Chapter 5

Implementation

This chapter provides a technical description of the implementation of two scenarios, one using two distributed Responsive Workbenches and the other one using a distributed Responsive Workbench and a CAVE. The first section describes the setup with respect to the hardware configuration used, introducing the rendering and interaction equipment. The second section talks about the software configuration describing the audio/video conferencing as well as rendering and distribution. Code fragments show the application programming briefly and flow charts are used to represent the combination of different techniques, operations, data and equipment.

5.1 Hardware Configuration

Most projection-based VEs and CVEs show almost the same hardware configuration. In the first place there is a computer that processes data and renders it to the screen as well as a tracking device that measures the position and orientation of the user's viewpoint. This tracking data is read by the rendering machine in order to determine the correct perspective view onto the virtual scene from the user's viewpoint. The hardware configuration implemented in this thesis includes input devices for interaction, computers for rendering and distribution and computers for video and audio streaming. Additionally it includes equipment like shutter glasses, infra-red emitters, cameras as well as microphones and headphones.

5.1.1 RWB-RWB configuration

For the distributed RWB-RWB setup the hardware configuration shown in Figure 5.1 is used [47, 51]. For the rendering two SGI ONYX workstations are used with at least one Infinite Reality (IR2) graphics pipe each. The reason is

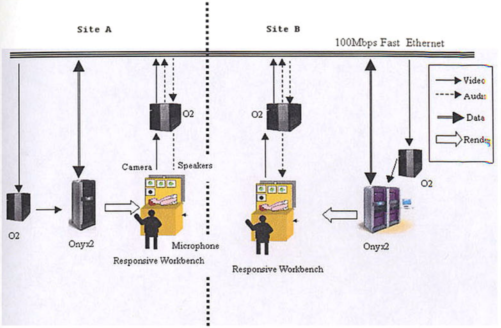


Figure 5.1: Schematic of the implemented setup with two Collaborative Reon- sive Workbenches.

that the SGI pipe architecture consists of two channels that can each rder a stereo image of a typical resolution of 1280x1024 pixels at either 96 H or 120 Hz stereo. Additionally a minimum of four MIPS R10000 processor are needed. Obviously, a lower cost hardware can be used both for renderingand for the audio/video communication. The availability of the particular eqip- ment and its high performance provide an easy choice.
For the tracking of the user's head and input devices the Fastrak trackingsys- tem from Polhemus is used. Therefore a Polhemus sensor is attached to thleft earpiece of the Crystal Eyes shutter glasses. For interacting within the Vitual Environment a Polhemus stylus and an own custom-made three buttonool are provided. The tracking sensors are attached to the input device to as the computer needs to know where the user holds the input devices and pints them to. The sensors' position and orientation are measured electronicall by a receiver attached to the front side of the Responsive Workbench. Thrugh calibration and transformation the co-ordinate system of the tracking syem is matched with the world co-ordinate system of the CVE.
For communicating with the remote partner wireless microphones and lad- phones are used. The video and the audio conferencing is handled by twO2

workstations. The decision for using these type of computers was mainly influenced by the availability of the O2 MVP video cards. These special video cards are equipped with a special motion jpeg encoding chip. The video streams are grabbed directly from the infra-red video camera. The infra-red cameras are necessary since the light in the laboratory is dimmed in order to perceive the rendered images with high contrast and brightness. After the video is received by the O2 workstation the stream is transferred to the DIVO video boards of each ONYX workstation. The same O2 that manages the video conferencing also manages the audio connection. The audio stream is grabbed from the wireless microphones and then send to the other O2 where the headphones are plugged in. The reason for using headphones is to avoid acoustic feedback loops which disturb the communication between the remote partners. As soon as more than one user is working at the same site more headphones or normal speakers have to be taken in order to provide the other person(s) with audio perception too.

5.1.2 RWB-CAVE configuration

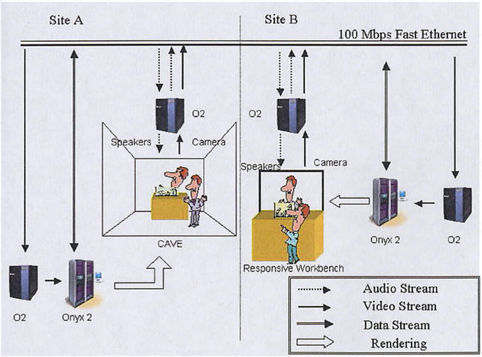


Figure 5.2: Schematic of the implemented setup with one Responsive Workbench and the GMD's cave-like CyberStage.

