VII Symposium on Virtual Reality, p.51-62. São Paulo, SP, Brasil, 2004

ViRAL: Um Framework para o Desenvolvimento de Aplicações de Realidade Virtual

Thiago de A. Bastos, Romano J. M. da Silva, Alberto B. Raposo, Marcelo Gattass

Tecgraf – Grupo de Tecnologia em Computação Gráfica / Dep. de Informática
PUC-Rio – Pontifícia Universidade Católica do Rio de Janeiro, Brasil
{tbastos, romano, abraposo, mgattass}@tecgraf.puc-rio.br

Abstract. *This paper introduces ViRAL (Virtual Reality Abstraction Layer), a graphical component-based framework for Virtual Reality (VR) applications development. We briefly explain the motivations for the development of a new VR framework and then describe ViRAL, detailing its architecture and pointing out its most important features as compared to existing VR frameworks: Simplicity, a component-based design and integrated graphical user interfaces.*

Resumo. *Este artigo apresenta o ViRAL (Virtual Reality Abstraction Layer), um framework gráfico, baseado em componentes, para o desenvolvimento de aplicações de Realidade Virtual (RV). Após breve discussão sobre a motivação para o desenvolvimento da ferramenta, o ViRAL é apresentado, detalhando sua arquitetura e destacando seus recursos mais importantes em comparação com outros frameworks de RV existentes: simplicidade de uso, design baseado em componentes e interface gráfica integrada.*

1. Introdução

As aplicações de *desktop* baseadas no paradigma WIMP (*Windows, Icons, Menus and Pointing Device*) usam dispositivos de entrada convencionais como o teclado, mouse e joystick. O dispositivo de saída é normalmente um monitor ou um sistema de projeção que replica a imagem do monitor. Aplicações de RV, por outro lado, são desenvolvidas para dar aos usuários a sensação de imersão, fazendo-os se sentirem dentro de um ambiente virtual (AV). Para isso, muitos dispositivos de RV têm sido criados, inclusive complexos sistemas de projeção, como a CAVE [Cruz-Neira 1993], e dispositivos de entrada variando desde mouses 3D até sensores sem fio com seis graus de liberdade.

Uma outra característica das aplicações de RV é que, geralmente, elas não são projetadas para serem usadas com dispositivos específicos. É desejável que funcionem com quaisquer dispositivos que atendam aos requisitos mínimos. Contudo, desenvolver aplicações independentes de sistema de RV é uma tarefa complexa e custosa.

Os frameworks de RV oferecem design e construtos reutilizáveis para o desenvolvimento de aplicações de RV. Eles solucionam problemas notórios desta área, como lidar com dispositivos de entrada e saída ou renderizar para diferentes sistemas de projeção. Alguns frameworks oferecem abstrações para que as aplicações se adaptem a qualquer sistema de RV [Bierbaum 2000]; outros oferecem suporte para simulações distribuídas [Arsenault 2001, Tramberend 2001]. Todos estes frameworks têm algo em comum: são profundamente complexos, e ficam ainda mais difíceis de usar à medida

que evoluem e passam a oferecer mais recursos. Toda esta complexidade afeta negativamente os desenvolvedores de aplicações de RV e, conseqüentemente, impactam na qualidade da aplicação final, tornando-a mais difícil de manter e utilizar.

O projeto do ViRAL surgiu pela carência de um framework que fosse pequeno e simples – adequado para sistemas de RV de baixo custo. Os frameworks existentes se mostraram complexos demais para nossas necessidades, visto que a maioria de nossas aplicações funcionaria em sistemas de RV simples, com um único PC. Os principais operadores dessas aplicações seriam técnicos não especializados, reforçando assim a necessidade delas serem fáceis de configurar e usar. Os operadores não deveriam precisar lidar com código ou arquivos de configuração, mas sim poder trabalhar com interfaces gráficas intuitivas. Por fim, os requisitos do projeto – em especial dar suporte a interfaces gráficas e reconfiguração em tempo de execução, levaram à construção de um framework *grey-box* com componentes gráficos integrados.

A próxima seção faz uma breve análise dos frameworks de RV atuais, destacando suas diferenças e similaridades com o ViRAL. As seções seguintes cobrem os principais aspectos do projeto do ViRAL e como ele é utilizado.

2. Trabalhos Relacionados

Os principais frameworks de RV existentes são ferramentas destinadas a pesquisadores ou programadores experientes. São projetos que desenvolvem e oferecem tecnologia de ponta, mas sem a preocupação de torná-la acessível a grupos menos especializados. O projeto do ViRAL, por outro lado, busca simplificar o desenvolvimento de aplicações de RV, torná-las mais fáceis de serem operaradas e, até certo ponto, permitir que técnicos e artistas, sem precisar programar, componham suas próprias aplicações.

Aplicações que se baseiam em frameworks *white-box* convencionais, como o VR Juggler [Bierbaum 2000], são desenvolvidas e mantidas inteiramente por programadores. Cada aplicação é desenvolvida como um produto fechado. O ViRAL se distancia desse modelo, seguindo uma arquitetura *grey-box* baseada em componentes [Szyperski 1997], onde não há o conceito de aplicação fechada: uma aplicação passa a ser definida pelos componentes que estão instanciados e conectados no momento. Uma vez programado, um componente pode ser usado em qualquer aplicação ViRAL da forma que o operador definir, possibilitando a criação dinâmica de novas aplicações.

