diferentes da cena. Em determinadas posições, por exemplo, a câmera enxergará todos os objetos da cena, enquanto que em outras, a câmera enxergará um número muito pequeno ou nenhum objeto.

Esta variação do número de objetos visíveis produzirá medidas com desvio padrão muito acentuado em cada *benchmark*. De qualquer forma, os resultados serão comparados conforme os valores médios obtidos e a amplitude da variação das amostras em torno deste valor médio.

No entanto, como esta variação ocorre em função do caminho a ser percorrido pela câmera, as amostras não terão uma distribuição normal (gaussiana). Sendo assim, para melhor representar a variação das amostras em torno do seu valor médio, os gráficos apresentados na seção seguinte adotarão o valor do 3º Quartil, ao invés do desvio padrão tradicional. O valor do 3º Quartil representa o intervalo onde se encontram 75% de todas as amostras obtidas. Além disto, como a distribuição não é simétrica em torno do valor médio, os gráficos apresentam dois intervalos para o 3º Quartil, um contendo as amostras acima do valor médio, e o outro contendo as amostras abaixo do valor médio.

6.4 Análise dos resultados

Para analisar os resultados, é importante dispor de medidas de referência, obtidas com a aplicação executando em uma única máquina (local), e com o *driver* OpenGL original. A configuração da máquina utilizada neste caso é igual ao apresentado na seção anterior, sendo que a mesma também está conectada a dois projetores DepthQ.

Os modos de renderização apresentados nesta seção são os mesmos mencionados no capítulo 5. Porém, quando o modo mencionado estiver fazendo uso do algoritmo de *culling*, seu nome terá o sufixo *cpucull* ou *gpucull*, caso este algoritmo esteja executando na CPU ou na GPU, respectivamente. Quando o nome do modo não tiver estes sufixos, isto significa que o modo não está utilizando *culling* e, portanto, que o mesmo submete todas as geometrias da cena para a GPU, independente do que está visível ou não para o usuário.

A tabela 3 apresenta o número de chamadas de renderização utilizadas pela aplicação. Como o número de objetos visíveis varia conforme o trajeto percorrido pela câmera, estas medidas foram realizadas na sua posição inicial, na origem da cena. Neste teste, a aplicação utiliza um FOV (*Field of view*) vertical de 45° e aspecto (*aspect ratio*) proporcional à resolução da tela ($2 * 1280/720 \approx 3,56$).

Tabela 3: Número de chamadas de renderização por quadro (local).

Modos de renderização	Qtd. de modelos			
	1	27	64	125
<i>ubosub</i>	11.103	299.781	710.592	1.387.875
<i>ubosub_cpucull</i>	11.103	44.412	178.944	213.746
<i>uborange</i>	11.103	299.781	710.592	1.387.875
<i>uborange_cpucull</i>	11.103	44.412	178.944	213.746
<i>indexedmdi</i>	326	9.100	21.569	42.126
<i>indexedmdi_gpucull</i>	326	9.100	21.569	42.126
<i>indexedmdi_unified</i>	1	1	1	1
<i>indexedmdi_unified_gpucull</i>	1	1	1	1

Esta tabela revela características importantes de cada técnica. Por exemplo, o número de chamadas realizadas pelo *ubosub* e *uborange* são iguais, pois ambas são muito semelhantes e utilizam o mesmo algoritmo de *culling* na CPU. Além disto, este algoritmo de *culling* produz resultados notáveis com o aumento do número de modelos.

Já nas técnicas de renderização indireta (*indexedmdi*), observa-se que o número de solicitações é reduzido drasticamente, pois cada solicitação contempla diversos comandos

Figura 24: Tempo médio de renderização (local).

de renderização, que serão processados pela GPU. Na abordagem *indexedmdi_unified*, por definição, toda a cena é renderizada com uma única chamada.

Nos casos de renderização indireta (*indexedmdi*), o *culling* não altera o número de solicitações realizadas pela CPU. O resultado deste algoritmo é armazenado diretamente na memória da placa gráfica, com o intuito de reduzir o número de comandos processados pela GPU. Como a imagem produzida por cada modo de renderização é idêntica, pode-se concluir que no modo *indexedmdi*, por exemplo, as 326 solicitações realizadas com um único modelo CAD irão produzir as mesmas 11.103 renderizações, ou seja, cada solicitação acarreta no processamento médio de 34 comandos.

O caso mais extremo ocorre no modo *indexedmdi_unified* com 125 modelos, onde uma única solicitação requer o processamento de 1.387.875 comandos, ou 213.746 comandos se o *culling* estiver habilitado. Sendo assim, o algoritmo de *culling* do modo *indexedmdi_unified_gpucull* exclui 1.174.129 comandos da renderização ($1.387.875 - 213.746$), somente através da alteração dos atributos `instanceCount` nestes comandos.

A figura 24 apresenta os tempos médios (e 3º quartis) de renderização de cada quadro, para a mesma aplicação executando em uma máquina local. Os tempos obtidos com um único modelo 3D foram omitidos, pois não seria possível observá-los na escala do gráfico. Sendo assim, os mesmos valores também estão presentes na tabela 4.

Nesta tabela, é possível notar que as técnicas mais modernas, que reduzem a comunicação CPU/GPU, invariavelmente produzem melhor desempenho. No entanto, este ganho só se torna perceptível em cenas mais complexas. Por exemplo, ao comparar o *uborange*,

que evita a transferência de matrizes e materiais em todo quadro renderizado, com o *ubosub*, os dois apresentam resultados praticamente idênticos com um único modelo. Porém, com 125 modelos, os ganhos de desempenho podem chegar a 30%.

Tabela 4: Tempo médio de renderização local (milissegundos).

Modos de renderização	Qtd. de modelos			
	1	27	64	125
<i>ubosub</i>	2.94	43.77	103.31	200.92
<i>ubosub_cpucull</i>	2.11	12.17	26.90	52.21
<i>uborange</i>	2.93	31.82	92.64	177.04
<i>uborange_cpucull</i>	2.12	9.04	21.60	42.03
<i>indexedmdi</i>	2.66	23.87	53.20	101.15
<i>indexedmdi_gpucull</i>	2.54	16.16	34.97	65.21
<i>indexedmdi_unified</i>	2.64	23.30	51.75	98.26
<i>indexedmdi_unified_gpucull</i>	2.53	15.27	32.95	61.27

O algoritmo de *culling* também se mostra indispensável em todas as circunstâncias, com exceção das cenas com um único modelo. Neste último caso, como o modelo está quase sempre visível, o algoritmo produz pouco benefício. Entretanto, estas medidas nos permitem aferir o *overhead* do *culling*. No caso da versão implementada na CPU, a técnica utilizada consiste na verificação dos objetos de forma hierárquica, ou seja, o *overhead* varia conforme o número de objetos visíveis. No caso do algoritmo implementado na GPU, o mesmo não utiliza uma abordagem hierárquica e, portanto, sempre verifica a mesma quantidade de comandos, variando apenas com a complexidade da cena.

