Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

LONG DISTANCE DISTRIBUTION OF VIRTUAL AND AUGMENTED REALITY APPLICATIONS

Ausgeführt am
Institut für Softwaretechnologie und Interaktive Systeme
der Technischen Universität Wien

unter der Anleitung von
Univ.Ass. Mag.rer.nat. Dr.techn. Hannes Kaufmann

durch
Mathis Csisinko
Treustraße 49/34
1200 Wien

5. Mai 2006

Abstract

Collaboration is an intriguing and promising aspect in *Virtual* and *Augmented Reality* systems, whether participants are co-located or not. At least, distant *collaboration* requires distribution capabilities to replicate the *Virtual Environment* on all participating sites.
Studierstube is a *Collaborative Augmented Reality* system, providing *reliable* distribution in a limited way. This thesis extends these capabilities to long distance distribution, supporting ordinary *IP*-based networks like the *Internet*. *Collaboration* between participants located in different cities, countries and even continents becomes possible.
In order to apply these new features, *Construct3D*, a dynamic geometric construction tool in 3D for educational purposes, was adapted. By reducing the data amount to transmit, distribution efficiency was increased. Furthermore, robustness, flexibility and *scalability* capabilities were improved. Distribution and *collaboration* features directly profit from all of these efforts.

Zusammenfassung

Kollaboration ist ein faszinierender und vielversprechender Aspekt von *Virtual* und *Augmented Reality* Systemen, unabhängig von der Tatsache, ob die Teilnehmer am selben Ort versammelt sind. Zumindest *Kollaboration* über Entfernung erfordert die Fähigkeiten zur Verteilung, um das *Virtual Environment* an jeden teilnehmenden Ort replizieren zu können.
Studierstube ist ein *kollaboratives Augmented Reality* System und stellt *zuverlässige* Verteilung in einem beschränkten Ausmaß zur Verfügung. Diese Diplomarbeit stellt eine Erweiterung dieser Fähigkeiten um die Möglichkeit zur Verteilung über weite Strecken auf herkömmlichen *IP*-basierenden Netzwerken wie dem *Internet* dar. *Kollaboration* zwischen Teilnehmern aus verschiedenen Städten, Ländern, ja sogar Kontinenten wird damit möglich.
Um diese neuen Möglichkeiten anzuwenden, wurde *Construct3D*, ein dynamisches Konstruktions-Programm für 3D-Geometrie zu schulischen Zwecken, adaptiert. Durch Reduktion des zu übertragenden Datenvolumens ließ sich die Verteilungs-Effizienz steigern. Des weiteren wurden Eigenschaften betreffend Robustheit, Flexibilität und *Skalierbarkeit* verbessert. Die Fähigkeiten zur Applikations-Verteilung und *Kollaboration* profitieren von all diesen Bemühungen direkt.

Acknowledgements

As I finally completed this thesis, I want to thank everybody, who made this possible. First of all, thanks to my parents for supporting me during my years of being an undergraduate student. Many thanks go to my supervisor Hannes Kaufmann for his guidance and enormous patience. I wish to express thanks to Gerhard Reitmayr for providing creative input and background information, especially in the difficult initial phase of this work. Special thanks also to Alexander Bornik for providing a remote testing environment in Graz and helping me to resolve all building problems. For giving me the chance to present my work in Graz, I also want to express thanks to Prof. Dieter Schmalstieg. Big thanks also to Morten Erikson, chief developer of Coin3D, for his support while creating an add-on patch. Not to forget, I would like to thank everybody involved into proofreeding this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern technologies like *Virtual* and *Augmented Reality* have emerged in the past as interesting media for work, education, and even entertainment. In the meantime, these technologies have matured to the point, where they can be applied to a wide range of application domains. For example, they can assist in daily life, allowing new forms of knowledge aquisition, making use of objects not present in the real world, manipulating them and interacting with them.
The ability to collaborate with collegues is an additional fascinating and very useful concept in context to *Virtual* and *Augmented Reality*. It allows working together on one project by utilizing modern communication technologies, even if some participating people are geographically located hundreds of miles away.

1.1 Problem statement

As sharing a virtual workspace offers the possibility to work together with people cituated all over the world, distribution and replication of data has to be taken into account.
Ideally, data transmission should be fast to achieve quick response times. Especially in long distance distribution this aspect is crucial, as the travelling time usually depends on the distance to cover. Also data transmission has to be (in most cases) reliable. As an additional attempt for fast response times, to prevent network congestion, and to increase efficiency, transmitted data should be reduced to the lowest possible amount.
Depending on the application, distributed data can contain several aspects.

- Input data distribution:
  By sharing *tracked* input device data, the actions and movements of other participants can be visualized, especially of those working in distant locations.

- Output data distribution:
  With replication of application content, directly perceivable by participants, the illusion of working closely together and collaboration in a shared environment can be created, even if participants are geographically separated. Traditionally, application content stimulates mainly the human visual sense.

- Intermediate data distribution:
  Sharing the application state, in the form of compacted meta information reduces the amount of transmitted data. This allows to regenerate some parts of application content by additional processing without the need for network transmission.

To help creating the illusion of being immersed and involved in the shared workspace, the application state has to stay consistent for all participating sites to a certain degree. This implies that each participant perceives a similar stimulus of the objects in the workspace, although some slight differences may be observed. Viewing conditions depending solely on the relative position and custom settings of a single participant are common examples of these perception differences. User roles to achieve different appearance or hiding certain parts of shared information from users with lower privileges may be also desirable.

1.2 Contribution

Hesina [Hes01] introduced distribution features in the *Studierstube*[1], an *Augmented Reality* software framework (for details see section 2.1 and section 2.8). A series of remaining shortcomings had to be resolved, as these features were more or less restricted to small local networks, because the implementation makes use of a special networking mode. This networking mode called *multicast UDP* (for details see section 2.2) lacks of support for long distance distribution: As a major drawback, *multicast UDP* traffic cannot go directly beyond the borders to world wide networks as they exist today. Thus, *multicast UDP* data packets are usually not *routed* into the world wide *Internet*, even if it is possible to *route* between two private local networks.
This work is supposed to fill this gap, enhancing distribution features, overcoming this rigid restriction in terms of networking, and offering the possibility to truly distribute *Studierstube* applications around the world. Beyond that, this work also covers the aspect of distributing data of *tracked* input devices (for details see section 2.4), as this is also part of shared data used by *collaborative* applications.
As another aspect of this work, an example application called *Construct3D* [KSW00] (for details see section 2.9) was selected to make in-depth long distance distribution tests. As useful tool in geometric education, it allows creating and editing of three-dimensional geometric constructions in an intuitive way by participating in a *Collaborative Augmented Reality* system.
Apart from extended distribution functionality, the replication behavior of this application was adapted. Instead of transmitting the whole application state including full visual representation, distribution is restricted to proper state data, capable of regenerating all remaining application state. This means that distribution is basically reduced to invisible meta information. This meta information consists of application commands, containing data essential for regenerating the application state. Processing these commands generates also instantly visible geometric results to the observer.
Another huge amount of this work was spent into massively increasing stability of present features by bug-fixing and reimplementation as well as extending

[1] *http://www.studierstube.org/*

them and introducing new functionality to push the application further. Details on this important part are not always easily describable. Nevertheless, some aspects are included throughout this thesis and located at appropriate places. Huge efforts are put into enhancing flexibility, concentrating on multi-user as well as distribution capabilities, application operation history and layer features.

1.3 Structure of this thesis

Chapter 2 gives a rough overview of related work as well as theoretical definitions and fundamentals needed later. It also shortly outlines the frameworks used in the implementation.
In chapter 3, design aspects of the implementation are thoroughly presented. Apart from the contribution, preexisting functionality is also described in short, especially if no or only few documentation exists. Also, well-documented aspects are depicted focussing on the context of this work.
Chapter 4 is going further into implementation details by supplying additional information about all design issues.
Finally, chapter 5 concludes this thesis by presenting results and outlining future work possibilities.

Chapter 2

Related work

Before presenting related work, some necessary theoretical foundations have to be described to define *Virtual* and *Augmented Reality* as well as basic networking terminology. This will be useful for characterizing distribution capabilities of selected related work as well as frameworks and applications depicted throughout this work.

2.1 Virtual and Augmented Reality

Virtual and *Augmented Reality* are closely related to each other. In the concept of *Virtuality Continuum*, as proposed by Milgran and Kishino [MK94], these two technologies differ in the proportions of combination of real and virtual world.

2.1.1 Virtual Reality

According to Rheingold [Rhe91], *Virtual Reality* is an environment, where a person experiences being surrounded by a three-dimensional computer-generated representation of an artificial world. In this environment, the person is able to move around and see it from different angles. He can reach into it, grab it and reshape it.
This is a definition closely related to optical senses, but *Virtual Reality* environments (*Virtual Environments*) applying to nearly all human senses increase the sensation of being fully immersed.

2.1.2 Augmented Reality

Augmented Reality differs from *Virtual Reality* by the way the real world is treated. While in *Virtual Reality* the real environment is completely replaced by a virtual counterpart, *Augmented Reality* enhances the real world by superimposing or compositing with virtual objects. So, while being immersed in *Augmented Reality*, some coexistance of the virtuality and reality can be observed. Even better, a user experiencing *Augmented Reality* should discover that the seams between virtual and real realm are beginning to blur.
The following characteristics apply to *Augmented Reality* systems, as defined by

Azuma [Azu97]: These systems combine real and virtual, are registered in 3D and interactive in real time.

2.1.3 Collaborative Augmented Reality

Providing *collaboration* features is an intuitive logical next step to enhance *Augmented Reality*. These *collaboration* aspects offer rich possibilities, allowing several users to work together. Billinghurst and Kato [BK99] pointed out the benefits of *Collaborative Augmented Reality* systems.
Collaboration should not be restricted to participants using a single computer system. In order to work together with users located at different places somewhere around the world, networking and distribution techniques have to be integrated into *Collaborative Augmented Reality*. As *Collaborative Augmented Reality* relieves from the need of being co-located (physically present in a shared workspace), this illustrates how naturally important networking is to overcome geographical boundaries, building somehow a symbiotic relationship between these two distinct concepts.

2.2 Networking

Networking is a key aspect to long distance distribution: Amounts of data have to be transmitted over network lines to achieve the goal of working collaboratively together in a shared workspace.

2.2.1 Networking basics

Before describing distribution capabilities, some key terminologies on networking have to be explained.

Network definition

A network links several *nodes* (i.e. computers) together to share ressources. The network hardware including connection lines and interfaces can be interpreted as the physical network. In the software realm, networks are often seen on a logical level when considering, how other network participants are discovered and data is exchanged.

Network topology

Network topology describes the pattern of *links* of connecting pairs of *network nodes*. These patterns, as there are several possibilities, are usually depicted by a shape (*line*, *star*, *mesh*, ...). *Network topologies* often depend on the point of view: *Physical network topology* refers to the geographical layout, while *logical network topology* describes the path, data takes as it travels through the network.
The *topology* on the physical level may be completely different from the one on the logical level. Today, *Ethernet* is best characterized by *star topology* on the physical level. But there is no problem achieving *mesh topology* on the logical level by running a common file-sharing protocol. (In its beginnings, *Ethernet*

originally conformed physically to *bus topology*, but as technological progress was made, it complies more to *star topology* by now.)

