LVRL: Um Framework para RV

Authors Name/s per 1st Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

Authors Name/s per 2nd Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

***Resumo*—Com o barateamento dos hardwares para ambientes imersivos, o interesse por aplicações de Realidade Virtual (RV) vem aumentando consideravelmente. Entretanto os frameworks atualmente disponíveis exigem que as aplicações do usuário sejam desenvolvidas especificamente para elas. Isso faz com que o custo de se portar uma aplicação legada para ambientes RV seja alto, na maioria das vezes até proibitivo. Este artigo propõe um novo framework, o LVRL (*Lightweight Virtual Reality Libraries*), que permite a criação ou conversão de aplicações existentes para RV sem alteração na estrutura da aplicação. O principal objetivo do LVRL é fornecer uma interface de programação (API, aplication program interface) minimalista e não intrusiva permitindo o desenvolvimento de aplicações RV por desenvolvedores sem conhecimento específico em RV. Este artigo descreve a arquitetura do LVRL,suas funcionalidades, forma de usar e os benefícios obtidos pelas aplicações que o utilizam.**

***Keywords*-component; formatting; style; styling;**

I. Introdução

Nos ultimos anos a RV vem se popularizando devido a redução dos custos de hardware e consequentemente dos sistemas imersivos. No passado tais ambientes eram extremamente caros, chegando a custar alguns milhões de dólares no caso de CAVEs [1]. Com o advento das televisões 3D de alta definicao a preços acessíveis, ambientes imersivos como a NexCAVE [2]. estão ao alcance da maioria das empresas de engenharia, design, jogos e outras que trabalham com 3D no dia a dia. A NexCave utiliza de 9 a 21 televisões afixadas num suporte com a forma de um poliedro de ”n”lados inscrito numa a seção de esfera suficiente para cobrir ao maximo o campo de visão do usuário.

Apesar da redução dos custos do hw a produção de sw para ambientes imersivos ainda é uma atividade custosa. Isto se deve ao pequeno numero de ferramentas de baixo custo para a produção de tais sistemas, limitando dessa forma o numero de aplicações que estejam aptas a executar nesses ambientes. Um outro aspecto que dificulta a criação de novas aplicações é a complexidade para transformar uma aplicação desktop funcional em uma aplicação RV. Além das peculiaridades desse tipo de ambiente, pesa ainda o fato que praticamente todos os SDKs para RV modernos estarem amarrados a algum tipo de renderizador, como é o caso do VrJuggler [3], BlenderCave [4], Eon Studio [5], Avango NG [6], 3DVIA Virtools [7] e INVRS [8]. Por consequência as aplicações imersivas precisão ser inteiramente refeitas para se adequar a uma dessas plataformas.

Hoje em dia já existem muitas aplicações 3D baseadas em renderizadores específicos como os visualizadores de modelos massivos [9], de modelos de elementos finitos [10], de nuvens de pontos [11], de dados sísmicos [12] e de dados visuais resultantes de simuladores científicos em geral. O custo de reescrevê-los dentro de plataformas especificas seria demasiadamente alto e as vezes impossível, dado o nível de especificidade do renderizador ou a plataforma ser fechada como no caso do EON Studio e do Virtools.

No entanto isso não é necessário pois os componentes necessários para a criação de uma aplicação RV podem ser construídos de forma que possam ser utilizados pela aplicação já existente. Uma aplicação imersiva se diferencia de uma aplicação desktop pelo fato de ter de suportar múltiplas saídas de vídeo de pontos de vista diferentes. Ainda, o usuário não pode usar mouse e teclado por estar em pé em frente ao ambiente imersivo, como pode ser visto na figura 1.

Nesse trabalho propomos um framework não intrusivo que possibilita a conversão de aplicações desktop em aplicações imersivas por um desenvolver não especializado em RV. O framework permite que o desenvolvimento da aplicação seja feito exclusivamente em desktop e garante que ao mudar para o modo RV os dados serão visualizados considerando o formato do ambiente imersivo e a navegação terá as a mesmas funcionalidades adaptadas para dispositivos RV.

Na seção II apresentamos as ferramentas existentes hoje e as razões que dificultam a conversão dos software 3D desktop para ambientes imersivos. Em seguida, na seção III explicamos cada um dos componentes que compõem nosso framework e mostramos como cada um deixa transparente para o usuário o fato da aplicação estar sendo executada em ambientes desktop ou imersivos. Na seção IV discutimos as características e contribuições que o nosso framework propõe. Por fim, algumas conclusões e trabalhos futuros swão apresentados na seção V.

II. Trabalhos Relacionados

Os frameworks para RV existentes tentam assumir algumas funções. São elas: renderização de múltiplos pontos de vista, captura de eventos de dispositivos de interação de RV e distribuição da renderização em diferentes maquinas da rede. Como já citado, ao mesmo tempo em que tentam resolver esses problemas, também requerem, em contrapartida, que o desenvolvimento seja orientado para suas plataformas específicas.

BlenderCave [4], Virtools [7] e EON Studio [5] são ferramentas de autoria para desenvolvimento de aplicações 3D interativas. Elas tem um sistema de renderização interno o qual o desenvolvedor da aplicação não tem acesso. Os dados são modelados em ferramentas de terceiros como o 3DStudio Max e importados para dentro da ferramenta. Nesse tipo de software não é possível fazer uma renderização específica, como por exemplo técnicas de renderização baseadas em pontos. Quanto a interação, esses frameworks são capazes de ler os dispositivos de RV, mas o desenvolvedor precisa desenvolver o suporte e o mecanismo de interação necessário para o dispositivo que deseja suportar.

Frameworks programáticos de mais baixo nível, como o VrJuggler [3], Avango NG [6] e INVRS [8] em geral são baseadas em grafos de cena desenvolvidos por terceiros para fazer a renderização. Grafos de cena bastante comuns são o OpenSceneGraph [13], OpenSG [14] e SGI OpenGL Performer [15]. Caso a aplicação a ser convertida já use um desses grafos de cena, provavelmente a parte do render será mais facilmente convertida para ambientes imersivos usando esses frameworks, uma vez que eles já resolvem, ou podem ser facilmente adaptados para resolver, questões como a renderização de múltiplos pontos de vista e a distribuição do rendering por diferentes maquinas da rede. No entanto, em geral, é necessário se manter duas versões do software, uma desktop e outra para ambientes imersivos. Isso acontece pois essas componentes precisam controlar o laço principal de controle da aplicação para ser possível fazer o desenho sincronizado em todas as telas. Já com relação a interação 3D elas tem o mesmo problema das ferramentas de autoria, isto é, elas passam o dado cru dos dispositivos e o desenvolvedor é o responsável por tratar cada tipo de dispositivo para o qual queira usar na aplicação.