Este último aspecto, da diferença entre os dois algoritmos de *frustum culling*, aparece de forma evidente neste *benchmark*. Apesar dos modos indiretos (*indexedmdi* e *indexedmdi_unified*) se mostrarem consistentemente mais vantajosos que os demais modos (*ubosub* e *uborange*), ao habilitar o *culling*, observa-se uma inversão ondes os modos tradicionais passam a ter melhor desempenho. Este resultado mostra que, a partir de uma determinada complexidade de cena, com mais de 1 modelo, o algoritmo hierárquico de *culling* utilizado pela CPU apresenta resultados superiores do que o paralelismo (força bruta) implementado na GPU. Apesar de existirem técnicas mais sofisticadas de *culling* na GPU, com recursos limitados de hierarquia, este mecanismo não foi considerado neste trabalho.

Além disto, apesar do modo *indexedmdi_unified* utilizar um único comando para solicitar a renderização de toda a cena, o mesmo apresentou um ganho muito pequeno de desempenho quando comparado com o modo *indexedmdi*. No caso de 125 modelos, apesar do número de solicitações ser reduzido de 42.126 para apenas 1 (uma), o ganho foi inferior

Figura 25: Tempo médio de renderização (1 máquina remota).

a 3% com o *culling* desabilitado e de apenas 6% após habilitá-lo.

6.4.1 Uma única máquina remota

Esta seção apresenta os primeiros resultados obtidos com o *driver* OpenGL modificado, sendo as suas funções transferidas pela rede *Ethernet*. Neste caso de uso, foram utilizados dois computadores iguais: um deles para a aplicação e o outro somente para a exibição do conteúdo.

Os tempos médios de renderização estão representados na figura 25, e seus valores na tabela 5. Apesar desta infraestrutura não representar um sistema imersivo real, os valores obtidos nos permitem analisar o *overhead* do *driver* OpenGL e de sua respectiva abstração da rede. Esta comparação pode ser realizada, pois a imagem obtida nesta configuração é exatamente igual à obtida com um único computador local. Em ambos os casos, a imagem é exibida em dois projetores DepthQ, na mesma resolução e com o mesmo *frustum* de visualização.

Sendo assim, é possível observar o impacto da distribuição no desempenho das alternativas mais convencionais de renderização. No caso do modo *ubosub_cpucull* com 64 modelos, por exemplo, o desempenho piorou, em média, mais de 600%, reduzindo consideravelmente o número de quadros renderizados por segundo, quando comparado com um computador local.

É importante notar que embora exista uma comunicação através da rede, a maioria das informações não são de fato enviadas, uma vez que o *driver* utiliza uma codificação

Tabela 5: Tempo médio de renderização com 1 máquina remota (milissegundos).

Modos de renderização	Qtd. de modelos			
	1	27	64	125
<i>ubosub</i>	5.02	124.74	297.47	579.60
<i>ubosub_cpucull</i>	5.59	74.21	188.83	329.32
<i>uborange</i>	5.09	87.71	209.81	410.92
<i>uborange_cpucull</i>	2.89	25.05	68.16	113.16
<i>indexedmdi</i>	3.16	25.00	55.85	108.42
<i>indexedmdi_gpucull</i>	2.94	16.11	35.57	67.57
<i>indexedmdi_unified</i>	3.16	23.64	51.74	98.21
<i>indexedmdi_unified_gpucull</i>	2.93	14.55	30.90	56.51

em delta, onde apenas as informações diferentes do quadro anterior são transferidas. No entanto, grande parte da perda de desempenho é causada pelo processo de verificação, onde todas as informações do quadro corrente precisam ser comparadas com as do quadro anterior.

Apesar de existir um *overhead* estabelecido pela comunicação e pelo sincronismo, este *overhead* pode ser diluído ao utilizar um conjunto maior de objetos, juntamente com as técnicas mais modernas de renderização discutidas neste trabalho. Este fato pode ser observado, por exemplo, no modo *indexedmdi*, onde para um único modelo o desempenho piorou em 18%, porém para 125 modelos, o impacto foi de apenas 7%.

Outro fato interessante pode ser observado nos modos *indexedmdi_gpucull* e *indexedmdi_unified_gpucull*, onde a renderização indireta é utilizada junto com o *culling* na GPU. No *benchmark* com uma máquina local, estes modos apresentaram desempenho inferior aos modos tradicionais, em função algoritmo do *culling*. No entanto, ao introduzir a rede como elemento intermediário de comunicação, o desempenho das técnicas modernas apresentam resultados extremamente vantajosos.

Além disto, especificamente no modo *indexedmdi_unified_gpucull*, a aplicação apresentou melhora de desempenho no caso distribuído, quando comparado com uma máquina local. Apesar de não ter sido antecipado, este comportamento pode ser explicado através de um grau reduzido de paralelismo, obtido através da execução de tarefas distintas nas duas máquinas do teste.

No caso específico da cena com 125 modelos, o tempo de renderização reduziu quase 8%. Entre os motivos identificados para este grau de paralelismo, podemos mencionar por exemplo, o fato de que quando o algoritmo de *culling* na GPU está habilitado, o mesmo faz uso de uma barreira de memória. Esta barreira na GPU existe para garantir que a

Figura 26: Taxa de transferência na rede.

solicitação de renderização seja executada apenas após a conclusão do algoritmo de *culling*. No entanto, ao executar a aplicação em um sistema distribuído, a barreira introduzida pelo algoritmo afeta apenas a máquina responsável pela exibição, garantindo que a máquina da aplicação seja liberada para executar outras instruções. Este fato promove certo grau de paralelismo, criando oportunidade para um ganho marginal de desempenho.

Novamente, as vantagens das técnicas mais modernas de renderização são evidenciadas em cenas mais complexas. Considerando o caso com 64 modelos, por exemplo, um modo como o *ubosub_cpucull* funciona com uma taxa inferior a 6 quadros/segundo. O modo *uborange_cpucull* proporciona uma melhora considerável, operando à 14 quadros/segundo. No entanto, ambos com o desempenho aquém do considerado satisfatório. O modo *indexedmdi_unified_gpucull*, por sua vez, proporciona um desempenho bem melhor, a 32 quadros/segundo.