Network interconnection

Networks can be interconnected by special devices, classified by their features: *Repeaters* represent the simplest device type. Also known as *hubs*, these devices retransmit physical signals without further investigation between a number of networks with identical physical specifications, while *bridges* and *switches* forward network traffic even between physically heterogenous *network segments*. The latter manage this by inspecting and interpreting traffic data. As a consequence of this, these devices usually have to maintain at least a local view about the network structure.

Server and client roles

A *server* provides network services to other computer systems, consequently called *clients*. Network participants (*hosts*) with ambigous classification, whether acting as *servers* and *clients* alternatively or simultaneously, are simply called *peers*.
A common *server-client scenario* is that a *client* actively contacts a *server*, which is passively listening for incoming *requests*. After handling the *request*, the *server* sends a proper *response* back to the *client*. This mechanism is also known as *request-response mechanism*.

Data flow directionality

If sender and receiver roles are predefined and do not alternate, data always travels in one direction. The opposite of this *unidirectional* property is *bidirectionality*, where data can be transmitted in either direction of a connecting line.

Network connection types

Two different *connection types* exist: In *connectionless* networks, packets of data are exchanged, whereas *connection-oriented* networks establish true connections, even if they are only virtually present on a logical level. Packets in *connectionless* networks are also called *datagrams*, while data exchange in *connection-oriented* networks is often handled in a *stream* of continuous data.

Unicast, multicast and broadcast data delivery

Considering which *nodes* receive data, there exist three types: *Unicast* (*one-to-one*) delivery features a single receiver, while *broadcast* (*one-to-all*) describes the mode, where each participant of the network receives the same data. In between exists *multicast* (*one-to-many*) mode to address a configurable number of *nodes*.

Network reliability

In *unreliable* networks, there is no guarantee that transmitted data actually reaches its destination. Furthermore, it is even not necessary to report an error,

if transmission failures occur, no matter if they are of *permanent* or *transient* nature.
Reliability in networks states that *omission failures* (data loss) are masked. This is typically achieved by basic acknowledging and retransmission strategies. Unmaskable transmission errors have to be reported to the sender, causing indication of failure to the sending process. Such strategies to ensure *network reliability* on the logical level are needed to weed out the weaknesses of physical *unreliable* networks.

Network scalability

Scalability describes the capability of a system to increase total throughput under increased load, when additional ressources are added. In terms of networks, this increase can be heightening the number of *nodes*, while maintaining system performance and usability at the same time. In network systems without *scalability* features, performance could decrease, when adding a number of extra *nodes* participating in the network.

2.2.2 Network protocols

Network protocols describe sets of rules of how communication is handled between participants of a network. These rules cover various aspects including data representation, error detection, and connection negotiation.
Stacking is a common mechanism to combine features of several *protocols*.
In the following some key *protocols* of the *Internet* are described.

IP

IP (*Internet Protocol*) is one of the most fundamental *network protocols*. With *IP*, each *host* in a network can be addressed by means of an *IP address*. This allows connecting heterogenous networks together and directing network data packets properly so that the destination is reached by maintaining information about network interconnections. This important network traffic forwarding process is called *routing*.
Routers perform interconnection of several networks on a higher protocol level than *bridges* and *switches*, as *routing* devices have to inspect *IP* header data. *Routing* capabilities can be integrated in any *host* participating in the *IP* network. But these features can also be implemented by special devices for the single purpose of *routing*.
A *firewall* is a *host* to control network traffic for security issues. It is characterized by performing network data filtering on a per packet basis. By inspecting properties of each packet like source and destination address, this filtering strategy is defined by a set of rules.

TCP

TCP (*Transmission Control Protocol*) is another part of the well-known *TCP/IP protocol suite*. Being a *reliable protocol* basing on *IP*, it establishes virtual *connections* between two participants. Maintaining these *connections* requires some synchronization overhead. In this *unicast*, *point-to-point connection*, data exchange is abstracted by *streams*, delivering

payload correctly in-order.
To allow multiple independent applications running *TCP* on a single *host*, *port* information is introduced: *Socket* information, containing *IP address* and *port number*, uniquely identifies an application.

UDP

UDP (*Universal Datagram Protocol*) is the *unreliable*, *connection-less* counterpart to *TCP*, offering networking by means of *datagrams*. Usually *unicast*, but also offering (limited) possibilities for *multicasting* and *broadcasting*, it allows efficient data exchange without much data overhead and synchronization effort. In *multicasting* and *broadcasting* mode, packets usually cannot go beyond *routers* (mainly because of *scalability* reasons and limited network ressources).
UDP introduces *ports* in the same way as *TCP* does, resulting in a similar *socket* definition.
The *MBONE* (*multicast backbone*) [Eri94] network is an attempt to be capable of *routing* also *multicast* network traffic under certain conditions. *Tunnelling* otherwise *unroutable* traffic (encapsulating *unroutable* data packets in *routable* packets) is another strategy. However, these possibilities are no general option in long distance distribution because of the requirement of being part of an additional network or difficult specific configuration. Also, *MBONE* is not likely going mainstream.

2.3 Tracking data distribution systems

Specialized software frameworks fill the gap between *tracking* devices and *Virtual* and *Augmented Reality* toolkits. These *middleware* systems offer services for other software systems and focus on generalization of *tracking* data format in order to support various different input devices. In addition to these abstraction capabilities, these frameworks usually allow device data preprocessing and network transmission.
Currently, *VRPN* and *OpenTracker* belong to the major *tracking* device frameworks with comprehensive features. While *VRPN* is described in the following, details about *OpenTracker* can be found in section 2.4.

2.3.1 Tracking

Tracking is, in context to *Virtual* and *Augmented Reality* systems, measuring *poses* of bodies (whether subjects or objects) in space. *Poses* consist of positional information and orientation, giving six *degrees of freedom* (*6DOF*). Not all devices are able to support *6DOF*, usually because positional or orientation information is completely missing and cannot be regained.
When thinking of a universal *tracking* data format, some characteristics can be observed. Usually, input device data can be classified into two major categories:

- Accurate positional and orientation information (*pose*) has to be present at any instant. This is best realized by *streaming* capabilities. Data from the previous instant is constantly replaced by more recent information.
- Further information usually does not change as often as positional data. Therefore, this data can be treated as *events*, reflecting the occured state

change. Moreover, this can be also handled by integrating it into the *pose* information *stream* and neglecting its *event* type nature. Examples of this additional data include digital button press events and analogue device data.

Tracking device data flow is usually *unidirectional*: Data is transferred from the origin, the input device to the sink, the interface to the *Virtual* or *Augmented Reality* system.
An example of an input device used in the *Studierstube*[1] [SFH+02] environment is shown in figure 2.1: This pen's position and orientation is *tracked* optically by evaluation of the positions of attached spherical markers with retroreflective surface. Being wireless, a button event is transmitted by radio signal to a receiver station.

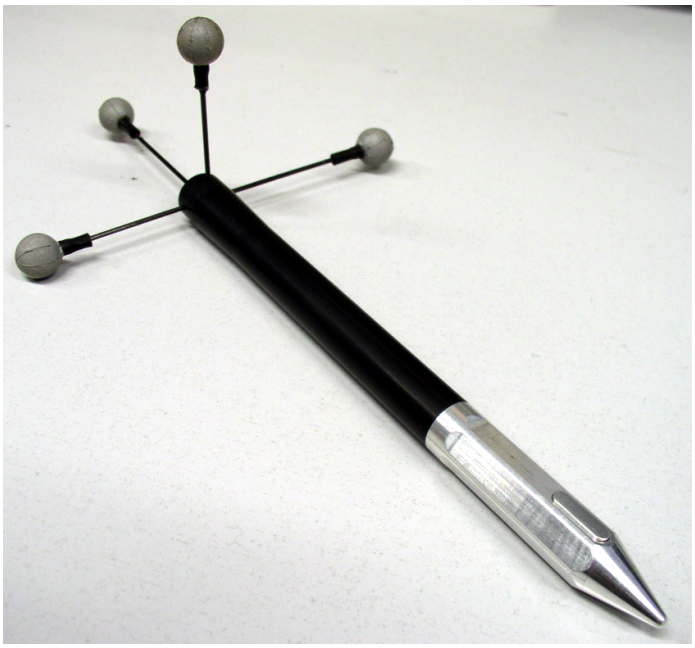

Figure 2.1: Example of a wireless input device: An optically *tracked* pen

The evaluation, mentioned before, is performed in a postprocessing step by inspecting the images taken by a set of cameras like the ones shown in figure 2.2: The reflections of the infrared light beams, actively transmitted by each of these cameras, are used to geometrically calculate position and orientation.

2.3.2 VRPN

VRPN[2] (*Virtual-Reality Private Network*) [THS+01] is an example of a device-independent and network-transparent framework for peripheral devices used in *Virtual* and *Augmented Reality* systems. It is written in *C++*.
Devices are classified into a wide range of different types, depending on the kind

[1] *http://www.studierstube.org/*
[2] *http://www.cs.unc.edu/Research/vrpn/*

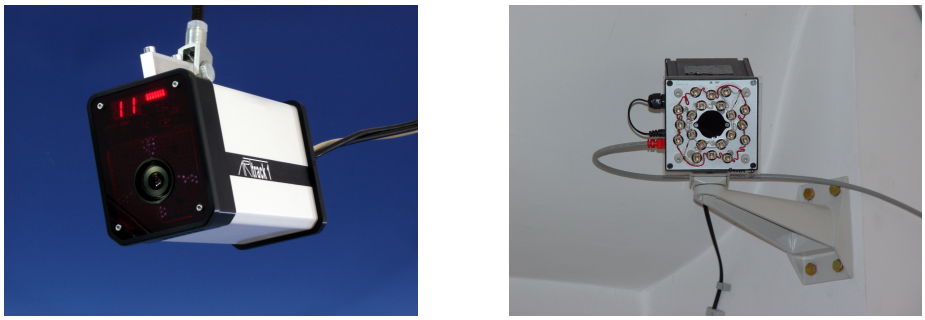

Figure 2.2: Optical *tracking* devices: On the left an *ARTTrack1* camera (provided by *A.R.T.*), on the right a camera used in an alternative custom setup, developed by the *Vienna University of Technology*

of *tracking* data: *Pose* data, button states, analogue values and incremental rotations belong to these data types. A device can offer interfaces for several types, and devices can be layered by connecting device outputs to inputs of other devices.
Networking is built upon *UDP* and *TCP*. Depending on the *reliability* delivery property of *tracking* data type, the protocol is chosen on a per message basis.