The hardware configuration for the distributed RWB-CAVE setup is sim-

ilar to the described RWB-RWB setup. Again two SGI ONYX workstations are used. The same computer is used for driving the RWB as in the other setup. For rendering four stereoscopic images (eight images in total) on the four walls of the CAVE, an ONYX workstation with at least two graphics pipes is needed. In the used configuration two or four Infinite Reality (IR2) graphics pipes are used. Each machine needs at least four MIPS R10000 processors. The performer application, drawing and culling processes are each running on one processor and the process running on the forth processor is importing the tracking data to the scene graph. As the application needs video conferencing facilities and tracking of two input devices in addition to the tracking of the viewpoint and the rendering, six MIPS R12000 processors are used to ensure a rendering frame rate of approximately $30Hz$.
As the video representation of the remote partner in the CAVE is offered without camera background additional video hardware is necessary (see Figure 5.3). Therefore special hardware chroma keyers are used for segmentation in order to determine the regions in the video images where the participant appears. The video streams sent and received by the O2 workstations are transferred to an Ultimate hardware keying device. After the remote partner has been cut out of the surroundings through filling the subtracted regions with transparency values the remaining video stream is transferred to the DIVO video boards of the ONYX workstation. For a clean and precise chroma keying homogeneous lighting is essential. This is contradictory to the light in the laboratories which is dimmed for perceiving the rendered images with high contrast and brightness. Additionally, the infra-red cameras produce black/white images only. However, if the user is wearing bright clothes and the keying color is black it is possible to subtract the user from its background. Especially helpful is the design of a ring which consists of five to ten infra-red LEDs. This ring attached around the infra-red camera makes it possible to illuminate the user directly as the user is supposed to look into the camera. When using infra-red light it should be ensured that the frequency does not disturb the infra-red driven shuttering of the stereo glasses.

5.2 Software Configuration

The software used for the implementation of the CVE is mainly Avango. The basic concepts of Avango are described already in section 1.3.3.
Several attempts to offer toolkits for distributed VE application development have been made recently (eg. VR Juggler [5], DIVE [24], WTK [84, 85]). These toolkits provide various degrees of support for network based communication between the distributed processes that form an application. However, using

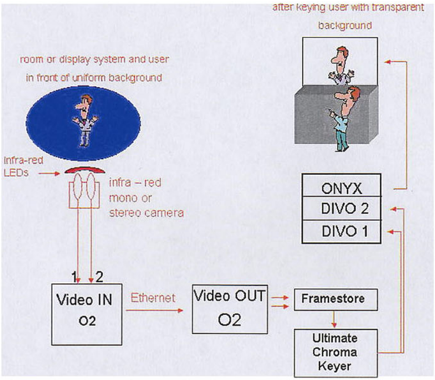

after keying user with transparent background
room or display system and user in front of uniform background
Infra-red LEDs
infra – red mono or stereo camera
ONYX
DIVO 2
DIVO 1
1
2
Video IN O2
Ethernet
Video OUT O2
Framestore
Ultimate Chroma Keyer

Figure 5.3: The video streams sent and received by the O2 workstations are transferred to an Ultimate hardware keying device. After the remote partner has been cut of out the surroundings through filling the subtracted regions with transparency values the remaining video stream is transferred to the DIVO video boards of the ONYX workstation.

these facilities often requires significant effort from the programmer. Normally, only parts of the application state are shared between the distributed processes such as transformation matrices which describe object positions. Sometimes explicit specification of communication endpoints for shared object attributes is necessary. The resulting database duplication problem, ensuring that all processes work on consistent copies of the shared database, is left as a challenge to the application programmer. This can be a tedious and error prone task especially when additional processes join an already running distributed application. Avango provides programmers with the concept of a shared scene graph, accessible from all processes forming a distributed application.

The software that handles the audio/video conferencing, however, is designed to run independently from Avango. Before implementing conferencing software, already existing audio/video conferencing toolkits were considered for integration. The requirement of having PAL or at least NTSC sized video resolution at 20-30 fps is the reason why toolkits like Mircosoft's netmeeting, public domain MBone tools or the well known public domain Vic and Rat tools cannot be integrated in the CVE. These toolkits are designed for low bandwidth connections such as modem, ISDN or DSL links, transferring icon-sized video conference images. Although some tools like the H263 encoding algorithm from Telenor are able to compress and send 4CIF (704x576 Pixels) sized video streams, but the frame rate drops to almost 0.5-1.0 fps. The developed audio/video conferencing software within this thesis is able to send PAL video streams over the network at 25fps and to integrate them into a CVE in real-time. Therefore the software handles bi-directional audio/video connections.

The first part of the following section describes the Avango field interface and how scene graphs are distributed. Additionally it provides code fragments to give an idea about the programming effort for implementing CVE applications.
The second part deals with the software that enables the audio/video conferencing in mono and stereo. In addition, modifications are described which are necessary for integrating stereo video textures into a VE using Avango.

5.2.1 Distributed Scene Graphs with Avango

Avango combines the familiar programming model of existing stand-alone toolkits with built-in support for data distribution that is almost transparent to the application developer [96]. The Avango framework is based on Performer to achieve the maximum possible performance for VE and CVE applications. Performer provides a C++ scene graph API which allows for flexible representation of complex geometry [87, 88, 89]. Advanced rendering tasks like culling, level-of-detail switching and communication with the graphics hardware are all handled by Performer. Whenever the underlying hardware allows, Performer utilizes multiple processors and multiple graphics pipelines. The toolkit also provides a collection of geometry loaders for the most popular as well as some uncommon file formats.