A figura 26 apresenta a taxa de transferência utilizada na rede por cada modo de renderização. Em função das otimizações aplicadas pelo *driver*, principalmente devido à codificação em delta, pouca informação é de fato transferida.

Conforme pode ser observado na figura, apenas os modos de renderização *ubosub_cpucull* e *uborange_cpucull* apresentam taxa de transferência com valores significativos. Isto ocorre pois estes são os únicos modos que utilizam *culling* na CPU. Ao utilizar este algoritmo de *culling*, as chamadas do OpenGL, enviadas através da rede, sofrem mudanças constantes em função de alterações na posição da câmera e, conseqüentemente, no conjunto de objetos visíveis. Estas alterações prejudicam o recurso de codificação em

delta praticado pelo *driver*, o que aumenta consideravelmente as informações enviadas na rede. Mesmo assim, a taxa de transferência é relativamente pequena, sempre inferior a 10 MB/s.

Os modos que não possuem *frustum culling* habilitado sempre renderizam todos os objetos da cena, independente da posição da câmera. Embora esta abordagem deteriore o desempenho gráfico da aplicação, a mesma garante certa uniformidade nas chamadas de função OpenGL entre quadros consecutivos, o que facilita o recurso de codificação em delta e, conseqüentemente, reduz a taxa de transferência nestes modos. Nas outras alternativas, que utilizam *culling* na GPU, a justificativa é semelhante - o algoritmo de *culling* não altera de forma significativa as chamadas OpenGL efetuadas pela aplicação, uma vez que o algoritmo é executado diretamente na GPU. Sendo assim, esta última abordagem também facilita o recurso de codificação em delta.

Para melhor investigar a influência da latência da rede sobre o desempenho da aplicação, selecionou-se a alternativa com maior taxa de transferência média: o modo *ubosub_cpucull* com 27 modelos na cena. Nesta situação, a aplicação possui taxa de transferência inferior a 8 MB/s. Para aferir a latência, o comando *ping* foi utilizado para realizar medidas de RTT (*round-trip time*) na mesma infraestrutura onde são executados os testes gráficos. Este comando foi utilizado transferindo pacotes de 64 KB em intervalos de 8 ms. Estes parâmetros aproximam o perfil de uso da rede efetuado pela aplicação. Neste cenário, o valor médio obtido para o RTT foi de 2 ms e desvio padrão de 0.4 ms.

Sendo assim, pode-se concluir que grande parte do *overhead* da aplicação ocorre na interceptação e no processamento das chamadas OpenGL, dentro do *driver* adaptado. Ao reduzir o número destas chamadas utilizando um mecanismo moderno de renderização, por exemplo, o impacto no desempenho da aplicação é reduzido de forma significativa, podendo até promover um ganho de desempenho, como observado na alternativa *indexed-mdi_unified_gpucull*.

6.4.2 Três máquinas remotas (*Powerwall*)

Este teste faz uso de 3 máquinas remotas, especificamente para exibição, conectadas à 6 projetores. Com a mudança da geometria da tela, o FOV (*Field of view*) vertical foi ajustado para 90°, e o aspecto (*aspect ratio*) para 1 (tela quadrada). Com o aumento vertical do *frustum*, o número de objetos visíveis aumentou nos casos envolvendo cenas mais complexas.

É importante ressaltar que para ambos os modos *ubosub_cpucull* e *uborange_cpucull*, o algoritmo de *culling* é executado na CPU e, portanto, utilizará o *frustum* de visualização original, antes de ser corrigido pelo *driver*. Desta forma, para que não haja inconsistências na imagem apresentada, como por exemplo, a exclusão indevida de determinados objetos, é importante configurar a aplicação para que o seu *frustum* seja igual ao do sistema de projeção.

Tabela 6: Número de chamadas de renderização por quadro (3 máquinas remotas).

Modos de renderização	Qtd. de modelos			
	1	27	64	125
<i>(ubosub/uborange)_cpucull</i>	11.103	110.991	274.879	354.535

Mesmo em aplicações com código fonte não disponível, diversas ferramentas oferecem opções para configurar o FOV vertical. Já o aspecto da imagem, normalmente, pode ser alterado através do redimensionamento da janela da aplicação, para que a mesma fique com aspecto (largura/altura) igual ao do sistema de projeção. Estas alterações foram realizadas neste teste, de forma a garantir que a aplicação operasse com *frustum* compatível com o arranjo dos projetores do sistema, o que é fundamental para que as técnicas que operam com *culling* na CPU produzam resultados coerentes.

A tabela 6 apresenta o número de chamadas de renderização realizadas em cada quadro. Com exceção dos modos *ubosub_cpucull* e *uborange_cpucull*, todos os outros valores são iguais ao do sistema local (tabela 3). Isto porque os demais modos não utilizam *culling* na CPU e, portanto, o número de chamadas de desenho será sempre o mesmo, independente da geometria do *frustum*, variando apenas com a complexidade da cena. Assim como a tabela de renderização local, estes valores também foram obtidos com a câmera na posição inicial, no centro dos modelos.

Esta tabela, no entanto, nos permite verificar o aumento do número de modelos renderizados, em função do aumento do FOV vertical de 45° para 90°. Sendo assim, a infraestrutura de *powerwall* nos permite visualizar um número $\approx 150\%$ maior de geometrias em média, utilizando 3 vezes mais resolução.

A figura 27 apresenta os tempos médios de renderização para este sistema. Os valores utilizados no gráfico estão na tabela 7, incluindo o caso com um único modelo. Apesar do uso do protocolo *multicast* em grande parte das transferências realizadas na rede, o sincronismo ocorre de forma individual, através do protocolo TCP. Este fato implica em uma queda de desempenho, como pode ser observado em todas as medidas obtidas. Este

Figura 27: Tempo médio de renderização (3 máquinas remotas).

overhead apresenta impacto maior nas cenas de menor complexidade, pois nestes casos, o número de sincronizações por segundo é maior.

No entanto, é possível observar que o impacto mais severo ocorre no desempenho dos modos *ubosub_cpucull* e *uborange_cpucull*, pois uma vez ajustado o FOV vertical da aplicação, a quantidade de geometrias renderizadas aumenta, o que afeta principalmente estas alternativas, onde o algoritmo de *culling* é executado na CPU. Já nos casos em que o algoritmo é executado na GPU, o desempenho não é afetado de forma tão representativa, pois as tarefas deste algoritmo são divididas de maneira razoavelmente uniforme nas 3 GPUs do aglomerado, uma em cada computador.

