Fourth Eurographics Workshop on Parallel Graphics and Visualization (2002)
D. Bartz, X. Pueyo, E. Reinhard (Editors)

Approach for software development of parallel real-time VE systems on heterogenous clusters

C. Winkelholz and T. Alexander

Research Institute for Communication, Information Processing, and Ergonomics,Wachtberg, GERMANY

Abstract
This paper presents our approach for the development of software for parallel real-time virtual environment systems (VE) running on heterogenous clusters of computers. This approach is based on a framework we have developed to facilitate the set-up of immersive virtual environment systems using single components coupled by an isolated local network. The framework provides parallel rendering of multiple projection screens and parallel execution of application and interaction tasks on components spread across a cluster. Main concept of the approach discussed in this paper is to use the virtual reality modeling language (VRML) as an interface definition language (IDL) for the parallel and distributed virtual environment system. An IDL-compiler generates skeleton-code for the implementations of the script nodes specified in a VRML-file. Components created this way can be reused in any VE by declaring the same interfaces. Instances of the implemented interfaces can reside in any application. By this approach commercial-of-the-shelf software can easily be integrated into a VE application. In this connection we discuss the underlying framework and software development process. Furthermore, the implementation of a VE system for a geographic information system (GIS) based on this approach is shown. It is emphasized that the components are used in various different applications.

Categories and Subject Descriptors (according to ACM CCS): D.1.3 [Programming Techniques]: Concurrent Programming D.2.11 [Software Engineering]: Software Architectures D.2.12 [Software Engineering]: Interoperability D.2.13 [Software Engineering]: Reusable Software I.3.3 [Computer Graphics]: Graphics Systems

1. Introduction

Virtual Environments (VE) are characterized as a computer-based generation of scenes of abstract or realistic environments, which can be perceived and interacted with consistently[1]. Immersive virtual environment systems have to compute the spatial superposition of user and virtual environment in real-time. The response time and update rate of the system have to be high enough to generate the experience of continuity and a perceptible 3D-environment. This requires not only 3D-based real-time computing and rendering systems but also real-time spatial registration of the user and his behavior. The implementation of interaction techniques with virtual environments requires the computation of time-consuming tasks for determining intersection and collision of objects with the virtual environment. Because of these high requirements on the processing power VE systems are usually realized with specialized and expensive parallel mainframes like Onyx infinite-reality systems from SGI. Recently more and more systems for PC clusters have occurred. These systems appear to be more cost effective and scalable. But beside the issue to realize such systems in general, questions still remain on how to design an appropriate programming environment for this kind of parallel and distributed systems.

When designing an object-oriented application, the programmer usually starts with creating high-level domain-dependent abstractions and transfers them into object classes. These classes are related to each other by inheritance or composition. Inheritance can only be used on a code-implementation-level, whereas composition can be used at runtime. Therefore composition is commonly used at a higher level of implementation. Containers of objects linked together by composition at runtime are also called software components. To allow the composition of components of different vendors across different platforms, the invocation of methods has to be performed through a standardized protocol. The software implementing such a protocol is called middleware. When middleware is commonly available for acquisition or purchase, it becomes commercial-off-the-shelf (COTS). Popular object oriented middleware is Microsoft's ActiveX and some implementations of CORBA[2] from different vendors. To achieve cross-platform interoperability the interfaces and the implementation of an object is handled separately. The interfaces are defined in a specific interface definition language (IDL). An IDL-compiler generates skeleton-code for the implementation of the object itself and stub-code for proxy-objects that can be used in other application to have access on the services provided by the implemented objects. Instances of the implemented objects can reside in arbitrary applications. A proxy can be assigned to the instance of an object with an appropriate interface at runtime. This makes the systems developed by such approaches highly scaleable, reliable, and easy to configure. The middlewares mentioned are developed for systems that are distributed across a local area network or a wide area network. Furthermore, they are only designed to have access to objects distributed in several components and not to implement real time parallel applications.

The aim of the framework discussed in this paper is to transfer the advantages of these established general approaches to parallel real time virtual environment systems on a cluster. Requirements of the systems addressed are different to the conventional application area of these approaches. Main differences are based on the demand for very low latency and the necessity to distribute the same data from one component to multiple components. ActiveX or middleware based on CORBA use point to point connections. This make them unsuitable for the broadcasts of data needed for a VE system. Data can only be broadcasted consecutively. This scales badly with data size and number of components that have to be supplied with the data. The aim of our developed framework is to facilitate set-ups of immersive virtual environment systems like Workbenches[3] and CAVEs [4] by assembling single components connected by an isolated local network. For this purpose we exploit the instance, that in modern stand-alone 3D-applications dataflow graphs are found and the whole application is scheduled in frames. The dataflow graphs are a high-level expression of the fundamental abstractions and operations in the system. In this context we adapt the dataflow graph to a particular meta-computing environment. This includes deciding which nodes in the dataflow graph should be executed on a dedicated workstation in the cluster. The decision how to distribute the activity of the dataflow graph is definitely a lower-level issue in the context of application development. As a matter of fact, task distribution on a heterogeneous cluster is always constrained by the actual system the VE application should be implemented on. Therefore a programming environment should separate these concerns. While implementing on a high-level abstraction the application programmer building an interactive VE should not be concerned about the distribution.