A Cavelib [16] foi a primeira componente para ambientes imersivos e foi criada junto com a primeira CAVE. Ela até hoje é mantida e vendida. A única diferença dela para componentes programáticas citadas a cima é que ela não está amarrada a um grafo de cena. Sua estratégia é assumir que toda a informação dinâmica de desenho precisa estar numa área de memória protegida. Essa memória é compartilhada e pode ser distribuída entre os computadores. No entanto, mesmo não amarrada a um grafo de cena específico, ela ainda exige que o desenvolvimento seja orientado a ela. Adicionalmente, ela também precisa que os manipuladores

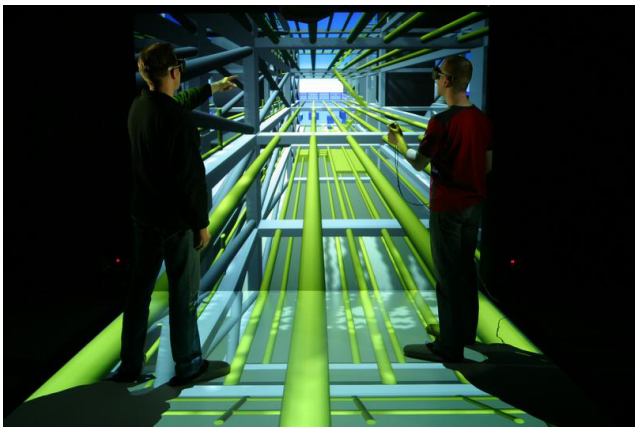

Figura 1. Sistema Imersivo em formato de L. fonte Fraunhofer IAIS

de câmera sejam feitos da mesma forma que as outras componentes, só que usando uma camada de entrada de dispositivos que tem uma interface parecida com o VRPN [17]. Essa amarração com grafos de cena ou estratégias de uso de memória em todas as soluções citadas está intimamente ligada com a distribuição da renderização por várias máquinas. O componente gestor dessa distribuição precisa saber o que distribuir e assim, poder sincronizar o desenho nas diferentes maquinas. No entanto, assim como fornece uma importante funcionalidade, ele adiciona uma restrição muito forte ao framework. Ao mesmo tempo, existem outras soluções de renderização distribuída genéricas [18]–[20], que não tem o cálculo dos múltiplos pontos de vista nativo, mas fornecem a mesma funcionalidade se a aplicação for capaz de calcular internamente os múltiplos pontos de vista.

O framework proposto fornece toda a parte de manipulação de câmera e os cálculos dos múltiplos pontos de vista. Supomos que os métodos de interação de câmera devem convergir a formatos já sedimentados e testados. Imbutindo-se esses componentes também em uma componente, acreditamos que pode-se reduzir o esforço da grande maioria dos desenvolvedores na criação de aplicações que são ao mesmo tempo desktop e imersivas. Sendo assim, ao invés de eventos de dispositivos de entrada, o framework já fornece uma posição e orientação 3d.

III. Framework

O framework provê ferramentas para o programador leigo em algoritmos de RV converter um programa desktop em um programa multimodal que funcione em ambientes desktop e RV apenas trocando a configuração do ambiente em tempo de execução. Para isso foram criados componentes não intrusivas que realizam as seguintes tarefas: auxiliar o sistema de renderização já existente a criar as câmeras virtuais para os múltiplos pontos de vista; fornecer manipuladores de câmera já projetados para funcionar nos dois tipos de ambiente de forma idêntica apenas mudando o tipo de

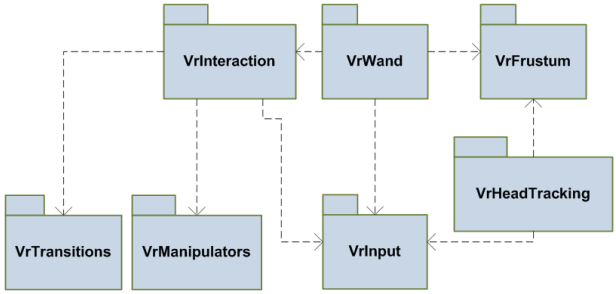

Figura 2. Diagrama de componentes com as dependências entre as componentes

dispositivo utilizado; fornecer uma interface de *Wand*, como uma generalização do mouse para interação 3D. Abaixo segue a descrição da arquitetura geral do framework e cada um dos componentes que o integram.

A. Arquitetura

O framework é formado por 7 componentes. O diagrama de dependência pode ser visto na figura **??**. Três dessas, são componentes matemáticas sem dependência nenhuma, que inclusive podem ser usadas em outros frameworks de RV. São eles o *VrFrustum*, *VrManipulators* e *VrTransitions*. As outras quatro são de mais alto nível e fazem todo o tratamento de eventos e interação para câmera. Eles também podem ser usadas com outros frameworks, mas é necessario usar o componente de leitura de eventos de dispositivos, *VrInput*, ao ínves do fornecido pelos outros frameworks.

O *VrFrustum* é responsável por gerar as câmeras dos diferentes pontos de vistas correspondentes a cada tela do sistema imersivo dada a configuração física das telas e a posição da cabeça do usuário. No modo desktop ela gera uma única câmera correspondente ao monitor.

O *VrManipulator* contêm um conjunto de manipuladores de câmera que foram projetados para funcionar com dados de entrada de mouse e teclado da mesma forma que com dados de entrada de dispositivos de RV. Esses manipuladores também são projetados para funcionar tanto em ambientes imersivos quanto em desktop.

O *VrTransition* é um componente que faz a interpolação entre duas poses de câmera de forma suave para evitar desorientação dentro do ambiente imersivo e em desktop.

Os outros quatro componentes são ligados a captura e interpretação dos dispositivos de entrada. Elas são *VrInput*, *VrInteraction*, *VrHeadTracking* e *VrWand*.

O componente *VrInput* é responsável pelo gerenciamento dos dispositivos de entrada, criando uma camada de abstração para os outros componentes. Todo o gerenciamento dos dispositivos, controle do ciclo de vida dos drivers proprietários e demais dependencias são controlados pelo *VrInput*. O *VrHeadTracking* monitora os eventos do sensor que lê a posição da cabeça do usuário e alimenta o *VrFrunstum* que precisa conhecer essa posição para calcular as múltiplas câmeras. A *VrWand* informa a direção para onde o usuário esta apontando o dispositivo de entrada. No modo RV os dispositivos geralmente fornecem a posição absoluta dentro do ambiente imersivo, assim como a orientação do dispositivo. Usando a componente *VrFrunstum* para transformar a posição do dispositivo de entrada de coordenadas do ambiente imersivo para coordenadas do mundo, a componente é capaz de calcular um vetor direção e uma posição para o raio. Já no modo desktop o processo é semelhante. No entanto o mouse só fornece uma posição 2D do cursor na tela. Para transformar isso num raio calculamos o vetor que sai da posição da câmera no mundo e passa por aquela posição no plano de projeção da câmera (o plano near).