Apesar das grandes diferenças arquiteturais, o ViRAL se aproxima bastante do VR Juggler em termos de recursos. Ambos são capazes de adaptar-se a praticamente qualquer sistema de RV. Seus ambientes virtuais podem ser renderizados com qualquer biblioteca que faça uso do OpenGL internamente. Ambos são também capazes de capturar e abstrair os dados de entrada de inúmeros dispositivos de RV, mantendo suas aplicações independentes de dispositivos específicos.

3. Design do ViRAL

O ViRAL pode ser visto como uma camada interposta entre as aplicações e os sistemas de RV. Essa camada é capaz de abstrair particularidades dos sistemas, tais como a forma de capturar entrada ou renderizar gráficos. Ela se baseia em um conjunto extensível de componentes de software com representação gráfica, e objetiva facilitar o desenvolvimento de aplicações de RV que sejam operadas por interfaces WIMP.

Um *componente* é um artefato de software, com interfaces internas e externas bem definidas, que é independente de outros artefatos de software. Este é um conceito central no design do ViRAL: dispositivos de entrada, observadores e ambientes virtuais são todos *componentes*, que podem ser carregados de bibliotecas e integrados em GUIs. Componentes são naturalmente independentes entre si, sendo capazes de se comunicar e trabalhar em conjunto sem conhecer um ao outro. Tal propriedade é útil, por exemplo, para manter as aplicações independentes de um dispositivo de entrada em particular sem limitá-las a um número pré-definido de eventos, como normalmente acontece.

O framework é portável para diversos ambientes, já que não contém código dependente de plataforma. Ao invés de desenvolver uma camada de abstração de plataformas própria, preferimos adotar uma solução existente: o Qt, da Trolltech, que atende a praticamente todas as nossas necessidades, inclusive interfaces gráficas.

Uma aplicação ViRAL é mantida independente de sistema de RV por meio de três camadas intermediárias (Figura 1b): o Qt abstrai a plataforma e o S.O. em uso; o ViRAL abstrai o sistema de RV, tratando a entrada de dados e gerando saída para os dispositivos disponíveis; por último, os plugins do ViRAL encapsulam outras APIs específicas para fazer interface com dispositivos, gerar som, gráficos e assim por diante.

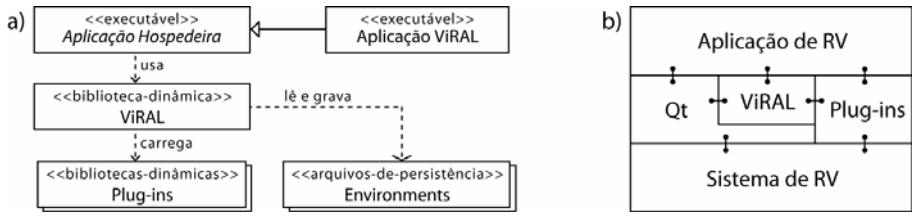

Figura 1: a) Diagrama de componentes; b) Acoplamento entre camadas.

O ViRAL pode ser estendido através de plugins: novos componentes podem ser importados de bibliotecas dinâmicas ou exportados pelas aplicações de RV. Eles podem adicionar suporte a novos dipositivos de entrada, novos ambientes virtuais, ou sistemas completamente originais, como será mostrado.

O framework pode ser utilizado de duas maneiras distintas: como um conjunto de components embutidos em uma *aplicação hospedeira* (qualquer aplicação onde o Qt pode ser incorporado; tipicamente uma aplicação de RV), ou como uma aplicação autônoma e extensível (aqui referida como "a aplicação ViRAL"). Evidentemente, qualquer aplicação que embute o ViRAL é automaticamente extensível, e a aplicação ViRAL é apenas uma *aplicação padrão*, que entretanto é útil em muitos casos.

A aplicação ViRAL funciona como uma ferramenta. Sua interface gráfica pode ser usada para preparar e executar aplicações de RV relativamente complexas através da combinação de componentes carregados de bibliotecas dinâmicas. Por exemplo, é possível instanciar e configurar um componente de ambiente virtual (*Scene*), vários dispositivos de entrada (*Devices*), um observador (*User*), várias projeções de vídeo (*Projections*), e então conectar os componentes apropriadamente (e.g. conectar um mouse 3D a um observador, tornando possível navegar na cena) para obter uma simulação de RV funcional. A partir de um único executável, diferentes aplicações de RV podem ser obtidas instanciando-se diferentes combinações de componentes.

Tendo configurado uma aplicação ViRAL, é possível iniciar, pausar e continuar uma simulação de RV a qualquer momento. Mesmo enquanto uma simulação está ativa, é possível que um operador acesse as janelas da aplicação e a reconfigure, ao vivo, sem interromper a simulação, desde que a estação possua um visor adicional independente.

O estado de uma aplicação ViRAL pode ser preservado através de arquivos de persistência, denominados *environments* (Figura 1a), um subsistema do framework. Todos os objetos do ViRAL têm a chance de armazenar um conjunto arbitrário de dados em um *environment*, incluindo todos os dados necessários para restaurar uma simulação de RV do ponto em que ela foi suspensa. Os arquivos de configuração, populares no VR Jugger, são substituídos pela combinação de GUIs e persistência.