2. Related Work

There have been various approaches to distribute virtual environments. Commonly they were used for large-scale multi-user environments like the DIS/MMA[5] and DIVE[6]. An overview of the state of the art of such systems is given in [7]. These approaches are designed to visualize a distributed virtual environment to multiple spatially separated users across a wide area network (WAN). In this case the requirements for synchronization and low latency are not so strict as for single user purposes. In contrast to these large-scale multi-user environment our focus is to substitute the expensive high-end graphic mainframes commonly used in single user set-ups by a cluster of PCs. Applications for these high-end graphic real-time systems are implemented using Performer[8]. Performer allows the distribution of an graphical application to several processors in a parallel computer, but not on a cluster of PCs.

Recently a lot of research work has been published on VE systems distributed on PC clusters. Mainly there are two approaches that can be distinguished. The first approach focuses on the distribution of the rendering task itself. The implementation of such an approach is WireGL[9]. Here the display is divided into several tiles that are rendered from different servers. The protocol for the communication of the client to the servers is based on the level of the graphic application programming interface (API), in this case OpenGL. The OpenGL-function calls are not directed to the local host, but after some pre-processing and compression, to the servers. The advantage of this approach is the portability of exciting graphics applications that uses OpenGL. But a VE application does not only consist of rendering tasks. In addition input devices have to be integrated and a lot of processing has to be done to integrate the users interaction into the virtual environment. Therefore the second approach provides the programmer with the concept of a shared scene-graph, accessible from all processes forming a distributed application. Each process owns a local copy of the scene graph and the contained state information, which is kept synchronized. There are several works based on this approach like Avocado[10], Lightning[11], Distributed Open Inventor[12] (DIV), SGI Graphic Cluster[13] and NetJuggler[14]. But in their current implementation most of them like [11, 13, 14] simply multiply the application and broadcast the data of the input devices. In contrast to this, with the approach presented in this paper even the computation of single interaction and application specific tasks can be assigned to dedicated workstations. Avocado and DIV allow different distributed tasks to change the shared scene-graph, but the way tasks interact among each other and with the scene-graph is hidden in the implementation. Our approach separates the interfaces through which the tasks of different components interacts from their

implementation. This approach is established in the development of common application components using CORBA or ActiveX. However, these common middlewares, do not provide performance required for real time VE systems.
With regard to parallelization the programming model of our work is more related to the work of Rischbeck and Watson[15]. They adopt the programming model of coexisting active and passive objects similar to ProActive[16] to implement a parallel VRML Server. But their framework focuses on the distribution of many fine-grained script-objects that do not need a synchronised local copy of the world model. In contrast to this our framework provides every node in the cluster with a synchronized model. As it will be explained in the next section we achieve this by distinguishing three roles of an object: active, passive, and neutral.

3. Cluster Event Broker Architecture

The implementation of a virtual environment is typically divided into an application task and a rendering task. The rendering task can further be divided into a culling and a drawing task. The application, culling and drawing tasks can be executed in a kind of pipeline parallelism. In a setup with multiple projection screens the rendering tasks for each projection screen are executed in a kind of task parallelism. The application task commonly is further divided into several tasks that mainly apply to the computation of the interaction with the virtual environment and the generation of the virtual environment itself. On the division of the application task into subtasks all kind of parallelism may occur.

For a VE system designed to improve the immersion into the virtual environment it is essential, that all the rendering and interaction tasks use exactly the same model of the virtual environment. Because of the limited bandwidth of the network, local copies of the model have to be available in each component, and it has to be ensured that the models are synchronized each frame. The virtual environment seen as 3D graphical user interface (GUI) is mainly event-driven corresponding to conventional 2D GUIs. Interaction devices supply the application with position and hit events. These events are used to trigger the appearance of the GUI and the data displayed. In 2D GUIs a desktop mouse is frequently used as an interaction device. In VEs, data gloves or 6DOF styluses are used as main input devices. In dependence on the implemented interaction metaphors suitable event types can be defined. Thus, the interaction with an VE can be modeled by dataflow graphs. The nodes of the dataflow graph are either scene-objects which determines the appearance of an object in the virtual environment or script-objects which perform further computation on the data received. Therefore scene-objects are the leaves of the dataflow graph. The root nodes are script-objects that read data from physical input devices. Because of this strict description of the interaction and the simulation in a dataflow graph with root and leaves, the problem of synchronizing the models on different nodes on the cluster reduces to the problem to direct the events appropriately. Sending an event is like a one-way function call. In a VE simulation the dataflow is scheduled in frames. Each node possesses input interfaces to receive events and output interfaces to send events. At the beginning of each frame the nodes route changed data of their output interfaces to the input interfaces of the connected nodes. Then the script-objects processes the data and the scene-objects are rendered. There are two possible strategies how to procceed if the computation of a script-object changes the values of its output interfaces: The data can be routed in the same frame or be delayed until the next frame. The first strategy should be used if computation time is only a small fraction of the cycle time of the simulation loop. This avoids unnecessary increase of latency in this branch of the dataflow. If computation time is comparable long to the cycle time of the simulation loop also the second strategy is appropriate. This is the starting point of our framework to distribute and parallelize the VE simulation.