2.4 OpenTracker

Virtual and *Augmented Reality* applications depend on input data typically supplied by special input devices covering spatial information. These *tracking* devices usually generate a continuous *stream* of input data, an application has to work on. But data coming from these devices may require some preprocessing like filtering, merging and transformations before.
OpenTracker[3] [RS01], written in *C++*, is a universal and configurable framework being capable of performing such operations on *tracking* data. A wide range of *tracking* devices are supported by implemented *modules*. Support of new input devices is achieved by extending *OpenTracker* with additional *modules*. *Tracking* data records, as an attempt to achieve device abstraction, support devices with up to six *degrees of freedom* (*6DOF*): typically spatial information containing a position in space and orientation. An adequate number of buttons, confidence information and timestamps are also part of this record. This *middleware* solution is used in *tracking* device support in the *Studierstube*[4] [SFH+02] framework (for details see section 2.8). It is implemented by an integrated *OpenTracker* client in this *Virtual* and *Augmented Reality* system. This also works well in combination with stand-alone *OpenTracker* instances.

2.4.1 Node concept

An *OpenTracker* specification is given in an *XML* file, representing the *data flow graph*: *Tracking* data is typically inserted into this graph by *nodes* called *sources* and forwarded to external outputs by *sinks*. Intermediate

[3] *http://www.studierstube.org/opentracker/*
[4] *http://www.studierstube.org/*

nodes are also called *filter nodes* and perform preprocessing mentioned before. *Tracking* data flows along the directed edges of the graph, occuring from *sources*, processed by *filters* and reaching its final destination in form of *sinks*.
An illustration of how *tracking* data is propagated from *sources*, passing *filter nodes* along the edges in a *directed acyclic graph* to *sinks*, is given in figure 2.3.

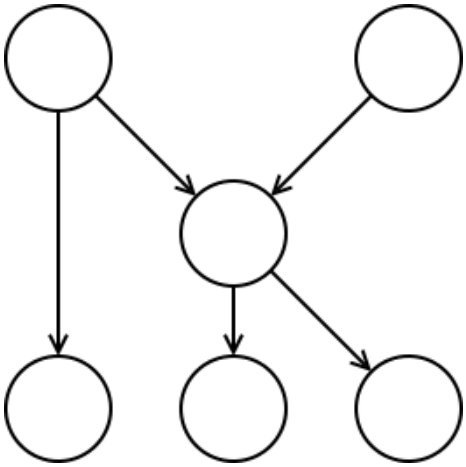

Figure 2.3: Example of a *data flow graph* in *OpenTracker*

2.4.2 Module concept

Modules are used in *OpenTracker* to apply some further structuring: A *module* usually represents a certain *tracking* device type or function.
Modules are capable of performing actions common to all *nodes* of certain type and, when implementing the *node factory* interface, handle *node* creation.
This concept is also reflected in the *XML* configuration file: With *modules*, it is also possible to apply some global configuration to all *nodes* of certain type.

2.4.3 Tracking data distribution

A key feature of *OpenTracker* in distribution is to transmit *tracking* data over networks. This can be simply done by means of special *sinks*. On the other side, corresponding *sources* pick up received data from the network in *OpenTracker* instances on other hosts. Inserting it into the *data flow graph*, received *tracking* data is treated just like input data occured from any other true *tracking* device: To *OpenTraker*, there is simply no difference, where data comes from.
Usually, *tracking* data transmission has not to be *reliable*. Dropped data is likely to be replaced by more recent *tracking* data just in the next instant. Thus, missed data has no negative impact on the application.
Originally implemented networking functionality is built upon *multicast UDP* and therefore not practicable for long distance distribution. But there is also support for *VRPN* (*Virtual-Reality Private Network*) [THS+01]: An *OpenTracker* program can receive data from a *VRPN server*, but is also capable of acting like a *VRPN server* as well.

2.5 Scene graphs

In 3D rendering systems, the *scene graph* concept is often and widely used to achieve a certain degree of database structuring. Other database approaches usually lack in suitability for rendering needs.
A *scene graph* is a common data structure used in vector-based graphics (mostly 3D rendering) systems. Being of *tree* structure, it resembles the rendering content in a hierarchical, object oriented way: A single *tree node* resembles a graphic element (for example a geometric primitive, visual appearance settings or a transformation operation). The *tree* arranges these elements logically and usually also spatially.
Scene graph trees are usually *directed acyclic graphs* (although in *fractal* applications *cyclicality* can be useful). The *directional* nature resembles the *parent-child* relation of each *node*. Figure 2.4 illustrates an example of this graph type.

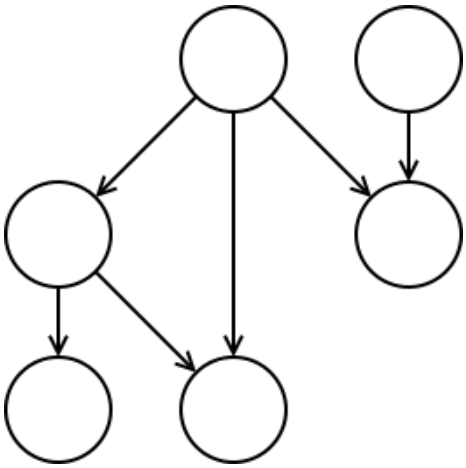

Figure 2.4: Example of a *directed acyclic graph*

If each *node* must have at most one *parent*, the *scene graph* has exactly one *root node*. In order to effectively reuse some *scene graph* parts, this restriction is too obstructive: To add a certain part several times (as also illustrated in figure 2.4), more than one *parent node* has to be allowed.
A common operation on a *scene graph* is the so-called *traversal*: The *tree* is processed (*traversed*), starting from a certain *node* (usually a *root*), visiting redundantly all *child nodes*. This *traversal* mechanism is excessively used in rendering. But also a wide range of other preprocessing, calculation and search operations take advantage from this mechanism. An important property of this mechanism is the visiting order, whether deterministic or not. By preventing *cyclicality*, *traversal* operations in the *scene graph* cannot consume infinite processing time: A *direct acyclic graph* implicitely guarantees termination of the *traversal* algorithm, which is usually implemented in a recursive way.

2.5.1 Distributed scene graphs

Although *shared memory* systems are capable of directly sharing data, they have additional hardware requirements. Granting access to memory for several participating sites is complicated and requires synchronization and locking strategies. Distributing and replicating *scene graphs* among heterogenous computer systems do not require any additional hardware. Considering this replication mechanism and the concept of *shared memory*, this replicated data can be seen as residing in *distributed shared memory* [CD88].
In the following, some *scene graph* systems with distribution features are shortly presented, including *blue-c Distributed Scene Graph*, *Avango* and several *Distributed Open Inventor* attempts.

2.5.2 blue-c Distributed Scene Graph

blue-c Distributed Scene Graph (*bcDSG*) [NLSG03], written in *C++*, is based on the *OpenGL Performer*[5] toolkit [SC92]. Distribution features are built on top of this framework and therefore cannot be integrated into *Performer*.
The whole *scene graph* can be divided and split into a shared and local partition. Shared parts of the *scene graph* have to be created solely using customized *nodes*, as standard *Performer nodes* do not work in distribution. This is caused by the unsuitability of the *node* identification mechanism (using pointers to memory), when crossing local machine boundaries. So, it is required to replace this information by a globally unique identifier among all participants. In the implementation, this identifier is assigned on a central session manager site.
Scene graph synchronization is performed in a *traversal* operation each rendering frame. This mechanism includes consistency, locking and ownership features. A relaxed locking scheme was implemented: Manipulations on any *node* are possible, requesting and claiming ownership by a *handshaking protocol* directly after these modifications. Even though this results in temporal inconsistencies, they are usual acceptable.
Data transfer is done with the *UDP* protocol, enabling *multicasting* support in system setups with more than two participating sites: While *scene graph* synchronization messages are transmitted to any participating site, locking operations are of *unicast* nature. For a very high number of participants, *unicast* data channels are eliminated and their network traffic is transferred to their *multicast* counterparts. On startup, network communication channels are established using *CORBA*.
Relying on *multicast UDP* and its *routing* deficits, the system is not suitable for long distance distribution. Apart from the requirement of exclusively using customized *nodes* in the *scene graph*, its synchronization features are based on *nodes* as atomic units: Changing a single *field* causes the whole *node* contents to be transferred. This can be problematic, when having a huge amount of data belonging to a single *node*. Also, late joining of participants was not originally implemented.

[5] *http://www.sgi.com/products/software/performer/*

2.5.3 Avango

Avango (formerly known as *Avocado*) [Tra99] is also based on *Performer*[6] [SC92] and implemented in *C++*. Similar to the *Inventor* toolkit [Str93], its own customized *scene graph nodes* act as *field containers*, storing data in terms of *fields*. Also the *field connection* and streaming mechanism are introduced and much like in *Inventor*.
Distribution features are based on so-called distribution groups. To build a shared object, first a local object has to be created and migrated to a distribution group. On the other side, all group members reverse this process by creating a local copy from this distributed object.
Networking is based on *Ensemble* [Hay98], making use of the *Maestro* toolkit. It seems that nonstandard micro protocols are used in network communication, but there is no precise information about this.
Late joining is supported by *Avango*, transferring the *scene graph* properly. However, *Avango* relies completely on its subclassed *node* types in distribution.

2.5.4 Distributed Open Inventor

Distributed Open Inventor is based on *Open Inventor*, a popular rendering toolkit thoroughly described in section 2.6. Several attempts in adding distribution features exist.

Hesina's approach

Distributed Open Inventor (*DIV*) [Hes01], as being part of the *Studierstube*[7] [SFH+02] toolkit (for details see section 2.8), is one implementation, elaborately described in section 2.7.

Pečiva's approach

A somehow similar idea was implemented by Pečiva [Peč02], also based on *master-slave* architecture: Opposed to previously presented solutions built on top of *Performer* [SC92], *Open Inventor* is modified directly. This is achieved by enhancing and extending the open source code base, written in *C++*. Benefiting from this, *scene graphs* can be easily set up for distribution without replacing each standard *node* by a customized counterpart.
The powerful *field connection* mechanism is enhanced by network capabilites. *Field containers*, to where *nodes* and *engines* belong, are also made network aware by additional methods. *Groups*, a special *node* type, being capable of storing *child nodes*, are enhanced by additional functionality targeted on these *children*. A custom *node* type supports sharing and distribution of a whole *scene graph*.
Networking is currently built upon the *Parallel Virtual Machine* (*PVM*) [GBD+94] framework, but subject to change.

[6] *http://www.sgi.com/products/software/performer/*
[7] *http://www.studierstube.org/*

Open Inventor by TGS

Open Inventor®[8] from *TGS*, recently acquired by *Mercury*, contains internal distribution management code, obviously and remarkably very similar to original *DIV* by Hesina. As this *TGS* implementation is not open source, it is not clearly evident, if and how this functionality is actually used.

2.6 Open Inventor

As *Virtual* and *Augmented Reality* are rooted in the real world, both of them inherit its 3D nature. *Open Inventor* is a *scene graph* based rendering library, offering an interface of *C++* classes with the ability to represent so-called *scenes* of (especially 3D graphical) objects. So it focuses on these objects, not on drawings.
Open Inventor is the open-source version of original *Inventor* [Str93]. The implementation used throughout this work is *Coin3D*[9] from *Systems in Motion* (*SIM*) and built on top of *OpenGL*[10].
In the following, some basic *C++* class types of *OIV* are presented, as also featured in the popular and famous *Inventor Mentor* [Wer93]:

- *Nodes* are basic *Open Inventor* (*OIV*) units and contain functionality to serialize to and read data from a stream of *ASCII* based or binary data. Being a basic *C++* class, these *nodes* cover also runtime type information. *Nodes* are part of the *scene database*. A hierarchical *tree* of *nodes* can be created by means of special *nodes* called *groups*. These *groups* can contain a number of *child nodes*. *Parent groups* and *child nodes* are fundamental to build up the hierarchical *scene graph*.
  Scenes contain one ore more *nodes* grouped together.

