VaiR: System Architecture of a Generic Virtual Reality Engine

Tom Fellmann & Manolya Kavakli
Department of Computing, Macquarie University, Sydney, NSW 2109, Australia
tom@ics.mq.edu.au, manolya@ics.mq.edu.au

Abstract

Investigating various virtual reality (VR) tools, we have described main components for a generic VR Engine. In this paper, we discuss the system architecture of a VR Engine (VaiR), and demonstrate the basic elements of this generic VR programming interface. The VaiR Engine integrates VR hardware and software within a graphics Application Programming Interface (API) (e.g. OpenSceneGraph). The main advantage of the VaiR Engine is the ability to use stereoscopic goggles, trackers, head mounted displays, etc with a number of 3D Modeling and Animation Packages (e.g. 3ds Max and Softimage) and scripting languages (e.g., XML). VaiR combines the important characteristics of many other VR tools and brings them together to generate a more powerful tool. Thus, it provides us a suitable ground to build a flexible multi-purpose VR Engine.

1. Introduction

Virtual reality is the simulation of a realistic world. Foley et al [1] describes VR as "completely computer-generated environments with realistic appearance, behavior and interaction techniques". A virtual reality environment provides three main facilities:

- Immersion that generates a feeling of presence in the environment with the users senses.
- Interaction that allows the user to interact with environmental objects (with the special case navigation).
- Imagination that generates a feeling of being part of the environment. The imagination is deeply dependent on the immersion and interaction.

All these facilities will be satisfied with specific software and hardware devices. The aim of a VR engine is to bring all the parts of a virtual reality environment together.

In this paper, we describe a VR Engine (VaiR) to be used as a base for a CAVE-like projection system called The Macquarie Engine with the participation of various departments within Macquarie University in a number of projects. A snapshot of head tracking in the VaiR Engine can be seen in Figure1.

The Macquarie Engine has a modular system architecture to connect VR hardware and software in a very flexible way. The Macquarie Engine integrates the following components:

- a 3D projection system (projection cube to generate stereoscopic view of the virtual world),
- a video tracking system (DVD cameras and recorders),
- a motion tracking system (Motion Builder and motion capture software),
- game engines,
- CAD packages (e.g., 3ds Max, Maya, Softimage),
- and databases with project-specific VR hardware and software.

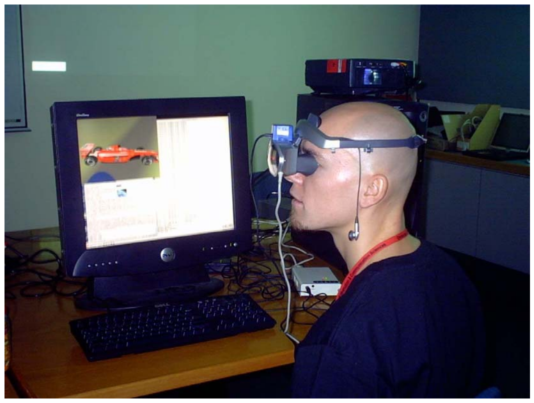


Figure 1. Head tracking in VaiR

The 3D projection system includes a head and hand tracking system to produce the stereo perspective and to isolate the position and orientation of a 3D input device. We plan to integrate a motion capture suit with this projection system.

The Macquarie Engine will allow the user to freely navigate, immerse, and interact with the objects in a virtual world. Implementation of the Macquarie Engine involves the development of an interface between the projection system, game engines, 3D Modeling packages, a motion capture system (Motion Builder and motion capture suit), and other VR tools. VaiR Engine provides us a suitable ground to build the Macquarie Engine.

2. Virtual Reality Tools

There are a number of virtual reality software tools currently available, serving different purposes. Some of these tools bring virtual reality on the web [e.g., 2], use virtual reality tools in games [e.g., 3] or just make a modular package for rapid prototyping of virtual reality applications [e.g., 4, 5, and 6]. Some of them have interfaces for virtual reality hardware resources, like [7] or [8], others place more emphasis on the physics [9]. The goal of VaiR is to provide a flexible environment for the development of various simulations. VaiR combines the important characteristics of above-mentioned VR tools together to generate a more powerful tool.