Tabela 7: Tempo médio de renderização com 3 máquinas remotas (milissegundos).

Modos de renderização	Qtd. de modelos			
	1	27	64	125
<i>ubosub</i>	5.91	126.61	290.45	618.27
<i>ubosub_cpucull</i>	4.71	75.03	189.46	342.49
<i>uborange</i>	6.25	89.94	212.41	429.14
<i>uborange_cpucull</i>	3.27	25.26	67.10	118.49
<i>indexedmdi</i>	3.71	25.49	56.52	109.42
<i>indexedmdi_gpucull</i>	3.57	17.09	37.21	71.50
<i>indexedmdi_unified</i>	3.70	24.15	52.42	98.57
<i>indexedmdi_unified_gpucull</i>	3.57	15.59	32.59	59.94

Outra observação importante é o comportamento das técnicas mais modernas de renderização. Mesmo com a mudança de infraestrutura, o que causou o aumento do número de objetos visíveis e o aumento da resolução, o impacto no desempenho destes modos foi

Figura 28: Tempo médio de renderização (9 máquinas remotas).

muito pequeno. Em especial, o modo *indexedmdi_unified_gpucull* manteve os seus resultados ainda melhores dos que os obtidos com uma única local, mesmo renderizando em 3 máquinas remotas.

6.4.3 Nove máquinas remotas (*Caverna*)

Neste teste foram utilizadas 9 máquinas remotas com um total de 18 projetores. Esta infraestrutura é dividida em três telas iguais: uma frontal e duas laterais. Cada tela deste sistema possui, portanto, uma configuração igual à utilizada no teste anterior (3 máquinas com 6 projetores).

As telas laterais são paralelas entre si e formam um ângulo de 90° com a tela da frente. Neste caso, as técnicas de *culling* em CPU, praticadas frequentemente pelas aplicações, não podem ser adaptadas através de uma simples alteração na configuração do FOV ou redimensionamento de janela. Sendo assim, ao utilizar um *driver* adaptado em aplicações com este tipo de *culling*, as telas laterais ficarão sem objetos, uma vez que estas geometrias serão excluídas pelo algoritmo antes de serem enviadas ao *driver*.

A figura 28 apresenta os tempos médios de renderização para este sistema, juntamente com os seus respectivos valores na tabela 8. Os resultados dos modos *ubosub_cpucull* e *uborange_cpucull* (representados de forma tachada) são apresentados apenas como referência, uma vez que o código do *culling* praticado nestes modos foi modificado para manter a compatibilidade com esta infraestrutura e, portanto, não representam soluções viabilizadas através do *driver*, como proposto neste trabalho.

Embora o código do *culling* na CPU tenha sido alterado para considerar as telas laterais, este recurso não representa o que é normalmente praticado nestas circunstâncias. Como, frequentemente, o código da aplicação não está disponível, a abordagem normalmente adotada é desabilitar o *culling* por completo. Desta forma, com exceção dos modos *ubosub_cpucull* e *uborange_cpucull*, as demais alternativas permaneceram intactas, pois não possuem *culling* ou trabalham com este algoritmo na GPU, o que por sua vez já considera a geometria do *frustum* correta para cada caso, uma vez que estas informações são corrigidas de forma apropriada pelo *driver*.

Apesar destas diferenças, os valores estão semelhantes aos obtidos nas configurações anteriores, porém com um *overhead* adicional. Estes últimos resultados demonstram que o aumento do número de computadores utilizados na exibição apresenta impacto severo nas técnicas convencionais. Em contrapartida, os modos de renderização mais modernos, que utilizam o mecanismo de renderização indireta, apresentam melhor escalabilidade com o aumento da infraestrutura.

Este fato é justificado pelo aumento no número de objetos visíveis e pelo desvio das amostras em relação ao tempo médio de renderização. O número de objetos visíveis afeta principalmente o desempenho dos modos com *culling* na CPU, uma vez que o número de verificações deste algoritmo sobrecarrega o processador.

Tabela 8: Tempo médio de renderização com 9 máquinas remotas (milissegundos).

Modos de renderização	Qtd. de modelos			
	1	27	64	125
<i>ubosub</i>	7.07	123.95	336.61	643.86
<i>ubosub_cpucull</i>	8.31	223.86	535.60	816.71
<i>uborange</i>	7.28	94.39	222.08	440.86
<i>uborange_cpucull</i>	5.11	77.85	180.04	294.07
<i>indexedmdi</i>	5.15	25.95	57.11	108.90
<i>indexedmdi_gpucull</i>	5.12	18.07	40.97	78.09
<i>indexedmdi_unified</i>	5.13	24.62	53.11	98.82
<i>indexedmdi_unified_gpucull</i>	5.09	16.66	36.59	67.00

Além disto, observa-se que o desvio das amostras em torno do valor médio é significativamente maior para os modos convencionais, independente do número de computadores envolvidos. Os modos modernos apresentam um desvio menor, pois maximizam o uso da GPU, que é um recurso de *hardware* dedicado para a aplicação, ao contrário da CPU, compartilhada com outros processos em uso na máquina. O uso exclusivo da GPU produz resultados mais estáveis, com pouca variação em torno do tempo médio.

Este desvio das amostras se torna relevante no momento do sincronismo. Antes de exibir o quadro renderizado, o *driver* precisa aguardar que todas as máquinas finalizem o processo de renderização. Sendo assim, a sincronização implica que o tempo de renderização aferido em uma determinada infraestrutura será sempre equivalente ao tempo da máquina mais lenta naquela infraestrutura.

Desta forma, ao aumentar o número de computadores, o tempo médio de renderização será invariavelmente maior, porém de forma proporcional ao desvio das amostras. Estatisticamente, quanto maior o número de máquinas, maior será a chance de pelo menos uma destas apresentar tempo de renderização próximo da faixa superior do desvio, atrasando o restante da infraestrutura.

Tabela 9: Número de chamadas de renderização por quadro (9 máquinas remotas).

Modo de renderização	Qtd. de modelos			
	1	27	64	125
<i>(ubosub/uborange)_cpucull</i>	11.103	244.266	555.458	966.421

A tabela 9 apresenta o número de chamadas de renderização. Novamente, esta tabela apresenta apenas os modos com *culling* na CPU, pois apenas estas alternativas apresentam valores diferentes dos demais testes. Como todos os modos apresentam a mesma cena, estes números servem como indicadores do total de geometrias visualizadas pelo usuário. Como esperado, estes números foram obtidos com a câmera em repouso na origem da cena, e os valores são consideravelmente maiores que o do teste anterior, uma vez que as vistas laterais também foram incluídas.