3.1. Arquitetura do Framework

Toda aplicação ViRAL é descrita por uma estrutura hierárquica que contém todos os seus objetos e componentes centrais. Essa estrutura pode ser vista como uma árvore heterogênea tendo, na raiz, o *singleton* [Gamma 1995] *RootSystem*; em seu primeiro nível, objetos do tipo *System*; e quaisquer outros objetos nos níveis mais abaixo (Figura 2). A árvore ViRAL simplifica a propagação de eventos e a implementação de persistência para todo o sistema, dentre várias outras tarefas.

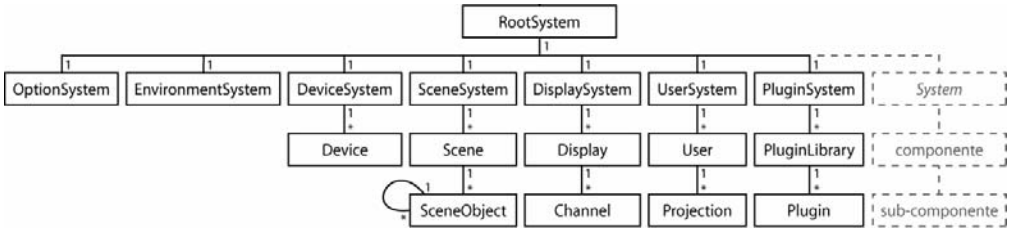

Figura 2: Esboço de uma árvore ViRAL contendo os *Systems* padrões.

O primeiro nível da árvore está restrito aos *Systems* – objetos *singletons* que agregam funcionalidades ao framework. Todo objeto não-*System* faz sempre parte de um *System*, devendo ser projetado de forma coerente. O ViRAL não define os tipos de objetos admitidos em um *System*, ou como eles devem se relacionar. Por exemplo, como visto na Figura 2, o *OptionSystem* não possui filhos; já o *SceneSystem* possui *Scenes*, que por sua vez podem conter um número qualquer de árvores de *SceneObjects*.

Objetos do ViRAL são componentes quando utilizam o modelo de *signals/slots* [Trolltech 2004] para expor suas funcionalidades. A aplicação desse padrão permite que componentes cooperem sem necessariamente se conhecerem. *Signals* são tipicamente estímulos emitidos por um componente para sinalizar mudanças em seu estado interno, enquanto que os *slots* são respostas aos *signals*. Tome como exemplo um simulador de direção (e.g. para treinar motoristas); o ambiente virtual seria um componente, assim como os carros. Cada carro teria um *slot* recebendo como argumento um valor que ajusta o ângulo do seu volante. Qualquer componente poderia ser usado para dirigir os carros, desde que forneça um *signal* que tenha como argumento um ângulo. Assim, qualquer dispositivo com pelo menos um eixo de rotação (e.g. um volante real) poderia ser usado para dirigir os carros, bastando que seu *signal* seja conectado ao *slot* do carro.

Vários eventos são propagados pela árvore ViRAL para comunicar mudanças de estado, como o início ou o fim de uma simulação. Esses eventos são úteis, por exemplo,

para um *Device* saber quando ele deve começar ou parar de receber entrada de um dispositivo real; ou para um *Display* saber quando criar ou destruir seu contexto do OpenGL. Os *Systems* ainda recebem eventos adicionais, sendo notificados, por exemplo, após toda a árvore ter sido comunicada sobre o início de uma simulação; esses eventos extras fornecem controle adicional sobre o funcionamento do sistema.

O nível dos *Systems*, na árvore ViRAL, é construído na inicialização do sistema. Os *Systems* padrões são automaticamente instalados, enquanto que outros novos podem ser plugados via aplicações hospedeiras ou bibliotecas. Um *System* pode ter sua própria tela de interface integrada em uma aplicação hospedeira, como uma forma de se expor aos operadores. Por exemplo, a tela de interface do *OptionSystem* permite o ajuste das opções globais do sistema. Como será visto, a maioria dos *Systems* padrões fornece telas de interface, e a aplicação ViRAL é meramente uma agregadora dessas telas.

Como um framework gráfico, o ViRAL oferece uma coleção de *widgets* para uso em interfaces de componentes e aplicações. Existem *widgets* para editar vetores e quatérnios, visualizar ramos e selecionar objetos de uma árvore ViRAL (Figura 3).

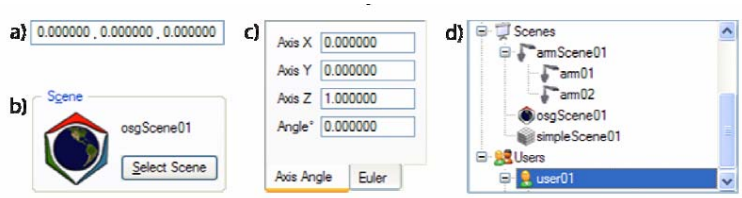

Figura 3: *Widget* de a) vetor; b) escolher objetos; c) quatérnio; d) árvore ViRAL.

Todos os objetos na árvore oferecem pelo menos um grau moderado de interação gráfica. Objetos simples possuem ícones, nomes únicos e podem responder a menus de contexto. Objetos mais avançados, como *Scenes* e *Devices*, podem, assim como os *Systems*, ter suas próprias telas de interface integradas em aplicações hospedeiras. É importante notar que objetos gráficos (com telas de interface) não sofrem penalidades em sua utilização no lado da programação: ao serem modificados, suas telas de interface são sempre atualizadas de forma transparente.