3. VaiR Engine

VaiR is an object-oriented system written in C++. With the help of the Standard Template Library (STL), VaiR has a reasonable performance at runtime.

The main design objective in the development of VaiR is to: Make everything as abstract as possible. This means that every concept used in VaiR is abstracted to be changeable. With the abstraction of all parts of the framework, we provide flexibility and portability from one platform to the other as well as extendibility of software and hardware resources.

As seen in Fig. 2, VaiR is a system that consists of 6 main parts. These parts are the VaiR-Core, the Graphics API that is the scenegraph library, the SW/HW manager, the configuration manager, the interaction manager and the VaiR-GUI. We discuss these parts in the following chapter.

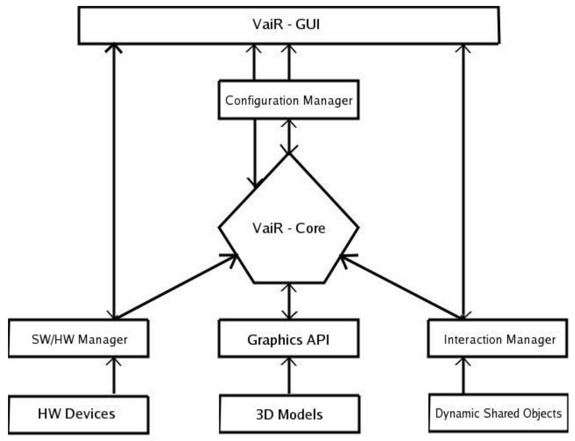


Figure 2. System Architecture of VaiR

3.1. VaiR Core

The VaiR Core is the connection between the components. The major content of the core is the abstraction of the scenegraph library. Through the abstraction of the scenegraph, we have the facility to manipulate our own scenegraph, without focusing on which scenegraph library is actually used. The abstraction works in a simple way. Every abstracted class/function in VaiR, calls the specific class/function in the scenegraph library.

The VaiR core provides our engine with its own viewer. The viewer is the display manager for our VR scene and controls what to display to the user. VaiR provides different levels of information, depending on the user. The “normal” user has the choice between 2 levels (On/Off), and the developer of VaiR can change the output of information in a scale of 0 to 4. At level 0 no information will be displayed, while at level 4 all information will be available to the user. This provides a facility for debugging the system.

3.2. Graphics API

The Graphics API is an external part of VaiR. It is a full graphics library (scenegraph library), which includes the tasks of the organization and representation such as the geometry, rendering and loading of different models in a scene. The abstraction of the scenegraph in VaiR is the connection between our system and the graphics library.

3.3. SW/HW Manager

The SW/HW Manager consists of 2 parts: the software interface and the hardware interface. The software interface is the general interface for hardware devices. It consists of various classes, such as buttons and sliders to allow interfacing with a variety of hardware tools such as mice, keyboards, space balls, data gloves and trackers to name a few. The hardware interface represents the specific classes for the hardware. For example, if a specific Tracker is used, then there is a specific class dedicated for this purpose. Through splitting the interfaces between software and hardware, we have achieved the ability to use various types of hardware devices in our VR system.

3.4. Configuration Manager

VaiR information is stored in XML format. This way, we store the whole scenegraph structure with the attributes, the settings for the VR environment and the hardware in an XML file. It is possible to load the script written in XML with the configuration manager and build up saved VR scenes. It is also possible to write the XML file first to build the VR scene. Through the software tools available for manipulating or generating XML files, the user of VaiR has the power to make individual and complex VR scenes.