A taxa de transferência na rede não foi ilustrada nestes últimos dois testes, pois como observado inicialmente, este aspecto apresenta pouco impacto no desempenho final da aplicação.

6.4.4 Uma máquina local com especificações atualizadas

Para servir de referência para os *benchmarks* realizados neste capítulo, utilizamos a aplicação em uma máquina local, com o *driver* OpenGL original e, portanto, sem transferir os comandos pela rede. Os resultados deste *benchmark* de referência foram apresentados no início deste capítulo, sendo que a máquina utilizada neste caso possuía exatamente as mesmas configurações das máquinas usadas nos testes com sistemas imersivos, envolvendo computadores remotos. No entanto, por restrições orçamentárias, a infraestrutura utilizada nestes testes não corresponde com o estado da arte disponível no mercado. Por

Figura 29: Tempo médio de renderização local (máquina atualizada).

exemplo, a placa gráfica instalada no aglomerado de computadores é uma Nvidia Quadro 5000, lançada em 2011.

Para apresentar uma medida de referência em uma infraestrutura moderna. Os mesmos *benchmarks* foram executados em uma única máquina local (sem transferir informações pela rede), porém com especificações atualizadas. O computador utilizado neste teste possui um processador Intel Core i7-6700K, com 32 GB de memória RAM e uma placa gráfica da Nvidia modelo GTX 1080, com 8 GB de memória dedicada. Esta placa gráfica, em contrapartida, foi lançada em Maio de 2016.

Tabela 10: Tempo médio de renderização local (máquina atualizada, milissegundos).

Modos de renderização	Qtd. de modelos			
	1	27	64	125
<i>ubosub</i>	0.85	27.43	64.38	127.68
<i>ubosub_cpucull</i>	0.37	6.39	16.19	31.24
<i>uborange</i>	0.77	19.52	48.32	92.31
<i>uborange_cpucull</i>	0.36	5.50	13.79	26.59
<i>indexedmdi</i>	0.41	4.95	11.43	22.05
<i>indexedmdi_gpucull</i>	0.40	4.85	11.14	21.53
<i>indexedmdi_unified</i>	0.41	4.94	11.41	22.11
<i>indexedmdi_unified_gpucull</i>	0.40	4.57	10.44	20.17

Os resultados obtidos estão presentes na figura 29 e também na tabela 10. Como esperado, o desempenho de todos os modos apresentou melhora muito significativa, porém é notável que o suporte aos recursos mais modernos do OpenGL foi aprimorado de forma especial nestas últimas gerações de GPU. Um dos fatos que sustenta esta afirmação é que

neste *benchmark*, todos os modos de renderização indireta (*indexedmdi*), com o *culling* habilitado ou não, possuem desempenho superior aos demais.

Conforme apresentado anteriormente, este resultado não é observado com a GPU Nvidia Quadro 5000, onde o desempenho do modo *uborange_cpucull* é superior, em uma máquina local. Este último fato se deve ao algoritmo de *culling* praticado na CPU, que apesar de consumir mais banda de comunicação, é mais eficiente do o praticado pela GPU, em função da sua abordagem hierárquica. No entanto, ao utilizar uma GPU moderna, a vantagem da abordagem hierárquica não é mais notável.

Sendo assim, as técnicas apresentadas neste trabalho devem apresentar resultados ainda superiores em GPU's mais modernas. O que reforça o fato de que ao utilizar esta infraestrutura de maneira adequada, é possível viabilizar uma solução de comunicação que mantenha desempenho e escalabilidade.

6.5 Síntese

Este capítulo apresentou os resultados obtidos com a aplicação gráfica executando nas seguintes infraestruturas: uma máquina local (com o *driver* original), uma máquina remota, três máquinas remotas (*powerwall*) e nove máquinas remotas (*caverna*), todas com o mesmo modelo de GPU (lançado em 2011), sendo que nos sistemas com computadores remotos, o *driver* ClusterGL foi utilizado, viabilizando o uso destas infraestruturas de forma transparente. Além disto, um outro teste foi conduzido em uma máquina local, porém com uma GPU moderna (lançada em 2016), apenas como referência.

A aplicação também foi testada com 4 modos principais de renderização: *ubosub*, *uborange*, *indexedmdi* e *indexedmdi_unified*, todos com *culling* habilitado e desabilitado. Estes modos foram utilizados com cenas de complexidade diferente, variando entre 1, 27, 64 e 125 instâncias de um modelo CAD da Nvidia, contendo 218 mil vértices.

Os valores obtidos evidenciam o ganho de desempenho obtido através dos modos que utilizam as técnicas mais modernas de renderização. A vantagem proporcionada por estes modos, apesar de pequena em uma máquina local, ganha proporções representativas após a substituição do *driver* e o uso de máquinas remotas na rede.

O modo *indexedmdi* e suas variações foram as alternativas que apresentaram desempenho equivalente em todas as infraestruturas remotas, promovendo a escalabilidade do sistema de visualização, ao contrário do *overhead* perceptível introduzido nas demais alternativas. Além disto, o *indexedmdi* viabiliza o uso do algoritmo de *culling* na GPU,

sendo a única alternativa prática para sistemas com telas de geometria mais complexa, como a representada pelas 9 máquinas da *Caverna*.

7 CONCLUSÃO

Este capítulo apresenta as considerações finais sobre os resultados obtidos no capítulo 6 e uma breve discussão sobre os objetivos alcançados, outras possíveis abordagens e trabalhos futuros.

Conforme apresentado no capítulo 2, a evolução dos sistemas imersivos procura aprimorar continuamente a qualidade das imagens exibidas. No caso dos sistemas de grande porte, com uma infraestrutura distribuída, apesar dos avanços individuais promovidos na capacidade de processamento e resolução dos equipamentos, é comum encontrar sistemas que adotam um número cada vez maior de computadores e telas (monitores ou projetores). De certa forma, estas instalações apresentam trajetória semelhante ao universo de dispositivos móveis e computadores pessoais, buscando resoluções e densidades de pixel compatíveis com a acuidade do olho humano.