Os vários sistemas (*Systems*) padrões do ViRAL serão descritos nas subseções a seguir, passando uma noção mais profunda do funcionamento real do framework.

3.2. Devices – Sistema de Dispositivos

O sistema de dispositivos suporta a criação e interconexão de componentes do tipo *Device*, permitindo que operadores definam como eles devem funcionar baseados em seus *signals* e *slots*. O sistema serve como um centro de controle para a comunicação de componentes, sendo a rota padrão de entrada de dados para as aplicações de RV.

Um *Device* é um componente criado para gerar, processar ou transformar eventos. Eventos são emitidos como *signals* e tratados por *slots* – seguindo o paradigma de comunicação entre componentes. Contudo, eles podem ser entregues de duas formas distintas: síncrona (quando ocorrem) ou assíncrona (quando pedido). *Devices* podem ser conectados de forma que o recebimento de um evento causa o envio de outro, criando assim cadeias de eventos (Figura 5b). A entrega assíncrona otimiza a propagação de eventos através de cadeias longas. Eventos síncronos muito freqüentes (e.g. amostras de

rotação de um rastreador) podem ser acumulados por um *Device* intermediário, e só serem propagados quando necessário (e.g. logo antes de desenhar o próximo quadro).

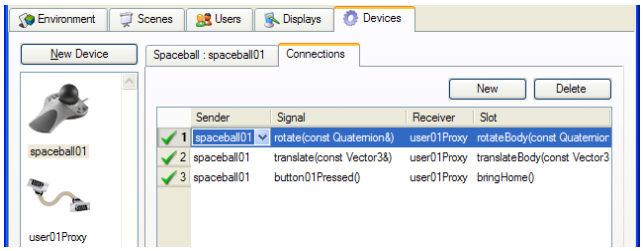

Figura 4: Tela de gerenciamento de conexões do sistema de dispositivos.

Dispositivos de todos os níveis de complexidade, de *wands* até braços com *force feedback*, podem ser representados dentro do ViRAL através de *Devices*. Porém, esses componentes não existem somente para fazer interface com dispositivos reais. Eles são usados para acumular, filtrar e transformar eventos, permitindo maior reutilização dos componentes. Devices podem ter telas de interface expondo suas configurações. Essas telas são integradas na interface do sistema de dispositivos, que também permite que operadores gerenciem conexões entre *Devices* utilizando uma tabela (Figura 4).

A comunicação entre *Devices* e outros tipos de componentes só é possível pelo uso de um *Proxy* – um *Device* que finge ter os *signals* e *slots* de outro componente. Eles são usados na maioria das aplicações de RV. No exemplo do simulador de direção, citado acima, um *Proxy* seria necessário para fazer conexões entre volantes e carros.

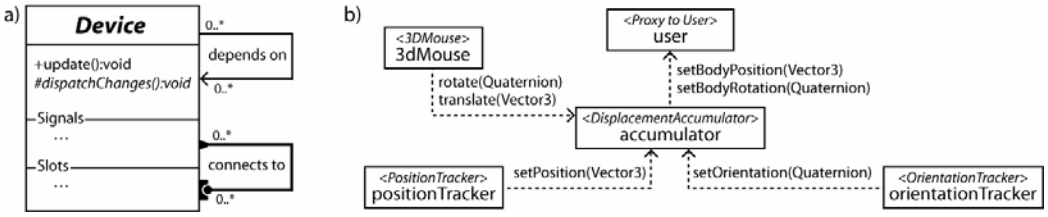

Figura 5: a) Esboço de um *Device*; b) *signals* percorrendo cadeia de *Devices*.

Componentes só podem se conectar quando o *signal* e o *slot* em questão têm parâmetros compatíveis. Um desenvolvedor pode criar qualquer espécie de *signal* e *slot* para seus componentes; contudo, tal liberdade só vale caso eles possam se comunicar corretamente. Deste modo, a definição de regras pode ser muito favorável, propiciando interfaces mais claras, reutilizáveis e eficientes. A primeira regra aplicada no projeto do ViRAL é não utilizar matrizes para representar posições, orientações, translações ou rotações. Vetores são usados para representar posições e translações, enquanto que quatérnios são usados para representar orientações e rotações. *Devices* devem sempre decompor seus eventos em vários *signals* elementares; por exemplo, um dispositivo com seis graus de liberdade deve prover pelo menos dois *signals*: um com um vetor (e.g. para translações) e outro com um quatérnio (e.g. para rotações). *Devices* com *slots* que recebem posições e orientações absolutas devem, sempre que possível, fornecer também *slots* que aceitem translações e rotações. Por exemplo, o componente *User* tem dois *slots* para definir sua posição e orientação absoluta, no espaço do ambiente virtual, e mais dois *slots* para transladá-lo e rotacioná-lo em seu espaço de coordenadas local.

O ViRAL distingue dispositivos que provêem posições e orientações absolutas – como rastreadores de posição, daqueles que provêem movimentos relativos – como os mouses 3D. Cada um desses métodos de entrada é explorado apropriadamente. Por exemplo, apesar de não ser possível usar diretamente múltiplos rastreadores de posição para controlar um observador, já que os eventos colidiriam, pode-se usar múltiplos dispositivos de movimento relativo, já que os eventos são cumulativos.