3.5. Interaction Manager

Within VaiR there are 2 types of interaction:

- Predefined interaction
    - navigation routes,
    - animation paths, etc.
- Self-defined (complex) interaction

Navigation routes are a subpart of predefined interaction. Within a number of navigation routes, the user specifies one as the current path. All pre-implemented interaction in VaiR is predefined interaction. Self-defined interaction is relatively complex and described by the user's own library. The user of VaiR has the option of describing the user's own acts in a Dynamic Shared Object (DSO; a unix specific compiled library) or a dynamic link library (DLL; a microsoft windows specific compiled library), which can be loaded into VaiR at runtime. To illustrate with an example, imagine we have two models in our VR scene. We would like to have a specific behavior between these models. We would describe the behavior in our specific compiled library and load the library at any time we wish during the execution of the VR program. Once the library is loaded, the behavior will be executed. This facility gives the user the opportunity to create complex interactions within the VR environment.

3.6. VaiR GUI

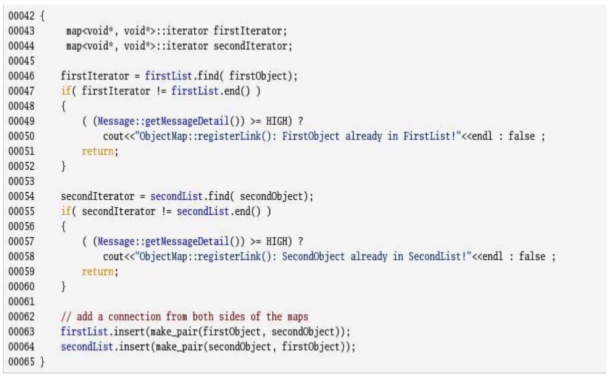
A graphical user interface (GUI) allows the user to build his/her own scene in a relatively short time. In the VaiR GUI, the whole scenegraph is shown as a tree and the user can manipulate the scenegraph as desired. The user can arrange the settings for the whole VR environment and the input/output devices. The standard user does not have to know the details of the VR engine, since the graphical user interface is self-explanatory.

4. Implementation

Following sections describes how VaiR system has been implemented integrating the main elements of the system architecture described above.

4.1. Implementation of The VaiR Core

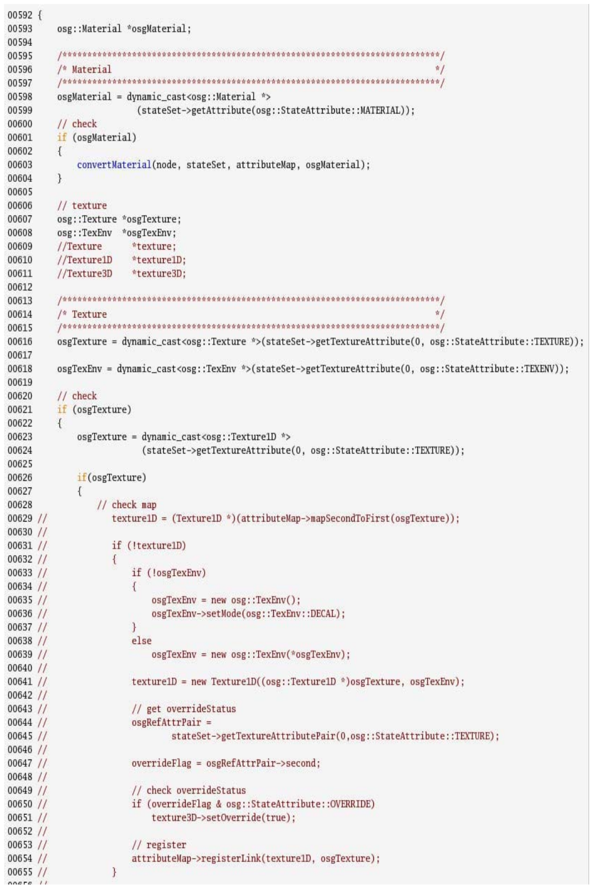
The connection between the VaiR objects and the graphics API is via an object map. This concept has been adapted from [6]. An object map makes it easy to have an overview about scenegraph objects. With an object map, every time we use an object from our scenegraph, we retrieve a similar object in the scenegraph library. The concept works in the whole abstraction of the scenegraph, including the nodes, the attributes, etc. Fig 3 shows the implementation of the Object Map (with the help of the STL), as well as the use of the information levels.



```
{
     map<void*, void*>::iterator firstIterator;
     map<void*, void*>::iterator secondIterator;

    firstIterator = firstList.find( firstObject);
    if( firstIterator != firstList.end() )
    {
        ( (Message::getMessageDetail()) >= HIGH) ?
            cout<<"ObjectMap::registerLink(): FirstObject already in FirstList!"<<endl : false ;
        return;
    }

    secondIterator = secondList.find( secondObject);
    if( secondIterator != secondList.end() )
    {
        ( (Message::getMessageDetail()) >= HIGH) ?
            cout<<"ObjectMap::registerLink(): SecondObject already in SecondList!"<<endl : false ;
        return;
    }

    // add a connection from both sides of the maps
    firstList.insert(make_pair(firstObject, secondObject));
    secondList.insert(make_pair(secondObject, firstObject));
}
```