Apesar de sistemas mais simples, contendo apenas um óculos de RV, apresentarem resultados extremamente favoráveis para a grande maioria das aplicações, existem casos de uso com requisitos específicos, cujas soluções mais simples não podem atender. Entre eles, podemos mencionar simuladores e aplicações de treinamento com espaço compartilhado entre diversos usuários, e também aplicações que necessitam da interação com instrumentos e periféricos reais.

Nesta classe de aplicações, o uso de infraestruturas distribuídas é prejudicado em função da sua complexidade. Conforme as soluções estudadas neste trabalho, diversas ferramentas de *software* já foram propostas e desenvolvidas com o objetivo de resolver ou amenizar estes problemas. No entanto, a falta de uniformidade nestas soluções não promove a interoperabilidade e dificulta o uso deste tipo de infraestrutura, principalmente por profissionais ou acadêmicos de outras áreas do conhecimento.

Por conta disto, este trabalho apresentou uma arquitetura para viabilizar a comunicação entre uma determinada aplicação e a sua respectiva infraestrutura de exibição. Esta arquitetura é baseada em conceitos já discutidos na literatura, e utiliza a substituição do

driver OpenGL como meio para interceptar, adaptar e redistribuir as suas respectivas funções para a infraestrutura desejada.

Esta abordagem, embora já discutida, possui problemas de desempenho que agravam a sua adoção, principalmente em aplicações com cenas de maior complexidade geométrica. Este trabalho, portanto, buscou explorar as alternativas existentes nas versões mais recentes do padrão OpenGL para contornar estes problemas, apresentando soluções e alternativas para viabilizar o uso desta técnica de distribuição, mesmo com cenas maiores.

Para verificar a contribuição das técnicas propostas e avaliar o seu impacto no desempenho final de uma determinada aplicação, o projeto adotou algumas ferramentas de código aberto, e introduziu as modificações necessárias para conduzir os testes desejados, procurando um ambiente que representasse de maneira apropriada os cenários encontrados em casos de uso reais.

Os resultados obtidos em diferentes infraestruturas distribuídas apontam que as técnicas propostas contribuem de forma muito significativa no desempenho da aplicação, quando comparadas com as técnicas tradicionais de renderização. Os valores indicam que as alternativas que reduzem a quantidade de alterações nos estados do OpenGL e, conseqüentemente, reduzem o número de solicitações para a GPU, podem se beneficiar de ganhos consideráveis de desempenho.

Além disto, entre as alternativas propostas, a implementação do algoritmo de *culling* na própria placa gráfica se mostrou como uma ferramenta importante para viabilizar o uso desta otimização em sistema imersivos, principalmente naqueles com geometria complexa, cujo *frustum* não pode ser configurado apropriadamente na aplicação.

Outro aspecto que pode ser observado através dos resultados é que os modos de renderização mais modernos, que promovem o uso mais intenso da GPU, apresentam não apenas os melhores resultados, mas também o menor desvio padrão nos tempos médios. Isto ocorre porque a GPU é usada exclusivamente para a aplicação em teste, o que representa o cenário mais comum. Sendo assim, este fato proporciona pouca variação em torno do valor médio no tempo de renderização.

Esta estabilidade se mostrou fundamental ao considerar o aspecto do sincronismo entre as diversas máquinas da infraestrutura. Quanto maior o número de computadores na infraestrutura de exibição, maior a probabilidade de uma destas máquinas apresentar um tempo de renderização compatível com a faixa superior do desvio padrão. Mesmo que uma máquina específica não seja consistentemente mais lenta, o que poderia ser corrigido, sempre haverá uma máquina mais lenta em cada quadro - cujo atraso será proporcional

ao desvio padrão do tempo médio.

Ao utilizar um modo de renderização mais moderno, os valores obtidos para o desvio padrão nestes casos é praticamente nulo, imperceptível nos gráficos apresentados. Desta forma, todas as máquinas da infraestrutura apresentam tempos de renderização muito semelhantes, de forma consistente. Este comportamento reduz a sobrecarga do sincronismo, promovendo a escalabilidade da aplicação em termos da infraestrutura utilizada.

Esta observação pode ser confirmada verificando os tempos médios de renderização da aplicação, ao utilizar variações do modo *indexedmdl*, em diferentes infraestruturas. Nestes casos, os tempos médios são muito semelhantes, principalmente nas cenas mais complexas, onde o *overhead* da rede é pequeno quando comparado com o tempo de renderização. Embora o número máximo de computadores testados tenha sido nove, os valores devem permanecer estáveis com um número maior de máquinas até que seja necessário realizar uma mudança na topologia da rede, como o acréscimo de um novo switch, por exemplo.

Este benefício da escalabilidade permite que uma determinada aplicação mantenha o seu desempenho em uma diversidade ampla de infraestruturas. Quando comparado com a execução em uma máquina local, a única diferença notável seria um *overhead* com valor proporcional apenas à complexidade da cena visualizada, independente do número de computadores.

Desta forma, os resultados obtidos ilustram um benefício claro que algumas das técnicas apresentadas podem causar sobre o desempenho da aplicação. Estes resultados confirmam a hipótese inicial e a expectativa do trabalho, de que é possível viabilizar uma interface de comunicação de baixo nível, entre a aplicação e um determinado sistema imersivo, mantendo a interoperabilidade da ferramenta e o seu respectivo desempenho, independente da infraestrutura deste sistema.

7.1 Desafios pendentes

A abordagem utilizada neste trabalho, no entanto, ainda possui incompatibilidades com um certo conjunto de aplicações, uma vez que as mesmas utilizam recursos do padrão OpenGL que não podem ser adaptados de forma apropriada pelo *driver*, sem que haja alguma intervenção no código da ferramenta.

Além disto, conforme discutido no capítulo 5, este fato foi agravado nas últimas versões do padrão. A flexibilidade introduzida pelo OpenGL, com o propósito de viabilizar a customização e a otimização do *pipeline*, prejudicou a identificação e a adaptação dos

comandos emitidos pela aplicação, o que é fundamental para viabilizar o seu uso em sistemas imersivos.

Sendo assim, ficou evidente neste trabalho que o mecanismo utilizado pelo *driver* para adaptar as chamadas do OpenGL deve ser diferente para cada conjunto de aplicações. Ou seja, cada aplicação necessita que funções distintas do padrão sejam adaptadas de formas diferentes.