Por fim o componente *VrInteraction* é responsável por interpretar os eventos de entrada e transformá-los em operações de manipulação da câmera. Essa componente fornece uma matriz de view que posiciona e orienta a câmera no mundo. Quando o *Vrinput* gera um evento de um dos dispositivos de entrada suportados pelo *VrIntecation*, esse interpreta o evento segundo um mapeamento e chama o *VrManipulator* para efetuar os cálculos matemáticos que atualizam a matriz de view corrente. Outra responsabilidade do *VrInteraction* é o gerenciamento das transições. Quando o usuário muda de manipulador ativo o *VrInteraction* verifica se existe algum pulo da última pose do manipulador que está sendo desativado para a posição inicial do manipulador que está sendo ativado. A seguir são apresentados maiores detalhes de cada um desses componentes nas próximas seções.

B. VrInput

O *VrInput* é o módulo responsável pelo acesso de dispositivos de entrada como mouse, teclado, joysticks e trackers. Nele, cada dispositivo é mapeado para um *leitor*, que faz o acesso direto ao dispositivo usando sua api nativa. Atualmente, existe suporte para os seguintes dispositivos:

- Teclado e mouse;
- Controle wii;
- Tablet Ipad;
- Celulares e tablets Android;
- Controle Flystick2;
- Tracker BraTrack;
- SpaceBall;
- Joysticks (somente Windows);
- Kinect (somente Windows);

Além dos dispositivos listados também há leitores de acesso às interfaces de entrada analógica, de botões e trackers presentes na componente VRPN [17]. Dessa forma, é possível acessar também dados para os dispositivos suportados por essa componente.

A definição de um *leitor* consiste em implementar um único método, chamado *poll*. Nesse método, é acessado

o estado do dispositivo e são criados *eventos*, os quais serão repassados para a aplicação através do *gerenciador de eventos*.

Um *evento* é uma mensagem que possui um identicador, informando seu tipo, e pode ou não conter dados associados. Por exemplo, o identificador *wii_btn_left_press* indica que a tecla esquerda do direcional do controle wii foi apertada. Nesse caso, não há nenhum dado associado ao evento. Já um evento de tracker contém um vetor e um quatérnio que indicam respectivamente sua posição e orientação.

O uso do *VrInput* é feito através da interface *InputController*, que é responsável por criar os leitores, além de permitir ativá-los ou desativá-los. Para garantir a estabilidade e o controle eficiente do ciclo de vida dos diversos leitores usados pela aplicação todo o gerenciamento fica a cargo do *VrInput*, sendo portanto vedada à aplicação instanciação direta de um leitor.

A criação de um leitor é feita através do método *createReader*. Como parâmetros de entrada devem ser fornecidos um identicador para o leitor, seu tipo e qual o canal de comunicação a ser usado. O identicador é uma string que permite à aplicação ativar ou desativar o leitor depois que esse últmo foi criado, assim como verificar seu status (se o mesmo se encontra ativo ou não).

O canal de comunicação define de que forma o *InputController* se comunicará com o leitor. Atualmente, há dois tipos de canais de comunicação: *bypass* e *ipc*. No primeiro o leitor é criado no mesmo processo da aplicação e a comunicação se dá por meio de chamadas diretas aos métodos do leitor. No segundo, o leitor é criado por um processo secundário e fica fora do contexto de execução da aplicação. A comunicação entre o *InputController* e esse processo secundário se dá através de chamadas entre processos com uso de memória compartilhada. O canal ipc tem a vantagem de isolar os dispositivos da aplicação. Dessa forma, uma falha no acesso ao dispositivo não derruba a aplicação principal, mas apenas o processo secundário do *VrInput*.

Só é permitida uma instância do *VrInput* por aplicação. Isso impede que haja acesso concorrente aos dispositivos entre diferentes instâncias que poderiam ter sido criadas pela aplicação. Por esse motivo, antes de se chamar qualquer método de *InputController*, deve-se chamar o método *initialize*. Ao ser chamado, esse cria a instância do *VrInput* e dispara o processo secundário responsável por se comunicar com os leitores quando o canal de comunicação ipc é usado.

Para receber os eventos gerados por um leitor, a aplicação deve implementar uma classe que extenda *EventObserver* e nessa registrar os eventos que deseja monitorar através do método *listenEvent* da interface *EventManager*. A listagem 1, apresenta um exemplo de implementação de um *EventObserver*. Nele, a aplicação recebe notificações da ocorrência de 3 eventos, sendo 2 desses relacionados ao leitor de identificador 'wii0' e 1 relacionado ao leitor 'fly0'.

```
Contrutor()
{
  listenEvent( "wii0", "wii_btn_left_press" );
  listenEvent( "wii0", "wii_btn_left_release" );
  listenEvent( "fly0", "flystick_btn_0_press" );
}

EventHandler( string id_leitor, id_evento )
{
  if( id_leitor == "wii0" )
    if( id_evento == "wii_btn_left_press" )
      AndeParaEsquerda(); // função da aplicação
    else
      Parar(); // função da aplicação
  else
      LigarRaio(); // função da aplicação
}
```

Listagem 1. Exemplo de observador de eventos

O método *EventHandler* deve ser implementado por todo *EventObserver*. Esse método é chamado pelo *EventManager* sempre que há ocorrência de um dos eventos que foram registrados através de *listenEvent*. Um *EventObserver* recebe apenas os eventos para os quais ele se registrou. No exemplo acima, quando o botão esquerdo do controle wii identicado por 'wii0' é pressionado, a ação *AndeParaEsquerda* é executada. Quando o botão é solto, é realizada a ação *Parar*. Da mesma forma, quando o botão 0 do controle flystick 'fly0' é pressionado, a ação *LigarRaio* é disparada.