The computation of the script-objects is directed to single components. Each component instantiates the whole model. The routing of events is executed in each component. The framework distinguishes between three roles of a script-object. A script-object can be neutral, active or passive. The computation of a neutral script-object is performed in every component. The computation of an active script-object is performed only in one dedicated component. In the other components the corresponding script-objects of an active script-object are passive. The data of the output interfaces of active script-objects are distributed to the corresponding passive script-objects at the very beginning of each frame, before the routing in the components is performed. Therefore passive script-objects do not have to perform the computation. The data of the output interfaces of all correlated script-objects is identical when the routing is performed in the components. In this way the model of the virtual environment is kept synchronized during the simulation. Figure 1 and Figure 2 illustrates this. At this point it has to be emphasized that in our terms the passive objects do not correspond to the passive objects in the work of Caromel et al.[16]. In fact in our terms the neutral objects correspond to the passive objects in their work. The role of a script-object does not have to be assigned at compile time but at the initialisation of the virtual environment system. During the implementation and design of the virtual environment the developer does not have to be concerned about the distribution of the final system.

The mechanism of distributing the events is hidden by some kind of middleware. Core of the middleware is one object we call an "Event Broker" (EB). The EB drives the simulation loop and exchanges the data of the interfaces. In each component there is an instance of an EB. Connected to the event broker is a model view controller (MVC) framework. The views lock the output to the physical output devices until they receive a swap signal from the event broker. This

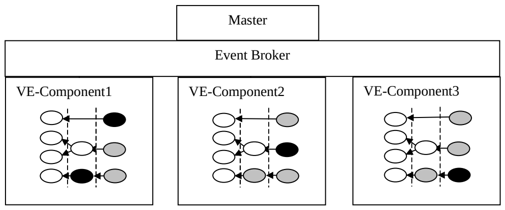

Figure 1: *Distributed computation of the script-objects in the dataflow of a VE. White spheres represent neutral script-objects, gray spheres represent passive script-objects, and black spheres represent active script-objects.*

enables parallel rendering of multiple projection screens. The global swap signal is send by a central master component. The master component also initializes the event broker of each component and assigns the active script-objects to the components. The middleware is designed to minimize high latency system calls to the network. This is achieved by amalgamating the events of the active script-objects in each component and broadcast them at the beginning of each frame. That way a simple ethernet as underlying network is sufficient for several set-ups. The middleware can be configured to use UDP or TCP as the network protocol. In case of UDP each workstation in the cluster has to be equipped with a second dedicated network card which are only connected among each other. This way the event brokers can use broadcast messages in a scheduled way that avoids collisions and lost of data packages. The middleware has been implemented on Win32, Linux, and IRIX platforms. The event data is converted into the platform independent network format in heterogeneous clusters.

We made performance measurements of the middleware. As the metric of performance we measured the extra time T_m needed by the middleware to synchronize the data of the output interfaces each frame. To achieve acceptable frame rates T_m should not exceed 10 ms. Table 1 shows how many hosts can be connected to a cluster under this condition. In

	10 kB	20 kB	30 kB	40 kB
homogen	55	40	30	25
heterogen	48	27	21	12

Table 1: *Maximum number of hosts at which $T_m < 10ms$ in dependence of the size of the data of the interfaces. Network bandwidth 100Mb/s, UDP as underlying protocol, and only one component per host.*

case of the usage of TCP only up to 3 distributed components can be connected for similar amount of data per frame. A more detailed description of the middleware and the performance measurements has been described in [17]. The event data size of 10-40 kB is sufficient in many cases. In the recently extended implementation of the middleware data is broadcasted concurrently distributed over multiple frames if data-size exceeds 40 kB. In this way also the coordinates of a large geometry or texture can be updated during an application without drop in the framerate. The developer only has to be aware of the fact, that if he uses events with large data sizes the event is not routed immediately, but with a delay of several frames.

4. Usage of VRML as an IDL

The most common way to make sure all components in the simulation use the same interfaces is to declare them in special files. Components can use these descriptions of the interfaces to generate proper objects, which can be integrated into the parallel and distributed simulation. Instead of defining a new interface definition language we use the virtual reality modelling language (VRML)[18] for this purpose. In a certain sense VRML can even be seen more as an IDL than a programming language. Because of the arrangement of the scene-objects in a scene-graph, VRML objects generally are called nodes. VRML defines some scene-nodes with fix-defined attributes. More important for our purpose are script-nodes, which can be defined with arbitrary fields. The fields are marked with the keys *eventIn*, *eventOut* or *field*. The *eventIn* fields are interfaces for the script-node to receive events and the *eventOut* fields specify which events are supplied by the script-node. VRML defines data-types that are suitable for VE applications, like *SFVec3f*, *SFRotation* or *SFNode*. A field marked with the *eventIn* or *eventOut* tag can participate in the event-driven VRML execution model. An event is sent from one node's *eventOut* to another node's *eventIn* field if a route exists. Routes are described in the routing graph, separate to the scene graph. The execution model of our framework differs from the execution model specified in VRML97[18] in one point. An event cascade is broken at script-nodes that are not neutral. As described above, events created while processing an incoming event of an active script-object have first to be distributed to the other components to assure that the models in all components in the cluster are kept synchronized. This delay in the routing of events may lead to conflicts, if a fan-out branch of the routing graph contains active script-nodes of which the siblings are neutral. The VRML97 specification does not define a processing order for fan-out or fan-in events. Therefore it is inconvenient to implement the behavior in a way that the order of processing incoming events is important. If the developer does so in our framework, he has to be aware that events in a branch with an active script-node are delayed by one frame. Future work may extend the framework to that effect, that also the routing of neutral script nodes are delayed if necessary and the execution model would be in conformance with the VRML97 specification.