Este último agravante é consequência do fato que o OpenGL não foi originalmente planejado para utilizar sistemas distribuídos. Desta forma, as limitações presentes na atual API do OpenGL exigem que as aplicações sejam corrigidas pelo *driver*, o que caracteriza um desafio pendente nesta arquitetura. No entanto, os resultados apresentados demonstram a viabilidade técnica e tecnológica para implementar a interface de comunicação proposta, ao mesmo tempo que aponta os problemas pendentes na API deste padrão. Estes resultados visam auxiliar estudos futuros na investigação de abordagens para resolver ou amenizar estas circunstâncias.

7.2 Outras abordagens e trabalhos futuros

De maneira geral, as incompatibilidades mencionadas são provenientes do fato que o OpenGL não possui meios para enumerar o conjunto de placas gráficas disponíveis, nem determinar a geometria da tela que está conectada em cada placa. Estas duas informações, juntamente com a posição relativa das telas, são fundamentais para a criação de aplicações para sistemas imersivos.

De posse destas informações, cada aplicação poderia determinar como usar os recursos disponíveis, configurando o *frustum* e as matrizes de forma adequada para cada parte da infraestrutura. Nesta abordagem, a aplicação assume a responsabilidade de usar ou não os elementos disponíveis na infraestrutura, não exigindo que o *driver* tenha que fazer adaptações neste sentido. Embora, em tese, estas informações possam ser obtidas através de outras API's, a sua ausência do OpenGL não garante a interoperabilidade das aplicações.

Além do mecanismo necessário para obter as informações da infraestrutura, o OpenGL também precisaria de recursos para produzir imagens com múltiplos pontos de vista, de uma mesma cena, de forma eficiente. Atualmente, o *pipeline* de renderização do OpenGL só consegue projetar a geometria da cena em um único plano. Em tese, para utilizar um sistema de exibição com múltiplos planos, como uma Caverna, a aplicação teria que submeter a cena para ser renderizada múltiplas vezes, o que deterioraria substancialmente

o seu desempenho.

Atualmente, a única opção disponível para renderizar mais de uma projeção, simultaneamente, é utilizando a extensão *NVX_linked_gpu_multicast*¹, específica da Nvidia. Esta extensão permite que um computador com diversas placas gráficas possa solicitar que cada placa renderize uma projeção diferente, ao mesmo tempo. No entanto, esta extensão não está amplamente disponível e é limitada à um único computador.

Estes são alguns dos problemas fundamentais relacionados ao uso do OpenGL em sistemas imersivos com infraestrutura distribuída. Apesar deste trabalho não ter investigado o Vulkan - versão do padrão que sucederá o OpenGL nos próximos anos, este não apresenta soluções para as questões expostas neste documento. Trabalhos futuros podem sugerir outras opções e alternativas para viabilizar o uso de uma interface bem definida, performática e interoperável, porém se beneficiando dos aspectos já identificados e discutidos neste trabalho.

¹https://www.opengl.org/registry/specs/NVX/nvx_linked_gpu_multicast.txt. Último acesso: 5 de outubro de 2016.

REFERÊNCIAS

- ALLARD, J. et al. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In: *Proceedings of the IEEE Virtual Reality Conference 2002*. Washington, DC, USA: IEEE Computer Society, 2002. (VR '02), p. 273–. ISBN 0-7695-1492-8. Disponível em: <<http://dl.acm.org/citation.cfm?id=580130.835869>>.
- ALLARD, J.; RAFFIN, B. A Shader-Based Parallel Rendering Framework. In: *IEEE Visualization Conference*. Minneapolis, USA: [s.n.], 2005.
- ARCHDEACON, J.; IWAI, N.; SWEET, B. Designing and Developing an Image Generator for the Operational Based Vision Assessment Simulator. In: *AIAA. Proceedings of AIAA Modeling and Simulation Technologies Conference*. [S.l.]: AIAA, 2012.
- ARCILA, T. et al. FlowVR: A Framework for Distributed Virtual Reality Applications. In: *1iAre journA(C)es de l'Association FranAaise de RA(C)alitA(C) Virtuelle, AugmentA(C)e, Mixte et d'Interaction 3D*. Rocquencourt, France: [s.n.], 2006.
- BHANIRAMKA, P.; ROBERT, P.; EILEMANN, S. OpenGL multipipe SDK: a toolkit for scalable parallel rendering. In: *Visualization, 2005. VIS 05. IEEE*. [S.l.: s.n.], 2005. p. 119–126.
- BIERBAUM, A. et al. Implementing Immersive Clustering with VR Juggler. In: *Proceedings of the 2005 International Conference on Computational Science and Its Applications - Volume Part III*. Berlin, Heidelberg: Springer-Verlag, 2005. (ICCSA '05), p. 1119–1128. ISBN 3-540-25862-0, 978-3-540-25862-9. Disponível em: <http://dx.doi.org/10.1007/11424857_120>.
- BRUZZONE, A. et al. Virtual simulation for training in ports environments. In: *Proceedings of the 2011 Summer Computer Simulation Conference*. Vista, CA: Society for Modeling & Simulation International, 2011. (SCSC '11), p. 235–242. ISBN 978-1-61782-950-5. Disponível em: <<http://dl.acm.org/citation.cfm?id=2348196.2348228>>.
- CRUZ-NEIRA, C.; SANDIN, D. J.; DEFANTI, T. A. Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE. In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1993. (SIGGRAPH '93), p. 135–142. ISBN 0-89791-601-8. Disponível em: <<http://doi.acm.org/10.1145/166117.166134>>.
- CRUZ-NEIRA, C. et al. The CAVE: Audio Visual Experience Automatic Virtual Environment. *Commun. ACM*, ACM, New York, NY, USA, v. 35, n. 6, p. 64–72, jun. 1992. ISSN 0001-0782. Disponível em: <<http://doi.acm.org/10.1145/129888.129892>>.
- DEFANTI, T. et al. The future of the CAVE. *Central European Journal of Engineering*, SP Versita, v. 1, n. 1, p. 16–37, 2011. ISSN 1896-1541. Disponível em: <<http://dx.doi.org/10.2478/s13531-010-0002-5>>.

DEFANTI, T. A. et al. The StarCAVE, a Third-generation CAVE and Virtual Reality OptiPortal. *Future Gener. Comput. Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 25, n. 2, p. 169–178, fev. 2009. ISSN 0167-739X. Disponível em: <<http://dx.doi.org/10.1016/j.future.2008.07.015>>.

DEFANTI, T. A. et al. The OptiPortal, a Scalable Visualization, Storage, and Computing Interface Device for the OptiPuter. *Future Gener. Comput. Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 25, n. 2, p. 114–123, fev. 2009. ISSN 0167-739X. Disponível em: <<http://dx.doi.org/10.1016/j.future.2008.06.016>>.