- *Node* information and associated data is typically stored by means of *fields*, another basic *OIV C++* class. *Fields* contain data of of simple (like string, integer and floating point numbers) or basic (like vectors and matrices) type. Considering the number of data elements, *fields* can be distinguished into two different types: *Single-value fields* contain exactly one data element, whereas *multiple-value fields* store a list of elements of the same data type. Just like *nodes*, *fields* support runtime type functionality and serialization ability.
  Apart from that, *fields* can be connected together to keep data synchronized between a *master field* and its connected counterparts called *slave fields*.

- As third fundamental *OIV* data unit, *engines* are also capable of aggregating *fields* and feature streaming capabilities as well as runtime type information. So *engines* belong to the important *field containers* group, just like *nodes* do. *Engines* are used to generate *field* data and manipulate data occuring from incoming *field connections*.

[8] *http://www.tgs.com/pro_div/oiv_main.htm*
[9] *http://www.coin3d.org/*
[10] *http://www.opengl.org/*

- *Sensors* can be attached to several entities in the database, especially *nodes* and *fields*. This allows observing changes and react properly, as callback code is executed. Due to the dependency to program instructions in this *notification* and callback mechanism, *sensors* cannot be represented in persistent storage.

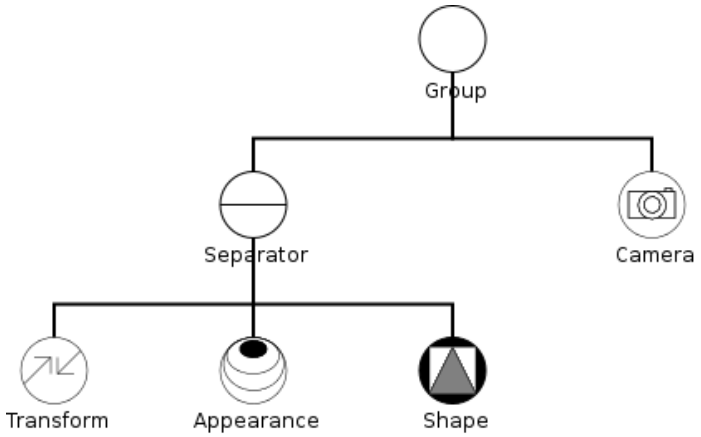

Figure 2.5: Example of a *scene graph* in *Open Inventor*

Nodes are the most generic *scene graph* elements. Usually they can be distinguished further into separate types:

- *Shape nodes* represent geometry and graphic primitives.
- *Property nodes* define the visual appearance of graphics including material properties, textures and lighting models.
- *Group nodes*, as already described, are essential for building up a *scene graph* hierarchy and can be equipped with various additional functionality.

Figure 2.5 shows a typical example of a *scene graph* containing some common *node* types. It also uses well-known *node* icons and indicates *scene graph* directionality as well as *parent-child* relationship implicitely by the vertical order instead of depicting arrows in graph edges.
Another important feature of *Open Inventor* is the *node kits* concept. *Node kits* encapsulate several *nodes* in self-contained subparts of the *scene graph*, allowing further restructuring of closely related elements in the *scene database*. *Node kits* are very powerful to compose compound and highly configurable *scene graphs*, acting like a single *node* and hiding their usually complex *scene graph* hierarchy from the outside.
In *Open Inventor*, *scene graph traversal* is done by applying an *action* to a starting *node*. *Actions* are passed down the *scene graph* recursively. Each *node* independently defines a proper behavior for each *action*. Rendering, searching, some computations, writing to stream and *event handling* are all done by applying certain *actions*. *Events* typically occur from input devices like mice and keyboards to allow interaction.

2.7 Distributed Open Inventor

Distributed Open Inventor (*DIV*) extends *Open Inventor* (see section 2.6) by distribution abilities. The whole or some selected parts of the *scene graph* can be shared among a network of participating sites. This is a fundamental prerequisite for the *collaboration* aspect. Sharing the *scene graph* avoids the *dual database problem* [MF98], being capable of distributing application and graphic library state altogether without separation.
Distribution makes use of the *notification* mechanism in *OIV* and observes occured changes by *sensors*. For convenience, a special group called *DivGroup* denotes a *subtree* for distribution, offering the possibility to share several independent parts of a *scene graph*.
Usually a single *master* hosts the original copy of the *scene graph* for replication to guarantee *total ordering* of messages. This *master* is responsible for transmission of *scene graph* changes to the network. In this transmission, the *node* name is used as unique identifier. So naming lies in the responsibility of the *master*. Considering that this is done just in the instant, when the name is needed the first time, it is called *lazy naming*. Naming messages are related to *nodes*, identified by *path* information. *Paths* contain a list of numerical indices, indicating the way to follow down to the target *node*, starting at the *scene graph root*. So *lazy naming* maps from *path* information to more convenient unique names. Except for its unhandiness, *path* information can change on structural *scene graph* modifications.
Scene graph modification messages, transmitted by the *master*, typically contain the name of the *node*, where the change occured. In addition to this, the remaining message body includes other information as well. This depends on the change, which has taken place. So, additional message data consists of:

- appropriate *field* name and data, if a *field* update occured,
- structural information, if the update is of structural nature (involving *group node* operations).

On the other side, *slaves* process received changes, modifying the *scene graph*. Initially, *slaves* are also capable of sending *polling* packets to the network, requesting the *scene graph* from the *master*. The *master* reacts on this message appropriately by transmitting the *scene graph* in its vcurrent state. This is actually an implementation of the late joining feature.
This predefined *master-slave* property can be dynamically changed: Any *slave* can request becoming the new *master*, waiting for acknowledgment of the current *master*. On approval, *master* status is transferred in the instance of this positive acknowledgment message. For consistency reasons, only one single *master* is recommended, while the number of *slaves* is irrelevant for this issue.

2.8 Studierstube

Studierstube[11] [SFH+02] (named after the german word for the study room of Goethe's famous character Faust, where he gained insight) is a system providing a framework for *Collaborative Augmented Reality* applications. It consists

[11] *http://www.studierstube.org/*

of a set of *OIV* extension classes to support rendering on various *Virtual* and *Augmented Reality* hardware devices and aquiring data input from 3D input devices.
The *Studierstube* framework integrates *DIV* for distribution capabilities and *OpenTracker*, supporting various input devices. It contains also widgets suitable for the 3D nature and an application concept to allow dynamic application loading and switching between them.
Figure 2.6 gives an idea about *Studierstube*, depicting two users collaboratively working on several applications, equipped with head mounted displays and interface devices in a highly immersive setup. Although wireless devices are more frequently used in these days, the idea is clearly transported. One major advantage of wireless devices is that they effectively diminish the feeling of being tethered.

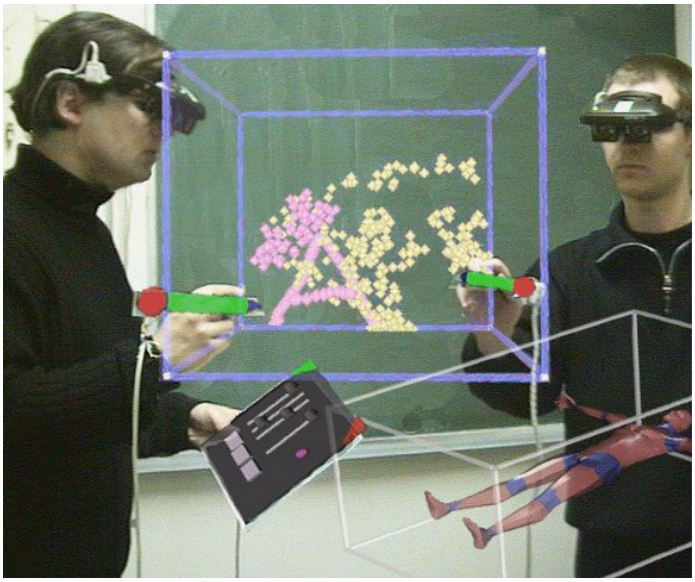

Figure 2.6: Two users collaborating in *Studierstube* in an early version

2.8.1 Tracking data processing

Studierstube acquires *tracking* data from *OpenTracker* (see section 2.4). This *OpenTracker* client seamlessly integrates into the framework. The interface is provided by means of special *sinks*. Such a *sink*, called *StbSink*, inserts data as *3D event* into the *scene graph* by applying a custom *action*. Just like *tracking* data in *OpenTracker*, these *3D events* consist of spatial (position and orientation) and button information. These *events* are passed down the *scene graph* by the *traversal* mechanism.

2.8.2 Application distribution

As previously published [SRH03], *Studierstube* includes distribution features of *Augmented Reality* applications based upon *Distributed Open Inventor* (see sec-

tion 2.7). Furthermore, distribution abilities are put on a higher level, making auto-configuration possible. This also allows applications to be implicitly distributed without further investigation.
All this is conducted by a special tool program, which manages parameters and settings of application distribution. As this program is known as *session manager*, it manages and maintains configuration data also referred as *sessions*. Further details about this *session manager* can be found in section 3.3.

2.8.3 Multiple users

In order to allow multiple users joining the shared workspace, a user concept is implemented to represent all participating sites by internal data structures. The number of users can be freely defined for each *Studierstube* instance, starting with at least one user.
Strongly related to user configuration are associated user ressources: Rendering output and interaction devices are specified on a per user basis.
Even if users can collaborate on a single application instance, common application scenarios typically include multiple hosts. This implies the need for *DIV* in *Studierstube*.
Utilizing their associated ressources, *collaborators* are able to define custom viewing preferences. Especially users supplied with head-*tracked* and head mounted displays profit from the ability to choose an individual viewpoint while retaining full stereoscopic graphics. This immersive hardware setup is chosen in figure 2.6.

2.8.4 Multiple applications

Studierstube supports dynamic loading of multiple applications. This concept includes the fact that applications are represented by *scene graphs*, avoiding the *dual database problem* [MF98]. By introduction of so-called *contexts*, application data, data representation and the application itself become united. *Studierstube* applications are composed by special *node kits* representing these *contexts*.
Multiple applications are accessible by a 3D interface analogue of the *multiple documents interface* in 2D. *Contexts* cover spatially a certain bounding volume, a 3D window (analogously to windows in 2D). The 3D window is visualized in figure 2.6 by wireframe boxes.

2.8.5 Multiple locales

By introduction of *locales*, multiple workspaces are possible, due to the fact that a *locale* represents a coordinate space. A *locale* embeds a number of applications and associated users. Workspaces can be separated from each other by defining multiple *locales*.
It is not mandatory that a *locale* uniquely corresponds to a physical place. This offers the possibility for long distance distribution and *collaboration*.