Fields and Fieldcontainers

The efficient implementation of a generic scripting and streaming interface for heterogenous objects requires additional *meta* information about object attributes and their types, and a way to access those attributes without knowing the exact type of the containing object. The C++ programming language does not treat classes as first class objects, so that this information is not easily available on a language level. Performer uses a member function API to access the state attributes of an object. A symmetric pair of get and set functions exists for each attribute. Setting one attribute may change another attribute of that object as a side effect. However, no abstract information about the number of attributes, their type, and their value can be obtained from an object via the Performer API. To overcome this problem Avango introduces the notion of *fields* as containers for object state attributes to Performer objects. Fields encapsulate basic data types and provide a generic interface for script-

```
template<class T> class fpSingleField : public fpField {

public:
  virtual fpType   getTypeId() const;
  virtual void     setValue(const T& value);
  virtual const T& getValue() const;
  virtual void     setSchemeValue(Scheme value);
  virtual Scheme   getSchemeValue() const;
};

template<class T> ostream& operator<<(ostream& stream,
                            const fpSingleField<T>& field);

template<class T> istream& operator>>(istream& stream,
                            fpSingleField<T>& field);
```

Figure 5.4: Part of the `fpSingleField` interface. `fpSingleField` is a template class and is used to instantiate single value fields on basic data types.

ing and streaming. They are implemented as public class member functions and are thus inherited by derived classes. They are directly accessible by client classes and are Avango's premier interface to object state attributes. There exist four different types of fields. *Single-fields* contain one basic data type value, whereas *multi-fields* contain a vector of values. A multi field can contain an arbitrary number of values. To bridge the Performer method based API to the Avango field oriented API, *adaptor fields* are used. They will forward `getValue()` and `setValue()` requests to the appropriate get and set functions of the Performer API. This ensures that Performer related state information is correctly updated according to field value changes, and possible side effects are properly evaluated.
The implementation of fields makes use of C++ templates which allow to parameterize the `fpField` class with the required data type. As an example for

the field class API, part of the template class definition for the single field is shown in Figure 5.4. The `getTypeId()` method provides access to the run-time type information for a field. The returned type identifier is used for run-time type checking and to provide type information for field values which are written to streams.
Access to the field value is provided by the `getValue()` and `setValue()` methods. Alternatively, a scheme representation of the field values can be provided or obtained via the `getSchemeValue()` and `setSchemeValue()` methods. For each field class a pair of stream operators exist which allow serialization of the field value into a stream, and the reconstruction of the field value from a stream.
Avango provides a *fieldcontainer* interface which represents the state of an object as a collection of fields. A fieldcontainer can be queried for its number of fields, and any of the fields themselves. Relevant parts of the fieldcontainer interface are shown in Figure 5.5. This, together with the generic scripting and

```
class fpFieldContainer {

public:
  int              getNumFields();
  fpFieldPtrVec&   getFields();
  friend ostream&  operator<<(ostream& stream,
                              fpLink<fpFieldContainer> fc);
  friend istream&  operator>>(istream& stream,
                              fpLink<fpFieldContainer> fc);

protected:
  virtual void notify(fpField& field);
  virtual void evaluate();
};
```

Figure 5.5: `fpFieldContainer` encapsulates the state information of an object and represents it as a collection of fields.

streaming interface of fields, allows to provide complete scripting and streaming functionality at the fieldcontainer level without knowing the exact type of the underlying object. This extents, at no extra cost, to all classes further derived from `fpFieldContainer`.

Fieldconnections

Much like SGI's Open Inventor Toolkit, Avango introduces the concept of *fieldconnections* [102]. Fields of compatible type can be connected in a way that whenever the value of the *source field* changes, it is immediately forwarded to the *destination field*. Using fieldconnections, a data-flow graph can be constructed which is conceptually orthogonal to the scene graph. Avango utilizes this mechanism to specify additional relationships between nodes which cannot be expressed in terms of the standard scene graph. This allows for the easy implementation of interactive behaviour and the import of real world data into the scene graph. The evaluation of the data-flow graph is performed once per rendering frame. Fieldconnections forward value changes immediately, so that there is no propagation delay for cascaded connection paths in the graph (see Figure 1.9). Loops are detected and properly resolved. A fieldcontainer can implement side effects on field changes by overloading the `notify()` and `evaluate()` member functions. Whenever a field is set to a new value, the `notify()` method is called on the fieldcontainer with a reference to the changed field as an argument. The fieldcontainer can do whatever is necessary to achieve the desired effect, including the manipulation of other fields. After all notifications for all fields on all fieldcontainers have been made for one frame, the `evaluate()` method is called on each fieldcontainer which had at least one of its fields notified. This allows the fieldcontainer to perform actions which depend on more than one updated field value for each frame.

Nodes

Fieldcontainer adaptions exist for all Performer node classes (`pfGroup`, `pfDCS`, `pfLOD`, etc.) and most of the Performer object classes (`pfGeoState`, `pfMaterial`, `pfTexture`, etc.). By convention the Avango nodes exchange the Performer `pf` prefix with the `fp` prefix. The Avango scene graph is built out of nodes as shown in Figure 1.9. The possibility to mix Avango nodes with Performer nodes in the scene graph can be conveniently used to define new nodes with interesting functionality. The `fpFile` node, for example, is derived from the adaption node `fpDCS`. It inherits the interface of `fpDCS` which consist of a `Children` and a `Matrix` field. The `fpFile` node adds a `URL` field of type `string`. With an overloaded `notify()` method, `fpFile` reacts to changes of the `URL` field by retrieving a geometry from the specified URL and adding it to its list of children. Thus, `fpFile` imports the geometry into the scene graph, and can be seen as an opaque handle to it. Subsequent changes to `URL` will result in replacement of the old geometry with the newly specified geometry.