DOERR, K.-U.; KUESTER, F. CGLX: A Scalable, High-Performance Visualization Framework for Networked Display Environments. *IEEE Transactions on Visualization and Computer Graphics*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 17, n. 3, p. 320–332, maio 2011. ISSN 1077-2626. Disponível em: <<http://dx.doi.org/10.1109/TVCG.2010.59>>.

EILEMANN, S.; MAKHINYA, M.; PAJAROLA, R. Equalizer: A Scalable Parallel Rendering Framework. *Visualization and Computer Graphics, IEEE Transactions on*, v. 15, n. 3, p. 436–452, 2009. ISSN 1077-2626.

EVERITT, C.; MCDONALD, J. *Beyond Porting: How Modern OpenGL Can Radically Reduce Driver Overhead*. Seattle, WA, USA: Steam Dev Days, janeiro 2014. <https://www.youtube.com/watch?v=-bCeNzgiJ8I>. Acessado em 1 de novembro de 2014.

EVERITT, C. et al. *Approaching Zero Driver Overhead*. San Francisco, CA, USA: Game Developers Conference, março 2014. <http://gdcvault.com/play/1020791/>. (GDC '14). Acessado em 1 de novembro de 2014.

EVL, E. V. L. *Bright Advanced Technology (BAT) CAVE*. 2009. <http://www.evl.uic.edu/core.php?mod=4&type=1&indi=161>. Acessado em 1 de novembro de 2014.

GARCÍA-FERNÁNDEZ, I. et al. New developments in simulation-based harbour crane training. *International Journal of Simulation and Process Modelling*, v. 6, n. 4, jan 2011. Disponível em: <10.1504/IJSPM.2011.048009>.

HUMPHREYS, G. et al. Chromium: A Stream-processing Framework for Interactive Rendering on Clusters. In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 2002. (SIGGRAPH '02), p. 693–702. ISBN 1-58113-521-1. Disponível em: <<http://doi.acm.org/10.1145/566570.566639>>.

ILMONEN, T.; REUNANEN, M.; KONTIO, P. Broadcast GL: An alternative method for distributing opengl api calls to multiple rendering slaves. In: *In WSCG'2005: The Journal of WSCG, volume 13, Plzen, Czech Republic*. [S.l.]: Science Press, 2005. p. 2005.

KACZMARSKI, H.; ZUFFO, M. K. Commodity clusters for immersive projection environments, course 47. In: *ACM SIGGRAPH 2002 Course Notes*. [S.l.: s.n.], 2002. (SIGGRAPH '02).

MOLNAR, S. et al. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 14, n. 4, p. 23–32, jul. 1994. ISSN 0272-1716. Disponível em: <<http://dx.doi.org/10.1109/38.291528>>.

NEAL, B.; HUNKIN, P.; MCGREGOR, A. Distributed OpenGL Rendering in Network Bandwidth Constrained Environments. In: *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2011. (EG PGV'11), p. 21–29. ISBN 978-3-905674-32-3. Disponível em: <<http://dx.doi.org/10.2312/EGPGV/EGPGV11/021-029>>.

PACHECO, P. S. *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN 1-55860-339-5.

PAPADOPOULOS, C.; PETKOV, K.; KAUFMAN, A. *Building the Reality Deck*. 2013. In POWERWALL: International Workshop on Interactive, Ultra-High-Resolution Displays, part of the SIGCHI Conference on Human Factors in Computing Systems.

RAFFIN, B. et al. PC Clusters for Virtual Reality. In: *Virtual Reality Conference, 2006*. [S.l.: s.n.], 2006. p. 215–222. ISSN 1087-8270.

RAFFIN, B. et al. Commodity clusters for virtual reality. In: *Proceedings of the IEEE Virtual Reality 2003*. Washington, DC, USA: IEEE Computer Society, 2003. (VR '03), p. 307–. ISBN 0-7695-1882-6. Disponível em: <<http://dl.acm.org/citation.cfm?id=832289.835985>>.

RODRIGUES, F. L. D. *Sistema de Realidade Virtual para Simulador Visual de Passadiço*. Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo, 2010.

ROTH, M.; VOSS, G.; REINERS, D. Multi-threading and clustering for scene graph systems. *Computers & Graphics*, v. 28, n. 1, p. 63 – 66, 2004. ISSN 0097-8493. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0097849303002310>>.

SCHAEFFER, B.; GOUDESEUNE, C. Syzygy: Native PC Cluster VR. In: *Proceedings of the IEEE Virtual Reality 2003*. Washington, DC, USA: IEEE Computer Society, 2003. (VR '03), p. 15–. ISBN 0-7695-1882-6. Disponível em: <<http://dl.acm.org/citation.cfm?id=832289.835927>>.

SOARES, L. P. et al. Virtual hang-gliding over rio de janeiro. In: *ACM SIGGRAPH 2005 Emerging Technologies*. New York, NY, USA: ACM, 2005. (SIGGRAPH '05). Disponível em: <<http://doi.acm.org/10.1145/1187297.1187327>>.

STAADT, O. G. et al. A Survey and Performance Analysis of Software Platforms for Interactive Cluster-based Multi-screen Rendering. In: *Proceedings of the Workshop on Virtual Environments 2003*. New York, NY, USA: ACM, 2003. (EGVE '03), p. 261–270. ISBN 1-58113-686-2. Disponível em: <<http://doi.acm.org/10.1145/769953.769984>>.

TEUBL, F. et al. FastFusion: A Scalable Multi-projector System. *14th Symposium on Virtual and Augmented Reality*, p. 26–35, Maio 2012.

VERHILLE, V. et al. How to Easily Develop a VR Experience from a 3D Desktop Application Thanks to the TechViz TVZLib API. In: PERRET, J. et al. (Ed.). *EuroVR 2014 - Conference and Exhibition of the European Association of Virtual and Augmented Reality*. [S.l.]: The Eurographics Association, 2014. ISBN 978-3-905674-76-7.

ZIELINSKI, D. J. et al. Enabling closed-source applications for virtual reality via opengl intercept-based techniques. In: _____. *IEEE Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. IEEE, 2014. p. 59–64. Disponível em: <<http://dx.doi.org/10.1109/searis.2014.7152802>>.

ZUFFO, J. A. et al. Caverna digital - sistema de multiprojeção estereoscópico baseado em aglomerados de pcs para aplicações imersivas em realidade virtual. In: *SBC 4th Symposium on Virtual Reality*. Florianópolis, SC, BR: [s.n.], 2001. (SVR '01).