2.8.6 Personal Interaction Panel

An established user interfacing method in *Augmented Reality* applications is the *Personal Interaction Panel* (*PIP*) [SG96], as featured in *Studierstube*. As an *interface metaphor*, a user can be equipped with a panel and a pen, resulting in a two-handed interface. Both panel an pen are spatially *tracked* input devices. The pen is stylus-like, containing at least one button for interaction. The lightweight, notebook-sized panel is just a passive object without any built-in intelligence. It provides basic haptic feedback when interacting and can be also used as real world notepad. Joined together, these two devices compose the *PIP*.
As the panel is just a simple prop, it has to be enhanced and augmented by virtual objects (usually 3D widgets). The user is able to interact with these *PIP* interface elements using a pen's button to perform selections. The virtual representation of both objects allows also interaction with the interfaces of other users, even if they and especially their interface props are not physically located nearby. This enriches the *collaboration* aspect, for example, by offering the possibility to teach interaction skills. In particular, it eases becoming familiar with how to trigger certain application functions by watching a tutor interacting with his or even demonstrating on the student's own interface.
Although displayed in an early development stage, a typical *PIP* can be observed in figure 2.6: Panel and pen are augmented with widgets and other graphics. Nowadays, the panel usually consists of a generic application management interface and other elements, accessing functions of a certain application. These remaining elements, defined by the application are part of so-called *sheets* of a *PIP*.

2.9 Construct3D

As an application in *Studierstube*, *Construct3D* [KSW00] profits directly from the 3D nature of *Virtual* and *Augmented Reality* and offers new possibilities in geometry and mathematics education:
The geometric construction is directly created in the surrounding space. It is best seen in a highly immersive hardware setup. To name some advantages in favor of classical education, natural viewpoint adaption by walking around and *collaboration* techniques belong to the features. But perhaps one of the most important educational intentions of this application is to train the power of spatial imagination, as previously published [KSDG05]. Immediately evident results of dynamic manipulations help exploring the nature of geometry and its behavior to gain insight.
Construct3D supports multiple users working collaboratively together. The *Personal Interaction Panel* is used as interface and contains a menu-driven system, used to trigger all operations.
Apart from illustrating *collaboration*, figure 2.7 shows hardware devices used in a typical setup: A see-through head mounted display, a *PIP* and optionally a headset to provide input to the alternative speech interface is given to each user. A virtual projection table is located in the background for presentation purposes.

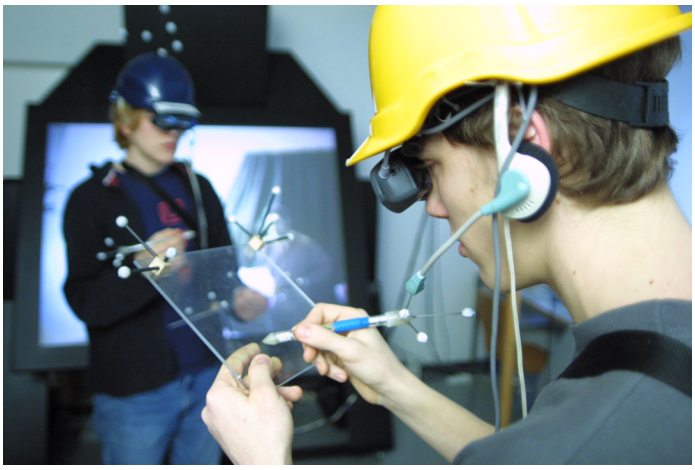

Figure 2.7: Hardware setup in *Construct3D* and two users collaboratively working together

2.9.1 Features

Application features include:

- Creation of some geometric primitives (points, lines, spheres, cylinders, ...) and basic objects (curves, planes, surfaces, ...) is available as well as the generation of compound objects by applying geometric operations (boolean operations, sweeps, transformations, ...).
  A preview function is offered for additional feedback.
- Dynamic modification of each geometric element is available at any time. This manipulation influences all depending objects, resulting into reevaluation and altering the geometric construction.
- By projecting 3D geometric constructions onto orthogonal planes, 2D views, best known from classic geometric drawings, are obtainable.
- Retaining the possibility to manipulate objects of other users, all objects are associated to a specific user and displayed in his own color theme.
- Several layers are present to enhance the capabilities for further logical structuring in geometric constructions.
- Persistant storage of construction results, based on *OIV* file format, is also supported to make basic loading and saving mechanisms available.
- An undo history provides more flexibility and control while editing.
- Distribution capabilities contribute to enriched *collaboration* aspects.

Construct3D handles geometric operations with the help of the commercial *3D ACIS® Modeler*[12] (*ACIS*, by *Spatial*). This geometry kernel is used in all geometry calculation operations and keeps track of all associated data. With

[12] *http://www.spatial.com/components/acis/*

such a powerfull toolkit, it is possible to update all depending objects of a geometric construction, when modifying a certain dependent object. Figure 2.8 shows a sample geometric construction result.

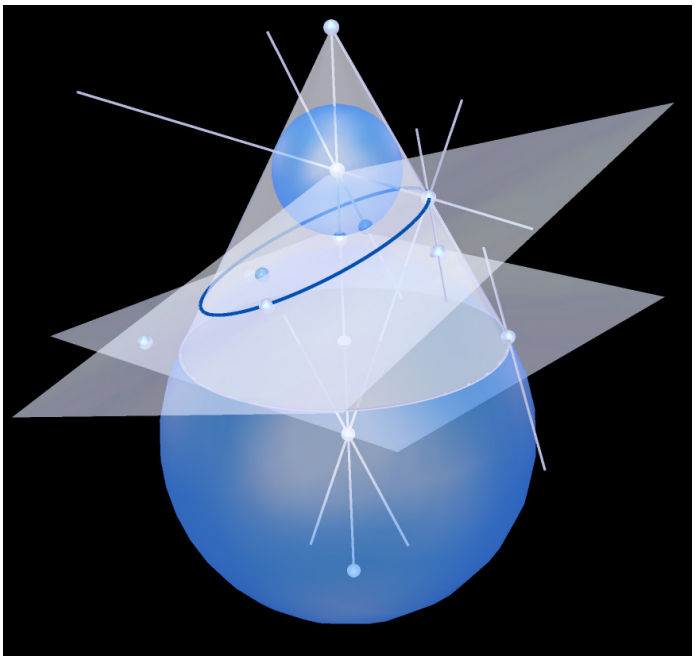

Figure 2.8: A result of a geometric construction in *Construct3D*

All geometric objects and results of operations are represented in a command list (a pleasent side effect of the implemented undo/redo feature). As this descriptive representation is already used in file storage, distribution features can also be built upon this.

2.9.2 Interface

Each actively collaborating user is given a *PIP* (composed by panel and pen, for details see section 2.8) to interact on, providing access to all application functions in conjunction with the surrounding space. Comparing figure 2.1 and figure 2.9, the similarities between the real pen and its virtual representation can be observed.
The construction process is actually carried out by interacting with the stylus-like pen in the surrounding space. The user's panel provides access to all application functions. In the working process of creating geometric constructions, objects are moved around and new objects are placed into the workspace. These two basic operations require spatial location information, which is directly provided by the *tracked* position of the pen. Similar in its requirements is the action of selecting an object for further operations: This is performed intuitively by pointing to its position and pressing the pen button.
The personal panel is customized and well-designed to fulfil the application needs better than in the generic *Studierstube* version. Its sophisticated menu

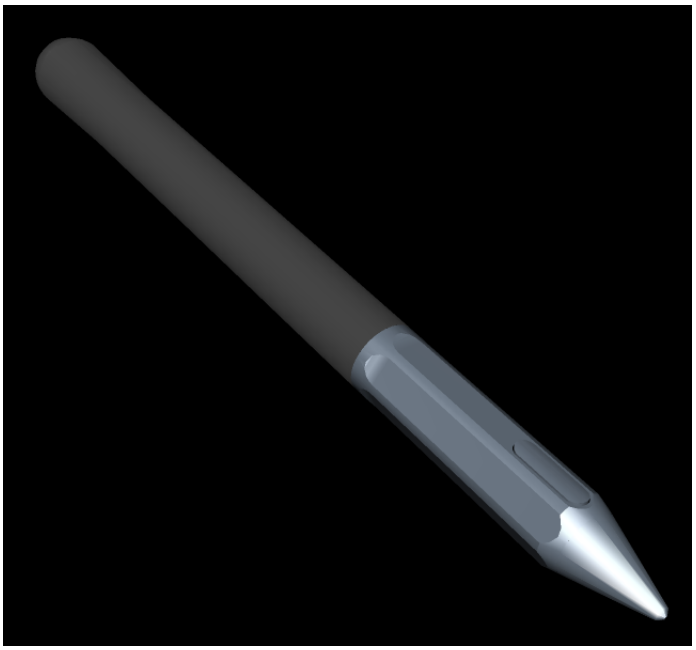

Figure 2.9: A user's pen in *Construct3D*

structure gives access to rich geometric functionality, focussing on usability, especially as this is difficult to achieve in 3D. This interface is grouped into some major areas (for details see figure 2.10):

- A vertical all purpose menu bar composed by *File*, *View* and *Edit*, providing access to some general program functions including file management and undo/redo history,
- a horizontal menu bar consisting of *Transform*, *Intersect*, *2D* and *3D* to perform all geometric operations,
- a single toggle button (*Point*) controlling, whether to draw points or enter selection mode on occurence of button events in virtual space,
- a *Layers* widget group, making layer functionality accessible,
- and finally, the *Help Notes* text area to assist the user with context sensitive help texts (analogous to *tooltips* used in *2D user interfaces*).

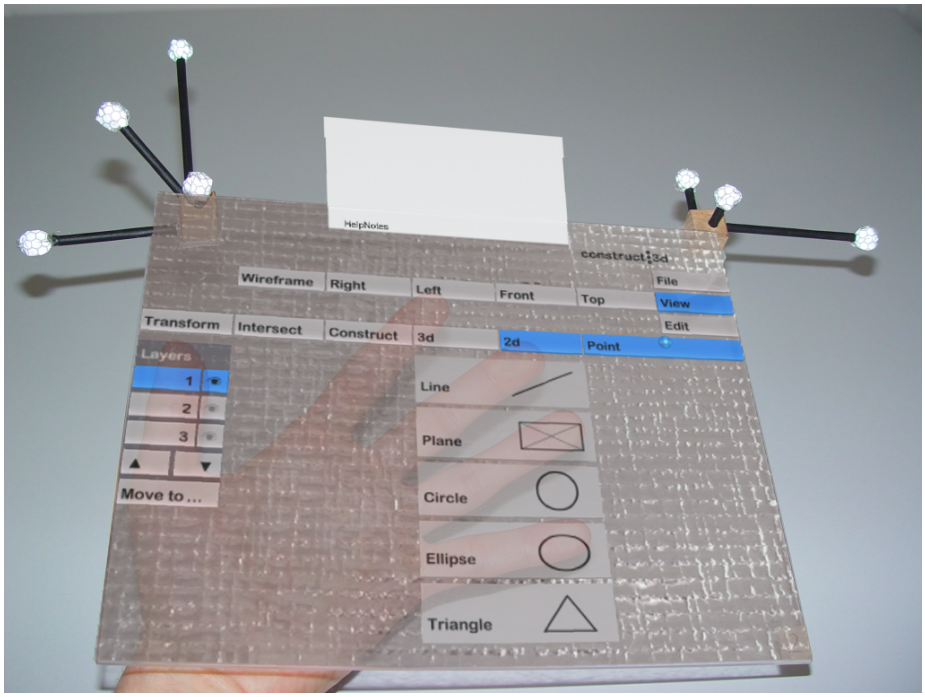

Figure 2.10: A user's panel, augmented with widgets, as used in *Construct3D*

Chapter 3

Design

Long distance distribution requires *Virtual* and *Augmented Reality* frameworks with appropriate capabilities. *Studierstube* bases on *Distributed Open Inventor* and *OpenTracker*. Each of these three frameworks is enhanced and adapted in context to long distance distribution. In application of these new features, *Construct3D* is also modified. All design considerations shall be presented in the following.