A Figura 5b mostra uma cadeia de *Devices* que poderiam ser utilizados para navegar em uma cena, juntamente com, por exemplo, um capacete de RV. As caixas representam *Devices*; as setas, a propagação de *signals*; os rótulos nas setas têm o nome do *slot* receptor e os seus parâmetros. Três dispositivos de entrada reais são utilizados: um mouse 3D, para movimentação irrestrita no AV; um rastreador de posição, para movimentos imersivos; e um rastreador de orientação, para ajustar a visão do capacete. Seria possível conectar esses três *Devices* diretamente a um *User*; porém, nesse caso os eventos absolutos dos dois rastreadores sobrescreveriam qualquer movimento feito com o mouse 3D. Um *Device* intermediário é então utilizado para calcular a posição final do *User* com base nos dados dos rastreadores e os movimentos do mouse 3D.

3.3. Scenes – Sistema de Cenas

Componentes *Scene* são responsáveis pela simulação de ambientes virtuais imersivos, renderizando gráficos para uma combinação qualquer de observadores (*Users*), visores e projeções (*Displays* e *Projections*, seção 3.4). Além de gráficos, estes componentes podem prover outros serviços relevantes, como testes de colisão e produção de som 3D.

Gráficos precisam ser renderizados com OpenGL, possivelmente por intermédio de uma API de nível mais alto, tal como um grafo de cena. Neste aspecto o ViRAL é bastante não-intrusivo, tentando ser o máximo compatível com todos os engines 3D e limitando suas chamadas do OpenGL a algumas operações de troca de buffer.

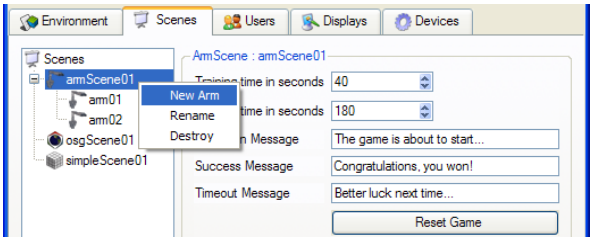

Figura 6: Sistema de cenas com três cenas, uma delas com dois objetos filho.

Para que se mantenham independentes de sistema de RV, as *Scenes* devem desconhecer totalmente o sistema em uso. Gráficos são renderizados chamando métodos que desenham um único quadro baseado em parâmetros passados como argumentos. Dentre esses parâmetros estão os componentes *User* e *Display* para o quadro sendo desenhado, e matrizes de projeção e visão derivadas de uma gama de parâmetros. Esta é a mesma solução utilizada no VR Juggler e outros frameworks.

Scenes podem ter árvores de *SceneObjects* como suas filhas na árvore ViRAL, um recurso valioso. Elementos das cenas, como um carro ou um avatar, podem ser feitos *SceneObjects* para que gozem de privilégios, como possuir uma tela de interface ou poder se comunicar com outros componentes do ViRAL. No caso de um simulador

de direção, por exemplo, um carro pode ser exposto como um *SceneObject* para que receba dados de entrada diretamente de um dispositivo de volante.

Como mostrado na Figura 6, *Scenes* e seus *SceneObjects* podem ter telas de interface próprias, permitindo que sejam configurados de dentro da interface do sistema de cenas (*SceneSystem*). Tal recurso também pode ser explorado para criar um esquema de edição e composição dinâmica de cenas.

3.4. Users & Displays – Sistema de Usuários e de Exposição

Users representam os usuários reais de um sistema de RV que estão imersos em um ambiente virtual. As principais propriedades de um *User* são: sua distância interocular, sua *Scene* e sua posição e orientação dentro da *Scene*. Um *User* tem quatro *slots* para fazer sua movimentação dentro de uma cena: dois para definir sua posição ou orientação atual, e dois para transladá-lo ou rotacioná-lo localmente. Esses *slots* permitem que um *User* receba dados de entrada diretamente de *Devices* que trabalham com dados absolutos (e.g. rastreadores) e movimentos relativos (e.g. mouses 3D).

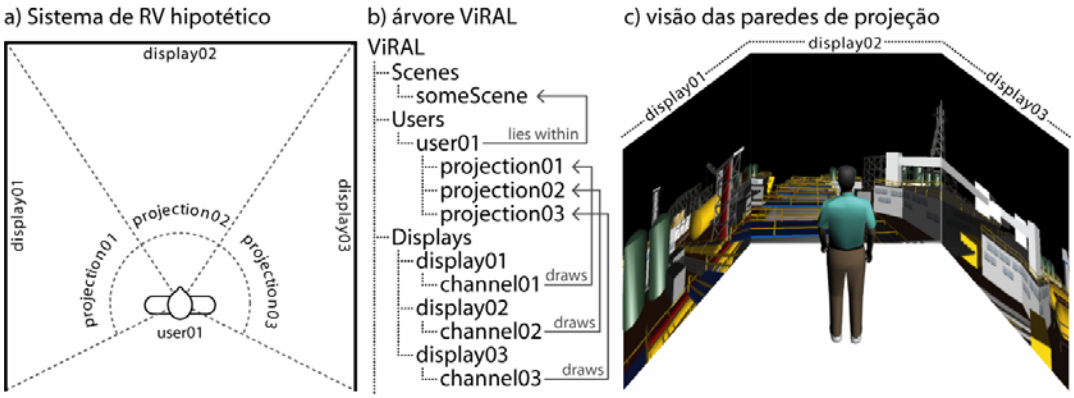

Figura 7: Uma árvore ViRAL refletindo um sistema de RV hipotético.