Sensors

Sensors represent Avango's interface to the real world (see Figure 1.9). They are derived from the `fpFieldContainer` class but not from any Performer node, and thus cannot be inserted into the scene graph. Sensors encapsulate

the code necessary to access input devices of various kinds. The data generated by a device are mapped to the fields of the sensor. Whenever a device generates new data values, the fields of the corresponding sensor are updated accordingly.
Fieldconnections from sensor fields to node fields in the scene graph are used to incorporate device data into an application. The `fpTrackerSensor` class provides an interface to six degree of freedom trackers like the Polhemus Fastrak devices (see Figure 5.6). Avango utilizes a device daemon process which

```
class fpTrackerSensor : public fpDeviceSensor{

public:
  fpSingleField<string>   Station; // inherited
  fpSingleField<fpMatrix> Transform;
  fpSingleField<bool>     Button;

};
```

Figure 5.6: The Avango sensor classes map data values from external devices to fields.

handles the direct interaction with the devices via serial line or network connections. The daemon updates the device data values into a shared memory segment, where the `fpDeviceSensor` classes can access them. A *station* name is used to identify the desired device data in the shared memory segment, and every `fpDeviceSensor` class specifies this identifier in its `Station` field. After connecting to the device daemon, the `fpTrackerSensor` class provides the current position and orientation information from the selected tracking device represented in form of a matrix in its `Transform` field. By connecting the `Matrix` field of a `fpDCS` node to the `Transform` field, the subtree rooted by the `fpDCS` node will move according to the tracker movement reported from the selected station.

Scripting

The development of Virtual Environment applications often follows a highly iterative approach. Many VE toolkits and frameworks do not account for this situation as changes and reconfigurations require recoding in C or C++ and recompilation of parts or even the whole application. An interpreted scripting language which has a binding to all relevant high-level object interfaces in a framework can greatly reduce the burden on the application programmer and

```
;; instantiate a fpFile node and attach it as child to the
;; scene-root node
(define geom (make-instance-by-name "fpFile"))
(-> (-> scene-root 'Children) 'add-1value geom)

;; load a geometry from the following path
(-> (-> geom 'Filename) 'set-value "./graphics/iv/jaw_bone.iv")

;; instantiate a Drag Tool Node and activate it
(define drag-tool (make-instance-by-name "fpDragTool"))
(-> (-> drag-tool 'TimeIn) 'connect-from (-> time-sensor 'Time))

;; instantiate a dragger - a special matrix dragger
(define geom-dragger (make-instance-by-name "fpMatrixDragger"))

;; assign the dragger to the geometry
(-> (-> geom 'Dragger) 'add-1value geom-dragger)

;; connect the dragger's matrix with the geometry's matrix
(-> (-> geom 'Matrix) 'connect-from (-> geom-dragger 'Matrix))
```

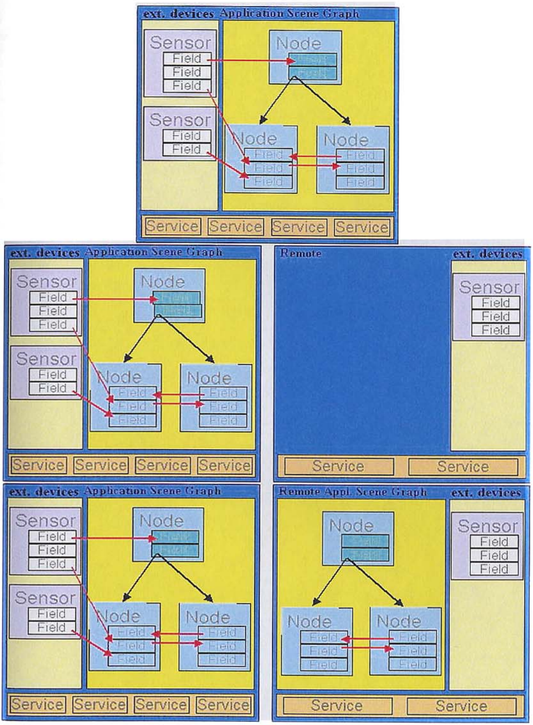
Figure 5.7: A `fpFile` node is instantiated and loads an Inventor file from the path specified in the *'Filename* field. By making the file node a child of the *scene-root* the associated geometry gets rendered. After instantiating a `DragTool` and configuring it, the `DragTool` checks for intersections between the input device representation geometry and all other geometry in the scene. If an intersection with *geom* is detected, a special mechanism looks for an instance of a dragger being assigned to *geom*. In this case the matrix field connection between *geom* and *geom-dragger* is executed and the jaw bone geometry follows the movements of the input device.

will significantly shorten the development cycle. No recompilation is necessary and modifications can be applied to running applications. As already described in section 1.3.3 Avango features a complete language binding to the interpreted language Scheme. It uses the *ELK* Scheme implementation which is a small and elegant Scheme interpreter and is especially suited to serve as an extension language for C and C++ programs. The Avango scheme binding is based on the field and fieldcontainer API's of the Avango objects. For all basic data types that are used to instantiate field classes a scheme representation with all necessary access functions exists. The basic data types are passed by value to and from the Scheme interpreter and can be handled like any other built-in Scheme type.
Avango objects such as nodes and sensors are handled by reference. This is