Figure 3. Object Map

Another important class in the VaiR Core converts the scenegraph from the graphics API. Fig. 4 shows a portion of that. Although it is planned to have every state's attributes handled, Texture1D and Texture3D attributes are not available at the current version of VaiR.

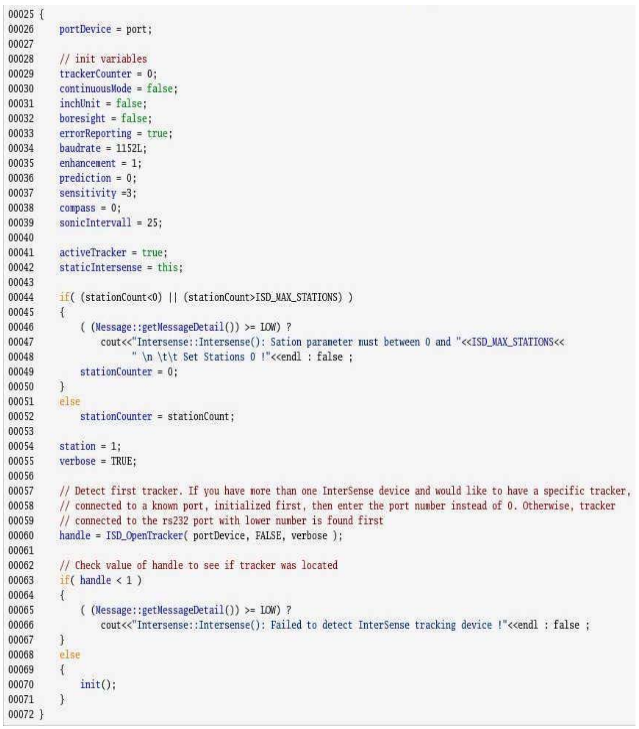


```
{
    osg::Material *osgMaterial;

    /**************************************************************************/
    /* Material                                                               */
    /**************************************************************************/
    osgMaterial = dynamic_cast<osg::Material *>
                    (stateSet->getAttribute(osg::StateAttribute::MATERIAL));
    // check
    if (osgMaterial)
    {
        convertMaterial(node, stateSet, attributeMap, osgMaterial);
    }

    // texture
    osg::Texture *osgTexture;
    osg::TexEnv  *osgTexEnv;
    //Texture      *texture;
    //Texture1D    *texture1D;
    //Texture3D    *texture3D;

    /**************************************************************************/
    /* Texture                                                                */
    /**************************************************************************/
    osgTexture = dynamic_cast<osg::Texture *>(stateSet->getTextureAttribute(0, osg::StateAttribute::TEXTURE));

    osgTexEnv = dynamic_cast<osg::TexEnv *>(stateSet->getTextureAttribute(0, osg::StateAttribute::TEXENV));

    // check
    if (osgTexture)
    {
        osgTexture = dynamic_cast<osg::Texture1D *>
                        (stateSet->getTextureAttribute(0, osg::StateAttribute::TEXTURE));

        if(osgTexture)
        {
            // check map
//              texture1D = (Texture1D *)(attributeMap->mapSecondToFirst(osgTexture));
//
//              if (!texture1D)
//              {
//                  if (!osgTexEnv)
//                  {
//                      osgTexEnv = new osg::TexEnv();
//                      osgTexEnv->setMode(osg::TexEnv::DECAL);
//                  }
//                  else
//                      osgTexEnv = new osg::TexEnv(*osgTexEnv);
//
//                  texture1D = new Texture1D((osg::Texture1D *)osgTexture, osgTexEnv);
//
//                  // get overrideStatus
//                  osgRefAttrPair =
//                          stateSet->getTextureAttributePair(0,osg::StateAttribute::TEXTURE);
//
//                  overrideFlag = osgRefAttrPair->second;
//
//                  // check overrideStatus
//                  if (overrideFlag & osg::StateAttribute::OVERRIDE)
//                      texture3D->setOverride(true);
//
//                  // register
//                  attributeMap->registerLink(texture1D, osgTexture);
//              }
```