In VRML97 the script-nodes contain a field with an URL to the code that has to be processed. This code implements functions associated with the *eventIn* fields. The code is executed in a runtime environment. The runtime environment interface, also called scene authoring interface (SAI), provides some services like for creating new nodes or for deleting and adding routes dynamically. In our framework these calls to the SAI are handled like special *eventOut*-fields, which are routed statically to the SAI. In some implementations the SAI can also be used by external applications to control the scene graph. In addition to the functions for the *eventIn* fields a script node may implement some other functions like *initialize()*, *shutdown()* or *eventsProcessed()*, which are called at specific points in the live cycle of the script-object. In our framework we added the function *preFrame()*. It is called at the beginning of each frame before data among the components is exchanged (see Figure 2). The *preFrame()* function is useful to implement script-nodes associated with physical input devices whose data has to be provided with minimal latency to the system.

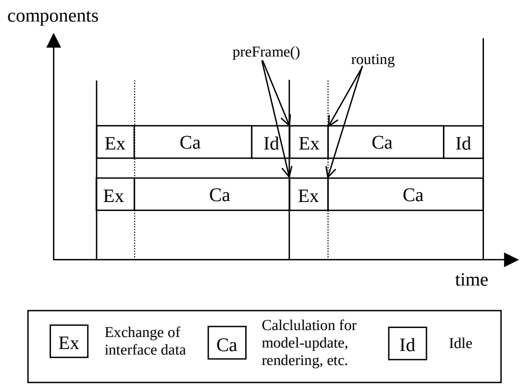
components
preFrame()
routing
Ex
Ca
Id
Ex
Ca
Id
Ex
Ca
Ex
Ca
time
Ex
Exchange of interface data
Ca
Calclulation for model-update, rendering, etc.
Id
Idle

Figure 2: *The preFrame()- function is called at the beginning of each frame. The routing is performed after the exchange of the interface-data.*

VRML97 defines language protocols for java and javascript. But there are also some propriate language protocols for C++ from different vendors. These propriate VRML browser are designed as simple web-browsers to run standalone and not to run parallel and distributed on a cluster. The SAI is designed in such a way, that the external application holds a reference to the browser enabling access to the scene-graph. In some implementations it is also possible to set callback handles for *eventOut* fields of nodes in the virtual world. By this it is not only possible to control the VE from an external application but also to include the external application in the event cycle. The disadvantage in designing the system in such a way is, that the interfaces of how the external application as a component of the VE system interacts with the VE are hidden in the implementation code. This makes the usage of this application as a component in other VE systems very inconvenient. Otherwise, if the external application itself is represented as a script with fields that connects them to the VE, it can be reused very easily. This separation of interface from implementation is a very common approach in the development of components in modern software systems. This technique is the essence of CORBA or ActiveX. It enables interoperability with all of the transparencies recommended in larger software projects. The interface to each object is defined very strictly. In contrast, the implementation of a component - its running code and its data - is hidden from the rest of the system (that is, encapsulated) behind a boundary that the client may not cross. Clients access objects only through their advertised interface, invoking only those operations that the object exposes through its IDL interface, with only those parameters that are included in the invocation. Another advantage of the approach to separate interface and implementation is, that kind of invocation, local or remote, can easily be changed. In the case of the VRML dataflow graph the invocations are only one-way.

To apply this approach to VE systems we developed an IDL-compiler that generates skeleton-code for the script-nodes defined in a VRML-file. In our current implementation the IDL-compiler generates C++ skeleton-code. The IDL-compile generates two kinds of skeleton code: lib-skeleton code and object-skeleton code. The lib-skeleton code maps the *eventIn* fields of a script-node to functions, which are exported as functions of a dynamic link library. The implementation of the lib-skeleton code has also to be compiled and linked in that way. The object-skeleton code maps the *eventIn* fields of the script node to virtual functions of a base-class. The developer implements the functionality of these virtual functions by inheritance. The code necessary to register this script-object at the event broker as an implementation of the VE script-node is contained in the object-skeleton code. The object-skeleton code is intended to implement script-objects that need to be integrated into other application frameworks to perform its task.

To allow an adequate application development a script-object has to be able to send events to other objects by reference. They may receive these references from their *eventIn* fields or through name services of the SAI. In the case of a parallel and distributed simulation, also events sent by references have to be delayed until the next frame. For this reason in our framework the SAI offers only proxy-objects to the implementations of the script-objects. The stub code of these proxies is integrated into the SAI of our runtime environment. Figure 3 shows how in this case the event is directed to the EB, which buffers it for synchronization. Stubs and skeletons serve as proxies for clients and servers, respectively.

In the parallel simulation the component with the lowest performance determines the cycle time of the simulation loop. This means, if there is a script which's computation needs more than the desired cycle time it will impair

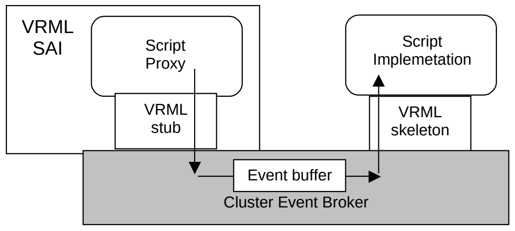

Figure 3: *How skeleton and stub code integrate into the event broker architecture.*

the whole simulation, also in the distributed case. Therefore the computation of such a script has to be broken down into further scripts that than can be distributed or if possible the computation of the script has to run asynchronous to the event and simulation loop in an extra thread. To facilitate the latter implementation we plan the IDL-compiler to provide compiler options that allow generation of skeleton code for script-nodes that has to be linked ’loosly’ or ’stricly’ into the event cascade. A ’loosly’ linked script-node means that it is running asynchronously in an extra thread and it is not guarantied, that the computation finish within the frame.