Uma *Projection* define uma visão de um *User* em um ambiente virtual. Suas propriedades incluem um volume de visão, uma transformação de visão e um plano de paralaxe. Um par *Projection*/*User* contém todos os dados necessários para renderizar um quadro de uma *Scene*. Na maioria dos sistemas de RV, várias *Projections* precisam ser definidas para um mesmo *User*; um exemplo está na Figura 7. *Projections* podem ser definidas manualmente ou calculadas automaticamente (Figura 8a). Quando em modo manual, um operador precisa especificar as propriedades da *Projection* com valores fixos. Quando em modo rastreado, a *Projection* recebe uma descrição física da tela de projeção, em coordenadas do mundo, e sabendo a posição real do usuário é possível derivar todos os parâmetros necessários. Este segundo modo é necessário em sistemas de RV não-triviais, onde o usuário pode se mover em relação a telas de projeção estáticas (e.g. um Workbench [Krüger 1995]).

Uma vez definidas, *Projections* precisam ser associadas a um canal de saída de vídeo para que sejam renderizadas e visualizadas. O sistema de exposição define como os gráficos são gerados e expostos por uma aplicação. Na nomenclatura do ViRAL, *screens* são canais lógicos de saída de vídeo com uma certa resolução em pixels. *Displays* representam contextos de renderização cujos quadros são expostos em uma *screen*; e *Channels* são utilizados para renderizar uma *Projection* em um *Display*.

Displays são meramente contextos de renderização, e dependem dos *Channels* para renderizar as *Projections*. Cada *Channel* referencia uma única *Projection*, que é renderizada em modo: mono, olho-esquerdo, olho-direito, *stereo double-buffer* ou *stereo quad-buffer*. Os modos definem como a distância interocular do *User* que possui a *Projection* afeta a renderização, e se o suporte a *stereo* por hardware será utilizado.

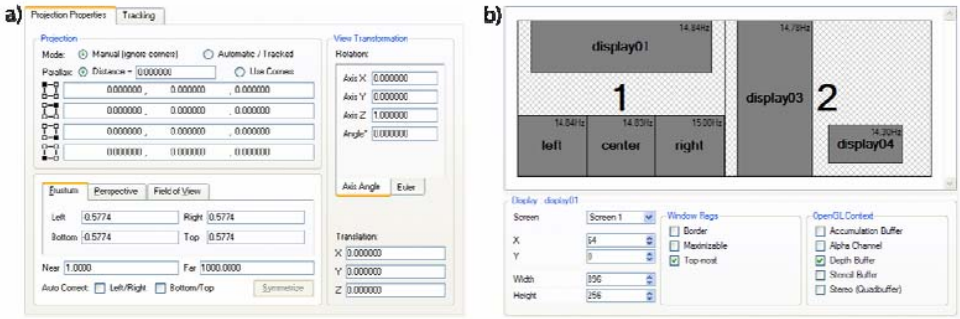

Figura 8: a) Tela de configuração de uma *Projection*, no sistema de usuários; b) Interface do sistema de exposição de uma aplicação-exemplo.

A Figura 8b mostra a interface do sistema de exposição de uma aplicação rodando em um PC com dois monitores. Os monitores são representados como *screens*, rotulados "1" e "2". Sobre as *screens* encontram-se seis *Displays*: três nomeados "left", "center" e "right", e outros com seus nomes padrões. Além de representar os *screens*, a interface exibe a posição e a taxa de quadros por segundo de cada *Display*.

3.5. Environments – Sistema de Persistência para Ambientes Virtuais

A árvore ViRAL é uma estrutura capaz de representar um ambiente virtual. Todos os dados pertinentes a uma simulação de RV fazem parte, direta ou indiretamente, dessa árvore. Fornecer persistência para ambientes virtuais se resume em tornar essas árvores persistentes; e esse é o objetivo do *EnvironmentSystem*.

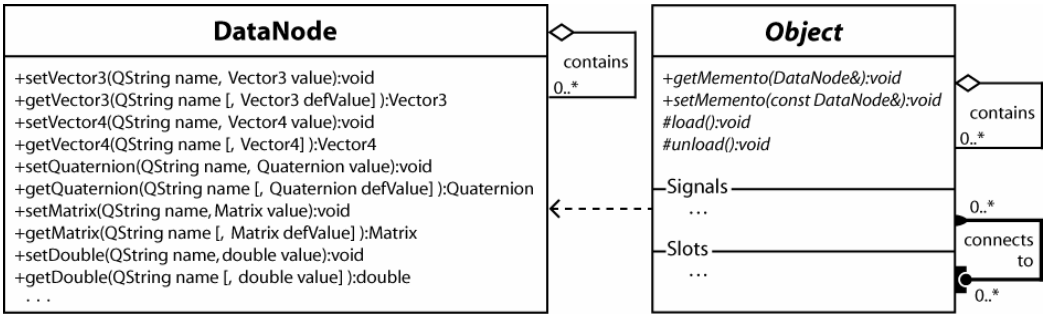

Figura 9: Esboço das classes *DataNode* e *Object*.