Uma vez criados os leitores e *EventObservers*, deve-se executar o método *ProcessEvents* de *InputController* sempre que um novo frame estiver para ser desenhado. É nesse momento que o *VrInput* chama os métodos *poll* de todos os leitores ativos e processa os eventos gerados, os enviando para os *EventObservers* registrados. Também nesse momento é verificado se houve erros de desconexão de dispositivos ou queda do processo secundário. No primeiro caso, é gerado um evento especial que indica a desconexão do dispositivo, o qual pode ser usado pela aplicação para tratar esse tipo de erro. No segundo caso, o *VrInput* tenta reiniciar o processo secundário e reativar os leitores. Esse processo é repetido por 3 vezes e caso não tenha sucesso o *VrInput* reporta o erro para a aplicação. Esse desacoplamento impede que a aplicação tenha que ser reiniciada no caso de falha em algum dispositivo.

C. VrFrustum

O *VrFrustum* é responsavel por criar as múltiplas câmeras que são necessárias para o ambiente imersivo a partir de uma câmera principal. Essa funcionalidade é comum nos frameworks de RV, no entanto o principal diferencial do *LVRL* é que o componente *VrFrustum* não precisa ter acesso a câmera principal, bastando conhecer a configuração que define a geometria do sistema de projeção e receber a pose da cabeça do usuário a cada quadro.

No caso de um ambiente desktop com um monitor o cálculo fica restrito a uma única tela. Porém, no caso de uma CAVE ou NexCAVE, comumente composta por mais de 3 telas, terão que ser calculados "N"pontos de vista

correspondente as ”N”telas do sistema. No caso de um sistema com suporte a estereoscopia serão ”2 x N”. Abaixo descrevemos como configurar e e usar o *VrFrustum* de forma a ser transparente a mudança do modo desktop para o modo RV.

O arquivo de configuração agrega os parâmetros que são inerentes ao ambiente imersivo. Esse arquivo é comum a todas aplicações que usem o *VrFrustum* para um determinado ambiente imersivo. Na listagem 2 é mostrado um exemplo de um arquivo de configuração para um ambiente imersivo em formato L, como na figura 1. O exemplo funciona com projetores de estéreo passivo lado a lado. Cada uma das duas telas é desenhada por uma maquina diferente.

O arquivo é escrito no formato XML. O marcador *ScreenSet* agrega o conjunto de telas (*screens*). Um tela pode ser um monitor, um projetor ou dois projetores (no caso de projeção com estéreo passivo). Para cada tela é informado a posição 3D dos 4 cantos (*corners*) da tela. Essa informação é suficiente para o cálculo do *frustum*.

O arquivo de configuração também contém informações sobre o mapeamento entre as diferentes saídas de vídeo dos projetores e as respectivas regiões (ou telas) do sistema de projeção (mapeamento de viewports). A viewport do olho direito é opcional, dado que se o sistema de projeção usar estéreo ativo ou não suportar estereoscopia só é necessário uma viewport. No caso do sistema imersivo funcionar distribuído, além da viewport, é necessario um identificador (*Machine*) que diz qual maquina é responsável por ela.

A última propriedade que uma tela pode ter é se ela é a *referência* (no caso do nosso exemplo é a primeira tela). Essa propriedade existe porque os planos de clipping *near* e *far* só podem ser definidos quanto a uma tela. As outras telas precisam ser calculadas em função da tela de referência para que os planos de clipping se encaixem sem descontinuidades.

Outras informações importantes para a criação de uma cena 3D em ambientes imersivos são o *Pivot* e o *OffsetTracking*. O *Pivot* é uma posição 3D na cena e tem duas funcões. A primeira é ser usada como a posicão padrão da cabeça do usuário no caso em que o sistema não tenha rastreamento de cabeça. A segunda é ser a origem do sistema de coordenadas da matriz de modelview fornecida pela componente para o posicionamento das câmeras no ambientes imersivo.

O *OffsetTracking* é uma posiçao 3D que representa a origem do sistema de coordenada do equipamento de rastreamento. Consideramos que todos sistemas de coordenada tem a mesma orientação e que a descrisão dos cantos da cave estão na mesma unidade fornecida pelo equipamento de rastreamento, em geral metros.

Na figura 3 é mostrada uma visualização simulada do que cada tela da cave exibe, no caso ela está sobre um navio plataforma de produção de petróleo.

A listagem 3 apresenta a simplicidade da interface de programação dessa componente e como ela é transparente

```
<ScreenSet>
 <Screen reference="true">
  <LowerLeftCorner X="0" Y="-3" Z="0"/>
  <LowerRightCorner X="3" Y="-3" Z="0"/>
  <UpperRightCorner X="3" Y="0" Z="0"/>
  <UpperLeftCorner X="0" Y="0" Z="0"/>
  <ViewportLeftEye X="0.0" Y="0.0" W="0.5" H="1.0"/>
  <ViewportRightEye X="0.5" Y="0.0" W="0.5" H="1.0"/>
  <Machine IP="10.0.0.1">
 </Screen>
  <Screen >
  <LowerLeftCorner X="0" Y="-3" Z="3"/>
  <LowerRightCorner X="3" Y="-3" Z="3"/>
  <UpperRightCorner X="0" Y="-3" Z="0"/>
  <UpperLeftCorner X="3" Y="-3" Z="0"/>
  <ViewportLeftEye X="0.0" Y="0.0" W="0.5" H="1.0"/>
  <ViewportRightEye X="0.5" Y="0.0" W="0.5" H="1.0"/>
  <Machine IP="10.0.0.2">
 </Screen>
</ScreenSet>
<Pivot  X="1.5" Y="-1.2" Z="1.5">
<OffsetTracking X="1.5" Y="-0.5" Z="1.5">
```

Listagem 2. Arquivo de configuração do VrFrustum.

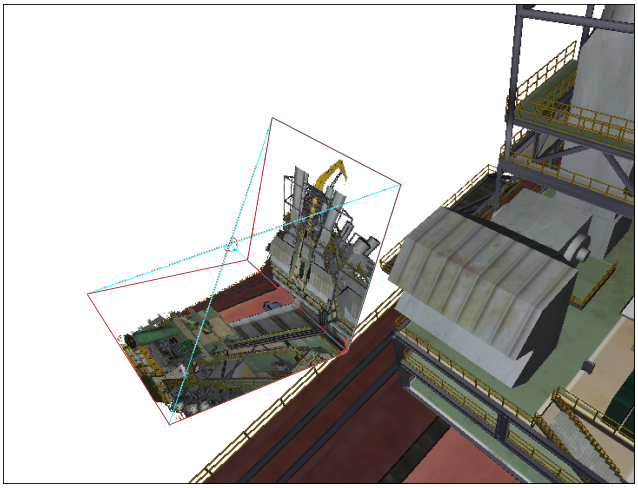

Figura 3. Visualização do sistema imersivo do arquivo de exemplo.

quanto aos ambientes imersivos. Na função de desenho da aplicação principal basta pegar o número de telas e fazer um laço. Em cada iteração obtem-se as matrizes da tela e atritui-se essas nas matrizes correntes da engine de desenho usada antes de chamar a função de renderização.