Figure 4. Converting Graphics API scenegraph to an application specific scenegraph

4.2. Implementation of SW/HW Manager

Fig. 5 shows a portion of the implementation of a specific tracker. It describes the initial settings of the tracker.

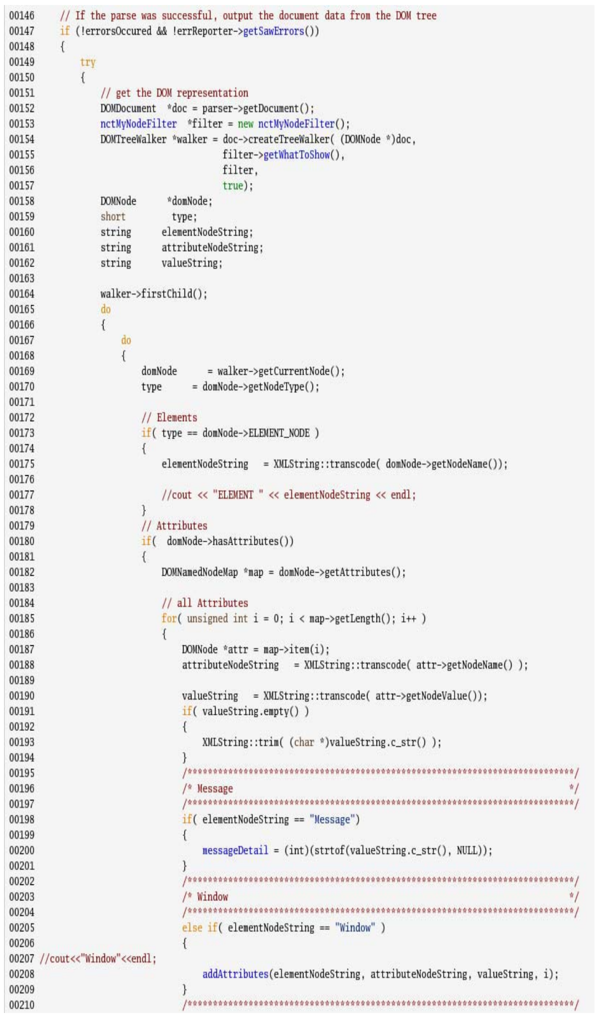


```
{
    portDevice = port;

    // init variables
    trackerCounter = 0;
    continuousMode = false;
    inchUnit = false;
    boresight = false;
    errorReporting = true;
    baudrate = 1152L;
    enhancement = 1;
    prediction = 0;
    sensitivity =3;
    compass = 0;
    sonicIntervall = 25;

    activeTracker = true;
    staticIntersense = this;

    if( (stationCount<0) || (stationCount>ISD_MAX_STATIONS) )
    {
        ( (Message::getMessageDetail()) >= LOW) ?
            cout<<"Intersense::Intersense(): Sation parameter must between 0 and "<<ISD_MAX_STATIONS<<
                  " \n \t\t Set Stations 0 !"<<endl : false ;
        stationCounter = 0;
    }
    else
        stationCounter = stationCount;

    station = 1;
    verbose = TRUE;

    // Detect first tracker. If you have more than one InterSense device and would like to have a specific tracker,
    // connected to a known port, initialized first, then enter the port number instead of 0. Otherwise, tracker
    // connected to the rs232 port with lower number is found first
    handle = ISD_OpenTracker( portDevice, FALSE, verbose );

    // Check value of handle to see if tracker was located
    if( handle < 1 )
    {
        ( (Message::getMessageDetail()) >= LOW) ?
            cout<<"Intersense::Intersense(): Failed to detect InterSense tracking device !"<<endl : false ;
    }
    else
    {
        init();
    }
}
```