5. Implementing the Components

This section describes the development process of the distributed application. For the implementation of components for visualization the framework provides view classes for user-centric projections screens. Objects of the view classes can be attached to a world model to render the scene. The following code fragment might give an impression on how components for the visualization are implemented within the framework.

```
void WinAppC::Init(long argn, const char** argv)
{
  ..
  ..
  CybSixDOFSensorC headtracker;
  CybGLWorldC world;
  CybGLStereoViewC *pView;
  pView = new CybGLStereoViewC(this,..);
  pView->setHeadtracker(&headtracker);
  world.addView(pView);

  EventBrokerC eb(name,endpoint,clusterIP);

  world.loadFromFile(eb.getInitData());
  eb.addModel(&world);
  eb.AsignInterface("headtracker",&headtracker);
  eb.initializationReady();

  while(processWinEvents()){
    if(!eb.processEvents()){
      this->Quit();
    }
  }
}
```

The implementation contains one instance of the event broker, one instance of the world model, and one or several objects of a view class. If necessary the views can be attached to a headtracker. The world model and the view objects are registered at the EB to receive update and swap events. The world model represents the local copy of the distributed virtual environment. It is initialized by loading the VRML-file that the event broker provides as init data. After the initialization is finished the component calls the *processEvents* function of the EB to run the simulation loop until a quit message is received. Components for visualization and components for reading sensor data have in common to link the VE to physical devices. For the implementation of these kinds of components it is therefore natural to use interfaces that provide position and/or orientation data in the coordinate system of the real world (e.g. headtrackers). It would be inappropriate to define these interfaces as script-nodes in the VRML-file. Instead, the framework includes a default interface for six degree of freedom sensor data. To achieve a seamless integration of sensors that intrinsically provide data in the coordinate system of the real world into the VE application we extended VRML by a native node that wraps the interface of a real world sensor registered at the EB with the name defined in a field of the native VRML-Node. The VRML-Node provides the data of the sensor as eventOut fields transformed to the coordinate system of the virtual environment.

The application specific components implement script nodes defined in the virtual environment. In some cases the implementation of these script nodes requires a programming environment which the SAI and standard C++ libraries cannot provide. In the applications discussed in section 7, for example, this is the necessity to implement a link to other applications via ActiveX/DCOM. The most convenient way to implement components using ActiveX is to use the programming environment "Visual Studio" from Microsoft. But this implicates to hook the code of the script-object into a foreign framework. In our approach the integration of a script-object into a foreign framework is achieved by using the object-skeleton code created with the IDL-compiler. Thus, the development process of the components using a foreign framework consists of the following steps:

1. Compile the VRML-file into object-skeleton code and lib-skeleton code.
2. Implement the lib-skeleton code with some default processing.
3. Implement the object for the full functional component by inheritance.
4. Implement and compile the component with the implemented object.

The default implementation of the lib-skeleton code may be default processing that does not need the special resources for performing as an active script-node. By using these default implementation script-codes the basic functionality of the VE can be tested in a standalone browser. To get a clear impression on how the implementation of the object-skeleton code is integrated into a foreign application framework, we give an example referring to the Microsoft's "Visual Studio" (MVS) framework. The event loop of the event broker has to run in a own thread to avoid impair on the VE simulation caused by the event loop of the MVS framework. Because the EB and the application framework both require access to the implemented script-object, the reference to the script-object is shared in a field of a data structure that is passed over to the thread. This data structure might look like:

```
 struct ThreadParams{
    CWinThread*       pThread;
    EventBrokerC*     pEB;
    CCriticalSection*  pCS;
    CybWorldC*         pWorld;
    Script1C_Impl*     pScriptImpl;
};
```

Furthermore, the data structure contains a field for a critical-section-object, which is used to synchronize the access to the script-object. The thread, as shown below, creates an instance of the event broker, initializes the simulation and registers the script-object as an implementation at the runtime environment.

```
UINT ThreadRoutine(LPVOID paramsIn)
{
  ..
  ThreadParams* params = (ThreadParams*)paramsIn;

  EventBrokerC* pEB;
  pEB = new EventBrokerC(name,endpoint,localIP);
  CybWorldC* pW = new CybWorldC;
  pEB->addDocument(params->world);

  params->pCS->Lock();

  pW->loadFromFile(pEB->getInitData());
  CybScripC* pScript;

  pW->setScriptImpl(params->pScriptImpl);

  pEB->initializationReady();
  params->pWorld = pW;
  params->pEB = pEB;
  params->pCS->Unlock();

  while(b){
    params->pCS->Lock();
    b=params->ptrObj->processEvents();
    params->pCS->Unlock();
  }
  // cleaning up
  ..
}
```

The application code might create the script-object and start the thread as the following code fragment shows.

```
ThreadParams threadParams;
threadParams.pScriptImpl = new Script1C_Impl;
threadParams.pCS = new CCriticalSection;
threadParams.pThread
 = AfxBeginThread(ThreadRoutine,&threadParams);
```