Um objeto *Environment* conserva uma árvore ViRAL utilizando uma estrutura de descrição similar a de um documento XML. É possível, a qualquer instante, capturar *snapshots* da ávore ViRAL para dentro de um *Environment*, e então salvar este em um arquivo. O inverso – carregar um arquivo de *Environment* e remontar a árvore ViRAL, pode ser feito sempre que não houver uma simulação ativa. Somente um *Environment* pode estar ativo numa aplicação em um dado momento, já que a ávore ViRAL é única.

Para a implementação de persistência, utilizou-se o *design pattern* Memento [Gamma 1995]. Como visto na Figura 9, objetos do ViRAL têm métodos para obter e restaurar seus estados encapsulados em um *DataNode*, que faz o papel do memento. Um *DataNode* é um dicionário capaz de armazenar todo tipo de dado, desde primitivas até quatérnios e matrizes. Ele pode conter outros *DataNodes*, formando uma árvore, e dessa forma um objeto ViRAL pode salvar e recuperar seus dados de forma modular.

Para salvar uma árvore ViRAL, visita-se ela em préordem coletando *DataNodes* e construindo uma árvore em paralelo. A árvore obtida é salva usando um formato de XML compactado. Para reconstruir uma árvore, criam-se instâncias de cada objeto e posteriormente recuperam-se seus estados repassando seus antigos *DataNodes*.

3.6. Sistemas de Plugins e Opções

Plugins têm um papel importantíssimo no framework: introduzir dinamicamente novos sistemas e componentes para as aplicações. Objetos do tipo *Plugin* são muito simples: servem basicamente para instanciar certos tipos de objetos, como *Scenes* (*ScenePlugin*) e *Devices* (*DevicePlugin*). O sistema de plugins permite que desenvolvedores definam livremente novos tipos de *Plugins* para estender seus componentes e sistemas.

O singleton *PluginSystem* funciona como um repositório geral de *Plugins*, sendo capaz de carregá-los automaticamente de bibliotecas dinâmicas localizadas em um conjunto pré-definido de diretórios. O singleton serve como um ponto de acesso a todos os *Plugins*, sendo capaz inclusive de listar plugins de certo tipo (e.g. *DevicePlugins*). Quase todos os sistemas são clientes do sistema de plugins. O *DeviceSystem*, por exemplo, consulta o *PluginSystem* para saber que *Devices* estão disponíveis para os operadores instanciarem, e dispõe uma lista com eles no diálogo de criação de *Devices*.

O sistema de opções (*OptionSystem*) é usado para conservar dados de forma independente de um *Environment*. Ele serve como um repositório de dados globais, funcionando tal qual um *DataNode*. Adicionalmente, este sistema provê uma tela de interface extensível projetada como um ponto central de edição de opções globais.

4. Utilizando o ViRAL

Essa seção retoma a aplicação de simulação de direção de automóvel, que foi citada algumas vezes ao longo do artigo. O objetivo é mostrar como o ViRAL pode ser usado na prática. Este exemplo utilizará um sistema de RV hipotético com três paredes de projeção e um único usuário, exatamente como na Figura 7. O usuário estaria sentado em um compartimento com um volante, pedais para acelerar e freiar, e uma alavanca de câmbio. Dados desses dispositivos são capturados por *Devices* e emitidos através de *signals* (Figura 10). O volante emite *signals* contendo um ângulo, em intervalos curtos de amostragem – assim como cada um dos pedais, emitindo seus níveis. O câmbio emite *signals* apenas quando é alterado pelo usuário, identificando a nova marcha escolhida.

Esta aplicação exige uma *Scene* projetada especialmente para simulações de direção. Os carros na cena seriam implementados como *SceneObjects*, e receberiam diretamente a entrada dos dispositivos de controle, por conexões com *Devices*. Cada *Car* teria quatro *slots*: um para receber o ângulo do volante, dois para os pedais e o quarto para receber mudanças de marcha, como mostrado na Figura 10.

Os carros, naturalmente, não estariam restritos aos dispositivos de entrada mencionados. Qualquer dispositivo com um eixo de rotação poderia ser usado como volante, o câmbio poderia ser controlado por botões, e os pedais poderiam usar botões ou qualquer dispositivo analógico. Quando um *signal* não é compatível com um *slot*, um *Device* intermediário pode sempre ser usado para transformar e adaptar os eventos.

O ViRAL está projetado para renderizar as visões de um *User*. Assim, para a simulação funcionar, um *User* deve seguir a posição do seu *Car* no ambiente virtual. A maneira mais direta de fazer isso com o ViRAL é usar o *Car* como dispositivo de entrada para o *User*, como um mecanismo de rastreamento. Sempre que o *Car* se move, um *signal* é emitido para atualizar a posição do seu *User* (como esboçado na Figura 10).