3.1 Tracking data distribution by OpenTracker

OpenTracker contains components providing *tracking* data transmission over network between several *OpenTracker* instances on different hosts. Just like *Distributed Open Inventor*, these capabilities are built upon *multicast UDP* so far.

3.1.1 Requirements

Following the *data flow* principle of *OpenTracker*, *tracking* data is inserted from the network into the *data flow graph* by means of a special *source* called *NetworkSource*, while a special *sink* named *NetworkSink* transmits data to the network. This implies that network traffic concerning *tracking* data is *unidirectional* and of *multicast* nature: Payload data is always transmitted by a single *NetworkSink* and received simultaneously by one or more *sources*, each of them a *NetworkSource*. Figure 3.1 illustrates, how *tracking* data is exchanged over network between different *OpenTracker* instances.
It is an important thing not to confuse *NetworkSource* and *NetworkSink*, as their names seem to be contrary to their behavior in the network. But on the other hand, these names reflect their role in *OpenTracker* and its *data flow graph*. However, these terms will be superseeded by more appropriate ones as network roles are determined in the following.

3.1.2 Solution

Considering the requirements, an alternative *unicast UDP* implementation to *multicast UDP* can be easily achieved: A *NetworkSink* generating *tracking* data packets has to deliver these data records simultaneously to associated receivers.

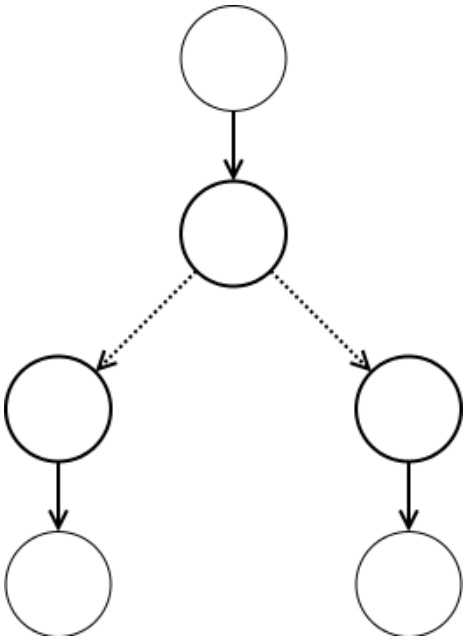

Figure 3.1: *Tracking* data distribution over network between several *OpenTracker* instances

To know where to send these *datagrams*, the *NetworkSink* has to maintain a list of counterparts. Each of these is usually a *NetworkSource* of an *OpenTracker* instance on the receiver side.
Although running *unicast UDP*, this data delivery strategy is clearly of *multicast* nature with *one-to-many* property: A central site has to transmit *datagrams* multiple times on a per receiver basis.

3.1.3 Multicast (one-to-many) data delivery on unicast UDP

As a consequence of *data flow* properties on *unicast UDP*, the *network topology* on the logical level is a *star*. Figure 3.2 illustrates delivery of data occuring in the central instance.
Keeping this *topology* in mind, *tracking* data of several devices can be distributed by a single network. As long as *tracking* data occurs on a single central location, multiplexing of device data is possible: *Tracking* data packets are equipped with so-called *station* information to distinguish between data of different *tracking* devices.
Of course building several independent networks (consuming more network ressources) is the alternative and more general way, as this allows distribution of *tracking* data occuring at different places.

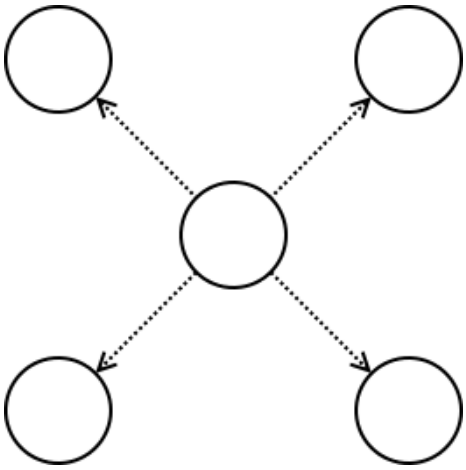

Figure 3.2: *Star topology* and *tracking* data flow paths on network

3.1.4 Managing tracking data sender and receiver associations

Establishing the *tracking* data network is initiated by each *NetworkSink*, similar to the traditional *server-client scenario*: Each *client* must have knowledge in advance about the *server* providing desired *tracking* data in terms of *socket* information (*host* and *port*).
To inform a *NetworkSink* about the willingness of receiving *tracking* data, a *NetworkSource* initially just transmits a *datagram* containing control information. It is an important aspect that this *datagram* travels in the opposite direction than packets containing *tracking* data, and therefore it cannot be mixed up with the payload. Reading this *datagram* on the *server* side also includes retrieving information about the origin, allowing the *NetworkSink* to provide *tracking* data.
These small packets, received by the *server* from each associated *client* and covering management information, come in two flavors:

- *Poll* commands indicate that the *server* should begin transmitting *tracking* data. If the *client* is already part of the list of receivers, this message is ignored. Considering the *unreliable* nature of *UDP*, random start order of *servers* and *clients* (and even the possibility to participate in several networks with contrary *server-client* roles in context with random start order), *poll* packets have to be sent regularly. As receiving data is implicitely an acknowledgment of a request for *tracking* information, *polling* can be suspended, while *tracking* data is constantly obtained from the network. But as soon as data retrieval is intercepted for a short time, *polling* has to be resumed, because there is no way to rule out that the *server* was not terminated (and restarted).

- A *leave* command is transmitted in the case of *client* termination to allow the *server* to delete the *client* from the list of *tracking* data receivers, preventing unsuccessful data delivery attempts to terminated *clients*.

With these semantics, no explicit acknowledgment has to take place. *Clients* indicate what they expect from the *server* without explicit confirmation and being not aware about the status end even the presence of the *server*. And this knowledge is simply not needed, as only *tracking* data is of interest.
Lost *poll* and *tracking* data packets due to *unreliable* networks are likely to cause hardly any harm, implying that they are regularly retransmitted. *Leave* packets cannot be retransmitted as the *client* is pushing for cleaning up and releasing ressources when terminating. But even if this packet disappears, it causes no serious trouble in most cases, although delivering *data* to terminated *clients* due to unreceived (and therefore unprocessed) *leave* request may not be the most efficient way.
As *polling* and *tracking* data reception is strictly performed alternately, *network* performance is not influenced negatively.

3.2 Distributed Open Inventor

Distributed Open Inventor requires the presence of an underlying network to operate on. As mentioned in section 1.2, *DIV* network traffic cannot go beyond *routers* (unless part of a *MBONE* [Eri94] network) in the current implementation [Hes01] due to the *multicasting* feature of *UDP* (see section 2.2 for reference). This restricts distribution to a large extent.

3.2.1 Requirements

The requirements of the underlying network include *reliability*: The underlying network layer has to guarantee that messages are delivered exactly as they are sent i.e. in the correct number and order without data loss. Also transmitted data has to be delivered to each participating site simultaneously and there is no need to identify source and destination. As there are no predefined sender and receiver roles, logical network links have to be present between each pair of network participant and *bidirectional* data transfer on these links has to be assumed.
In the past, a *multicast UDP* protocol, explicitely enhanced with additional *reliability* capabilities, was used.

3.2.2 Solution

To overcome the previously invincible borders of private local networks, *TCP* as a very widespread and *reliable network protocol* was chosen, but this implies a lot of changes. The difference between the more lightweight *UDP* and *TCP*, as shown in table 3.1, are essential:
In contrast to *multicast UDP* but similar to ordinary (*unicast*) *UDP*, *TCP* allows only *point-to-point* communication. But unlike in the *multicast UDP* implementation no *reliability* treatment has to be done manually in *TCP*, as this is implicitely taken care of in the protocol. In order to allow data delivery to all *network nodes* in *TCP* implementation, considering its *unicast* nature, it has to emulate *multicast* data delivery to comply to the requirements of *Distributed Open Inventor*. As any *network node* might act as *server*, a *many-to-many* property has to be taken into account.

UDP	*TCP*
connection-less	*connection oriented*
datagram abstraction	*stream* of data abstraction
unreliable	*reliable*
lightweight	bigger synchronization overhead

Table 3.1: Comparison between *UDP* and *TCP*

3.2.3 Multicast (many-to-many) data delivery on TCP

Multicast data delivery on *TCP* with multiple senders is simply done by building up a logical network of so-called *true mesh topology* (figure 3.3), a network, where each *peer* is logically connected to each other *peers*. As each other *peer* can be directly reached from each *peer*, payload data forwarding from any connection to any other has never to be done. The advantage of this is increased robustness, as data transmission is not dependent on any other *peer* but the directly involved sender and receiver. It also makes *peer* functionality quite easy. Ensuring *true mesh topology* at any time is the main challenge, while sending and receiving is, as mentioned before, fairly simple: Data is automatically transmitted to each connection simultaneously, on the other (receiver's) side nothing special has to be considered apart from the requirement not to mix up data of distinct connections. Processing order of data received from different connections is uncritical as the next higher network layer implies that critical data in terms of processing order is sent from exactly one *peer* at any time.

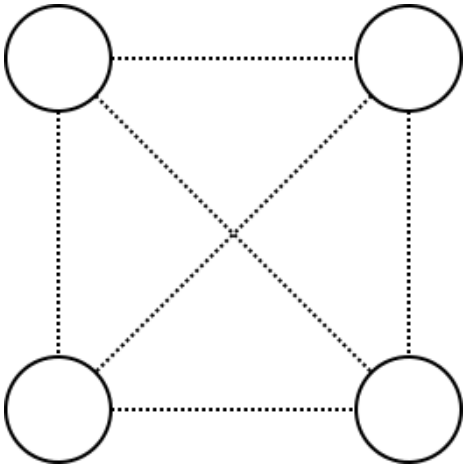

Figure 3.3: *True mesh topology* on network

3.2.4 TCP peer identification

For some reasons explained later, each *peer* has to be uniquely identified. The information of the *server socket* is an appropriate identification, as long as this information is globally valid along the network and identifies everywhere the same *peer*.

3.2.5 Arrival of new peers

Whenever a *peer* joins the distribution network, it has to know at least one *peer* of the existing network. Otherwise, it will become the single participant of a new network.
A new *peer* initially contacts the network by sending a special message, identifying itself just after connection establishment. This identification contains the *server port* of the *peer* (as each *peer* contains *server* as well as *client* functionality). The arrival of a new *peer* has also to be forwarded to all other participating sites of the network.