implemented by providing a binding for the `fpLink` class. `fpLink` values are again passed by value to the Scheme interpreter so that references to Avango objects are properly reference counted. Scheme uses a garbage collector to reclaim memory from objects which can no longer be accessed by the interpreter. When an `fpLink` value is garbage collected, the reference count on the associated Avango object is decremented accordingly.
Avango objects can be created from Scheme by providing the name of the desired object class as an argument to the `(make-instance-by-name class)` function. The object is instantiated, and a reference is handed back in the form of a `fpLink` value.
A set of access functions allows access to the fieldcontainer and the field interfaces of Avango objects. Figure 5.7 shows an example script which instantiates an `fpFile` node and connects it to an instance of a `fpMatrixDragger`. A `fpMatrixDragger` contains the position and orientation of the input device. With a field connection between this dragger and a geometry the latter one follows the movements of the input device. Because ELK Scheme is an interpreted language, every Avango application can provide a command-line interface, where Scheme commands can be entered at run-time. All Avango objects which are defined by a scheme script can be accessed and manipulated while the application is running.
The Avango scripting interface suggests a two phased approach for the application development. Complex and performance critical functionality is implemented in C++ by subclassing and extending already existing Avango classes. The application itself is then a collection of Scheme scripts which instantiate the desired Avango objects, set their field values and define relationships between them through field connections. The scripts can be written and tested while the application is running. This leads to very short turnaround times in the development cycle and provides a very powerful environment for rapid application prototyping.

Distribution

As already described Avango objects are fieldcontainers that encapsulate the object state in a set of fields. The streaming interface of the field and fieldcontainer classes allow for a very elegant implementation of the distributed object semantics.
Distributed object creation in Avango is a two stage process. First a local object is created which is then, in a second step, migrated to the desired distribution group (see Figure 5.8). The migration involves the announcement of a new object to the distribution group and the dissemination of the current object state to all group members. For this the streaming interface of the


ext. devices
Application Scene Graph
Sensor
Field
Node
Service
Remote
Remote Appl. Scene Graph

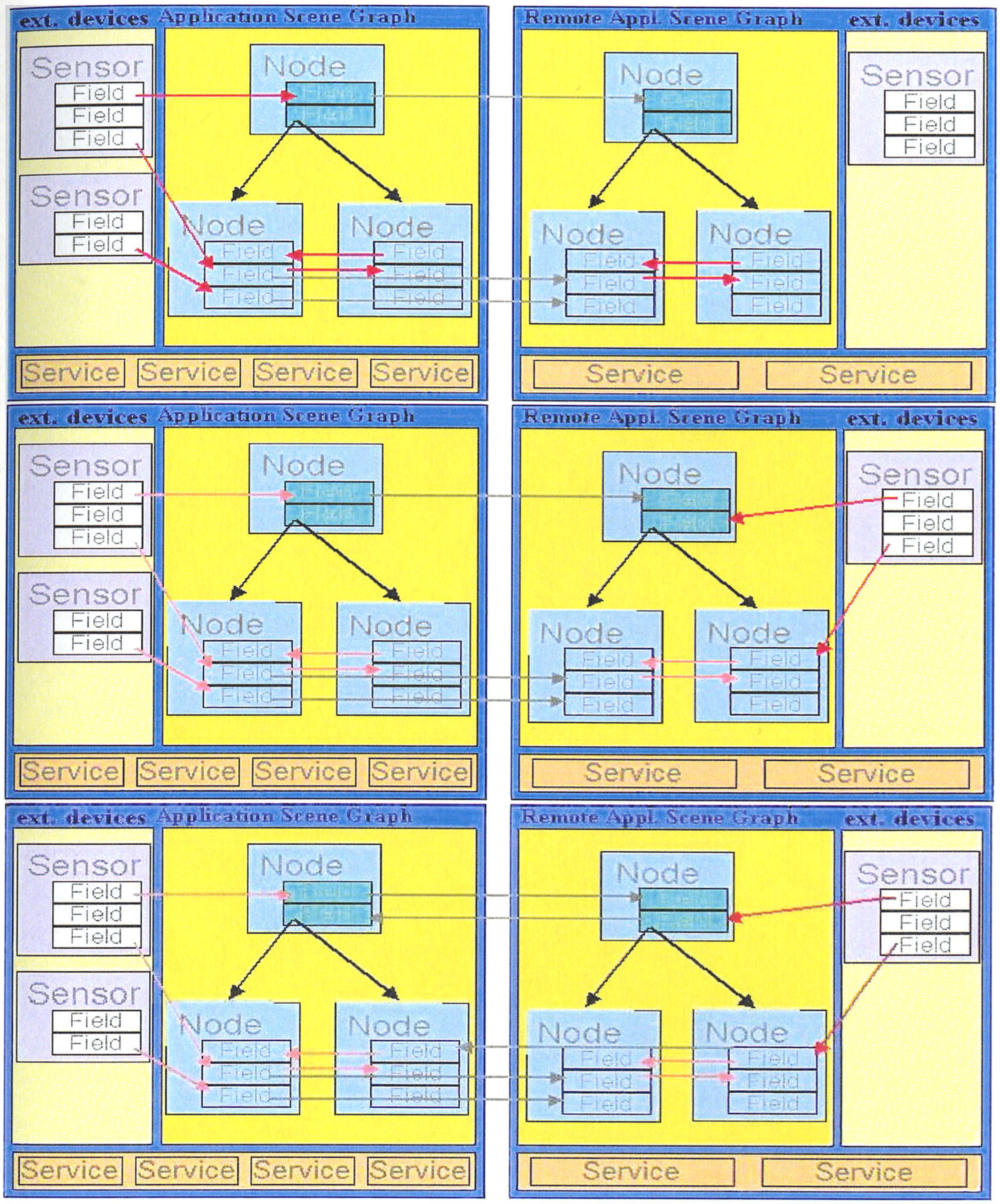


Figure 5.8: Illustration of a scene graph distribution using Avango. The six steps show how the scene graph is copied to a remote site and field values are updated. The last two steps illustrate how the participants who where joining the group at a later time are able to interact on the received data and propagate changes back to the first user.