Figure 5. Implementation of a specific Tracker

4.3. Configuration Manager

Parsing XML files is possible using the Document Object Model (DOM)-Parser. Fig. 6 demonstrates that the DOM parser is searching for specific words, and once it finds a word, it calls a function for saving the attributes.

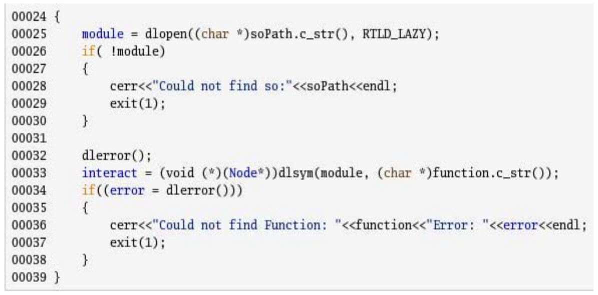


```
    // If the parse was successful, output the document data from the DOM tree
    if (!errorsOccured && !errReporter->getSawErrors())
    {
        try
        {
            // get the DOM representation
            DOMDocument  *doc = parser->getDocument();
            nctMyNodeFilter  *filter = new nctMyNodeFilter();
            DOMTreeWalker *walker = doc->createTreeWalker( (DOMNode *)doc,
                                    filter->getWhatToShow(),
                                    filter,
                                    true);
            DOMNode      *domNode;
            short         type;
            string       elementNodeString;
            string       attributeNodeString;
            string       valueString;

            walker->firstChild();
            do
            {
                do
                {
                    domNode      = walker->getCurrentNode();
                    type      = domNode->getNodeType();

                    // Elements
                    if( type == domNode->ELEMENT_NODE )
                    {
                        elementNodeString   = XMLString::transcode( domNode->getNodeName());

                        //cout << "ELEMENT " << elementNodeString << endl;
                    }
                    // Attributes
                    if(  domNode->hasAttributes())
                    {
                        DOMNamedNodeMap *map = domNode->getAttributes();

                        // all Attributes
                        for( unsigned int i = 0; i < map->getLength(); i++ )
                        {
                            DOMNode *attr = map->item(i);
                            attributeNodeString   = XMLString::transcode( attr->getNodeName() );

                            valueString   = XMLString::transcode( attr->getNodeValue());
                            if( valueString.empty() )
                            {
                                XMLString::trim( (char *)valueString.c_str() );
                            }
                            /***************************************************************************/
                            /* Message                                                                 */
                            /***************************************************************************/
                            if( elementNodeString == "Message")
                            {
                                messageDetail = (int)(strtof(valueString.c_str(), NULL));
                            }
                            /***************************************************************************/
                            /* Window                                                                  */
                            /***************************************************************************/
                            else if( elementNodeString == "Window" )
                            {
//cout<<"Window"<<endl;
                                addAttributes(elementNodeString, attributeNodeString, valueString, i);
                            }
                            /***************************************************************************/
```

Figure 6. Parsing XML-File

4.4. Interaction Manager

Fig. 7 demonstrates a portion of the Interaction Manager including the main function.