6. Setting up the VE system

Once components like display servers, sensor servers, and application script servers have been implemented and are available, they are installed on the workstations providing the resources needed for one component to perform its task. For example, a display server should be installed on a workstation with hardware accelerated 3D graphic capabilities and a projector attached to it, and a sensor server is installed on a workstation with the respective physical input devices. The executables of the components are added to the component repository on the host, which is used by a cluster daemon to launch components on demand. The composition of the VE is described in a VRML-file. As shown in the previous section the VRML-file is loaded by the components to initialize the local copy of the VE model. The master

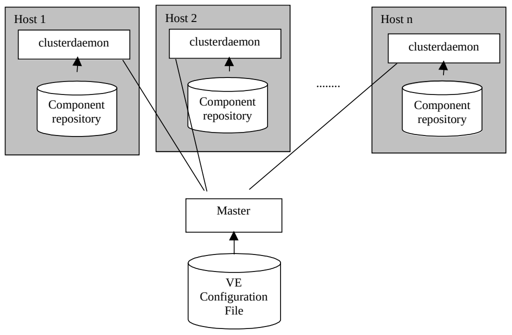

Figure 4: *The master connects to the clusterdaemons to spawn the components of the VE system.*

launches the components as can be seen in Figure 4. After the components have been started the master connects to the EBs of the components and communicates information about the URL of the VRML-file to be loaded and information about the active interfaces on each component. This kind of data is settled in a configuration file. Further, the configuration file contains a table with the components to be launched on which host and the parameter to pass over at startup. The assignment of the active interfaces is also arranged in a table. Interfaces of the VRML-file that do not occur in the table are declared as neutral objects. For each component it is mandatory, that the first parameter passed at startup is a string for its name in the distributed simulation. This name has to be passed to the instance of the event broker in the component. In this way the same executable can serve as multiple different components in the simulation. This will become clearer in the following section where some applications are described. By editing the configuration file a new virtual environment application can be assembled from consisting components.

7. Application

This section describes the implementation of three different VE systems for a geographic information system (GIS) based on the approach discussed in the previous sections. The VE systems implemented are used for the evaluation of ergonomic issues of geographic data visualization with semi-immersive VE systems (see Alexander [1]). The different set-ups differ mainly in the usage of different display technologies. Our main application set-up uses a semi-immersive virtual workbench as a display and a virtual laserpointer as a pointing device. At a second set-up we use a passive stereo projection screen, where the images for the left and right eye are rendered in parallel on different workstations. The third system realized is the prototype of a virtual 3D PDA using mixed reality technology. For this display the visualization component replaces a pattern in a video stream by the terrain model. The pattern been replaced is fixed on a palm sized table. Overall, we implemented the following executables to assemble the different set-ups:

StereoPSDisp Active stereoscopic rendering component with dynamic user centric projection for the workbench.
MonoPSDisp Monocular rendering component for parallel rendering.
MRDisp Rendering component for a mixed reality set up.
ScriptSrv Component to run common scripts that need not to be integrated into another framework.
AFlockOfBirds Server component for ascension flock of birds six DOF sensors.
MSpaceMouse Component to integrate a Magellan space mouse.
SpeechRec Component to integrate a speech recognition system.
VEGIS Component which implements the link to the GIS.
Browser Stand alone VRML browser for debugging.

These executables can be parameterized and used as different components in the set-up. The names given above are the names used in the component repository. The display components *MonoPSDisp* and *StereoPSDisp* can be parameterized with the location and orientation of the projection screen in world coordinates. The *MonoPSDisp* component can further be parameterized either to display the image for the right or for the left eye. This way the *MonoPSDisp* component can be used for stereoscopic passive multi projection screen systems, where each screen for each eye is rendered on a different workstation. We use this type of component in our showroom set-up. The component *StereoPSDisp* for active stereoscopic rendering with shutter glasses is used in the set-up of the virtual workbench.

The most general usable component of these is *ScriptSrv*. This component only contains an EB and the synchronized VE model with the VRML runtime environment. It is intended to run scripts that do not need to be embedded into a foreign framework. It is very useful to launch one instance of this component class for every script that uses the time-consuming collision detection services of the SAI. For the integration of input devices we have implemented the two components *AFlockOfBirds* and *MSpaceMouse*. One sensor of the flock of birds is mounted on one pair of shutter-glasses that is used as a headtracker. An other sensor is equipped with three buttons that is used as a six DOF mouse. For the speech recognition component *SpeechRec* we used commercial of the shelf software through an ActiveX interface. Because speech recognition consumes computing power it is advisable to run this on an extra workstation in the cluster. The script-object provided by the *SpeechRec* component contains one *eventOut* field for the strings recognized by the system. This field is routed to another script-object that converts the string to commands for the application. In the current implementation this mapping is fixed and is used only for menu item selection. For a more progressive integration of speech control the state of the virtual environment has to be taken into account to resolve ambiguities. In our approach the script performing the mapping can easily be extended for this task because it has access to the synchronised VE model. The most application specific component that is used in each set-up is the *VEGIS* component, which implements the link to a GIS. The component itself is assembled by several subcomponents. One subcomponent generates 2D maps from different database sources including vector data as VMAP and raster picture data like PCMAP. This subcomponent is also used in an ordinary 2D GIS system as a display component[19]. Another subcomponent generates the terrain model online from data stored in a DTED database and textures this with the maps generated by the map-component. The subcomponents are linked together to the *VEGIS* component by using standard techniques like dynamic link libraries and ActiveX. Beside the content of the texture map it is also necessary to display some entities with a geo-spatial reference as three-dimensional objects. The *VEGIS* component also manages the representatives and recalculates their position in the virtual environment when the terrain model changes, e.g. if another resolution of the grid is chosen. Furthermore, the *VEGIS* component advertises an ActiveX/DCOM interface to external applications that allows to control the terrain model and the entities of the virtual environment. The component itself is multi-threaded, because the task of the terrain generation cannot be performed within the cycle time of the simulation loop. For our workbench the six DOF mouse sensor provided by the *AFlockOfBirds* component is used to implement a virtual laserpointer as a pointing device. The laserpointer is implemented as a simple script node using the intersection services of the SAI. The functionality of the VE applications is composed in a VRML file. Beside the script-objects described above there are several other small script-objects that handle the appearance of menu item, etc. These script-objects are declared as neutral within all applications.