D. VrHeadTracking

O *VrHeadtracking* é uma componente cuja função é gerenciar os dados recebidos sobre o monitoramento da posição e orientação da cabeça de um usuário, e repassar esses dados para o *VrFrustum* realizar os cálculos dos novos pontos de vista do usuário em relação a cada uma das telas que compõe o sistema de visualização 3D. No modo de desktop, a posição do rastreamento de cabeça não tem efeito nenhum.

Internamente, o *VrHeadtracking* se conecta e recebe os dados do rastreamento da posição e orientação da cabeça do

```
  //funções de configuração dependentes do ambiente RV
  void loadConfigFile( string filename );

  void setHeadPose( double[3] position
                   ,double[16] orientation );

//funções condiguração dependentes do conteudo desenhado
  void setNearPlane( double value );
  void setFarPlane( double value );
  void setScale( double value );
  void setInterEyeDistance( double value );
  void enableDesktopMode( double fov, double aspect );
  void disableDesktopMode();

  //funções para o desenho
  int getNumberOfScreens();

  void getScreenMatrixs( int screen_index,
                         double* projection_esq,
                         double* modelview_esq,
                         double* viewport_esq,
                         double* projection_dir,
                         double* view_dir,
                         double* viewport_dir,
                         string* ip_machine );
```

Listagem 3. Interface de programação do VrFrustum

usuário usando a componente *VrInput*. Com esses dados o *VrHeadtracking* se conecta a uma instancia do *VrFrustrum* na qual já se encontram definidas um número de câmeras virtuais correspondentes ao numero de telas do sistema de visualização 3D. A cada quadro o sistema verifica a posição e orientação da cabeça do usuário dentro do sistema de visualização 3D, e as posições das câmeras virtuais são recalculadas. Esses dados são repassados para o sistema de renderização da aplicação, permitindo que cada tela do sistema imersivo tenha uma projeção do cenário virtual corretamente alinhada em relação ao ponto de vista corrente do usuário.

E. VrWand

A componente *VrWand* fornece um apontador virtual 3D. Este tipo de recurso é usado em ambientes virtuais imersivos para que o usuário interaja com os objetos do cenário virtual, realizando operações de seleção e manipulação de objetos. É uma generalização do conceito de mouse para 3D. Esse apontador é comumente chamado de *raio* por ser um vetor no espaço 3D. No *LVRL*, o componente *VrInput* fornece os dados de movimentação e direção obtidos dos dispositivos de rastreamento que ela suporta. Esses dados são usados para controlar a posição e orientação do raio.

O *VrWand* armazena e processa os dados vindos do tracker e fornece ao final um vetor posição e uma orientação já em coordenadas do sistema definido pelo cenário virtual. Isso é possivel usando as matrizes do *VrFrustrum* e do *VrInteraction* para transformar as coordendas do sistema do rastreador para o do cenario virtual.

No modo desktop, o raio virtual sai da câmera e passa pela posição do mouse no plano *near*. Já no caso do modo RV a posição é definida pelo sistema de rastreamento convertida para o sistema de coordenadas do cenário virtual, preservando detalhes de escala entre as unidades do sistema de rastreamento (metros) e a unidade de medida no cenário virtual.

F. VrManipulators

O *VrManipulators* contém as implementações dos manipuladores disponíveis no framework. Um manipulador é um objeto capaz de modificar a matriz de transformação da câmera de modo a produzir um tipo específico de interação. Atualmente existem quatro tipos de manipuladores:

- *Fly*: produz na câmera o comportamento de um voo livre. Dentre os manipuladores atuais é o menos restritivo. É possível apontar a câmera em todas as direções. A partir dessa direção de referência, movimentasse a câmera para frente, para tras, para os lados, para cima e para baixo.
- *Examine*: a câmera realiza movimentos de rotação em torno de um ponto, o *pivot*. O posicionamento do *pivot* em um ponto específico de um objeto do cenário virtual, por exemplo no centro desse último, permite simular a inspeção em torno do objeto escolhido para *Pivot*
- *Walk*: tem um comportamento similar ao do manipulador *fly*, mas com a restrição de que a câmera se movimenta sempre no nível do plano do chão, mesmo quando a câmera é apontada para cima. A referência de onde é o chão em um dado momento deve ser fornecida pela aplicação. Como resultado, tem-se a sensação de se estar caminhando pela cena.
- *Rail*: nesse manipulador o movimento de translação da câmera somente pode ser feito em cima de um caminho específico atribuido previamente pela aplicação, o *trilho*. É possível movimentar a câmera nos dois sentidos do *trilho*, ao mesmo tempo em que se pode apontar a câmera em todas as direções.

Os manipuladores possuem uma interface padrão composta basicamente po 3 métodos, que devem obrigatoriamente ser implementados:

- *setViewMatrix()*: Usado para atribuir a posição e orientação atual da câmera. Esse método deve ser chamado preferêncialmente uma vez, sempre que o manipulador começar a ser usado.
- *update(dt)*: Nesse método fica a implementação do comportamento do manipulador. Nele são feitos os cálculos necessários para produzir a mudança na câmera de acordo com chamadas realizadas pelos dispositivos de entrada. Por exemplo, apertar a tecla 'w' provoca a chamada do método *goForward* no manipulador *fly*. Esse último método tem como efeito empilhar a *requisição da ação* "mover a câmera para frente". Ao ser executado, *update* verifica a existência dessa requisição e calcula a modificação necessária na câmera. O parâmetro *dt* é o intervalo de tempo entre a

renderização do frame atual para o anterior e é usado no cálculo do movimento. O método *update* deve ser chamado sempre que um novo frame estiver para ser desenhado.
- *getViewMatrix()*: Usado para obter a matriz que deverá ser atribuida à câmera. Deve ser chamado sempre após o método *update*.

Além dos métodos descritos acima, um manipulador implementa também *métodos de requisição de ações*. Esses são específicos de cada manipulador e são chamados geralmente quando um novo evento de dispositivo chega. Por exemplo, o manipulador *fly* possui os métodos *goForward*, *goBackward*, *goLeft*, *goRight* e *Rotate* que provocam requisições, respectivamente, de mudanças na translação e orientação da câmera. Da mesma forma, o manipulador *examine* possui os métodos *zoomIn*, *zoomOut* e *Rotacionar*. Como mencionando antes, as ações desses métodos são realizadas no método *update*, quando este é chamado.