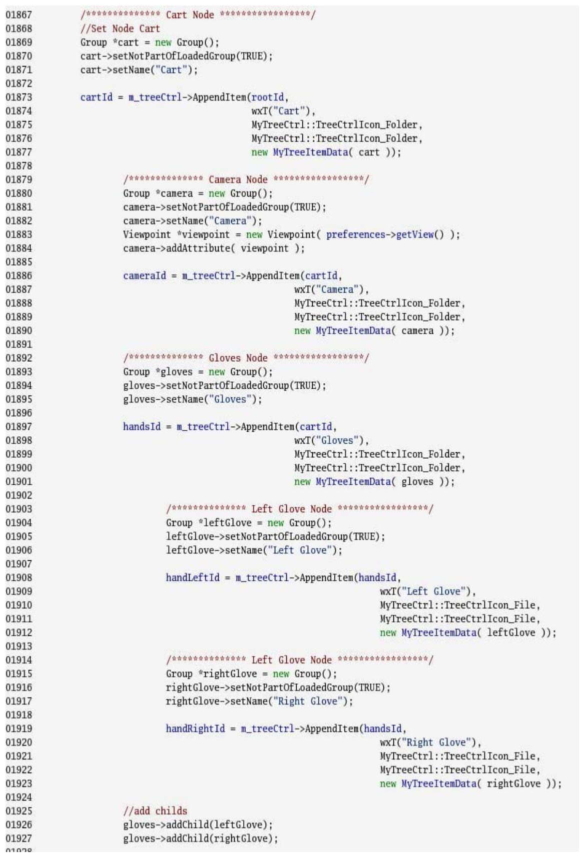


```
{
    module = dlopen((char *)soPath.c_str(), RTLD_LAZY);
    if( !module)
    {
        cerr<<"Could not find so:"<<soPath<<endl;
        exit(1);
    }

    dlerror();
    interact = (void (*)(Node*))dlsym(module, (char *)function.c_str());
    if((error = dlerror()))
    {
        cerr<<"Could not find Function: "<<function<<"Error: "<<error<<endl;
        exit(1);
    }
}
```

Figure 7. Loading DSO's

4.5. VaiR GUI

Fig. 8 is a portion of the scenegraph in our graphical user interface. The figure demonstrates the cart concept being pre-implemented in the GUI. The cart concept assumes that the user of a virtual environment is sitting on a carpet. However, the user can move in all directions. His view is the normal view with his eyes, and his hands are the virtual abstraction of these.



```
/************** Cart Node *****************/
//Set Node Cart
Group *cart = new Group();
cart->setNotPartOfLoadedGroup(TRUE);
cart->setName("Cart");

cartId = m_treeCtrl->AppendItem(rootId,
                         wxT("Cart"),
                         MyTreeCtrl::TreeCtrlIcon_Folder,
                         MyTreeCtrl::TreeCtrlIcon_Folder,
                         new MyTreeItemData( cart ));

    /************** Camera Node *****************/
    Group *camera = new Group();
    camera->setNotPartOfLoadedGroup(TRUE);
    camera->setName("Camera");
    Viewpoint *viewpoint = new Viewpoint( preferences->getView() );
    camera->addAttribute( viewpoint );

    cameraId = m_treeCtrl->AppendItem(cartId,
                             wxT("Camera"),
                             MyTreeCtrl::TreeCtrlIcon_Folder,
                             MyTreeCtrl::TreeCtrlIcon_Folder,
                             new MyTreeItemData( camera ));

    /************** Gloves Node *****************/
    Group *gloves = new Group();
    gloves->setNotPartOfLoadedGroup(TRUE);
    gloves->setName("Gloves");

    handsId = m_treeCtrl->AppendItem(cartId,
                             wxT("Gloves"),
                             MyTreeCtrl::TreeCtrlIcon_Folder,
                             MyTreeCtrl::TreeCtrlIcon_Folder,
                             new MyTreeItemData( gloves ));

        /************** Left Glove Node *****************/
        Group *leftGlove = new Group();
        leftGlove->setNotPartOfLoadedGroup(TRUE);
        leftGlove->setName("Left Glove");

        handLeftId = m_treeCtrl->AppendItem(handsId,
                                     wxT("Left Glove"),
                                     MyTreeCtrl::TreeCtrlIcon_File,
                                     MyTreeCtrl::TreeCtrlIcon_File,
                                     new MyTreeItemData( leftGlove ));

        /************** Left Glove Node *****************/
        Group *rightGlove = new Group();
        rightGlove->setNotPartOfLoadedGroup(TRUE);
        rightGlove->setName("Right Glove");

        handRightId = m_treeCtrl->AppendItem(handsId,
                                     wxT("Right Glove"),
                                     MyTreeCtrl::TreeCtrlIcon_File,
                                     MyTreeCtrl::TreeCtrlIcon_File,
                                     new MyTreeItemData( rightGlove ));

    //add childs
    gloves->addChild(leftGlove);
    gloves->addChild(rightGlove);
```