In the following the realisations of two different application set-ups are shown in the Figures 5- 6. The gray boxes represent physical devices like workstations and input output devices. The workstations contain the components they are performing represented as white boxes. The white boxes of the components contain two description fields. At the top the name of the executable of the component separated by a ’::’ from the name of the component instance. In the middle a field indicates the script-object that is assigned as active to the component. Figure 5 shows how the set-up of the workbench is assembled from different components. One SGI Onyx infinite reality rack serves as the host for the stereoscopic rendering component *StereoWBDisp* and the components *AFlockOfBirds* and *MSpaceMouse* for the input devices. Because our Onyx is only equipped with two processors, which are already busy to perform drawing and culling task, the script of the pointing-device is performed as an active object in the instance of the *ScriptSrv* component on a single workstation in the cluster. The performance of the system is increased by up to 25% this way. The components *SpeechRec* and *VEGIS* are running on dedicated workstations as well. We have no indication of how the performance of the system would decrease if we would execute these components on the Onyx, because they require a Win32 operating system.

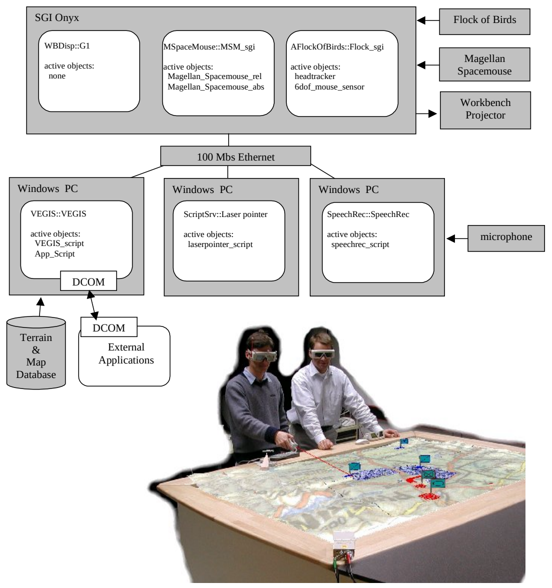

Figure 5: *Realisation of the workbench.*

Figure 6 shows the set-up for the stereoscopic parallel rendering in our showroom. Because no dynamic viewfrustum is needed the component for the head-tracker is omitted. The rendering is performed by two instances of the *MonoPSDisp* component on two different workstations. The composition of the application in the VRML-File differs from the workbench application mainly in the kind of navigation and how menu items and extra info is displayed in the VE. This is achieved by replacing some of the passive scripts we have not described in this paper. The showroom set-up could easily be extended to multiple projection screens later on.

8. Conclusions and Future Work

In this paper our approach to develop a highly flexible parallel real-time VE systems realized on heterogeneous clusters

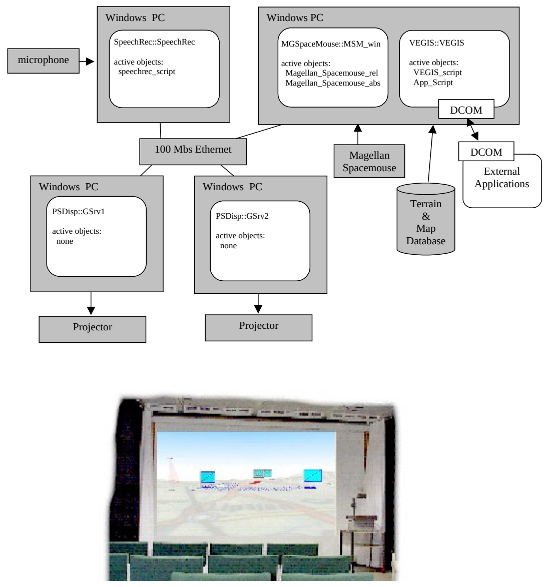

Figure 6: *Parallel rendering in the showroom.*

of workstations has been presented. This approach provides the developer with a meta programming environment. The concern of designing the high level abstractions for the virtual environment is separated from the low level issues of the actual distribution of the system. Commonly used middleware proved to be not suitable for the VE systems we were focused on. Because of this we have designed a cluster event broker architecture to implement a middleware that meets the requirements of the desired type of system. The designed middleware exploits the circumstance that in the implementation of virtual environments the interaction and animation are modeled in dataflow graphs and the running simulation is scheduled in frames. The simulation scheduled in frames allows to amalgamate the events each frame and to avoid unnecessary system calls with high latency. This way, the usage of common 100*Mbs* ethernet is sufficient for clusters with up to 20-30 workstations. Performance measurements have shown that the unavoidable system calls to the network is one limiting factor in clusters with a huge number of workstations. There are several projects trying to implement network drivers and network components with low latency and high bandwidth (see e.g. [20, 21]). The usage of such network components in connection with our framework promises the facility for clusters with more than 100 workstations.