Cada manipulador possui acesso a uma estrutura chamada de *ManipulatorState*. Um ponteiro dessa estrutura é passado ao manipulador através de seu construtor. É através do *ManipulatorState* que os parâmetros de configuração dos manipuladores podem ser alterados. Atualmente estão disponíveis os seguintes parâmetros:

- *Velocidade de navegação*: Usado no cálculo do movimento de translação da câmera.
- *Velocidade da rotação*: Usado no cálculo de uma rotação realizada na câmera.
- *Vetor Up do mundo*: Estabelece qual o vetor UP do mundo. Em nossa implementação todos os manipuladores podem realizar rotações apenas em torno de dois eixos: o eixo UP do mundo e o eixo RIGHT da câmera. Com essa restrição, garante-se que a câmera sempre fique alinha com o UP do mundo, o que se revelou ser um requisito importante em ambientes imersivos.
- *Ponto de pivot*: É o ponto em torno do qual a câmera fará rotações. Atualmente é usado apenas pelo *examine*.
- *Ponto no chão*: Em conjunto com o vetor UP do mundo, esse ponto estabelece qual o plano do chão usado pelo manipulador *andar*.
- *Caminho do trilho*: Estabelece qual o caminho que o manipulador *trilho* deve seguir.
- *Permitir rotação em torno do vetor RIGHT*: Em testes realizados, percebeu-se que em alguns momentos é conveniente limitar a liberdade de rotação da câmera afim de facilitar a conclusão de algumas tarefas de interação. Além disso, em alguns tipos de ambientes de visuaização, algumas rotações podem provocar desorientação ao usuário. Esse é caso por exemplo quando se permite em uma CAVE rotações em torno do vetor RIGHT. Isso tem o efeito de fazer que o usuário tenha a impressão de que a cena está "torta" para algumas das telas. Pensando nisso, esse parâmetro permite travar a rotação em torno do eixo RIGHT.
- *Permitir rotação em torno do vetor UP*: Tem o mesmo efeito do parâmetro descrito no tópico anterior, só que atua nas rotações em torno do vetor UP do mundo.

Os manipuladores descritos acima fornecem o suporte básico para tarefas de navegação e inspeção em aplicações 3D e foram desenvolvidas para funcionar em ambientes que vão desde o desktop padrão até ambientes imersivos como CAVEs, NEXCAVE. Há planos para a criação de novos manipuladores, alguns desses sendo apenas extensões dos já existentes.

G. Transições

Uma transição é uma interpolação entre duas poses diferentes de câmera. Seu objetivo é impedir que o usuário fique desorientado ao trocar entre manipuladores. Por exemplo, ao selecionar o manipulador *rail*, dependendo da posição corrente da câmera, a transição fará com que ela seja automaticamente posicionada para o início do caminho de câmera definido no manipulador *rail*. Isso é feito através de uma animação suave que leva da pose atual para a definida pelo manipulador *rail*. Caso a câmera fosse simplesmente transladada para a nova posição, isto é, sem a transição, o usuário poderia ficar desorientado, especialmente em ambientes imersivos como a CAVE.

Internamente, a transição é implementada como uma interpolação entre a *pose* inicial e final da câmera quando da troca entre manipuladores. Uma transição possui como parâmetros de controle sua duração e o tipo de interpolação usado. São fornecidos dois tipos:

- *Linear*: Consiste em uma interpolação linear da posição e orientação corrente da câmera para a nova posição requerida pelo manipulador selecionado. Ideal para transiçês de curta duração.
- *Slow Fast Slow*: Consiste em uma composição de interpolação não linear da posição e interpolação esférica (*Slerp*) da orientação corrente da câmera para a nova posição requerida pelo manipulador selecionado. A interpolação é dividida em três etapas: uma inicial e uma final que equivalem a 30% do tempo de duração da transição, e uma etapa intermédia equivalente a 70% do tempo restante. O comportamento é baseado em movimento uniformemente acelerado. A câmera acelera na etapa inicial até uma certa velocidade, se mantém constante na segunda etapa e, na etapa final, desacelera até que a câmera pare.

Para garantir que os modos propostos cumpram a tarefa de diminuir a desorientação do usuário, é recomendável avaliar o tempo de duração da transição em relação ao tipo de ambiente onde o usuário vai interagir. Por exemplo, num ambiente imersivo como uma CAVE é recomendável definir um tempo de transição que não seja pequeno (por exemplo ≥ 5 segundos). Caso o tempo seja pequeno demais,

Figura 4. LVRL sendo usado em uma CAVE com suporte a Flystick.

o efeito da transição será equivalente a um corte seco, aumentando o grau de desorientação do usuário. Já num ambiente desktop sem imersão, o tempo de transição alto pode ser desconfortável ao usuário, principalmente por causa da espera para o término da transição.

H. VrInteraction

O *VrInteraction* é o gerenciador da interação. Ele controla a forma como os dispositivos de entrada fazem uso dos manipuladores, assim como o processo de troca entre eles. Para isso ele faz uso dos módulos *VrInput*, *VrManipulators*, *VrTransitions* e de conjuntos de *mapeamentos* associados aos manipuladores.

Um *mapeamento* define a relação entre um dispositivo de entrada e um manipulador. Por exemplo, um mapeamento pode especificar que o movimento de arraste do mouse causa uma rotação proporcional no manipulador *examine*. Um mapeamento é implementado através de um *EventObserver*, projetado para escutar eventos de um dispositivo e mapeá-los em ações em um manipulador específico. Diferente dos *EventObservers* criados pela aplicação, um mapeamento no *VrInteraction* não é capaz de fazer chamadas diretas a métodos da aplicação, mas apenas àqueles definidos no manipulador envolvido no mapeamento.

Atualmente existem mapeamentos para teclado e mouse, e os controles wii e flystick (figura 4). Especificamente, os dispositivos teclado e mouse são usados em conjunto, formando os mapeamentos usados no desktop padrão, onde é garantido que eles estarão presentes. Esses mapeamentos foram feitos para os 4 manipuladores descritos na seção III-F. Dessa forma, hoje há um total de 12 mapeamentos no *VrInteraction*: 4 para teclado e mouse, 4 para o wii e 4 para o flystick. Novos mapeamentos envolvendo dispositivos como kinect, celulares android e ipads estão sendo desenvolvidos. Além disso, como a criação de um mapeamento consiste apenas na criação de um novo *EventObserver*, a inclusão de suporte a novos dispositivos no futuro é uma tarefa simples.