3.2.6 Peer arrival notification

Keeping in mind that in true mesh topology each *peer* is directly connected to each other and therefore there is the distance of exactly one *hop* between each pair of *peers* (as shown in figure 3.4), the arrival of a new *peer* can be communicated in a single step to all other *peers* currently present in the network.

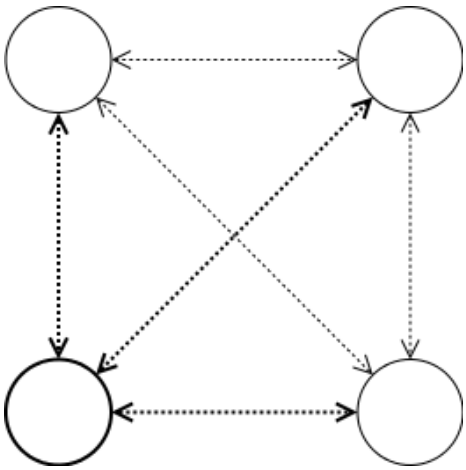

Figure 3.4: Each *peer* is one *hop* away from each other

But thinking of two (or more) *peers* independently and concurrently joining the network by contacting two distinct *peers*, which are currently part of the network, the problem arises that these two new *peers* probably will not get known of each other and therefore a *TCP* connection between them will not be established. This problem is illustrated in figure 3.5.
The solution to this is to embed lifetime information in the message, indicating a joining *peer*. This lifetime information is interpreted as the maximum number of *hops*, the message will travel and complies to the requirements of the *Time to live* concept (*TTL*). In the previous example of two *peers* concurrently joining the network a *TTL* value of 2 suffices. Bigger values can also cope with the rarely case of cascades of simultaneously joining *peers*. But a drawback of higher *TTL* is increased network load. So 2 is quite the optimum and a good trade-off between universality and efficiency.

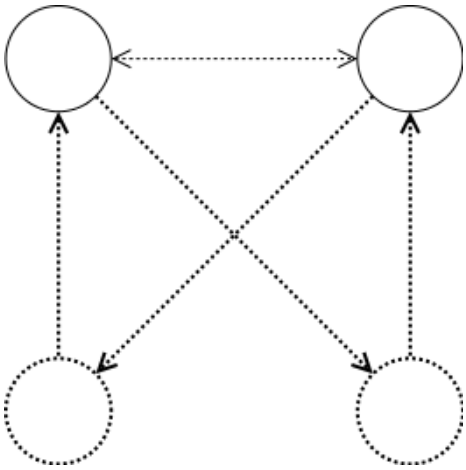

Figure 3.5: Two *peers* concurrently joining the network

3.2.7 Ensuring true mesh topology

When receiving the forwarded message of a new participating *peer*, some things have to happen as reaction:

1. First of all, the *TTL* value has to be examined.
   If it is positive, the message with decremented *TTL* value is forwarded to each connection but the one (for the purpose of optimization), where the message was received.

2. After that and without taking care of the *TTL* value, the *peer* has to check, if a connection to that *peer* currently exists. This implies that the *peer* has to keep track of its *TCP* connections.
   If no such connection exists, the *peer* has to establish a *TCP* connection to its counterpart by using the received *peer* information, which is, as mentioned before, at the same time the *peer* identification. Instantly after connection establishment, the *peer* identifies itself to the counterpart.

This strategy covers some potential problems:

- Not forwarding messages with exceeded lifetime information ensures that the network is not cluttered with such messages.

- Establishing connections dependent on current available connections guarantees that no undesirable *redundancy* is brought into the network.
  Network redundancy (as illustrated in figure 3.6) can cause network congestion, because traffic is unnecessarily higher. Especially *loops* over a single *peer* (*cycles*) have bad impact and cannot be detected in advance as easy as *redundancy* over more than one *peer*. This is done as reaction to identification messages.

Identification messages come in two flavors:

- A new *peer* arrival message is sent, when a *peer* initially contacts the network.

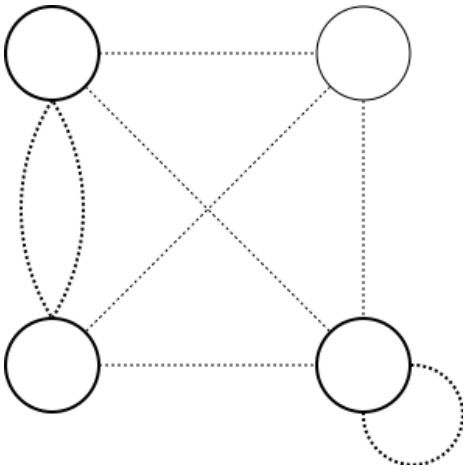

Figure 3.6: Examples of undesired *network redundancy*

- A simple identification message is sent, when a *peer* initially contacts another *peer*, for the purpose of identifying itself on the counterpart.

So, the simple identification message is the more common version of the new *peer* arrival message, which can be also seen in the reaction:

1. On receiving any identification message, the *peer* has to check, if another connection to the counterpart currently exists. If this is the case, the connection is closed immediately. As *network redundancy* involving more than a single *peer* is effectively prevented in advance, connection closing should only take place for *cycles*, even if it works for all types of *redundancy*.

2. If the identification message is of the special type indicating an initial connection to the network, the network is notified of the new *peer* by generating *TTL* equipped messages identifying the *peer* as described before.

Finally as all three control message types have been specified, a fourth message containing payload data is used for the actual data transfer. Considering strict *network redundancy* prevention (mentioned above), data messages are only valid for previously identified *TCP* connections (i.e. ignored on unidentified connections to prevent harm).
To summarize all that, figure 3.7 illustrates, how the information is distributed and *TCP* connections are established, when a new *peer* is joining the network:

1. The *peer* contacts another *peer* in the network, identifying itself.

2. The contacted *peer* notifies other *peers* participating in the network about the arrival of the new *peer*.

3. All other *peers* establish connections to the new *peer*, identifying themselves. (Additionally in case of a *TTL* value greater than 1, new *peer* arrival information is distributed at least once more, but causes no effect in this example.)

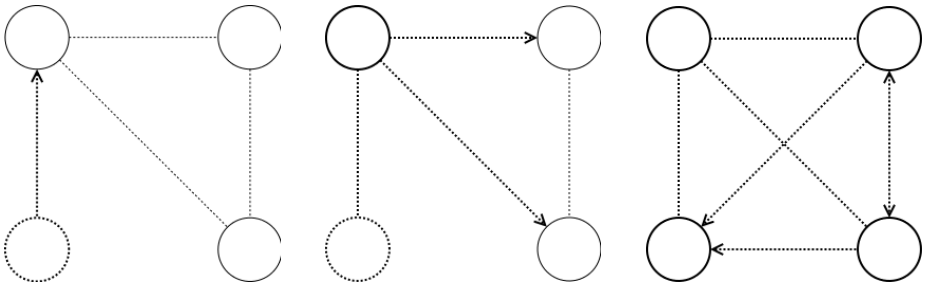

Figure 3.7: Sequence of a *peer* joining the network

3.2.8 Hybrid networks

As an extension and somehow located just slightly above both underlying network types, *hybrid networks* are also possible to profit from the advantages of those underlying network types.
TCP networks are not restricted to certain network structures like *multicast UDP*, but some inherent drawbacks are revealed when considering a huge number of *peers*. If n *peers* are part of the network, the number of connections according to *true mesh topology* is:

$$\sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2}$$

So the relation between the number of connections and *peers* is quadratic, even if only a small subset of these connections is used concurrently at each instant. This can be seen as wasting ressources. Considering a *peer* sending data to the network, one would observe that data is physically duplicated $n - 1$ times to have access to each other *peer*. An operating system running *multicast UDP* on a single physical network with *broadcast* characteristics (*shared media* networks like *busses*) is able to prevent that, because in such an *physical network topology* all network traffic reaches each of the *hosts* connected to the *shared media* anyway (regardless of the destination). On this physical level, the *network interfaces* performs filtering so that only data addressed to the corresponding *host* is forwarded to the operating system. As *TCP* allows only *point-to-point connections*, data duplication is already done in the application layer. This is essential for further considerations in this implementation. But relying solely on *multicast UDP* is not always possible because of its implied restrictions to supported networks. One way to deal with that is to group several participants together, which are able to run *multicast UDP* and connect these subnetworks by means of *TCP*. Figure 3.8 illustrates how *TCP* and *multicast UDP* networks are coupled.
It is essential that *hosts* running both network protocols perform some kind of *bridging* (not in strict conformance to networking terminology) between them to forward network data from one to the other. This implies that *hybrid networks* must not contain any *redundancy* to prevent data travelling endlessly. In contrast to *TCP* implementation, currently there is no robust algorithm implemented to cope with *redundancy*: *Bridging* functionality would have to be disabled on proper *hosts* so that the graph corresponding to the network is of

spanning tree nature to meet the requirements.

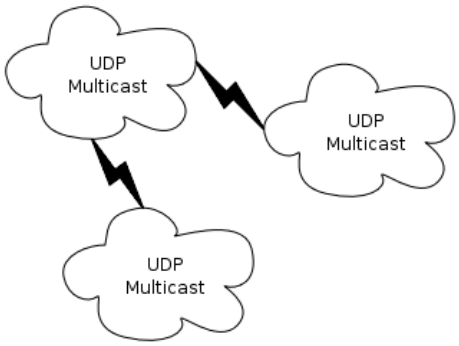

Figure 3.8: Example of a proper *hybrid network*, interconnecting several *multicast UDP* networks with *TCP*

3.2.9 Distributed Open Inventor functionality

On top of the established network *Distributed Open Inventor* commands are exchanged. On this level, the network is already abstracted so that no participating site needs to know anything about other participants: Messages are just sent to the network without explicit destination information and therefore received by each of the other participants. Consequently, received messages do not contain any information about their source. So the implementation of the underlying network requires that these messages are distributed to each participating site. *Reliability* and maintaining message order are additional requirements and mentioned before. As described earlier, the *TCP* implementation presented in this work complies to these requirements as well as the *reliable multicast UDP* implementation.
Usually, *DIV* features are enabled by utilizing *DivGroup*. Although *DIV* can be directly used, *DivGroup* integrates the distribution concept as special *group* into the *scene graph*. As shown in figure 3.9, the *scene graph* structure implies, what is replicated, allowing to have several independently configurable *scene graph* parts to distribute.

3.3 Studierstube

Among other extensions the enhancements on *Distributed Open Inventor* have to be reflected in *Studierstube*. Furthermore, *Studierstube* applications are distributed automatically by a *DivGroup*, implicitly created as *parent* of each application [SRH03]. The requirements of this functionality include configuration of distribution. This is handled by the core library with the help of a tool program called *session manager*.
With implicit distribution capabilities, network transparency is achieved. Without the effort of manually configuring settings, *Studierstube* applications are distributed in a standard way. These features are completely transparent to the application developer.