fieldcontainer is used to serialize the object state into a network data buffer which is then sent to all group members. The group members will reverse this process and create a copy of the object from the serialized state information. The newly created distributed object will now exist as a local copy in each of the participating processes.
In Avango object state is accessible in terms of object field values. Whenever a field value on a distributed object is locally changed the new value is streamed to a network buffer and sent to all members of the distribution group in order to keep the distributed copies of that object synchronized.
The parent child relationship between objects in the scene graph is represented by a multi-field of reference values on the parent node. Because all field values including `fpLink<>` types are streamable, the distribution scheme described above will not only distribute and synchronize singular objects but it is possible to distribute parent child relationships between these objects too. Additionally, it replicates a complete scene graph to all processes in the same distribution group.
Figure 5.8 illustrates all steps of the process when distributing a scene graph using Avango. In a first step the first participant loads up an Avango scene represented by the scene graph including nodes, sensors and services. After a remote participant joins the distribution group using local sensors and services (step two), the scene graph is copied according to the hierarchical order of the nodes (step three). In step four the fields of the local scene graph nodes are copied into the corresponding fields of the remote scene graph nodes which turns the remote scene graph into an identical copy. Now the remote sensors are connected to the nodes which allows interaction at the remote side as well (step five). As soon as these interactions change the state of the own nodes, represented by their field values, these changes are propagated back to the first participant again (step six). Step six occurs every time one participant in the distribution group interacts in the Virtual Environment coursing changes of the scene graph state.
Being able to distribute the entire scene graph with all parent-child relationships between the objects is a key feature on the way towards a simplified development of distributed applications. However, it is not sufficient when considering the case of a distribution group with two member processes A and B. Both processes have already created several distributed objects in that group. Now a third process C joins the group. From now on all three processes will be notified of future object creations and manipulations, but process C will not know of the objects that A and B already created before it joined.
Avango overcomes this problem by performing an *atomic state transfer* to every joining member. When a new process joins an already populated distribution group, one of the older group members takes the responsibility to

transfer the current state of the distribution group to the new member. This involves sending all objects, currently distributed in the group, with all their field values to the newcomer. After the state transfer the new member will have the proper set of object copies for this distribution group. To prevent consistency problems, the state transfer is performed as one atomic action by suspending all other communication while performing it.
The ability to replicate the entire scene graph paired with the state transfer to joining members effectively solves the duplicate database problem. New members can join an existing distribution group at any time and will immediately receive their local copy of the scene graph constructed so far in the distribution group. Furthermore, the application programmer does not need to be concerned with distribution details. The user can take the scene graph for granted on a per process level and can concentrate on the semantics of the distributed application. Figure 5.9 shows a brief scheme code example distributing a simple jaw bone geometry.

```
;; join the distribution group with the help of
;; string "codeword"
(define dist-group-node (av-join "codeword"))

;; instantiate a file node to load the geometry
(define geom (make-instance-by-name "fpFile"))

;; load the geometry (-> (-> geom 'Filename) 'set-value
         "http://viswiz.gmd.de/~gernot/graphics/iv/jaw_bone.iv")

;; distribute the geometry
(-> dist-group-node 'distribute-object geom)

;; after distribution add geom to the distribution group node
(-> (-> dist-group-node 'Children) 'add-1value geom)
```

Figure 5.9: The call of the `av-join` command creates a distributed session. This session can be joined by using the *"codeword"*. The instance of `fpFile` and thus the geometry can now be distributed applying *'distribute-object*. Joining Avango applications can access the geometry over the *URL* specified in the *'Filename* field since all fields are copied after the whole scene graph has been replicated. *geom* is attached as child to the distribution group node after being distributed.

5.2.2 Audio/Video Conferencing

The audio/video conferencing runs independently from Avango and thus independently from the scene graph distribution. In addition the video software sends and receives its streams independently from the audio source. However, the audio as well as the video setup show almost the same configuration. The flow chart in Figure 5.10 shows the single procedure steps of server and client. On the server site the frames are grabbed from the camera which is plugged

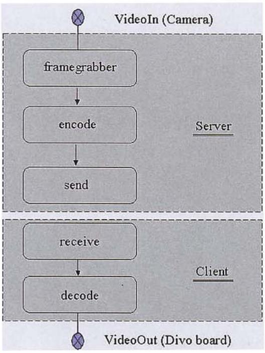


Figure 5.10: The figure shows the work flow of the video server and client. On the server side the frames are grabbed, encoded and then sent to the client side where the stream becomes decoded. The same sequence is used for the audio connection that runs independently from the video.