Os recursos usados por esses mapeamentos não devem ser acessados diretamente pela aplicação, pois isso pode gerar uma situação de conflito em que o recurso seria usado para duas tarefas distintas simultaneamente. Por esse motivo, cada mapeamento foi desenvolvido para aproveitar as funcionalidades do dispositivo da melhor forma possível usando o mínimo de recursos do mesmo. Por exemplo, o flystick possui 6 botões e no máximo 2 são usados no *VrInteraction*. Os botões restantes são deixados para que o desenvolvedor possa, através da criação de um *EventObserver* próprio, usá-los para realizar tarefas especificas da aplicação. Por exemplo, um dos botões restantes pode ser usado para iniciar a visualização de uma simulação.

A presença de mapeamentos pré-definidos tem como vantagem minimizar o trabalho do desenvolvedor que deseja criar uma aplicação RV de forma expedita. Não é preciso se preocupar com a criação de código relacionado a interação pois ele já está imbutido no *VrInteraction*. Também não é preciso pensar em um tratamento específico para diferentes ambientes RV uma vez que os manipuladores e mapeamentos foram criados para funcionar tanto em desktop quanto em ambientes imersivos. Por último, o *VrInteraction* padroniza a interação nas aplicações que o utilizam. Do ponto de vista do usuário, isso simplifica o aprendizado tornando-o mais rápido e eficaz uma vez que ele não terá que aprender um novo sistema de interação ao usar uma aplicação diferente.

As funcionalidades do *VrInteraction* são fornecidas através da interface *InteractionController*, cuja api é brevemente descrita abaixo:

- *setDeviceReader(deviceType, readerId)*: Esse método informa ao *VrInteraction* qual é o leitor que será usado para acessar um dispositivo. O parâmetro *deviceType* é uma string pré-definida no *VrInteraction* que identifica um tipo de dispositivo. Por exemplo, as strings VRINT_DESKTOP, VRINT_WII e VRINT_FLYSTICK referem-se respectivamente aos dispositivos teclado e mouse, wii e flystick. O parâmetro *readerId* é um identicador de um leitor previamente criado através do *VrInput*. Como exemplo, assumindo que um leitor para controles wii foi criado com o identicador ”wii0“, a chamada *setDeviceReader(VRINT_WII, ”wii0“)* faz com que esse leitor seja usado por todos os mapeamentos para o controle wii.
- *setMode(mode)*: Através desse método a aplicação escolhe qual manipulador será usado no momento. O parâmetro *mode* é uma string que identifia um manipulador. Por exemplo, *setMode(VRINT_FLY)* faz com o manipulador *fly* seja ativado. Quando chamado, esse método verifica se é preciso realizar uma transição entre o manipulador anterior e o atual. Por exemplo, se o manipulador *rail* for escolhido, o *VrInteraction* cria uma transição que leva a câmera até a posição inicial do trilho. Depois dessa etapa, procura-se por um mapeamento entre o manipulador que se quer usar

e o dispositivo que está ativo no momento. Caso não seja encontrado nenhum compatível, é escolhido um mapeamento nulo que não produz efeito na câmera. Por fim, a matriz de câmera resultante do manipulador anterior é atribuida ao atual. Isso garante, junto com as transições, que haja continuidade e fluidez na interação.

- *setDevice(deviceType)*: Esse método permite escolher qual dispositivo usar. O parâmetro *deviceType* é uma string que identifica o tipo de dispositivo. Por exemplo, *setDevice(VRINT_DESKTOP)* faz com que o teclado e o mouse sejam usados. Assim como acontece no método *setMode*, quando *setDevice* é chamado, o *VrInteraction* procura por um mapeamento compatível com o tipo de dispositivo escolhido e o manipulador atual. Caso não seja encontrado, um mapeamento nulo é escolhido.
- *setViewMatrix(m)*: O *VrInteraction* encapsula o uso do módulo *VrManipulators*. Dessa forma, a aplicação não tem acesso direto aos manipuladores. Como mencionado antes, é o *VrInteraction* o responsável por gerenciar o acesso aos manipuladores. Esse método tem como objetivo permitir atribuir a posição e orientação inicial da câmera nos manipuladores.
- *getViewMatrix()*: Esse método retorna a matriz de view do manipulador atual para que essa possa ser aplicada na câmera pela aplicação.
- *update(dt)*: Caso uma transição não esteja ocorrendo, esse método tem como efeito chamar o *update* do manipulador em uso. O parâmetro dt é um intervalo de tempo e tem o mesmo significado daquele passado para o método *update* do manipulador.

Para configurar os parâmetros de configuração descritos na seção III-F, o *VrInteraction* cria uma instância de *ManipulatorState* e a repassa para todos os manipuladores. Como a instância é única, garante-se uma uniformidade entre os comportamentos dos diferentes manipuladores. Para permitir que a aplicação mude esses parâmetros, é fornecida uma api na interface *InteractionController*.

Apesar de depender do módulo *VrInput*, o *VrInteraction* pode ter várias instâncias em uma mesma aplicação. Com isso é possível controlar mais de uma câmera. Isso é uma característica que pode ser usada em aplicações RV colaborativas, onde cada usuário teria controle de sua câmera. É importante mencionar também que o *VrInteraction* não realiza nenhuma chamada aos métodos da interface *InputController*. Ele apenas escuta os eventos gerados pelo *VrInput*. Logo, a criação dos leitores, assim como a chamada ao método *ProcessEvents* deve ser feita pela aplicação, como exposto na seção III-B. Caso isso não seja feito, não haverá geração de eventos nos dispositivos de entrada e, por consequência, o *VrInteraction* não será capaz de gerar uma matriz de transformação para a câmera.

Como mencionado antes, o uso do *VrInteraction* fornece formas de interação pré-definidas que permitem realizar as tarefas básicas de navegação e inspeção em novas aplicações 3D. Apesar disso criar certa rigidez, uma vez que os mapeamentos estão pré-definidos, isso contribui para minimizar o custo de criação de novas aplicações 3D, ou a transformação de aplicações legadas em aplicações imersivas. Além disso, espera-se que com a execução de testes de usabilidade e a opinião dos usuários dos sistemas que fazem uso do *VrInteraction*, os mapeamentos convirjam para formas ideias de interação.

IV. Discussão

Nesta seção discutimos as principais caracteristicas e contribuições do framework *LVRL*.

A. Interface de programação transparente

Utilizando o framework por completo, a aplicação só tem contato com os módulos *VrInteraction*, *VrWand*, *VrInput* e *VrFrustum*. Na seção anterior foi demostrado que todos esses componentes tem os mesmos resultados estando em modo desktop e RV. Dessa forma, o desenvolvedor pode programar sua aplicação sem se preocupar com detalhes inerentes ao tipo de ambiente de visuaização.