We implemented several different VE applications simply by assembling existing components. Nevertheless, there are some areas where the framework can be enhanced. So far we have implemented only a C++ script-protocol and C++-IDL-compiler to our framework, hence all lib-script-objects which are loaded at runtime have to be compiled for each platform. The implementation of a platform independent scripting language into the framework would facilitate the actual distribution of an application. Furthermore, the framework could be extended by some kind of sophisticated load-balancing. Currently the developer has to define which script-objects are to be active on a particular host. This means no problem or is even desired for script-objects requiring special resources or performing working tasks. Script-objects which are not assigned explicitly to a host are performed as neutral script-objects on each host. In dependence on number and granulation of such neutral script-objects they may decrease performance of the system. In this case it would be beneficial if these script-objects are assigned automatically as active to selected hosts. The load balancing in our framework would have also to take the currently available bandwidth of the network into account.

However the framework described makes efficient computing of VEs on heterogeneous clusters possible. This is not limited to rendering tasks, but also includes interaction tasks and other output of different modalities. The framework has been used successfully in our three different setups. It brings along flexibility, portability, and developer usability, which is especially important for our application as a research testbed.

References

1. T. Alexander. Ergonomic issues of data visualization with semi-immersive Virtual Environment Systems, *Human-System Interaction: Education, Research and Applications in the 21th Century* Shaker Publishing, 1999.
2. Object Managing Group. The Common Object Request Broker: Architecture and Specification. http://www.omg.org
3. W. Krüger and B. Fröhlich. The Responsive Workbench. *IEEE Computer Graphics and Applications*, pp. 12-15, May 1994.
4. C. Cruz-Neira, D. J. Sandin and A. T. DeFanti. Surround-scrren projection-based virtual reality: The design and implementation of the CAVE. *Proceedings of SIGGRAPH 93*, pp. 135-142, ISBN 0-201-58889-7.
5. F. Kuhl, R. Weatherly and J. Dahmann. *Creating Computer Simulation Systems - An Introduction to High Level Architecture.* Prentice Hall, 1999.
6. C. Carlsson and Hagsand. Dive - a multi-user virtual reality system. *IEEE VRAIS93*, pp 394-400.
7. S. Singhal and M. Zyda. *Networked Virtual Environment - Design and Implementation.* Addison Wesley acm Press, 1999.
8. J. Rohlof and J. Helman. IRIS-Performer: A high performance multiprocessing toolkit for Real-Time 3D Graphics. *Proceedings of SIGGRAPH 94.* pp. 381-395.
9. G. Humphrey, E. Matthew, I. Buck, G. Stoll, E. Matthew and p. Hanraham. WireGL: A Scalable Graphics System for cluster. *Proceedings of SIGGRAPH 2001*, ACM, Los Angeles, CA, 2001.
10. H. Tramberend. Avocado: A distributed virtual reality framework. *Proceedings of the IEEE Virtual Reality International Conference VR'99*, Houston, USA, March 1999.
11. M. Bues, R. Blach, S. Stegmair, U. Häfner and H. Hoffmann. Towards a scalable High Performance Applications Platform for Immersive Virtual Environments. *Proceedings of Immersive Projection Technology and Virtual Environments 2001*, pp. 165-174, Springer.
12. G. Hesina, D. Schmalstieg, A. Fuhrmann and W. Purgathofer. Distributed Open Inventor: A Pratical Approach to Distribute 3D Graphics. *Proceedings VRST99*, pp. 74-81, 1999.
13. SGI. White Paper: SGI Grpahics Cluster: The Cluster Architecture Challenges, the SGI Solution. http://www.sgi.com/Products/PDF/3088.pdf
14. J. Allard, L. Lecointtre, V. Gouraton, E. Melin and B. Raffin. Net Juggler Guide. *Technical Report, University of Orleons, LIFO*, Report Nr. 2001-02
15. T. Rischbeck and P. Watson. A Parallel VRML97 Server Based on Active Objects. *VECPAR 2000 Conference*, June 2000, Springer Verlag.
16. D. Caromel, W. Klauser and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, **10**(11–13), pp. 1043-1062, Wiley & Sons, 1998.
17. C. Winkelholz and T. Alexander. Rahmenwerk zur Realisierung einer immersiven virtuellen Umgebungen verteilt in einem Cluster. *FKIE-Bericht Nr. 37. Wachtberg-Werthhoven, FGAN-FKIE*. 2001
18. VRML Consortium Incorporated. Vrml97 Specification. *International Standard ISO/IEC 14772-1:1997*
19. A. Kaster and J. Kaster. Componentware Approaches in Management Information Systems. *Proceedings of the NATO RTO Symposium HFM-049/SY-005*, 10.-13.4.2000 Oslo, NO, Neuilly-sur-Seine: NATO RTO/RTA
20. G. Ciaccio and G. Chiola. GAMMA and MPI/GAMMA on Gigabit Ethernet. *PVM/MPI 2000*. pp. 129-136
21. N. Bode. Myrinet - A Gigabit-per second Local Area Network. *IEEE Micro*. http://www.myrinet.com