into the *VideoIn* port of the video card. Then the grabbed frames are encoded. After encoding, the reduced frames are packed into buffers and sent over the network to the client site that is already waiting. There the buffers are received and decoded. The inflated frames are then available at the *VideoOut* port of the O2 video card again.
The video setup decides about the next procedure. As shown in Figure 5.3 the video stream can either be chroma keyed or not.
Finally the video stream is transferred to the DIVO video board of the ONYX workstation. There the video is integrated and rendered in the CVE making

use of the Avango scene graph. Therefore the video frames are handled like a texture which can be mapped to a polygon representing the virtual video screen. Special video configurations in Avango download video frames from the video hardware continuously into the texture memory.
For sending and receiving the video and audio streams SGI O2 workstations are used in a way as shown in Figures 5.1 and 5.2. The decision for using these type of machines was mainly influenced by the availability of the O2 MVP video cards. These special video cards are equipped with a special motion jpeg encoding chip. The cards in general as well as the compressor chip in particular can be configured using SGI's *Digital Media Development Environment (DMdev)* library [86].
The audio/video conferencing software is developed on the basis of this DMdev library. This software handles the following different parts, namely: network communication, video path, video node, image compressor, image parameters, encoding and sending of the images. As the receiving unit works equivalent to the sending unit it is sufficient to focus on the latter. The program flow chart in Figure 5.11 shows the different initialization steps. The first step is to configure the network and the communication. Therefore three parameters are important: the protocol, the port and the host name of the receiving unit. Then the video path is configured which creates a link with the connected camera. For doing so the API functions of the VL (SGI's Video Library) are used. The VL allows also to configure video paths to more than only one connected camera. The video path consists of two nodes, the source and the target node. The source node is the camera and the target node is the memory segment to store the image. Later in the process the frames are grabbed from this memory for encoding.
The attributes of the video nodes that need to be set are image format, image size (PAL), zoom factor, color space etc..
The selection of the image encoder and its configuration needs as to be done according to the encoding requirements. The following encoders are available:

- Apple QuickTime Animation - 'rle'
- Cinepark - 'cvid'
- Intel Video- 'IV32'
- H.261 - 'h261'
- JPEG - 'jpeg'
- MPEG1-Video - 'mpeg'

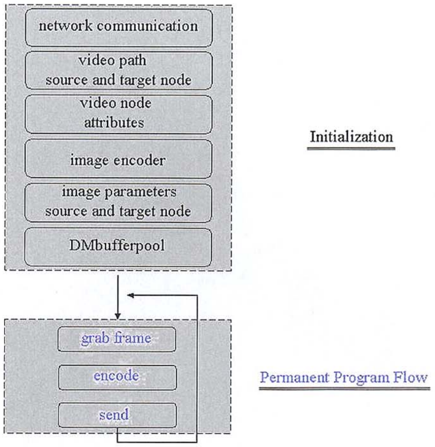


Figure 5.11: Program flow chart for the sending unit. The first block represents the initialization of the video board. The second block shows the permanent flow consisting of frame grabbing, encoding and sending.

The hardware encoder can use the jpeg standard. According to the requirements for the video conferencing the motion jpeg encoder is chosen. The parameters for this encoder type are set. The source and the target node are the uncompressed and compressed image respectively.
In the last step DMbufferpools are created. These pools are used to transport images on the video board. These pools are allocated in system memory to which all IO devices have a very fast access. Two DMbufferpools are created. One for the transport of the images from camera to memory (vid-to-mem) and one for transporting it to the image encoder. This last buffer is used again for writing the encoded image back into the same memory segment (mem-to-mem). With this the initialization phase is complete.
Then the permanent program loop is entered. This loop consists of the following three steps, grab a frame, encode and send. More precisely, every time a

new image is created at the memory target node an event is released. If this event (VLTransferComplete) occurs, the image frame is read into the DM-buffer and passed over (dmICSend()) to the encoder. dmICReceive() reads the compressed frame from the encoder again into the second DMbufferpool. Finally the call of dmNetSend() sends the compressed frame to the decoding client.
Since the video communication mainly occurs between the camera, the hardware encoder and the system's memory, the system CPU is mostly spared. Transfer and frame rate measurements showed a CPU load of less than 10% on a SGI O2 workstation with a MIPS R10000 processor. The separately handled audio connection adds another 5-7% to this CPU usage. These measurements are made having a video frame rate at the client side between 20-25 fps. The bandwidth necessary to ensure this transfer rate has to be at least 500 kbps. Similar results can be achieved using PCs with common video boards. VL- and DMdev-like digital media libraries are also available for PC hardware. The concept, however, remains the same.

Stereo-Video Conferencing

The stereo video conferencing is especially challenging because of two things:

- grabbing, encoding and transferring two synchronized images
- integrated rendering of synchronized stereo textures in Avango

When transferring two synchronized images, that correspond to each other, it has to be ensured that both images arrive completely at the other site. One image only is of no use. Due to this requirement the fast UDP protocol cannot be considered as it offers no confirmation mechanism. The TCP protocol instead is an acceptable option. An optimal solution for the stereo video conferencing is to send the images together at once and not after each other. Additionally this implies that both images, the one for the left and the one for the right eye, should be grabbed together as a mixed image from the camera. Here mixed means that both images share the 576 lines (fields) of the PAL sized images. How this could look like is shown in Figure 5.12.
It is possible to configure the video nodes in a way that only the odd fields are grabbed from the right camera image and the even fields of the left camera image. Then both images, having half size only, can be merged. This merging can either be done so that the upper part of the image is of the right camera and the lower part of the left camera or they both are merged alternating like shown in the right hand side of part of Figure 5.12. Merging half-images of both cameras has the advantage that the amount of data is the same as when using mono video conferencing. It is evident that extra CPU time needs to be

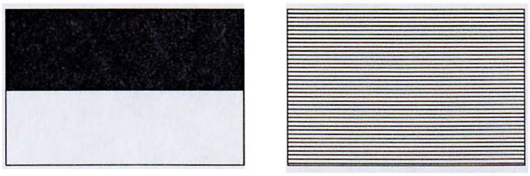


Figure 5.12: The video server can be configured to grab the even fields of the right and the odd fields of the left camera. Then both fields are assembled together in one image which will again be encoded and sent to the client. With this trick it is possible to keep a high frame rate even sending stereo images over the network.

spend to copy the half-images into the same DMbuffer on server and client site. Additionally it ensures a PAL sized mutual image which can be compressed by the video hardware which is optimized for PAL sized images only. As soon as the image resolution decreases, the encoding needs to be done by software which would result in a decreasing frame rate.