B. Mudança de desktop para imersivo em tempo de execução

Com o design do *VrInteraction* e do *VrFrustum* é possivel mudar de desktop para formato imersivo e vice-versa em tempo de execuçao apenas fazendo chamadas de métodos. Para reconfigurar as telas, basta carregar um novo arquivo de configuração no *VrFrustum*. Para usar um dispositivo de RV ao invés dos tradicionais mouse e teclado, é preciso somente dizer ao *VrInteraction* qual dispositivo usar.

C. Não intrusivo

Todas as componentes descritas não tomam conta da aplicação. São componentes que podem ser consultadas sempre que a aplicação principal precisar. O desenvolvedor pode usar a engine gráfica que quiser e não é exigido que sua aplicação seja reescrita em torno do LVRL. O controle de execução continua pertencendo a aplicação e não ao LVRL.

D. Multiplataforma

Todas as componentes não tem nenhuma dependência do sistema e foram desenvolvidas com o objetivo de rodar em vários sistemas operacionais. Atualmente há suporte para Windows e Linux. A exceção reside apenas no *VrInput*, que depende dos drives do dispositivo. Dessa forma, alguns dispositivos não são suportados em todas as plataformas. Apesar dessa dependência, o design baseado em um gerenciador central e vários leitores cujo o acoplamento são eventos baseados apenas em texto, permite que o *VrInput* seja usado em várias plataformas, mesmo quando um determinado dispositivo não possui driver para a plataforma.

E. Independência do Hardware

Com o uso do *VrInput* a presença ou não de um driver de dispositivos não influência no uso da componente. Apenas o leitor daquele dispositivo não estará disponível.

F. Portável

A implementação do framework usa apenas tipos nativos do C++. Dessa forma, conseguimos portá-lo facilmente para C e Unity3D, mostrando assim que a arquitetura do framework não impõe nenhuma restrição a portabilidade entre as linguagens.

G. Compatível com renderização distribuida

Apesar da componente não ter internamente primitivas de renderização distribuída, a configuração do VrFrustum fornece as ferramentas necessárias para se atingir esse objetivo.

V. Conclusoes e trabalhos futuros

O LVRL, com sua abordagem minimalista e não intrusiva, demostrou ser um framework com as características necessárias para ser usado na maioria das aplicações de visualizacao científica do grupo. Por esse motivo, acreditamos que isso possa ser extendido a várias outras aplicações existentes. O Design não intrusivo e a interface de programação transparente são as principais caracteristicas que permitem que programadores fora do universo de Realidade Virtual convertam ou desenvolvam novas aplicações para ambientes imersivos.

O principal trabalho futuro a ser feito é um estudo de ergonomia e de usabilidade para alcançar a melhor forma de interação para os mapeamentos presentes no *VrInteraction*. Além disso, há estudos em andamento sobre novas formas de interação envolvendo dispositivos como kinect, smartphones e pads.

Acknowledgment

omitido para revisão

Referências

[1] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart, "The cave: audio visual experience automatic virtual environment," *Commun. ACM*, vol. 35, no. 6, pp. 64–72, Jun. 1992.

[2] T. DeFanti, D. Acevedo, R. Ainsworth, M. Brown, S. Cutchin, G. Dawe, K.-U. Doerr, A. Johnson, C. Knox, R. Kooima, F. Kuester, J. Leigh, L. Long, P. Otto, V. Petrovic, K. Ponto, A. Prudhomme, R. Rao, L. Renambot, D. Sandin, J. Schulze, L. Smarr, M. Srinivasan, P. Weber, and G. Wickham, "The future of the cave," *Central European Journal of Engineering*, vol. 1, pp. 16–37, 2011.

[3] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira, "Vr juggler: A virtual platform for virtual reality application development," in *Proceedings of the Virtual Reality 2001 Conference (VR'01)*. Washington, DC, USA: IEEE Computer Society, 2001.

[4] J. Gascón, J. M. Bayona, J. M. Espadero, and M. A. Otaduy, "Blendercave: Easy vr authoring for multi-screen displays," *SIACG 2011: V IBERO-AMERICAN SYMPOSIUM IN COMPUTER GRAPHICS*, 2011.

[5] F. Wang, "Research on virtual reality based on eon studio," in *Proceedings of the 2010 Fourth International Conference on Genetic and Evolutionary Computing*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 558–561.

[6] R. Kuck, J. Wind, K. Riege, and M. Bogen, "Improving the avango vr/ar framework: Lessons learned," 2008.

[7] Dassault Systemes, "3DVIA Virtools," http://www.virtools.com, April 2012.

[8] C. Anthes, M. Satomi, A. Wilhelm, C. Sommerer, and J. Volkert, "Space trash : An interactive networked virtual reality installation." in *11th Virtual Reality International Conference (VRIC 09)*, Laval, France, April 2009, pp. 107–118.

[9] A. Raposo, I. H. F. dos Santos, L. P. Soares, G. N. Wagner, E. T. L. Corseuil, and M. Gattass, "Environ: Integrating vr and cad in engineering projects," *IEEE Computer Graphics and Applications*, vol. 29, no. 6, pp. 91–95, 2009.

[10] Y. Zhou, M. Garland, and R. Haber, "Pixel-exact rendering of spacetime finite element solutions," in *Proceedings of the conference on Visualization '04*, ser. VIS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 425–432.

[11] S. Rusinkiewicz and M. Levoy, "Qsplat: a multiresolution point rendering system for large meshes," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 343–352.

[12] P. Silva, M. Machado, and M. Gattass, "3d seismic volume rendering," in *Eighth International Congress of The Brazilian Geophysical Society*, 2003.

[13] "Openscenegraph," http://www.openscenegraph.org.

[14] "Opensg," http://www.opensg.org/.

[15] "Opengl performer," http://oss.sgi.com/projects/performer/.

[16] C. Cruz-Neira, "Virtual reality based on multiple projection screens: The cave and its applications to computational science and engineering," Chicago, Illinois, USA, 1995.

[17] R. M. Taylor, II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser, "Vrpn: a device-independent, network-transparent vr peripheral system," in *Proceedings of the ACM symposium on Virtual reality software and technology*, ser. VRST '01. New York, NY, USA: ACM, 2001, pp. 55–61.

[18] G. Humphreys and P. Hanrahan, "A distributed graphics system for large tiled displays," in *Proceedings of the conference on Visualization '99: celebrating ten years*, ser. VIS '99. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 215–223.

[19] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A scalable parallel rendering framework," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 3, pp. 436–452, May 2009.

[20] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MA, USA: MIT Press, 1994.