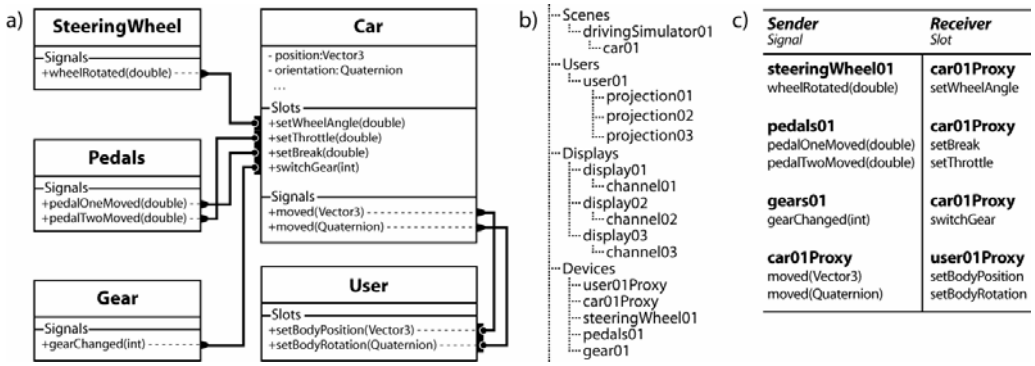

Figura 10: a) Principais componentes do exemplo, com seus *signals*, *slots* e conexões; b) Árvore ViRAL do exemplo; c) Tabela de conexões do exemplo.

Com os componentes *Scene* e *Devices* em mãos, configurar uma aplicação é apenas uma questão de criar os componentes no ViRAL. Por exemplo, para criar o simulador usando a aplicação ViRAL, um operador deve inicialmente criar um *Environment* vazio. Na interface de cenas, a *Scene* do simulador de direção deve ser criada e configurada, com um componente *Car* definido para ela. No sistema de usuários, um *User* deve ser criado com suas três *Projections* e associado à *Scene* do simulador. No sistema de exposição, três *Displays* precisariam ser criados – preenchendo três telas de projeção. Por fim, cada *Display* teria um *Channel* associado a sua devida *Projection*.

Neste ponto, a aplicação já é capaz de renderizar gráficos, mas ainda não é possível dirigir os carros – os *Devices* apropriados precisam ser criados e conectados. Pelo menos cinco *Devices* seriam necessários: três para a captura das entradas dos dispositivos reais e dois *Proxies* representando os componentes *Car* e *User*. Os últimos passos na preparação da simulação ocorreriam na interface de controle de conexões do sistema de dispositivos; os mesmos devem ser conectados como visto na Figura 10c.

5. Conclusão

Neste artigo foi apresentado o ViRAL, um *framework* de RV baseado em componentes gráficos. Ele está sendo usado para criar aplicações de RV operadas estritamente via interfaces gráficas, usando uma combinação de componentes gráficos e persistência ao invés de arquivos de configuração. O padrão *signal*/*slot* e a utilização de componentes mostraram-se muito adequados para a integração de dispositivos e aplicações de RV.

Atualmente, o ViRAL é usado principalmente em plataforma Windows, embora já tenha sido testado em estações SGI. Devido ao sistema de plugins, é relativamente simples adicionar suporte a novos dispositivos e engines de AVs. Em termos de dispositivos de entrada, já foram desenvolvidos plugins para mouses 3D, luvas P5, dispositivos de rastreamento da InterSense e um rastreador óptico [Silva 2004]. Em termos de sistema de projeção, o ViRAL já foi testado com capacetes, projetores estéreo ativos e passivos e sistemas de multi-projeção. As aplicações desenvolvidas vão desde pequenos jogos a complexos visualizadores de modelos CAD. A maior parte das *Scenes* têm sido desenvolvidas com o grafo de cena OpenSceneGraph [Osfield 2004].

O ViRAL está evoluindo e ganhando recursos mais complexos – como a distribuição de dispositivos, para permitir que componentes de instâncias remotas do ViRAL sejam conectados a componentes locais de forma transparente, pelo mesmo mecanismo de *signal/slot*. Nesta evolução, o crescimento do núcleo do framework será cuidadosamente controlado, de forma a manter seu principal objetivo: simplicidade.

6. Agradecimentos

A pesquisa em Realidade Virtual do Tecgraf/PUC-Rio é apoiada primordialmente pelo CENPES/PETROBRAS, pela FINEP e RNP. Alberto Raposo é bolsista de produtividade em pesquisa do CNPq, processo no 305015/02-8.

Referências

Arsenault, L., et al. (2001) “DIVERSE: a Software Toolkit to Integrate Distributed Simulations with Heterogeneous Virtual Environments”, Technical Report TR-01-10, Computer Science, Virginia Tech.

Bierbaum, A. D. (2000) “VR Juggler: A Virtual Platform for Virtual Reality Application Development”, Master Thesis, Iowa State University.

Cruz-Neira, C., et al. (1993) “Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE”, ACM Computer Graphics, 27(2): 135-142.

Gamma, E., et al. (1995) “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison-Wesley.

Krüger, W., et al. (1995) “The Responsive Workbench: A Virtual Work Environment”, IEEE Computer, 28(7): 42-48.

Osfield, R., Burns, D. (2004) “Open Scene Graph”, http://www.openscenegraph.org/

Silva, R. J. M. (2004) “Integração de um Dispositivo Óptico de Rastreamento a uma Ferramenta de Realidade Virtual”, Tese de Mestrado DI/PUC-Rio.

Szyperski, C. (1997) “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley.

Tramberend, H. (2001) “Avango: A Distributed Virtual Reality Framework”, In Proceedings of Afrigraph '01, ACM.

Trolltech (2004) “Qt Signals and Slots” http://doc.trolltech.com/3.3/signalsandslots.htm