After the stereo images are received by the client site and are decoded and splitted again they have to be texture mapped and rendered in the CVE. This integration into Avango is quite challenging as the draw traverser does not know anything about the synchronized images which arrive at the DIVO video hardware of the ONYX workstation (see Figure 5.3). For solving this problem a mechanism is created which knows about the synchronized video images and decides which one of them is rendered when. As already said video images are handled like textures which are permanently downloaded from the video hardware into the ONYX texture memory. Thus the geometry onto which these textures are mapped has to be added and removed to and from the scene graph according to the framerate. For doing this a fpDrawEyes node is implemented which offers two special methods, a pre_draw_callback() and a post_draw_callback() (see Figure 5.13).

According to the frame count the pre_draw_callback() switches the geometry invisible through overriding the geometries material properties. This happens before the draw traverser renders the scene. After the rendering the post_draw_callback() switches the geometries visibility again through disabling the material override mode. When the draw traverser arrives again at the beginning of the next frame the pre_draw_callback() is not going to be executed as the Performer function pfGetFrameCount() increases the currentFrameCount by one only every second frame, as one frame consists of two images to be rendered (left and right eye).

Figure 5.14 shows the corresponding scene graph created by an Avango scheme script. The whole scheme script is shown in Appendix B.

Both fpLoadFile nodes (they are similar to the fpFile nodes) lay on top of

```
int fpDrawEyes::pre_draw_callback(pfTraverser* trav)
{
  long int currentFrameCount = pfGetFrameCount();

  if (( _whichEye && currentFrameCount == _oldFrameCount) ||
      (!_whichEye && currentFrameCount != _oldFrameCount) )
    {
     pfOverride(PFSTATE_TRANSPARENCY, PF_ON);
     pfOverride(PFSTATE_FRONTMTL,     PF_ON);
     pfOverride(PFSTATE_BACKMTL,      PF_ON);
    }
  _oldFrameCount = currentFrameCount;

   return PFTRAV_CONT;
}

int fpDrawEyes::post_draw_callback(pfTraverser* trav)
{
  pfOverride(PFSTATE_TRANSPARENCY, PF_OFF);
  pfOverride(PFSTATE_FRONTMTL,     PF_OFF);
  pfOverride(PFSTATE_BACKMTL,      PF_OFF);
  pfPopState();

  return PFTRAV_CONT;
}
```

Figure 5.13: Implementation of fpDrawEyes' pre_draw_callback() and post_draw_callback(). The pre_draw_callback() switches the geometry invisible before the rendering whereas the post_draw_callback() switches the geometry visible again after the rendering.

each other since they have the same matrix transform and share the same parent node. Due to the switching of the fpDrawEyes nodes only one of them is going to be visible at a time. In the real implementation they are a little bit tilted to each other according to the user's eye position. This ensures that each eye's view direction stands orthogonal to the screen with the texture of the corresponding camera.

5.3 Conclusions

This chapter described the CVE implementation details with respect to the used hardware and the software configuration. The hardware configuration

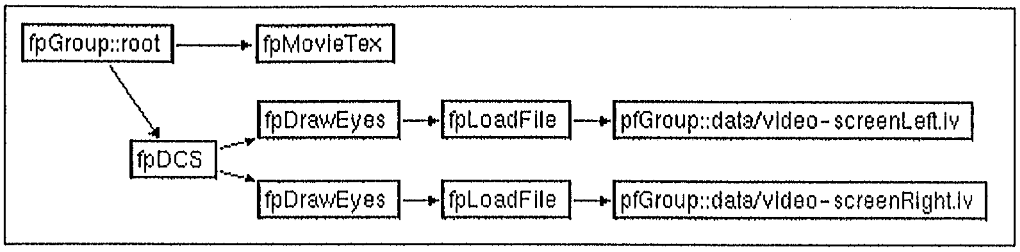


Figure 5.14: Scene graph showing the fpDrawEyes node. This node is responsible for switching between the two overlaying video textures.

is explained for a distributed setup using two Responsive Workbenches, and for a distributed setup using a Responsive Workbench and GMD's cave-like CyberStage. These two configurations include input devices for interaction, computers for rendering and distribution, and computers for video and audio streaming. Additionally it includes equipment like shutter glasses, infra-red emitters, cameras as well as microphones and headphones. This equipment can be used for any combination of display systems, including even more than two.

The software section described the Avango software framework which is used for rendering and distribution. Thereby Avango's field interface is introduced as well as the importance of field connections. Additionally it is explained how comfortable distribution mechanisms make use of this field interface in order to handle the database duplication problem. The remainder of this chapter describes the video conferencing software with respect to its mono and stereo video conferencing capabilities. A solution for sending synchronized stereo video frames is introduced as well as a solution for the integration and rendering of the synchronized stereo images in the CVE.