Figure 8. Visualization of the scenegraph

5. Testing

The VaiR-GUI was tested in a modular manner during development. Following the test results the configuration manager was adapted to the demands of the user. Our test results clearly indicate the advantages of having a GUI as an interface between the user and the VR system.

6. Conclusion

In this project, we integrated main components of a generic VR Engine. Our test results indicate that the

VaiR engine has proven to be a useful tool for a wide range of users (from digital artists to programmers). The use of a generic VR Engine will allow its user to focus on the research questions, without spending too much time with the integration of system components and the architecture of a VR Engine.

7. Acknowledgment

This project is funded by Australian Research Council Discovery Grant coded DP0558852 and Macquarie University Research Infrastructure Grant titled "Virtual Reality Engine". The fundamentals of this generic VR engine are adapted from Tom Fellmann's Diploma Thesis [10] at the University of Leipzig. Special thanks to Dr. Jens-Peer Kuska, who supervised this thesis, and to the industry partner, EADS Deutschland/Ottobrunn, which provided the task and the facilities to build up a VR engine prototype.

8. References

[1] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes & Richard L. Phillips, Computer Graphics: Principles and Practice in C(2nd Edition), volume 2, Addison-Wesley, USA, 1995.

[2] Dax, P. et all, VREng, http://vreng.enst.fr/net/vreng/html/, last accessed on 19.10.2005.

[3] Bíza, M., Cernohorsky, P., Fiala, M., VROnline, http://sourceforge.net/projects/vronline, last accessed on 19.10.2005.

[4] Nagel, H. R., Vittrup, M., Bovbjerg, S., Vester, J.P., VR++, http://www.idi.ntnu.no/~hrn/vrpp/, last accessed on 19.10.2005.

[5] Just, C., Bierbaum, A., Baker, A, Cruz-Neira, C., (1998), VR Juggler: A Framework for Virtual Reality Development. The 2nd Immersive Projection Technology Workshop (IPT98), Ames.

[6] Daly, J., Kline, B., Martin, G. A., VESS: Coordinating Graphics, Audio, and User Interaction in Virtual Reality Applications. In Proceedings of the IEEE Virtual Reality 2002 Conference, Orlando, FL, March 2002.

[7] Quest3D, http://www.quest3d.com/, last accessed on 19.10.2005.

[8] vrcom, Virtual Design 2, http://www.vrcom.de, last accessed on 19.10.2005.

[9] EZPhysics , http://ezphysics.org/, last accessed on 19.10.2005.

[10] Fellmann, T., Konzept einer Virtual-Reality Laufzeitumgebung und Implementierung des Rahmenwerkes basierend auf einem Open-Source Szenengraphen, Diploma Thesis, University of Leipzig, 2004.