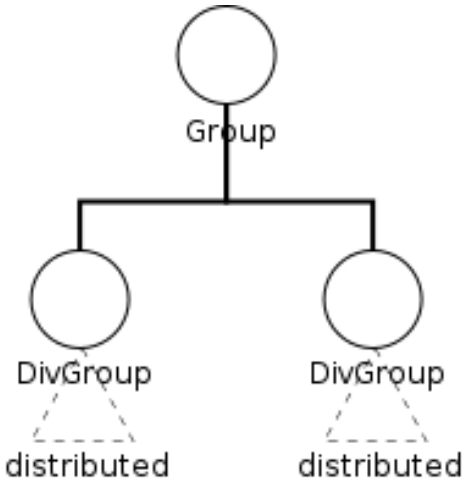

Figure 3.9: Example of a *scene graph* containing one *DivGroup* per distributed *scene graph subtree*

3.3.1 Distribution management

This *session manager* is the central point and manages network ressources. To achieve central management, each *Studierstube* application contacts the *session manager* and informs it about the so-called *locale* it wants to join. This *locale* represents the shared workspace of all participants. It is up to the *session manager* to reserve network ressources and notify all participants about distribution configuration. Apart of that, messages concerning users and applications are exchanged.
The participant, hosting a certain application, selects the underlying network type used for distribution. According to this preselection, the *session manager* takes care of requesting network ressources and transmits proper configuration data to all participating sites. From this point on, distribution is activated and performed independently, as the *session manager* itself is not included in the distribution mechanism. It still operates in the background, monitoring changes of participants and reconfigures them properly on demand. In other words, not only late joining, but flexible joining and leaving of participating sites is fully supported without having the need to know anything about potential participants in advance.
Distribution management message exchange is implemented in *TCP* and completely independent of any *DIV* functionality. Each participating site establishes a permanent *TCP* connection to the *session manager* sitting in the center in a network of *star topology* and *unicastly* exchanges distribution management messages.

3.3.2 Distribution configuration

The *session manager* assigns *master* property to the participant originally hosting a certain application. All other participating sites (*slaves*) receive the application *scene graph* by network due to the *node transfer* feature. Terminating

the *master* results in reassigning *master* property to another participant. This *master-slave* property assignment is conducted autonomously and cannot be influenced by *Studierstube* instances.
Another task of the *session manager* is to create network ressources according to the requested networking mode. In order to do this, a generator produces network configuration data:

- In *multicast UDP* mode, a single *multicast group* address and associated *port* number is generated on a per application basis.
- In *TCP* mode, each *peer* is given a unique *port* number to allow running several applications on a single machine.

As mentioned earlier, the *session manager* has to reconfigure the whole distribution network, if a participating site joins or leaves the workspace. Apart from the *master-slave* property, *TCP* mode is crucial in terms of how new participants join the network. But as the assignment of network ressources lies in the responsibility of the *session manager*, it simply creates a list containing proper contact information of all other participants for each *peer* and includes this in reconfiguration messages sent to each participating site. This strategy guarantees highest chances to contact any of the other *peers* succesfully to build up a network.

3.3.3 Distribution exclusion

Implicit distribution of an application as a whole can be problematic and undesirable. As long as there are no mechanisms to exclude certain parts of *scene graphs*, there is no chance to hide them from other participants. As the *DivGroup* is created implicitely, containing the whole application as *child*, undistributed *nodes* related to the application seem to be simply not possible. But keeping *scene graph* parts private is often desired, whether to realize local variations in appearance or to prevent network congestion by eliminating transmission of redundant information.
Previous attempts to have undistributed *subtrees* of application *scene graphs* within the reach of the *DivGroup* are based on filtering: Distribution functionality was prevented in the core of *DIV* by managing an exclusion list of *nodes* and *fields*.
A more elegant way is to circumvent *DIV* core of getting notified of private *subtrees*. This is achieved by breaking the propagation mechanism of update *notification*: To efficiently hide updates from being recognized by the *sensor*, used in *DIV* to detect changes, a *group* disables forwarding *notifications* upwards the *scene graph*. Considering the *node transfer* feature, the *group* has also to prevent writing its *children* to stream, as this feature makes use of the *write action*. For consistency reasons, also reading is prevented.
This special *group*, hiding its *children* from reading, writing and distribution, is called *HiddenChildGroup*. It is somehow the opposite of a *DivGroup*. Figure 3.10 illustrates, how *scene graph* parts in the reach of a *DivGroup* are prevented from being distributed by utilizing this *node* type.

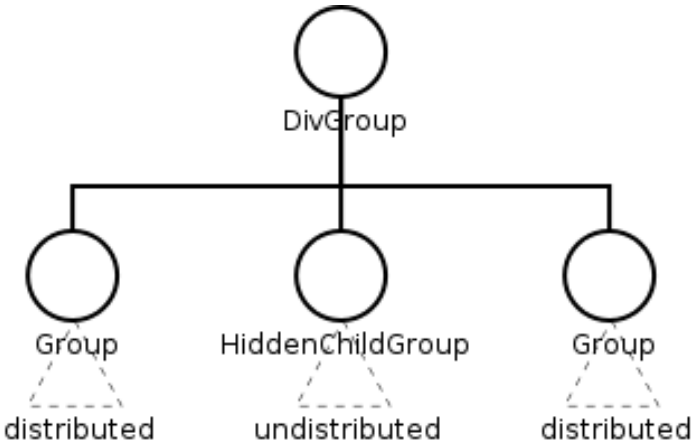

Figure 3.10: Example of a *HiddenChildGroup* as *child* of a *DivGroup* preventing distribution of a *scene graph subtree*

3.3.4 Implicit distribution domain

Implicit *DIV* distribution in *Studierstube* is bounded to the application *scene graph*. Being more specific, the *DivGroup* is the direct *parent* of the *ApplicationKit* containing the *ContextKit*. This *ContextKit* is strongly related to application specific data, covering also application defined geometry for a *PIP sheet*.
Other ressources, notably user ressources including general *PIP* geometry, are not part of this distributed *scene graph*. A *UserKit* instance contains information about a user, identified by its *user id*, and its associated devices (display, panel and pen).
A *LocaleKit* represents the *locale* and joins applications with user information: Each *UserKit* is directly part of the *LocaleKit* and each *DivGroup* used for distribution is also aggregated.
To accomplish full *Studierstube* application distribution, user information has also to be replicated. So, the contents of each *UserKit* is distributed by other means. Distribution management messages address this issue by including *UserKit* information to supply necessary data not distributed by *DIV*.

3.3.5 Application vs. tracking data distribution

As mentioned in section 1.1, three different types of distribution can be distinguished. They are classified by the level, where synchronization occurs:

- input data distribution,
- intermediate data distribution,
- output data distribution.

When distributing *Studierstube* applications, a proper distribution strategy has to be defined. Two extreme cases are possible:

- Application content or state is distributed solely by means of *DIV*, *tracking* data is not distributed at all on *slaves*.
  In variation to this, *tracking* device data can be used to position associated objects and user interface ressources (i.e. panel and pen). In order to prevent application state modifications, caused by *tracking* distribution, this data has to be ignored in (excluded from) further processing.
  Conforming to terminology used in section 1.1, this distribution policy can be interpreted as synchronizing application's intermediate or output data.

- Applications are not distributed at all. Instead, *Studierstube* instances operate in stand-alone mode relying on distributed *tracking* data and its consistency to achieve also consistent application state.
  This strategy represents synchronizing applications indirectly by means of input data, obtained by *OpenTracker*.

Table 3.2 gives an outline of differences of these two distinct distribution technologies.

OpenTracker distribution	*Distributed Open Inventor*
synchronizing input data	synchronizing intermediate/output data
tracking device data context	application state (*scene graph*) context
focus on fastness	focus on consistency
unreliable	*reliable*
self-contained data units	mutual data units dependencies
state-less	state-full

Table 3.2: Comparison between *OpenTracker* distribution and *Distributed Open Inventor*

Mixtures between these two extreme cases are possible, although this introduces some issues, which have to be addressed. These problems depend on how and to what extent these two distinct distribution strategies are mixed: If distributed *tracking* data as well as *DIV* can cause modifications in application state, potential conflicts will arise. Separating responsibility or defining rules of precedence are possible strategies to resolve these conflicts for individual solutions.
Somehow related to this issue is the problem, how *slave* associated users are able to perform operations in applications. Due to the fact that actively distributing application state on *slaves* is not directly possible, *tracking* data distribution has to be included in any case to let the *master* actually carry out the operation. This creates the illusion of being independent of *master-slave* property.
In section 3.4 these problems are addressed in detail and an example solution is presented.

3.4 Construct3D

As *Construct3D* is an *Collaborative Augmented Reality* application, it supports multiple users collaboratively working together. Giving a short overview, desired distribution capabilities are implemented on several levels of synchronization, as previously defined in section 1.1:

- Intermediate and output data synchronization:
  Distributed Open Inventor is responsible for synchronizing and replicating the database containing application operation history. This consists of all geometric operations (including final destination in object movements) and shared application state (synchronizing widgets on the *PIP*). Some remaining parts of the *scene graph* are also directly distributed by *DIV*.

- Input data synchronization:
  OpenTracker is used to position all user interface ressources (each *PIP*) accordingly to distributed *tracking* data. Relying on this distribution method, widget states on the panel are indirectly synchronized, but widgets representing application state are excluded from this. Also during movements of objects, spatial positions are temporarily updated by distributed *tracking* data.

In the following, these multi-user capabilities are described in more detail. Distribution features and conflicts that may occur between these different synchronisation strategies are also comprehensively depicted.

3.4.1 Multi-User concept

Construct3D is ready for supporting multiple users on different hardware and display setups on multiple sites. The application supports users dynamically joining and leaving the shared workspace.
Actively collaborating users are usually equipped with *tracked* devices, panel and pen for interaction as well as head mounted displays (as displayed in figure 3.11) for individual viewpoint adaption. Making use of the power of *Studierstube* to support various devices, alternative hardware setups are also possible.

Figure 3.11: A head mounted display and pen, as typically used in *Construct3D* setup

Of course, passively watching users (without associating a *PIP* to their

ressources) are also supported. This comes in handy to present to an audience, what is actually going on, or record movies for observations. Display hardware with large extents are recommended: A projection wall or a virtual table, as shown in figure 3.12, may be suitable for presentation purposes. Another intended use case of passive users might be having *Construct3D* running as some simple kind of service for active users to dynamically join the workspace by utilizing distribution features.

Figure 3.12: A *Barco* projection table

For easy visual user distinction, a color theme is assigned to each user. This color coding can be seen on their panels and all geometric objects, indicating the user being responsible for object creation. Figure 3.13 shows the virtual representation of two users collaborating when performing geometric operations in *Construct3D*: The results of geometric constructions can be seen beneath the coordinate space axes. The user's color theme of each panel is also related to geometric object colors.

3.4.2 Application scene graph details

As *Construct3D* is just another *Studierstube* application, it inherits automatically its distribution features. So, *Construct3D* is implicitely ready for distribution, sharing its *scene graph*.
The internal structure of the application's *node kit* (representing the *context*) is rather simple. Avoiding the *dual database problem* [MF98], it encapsulates all of the application's data. Basically the *scene graph* hierarchy is composed by command lists storing all *Construct3D* operations and its associated counterpart,

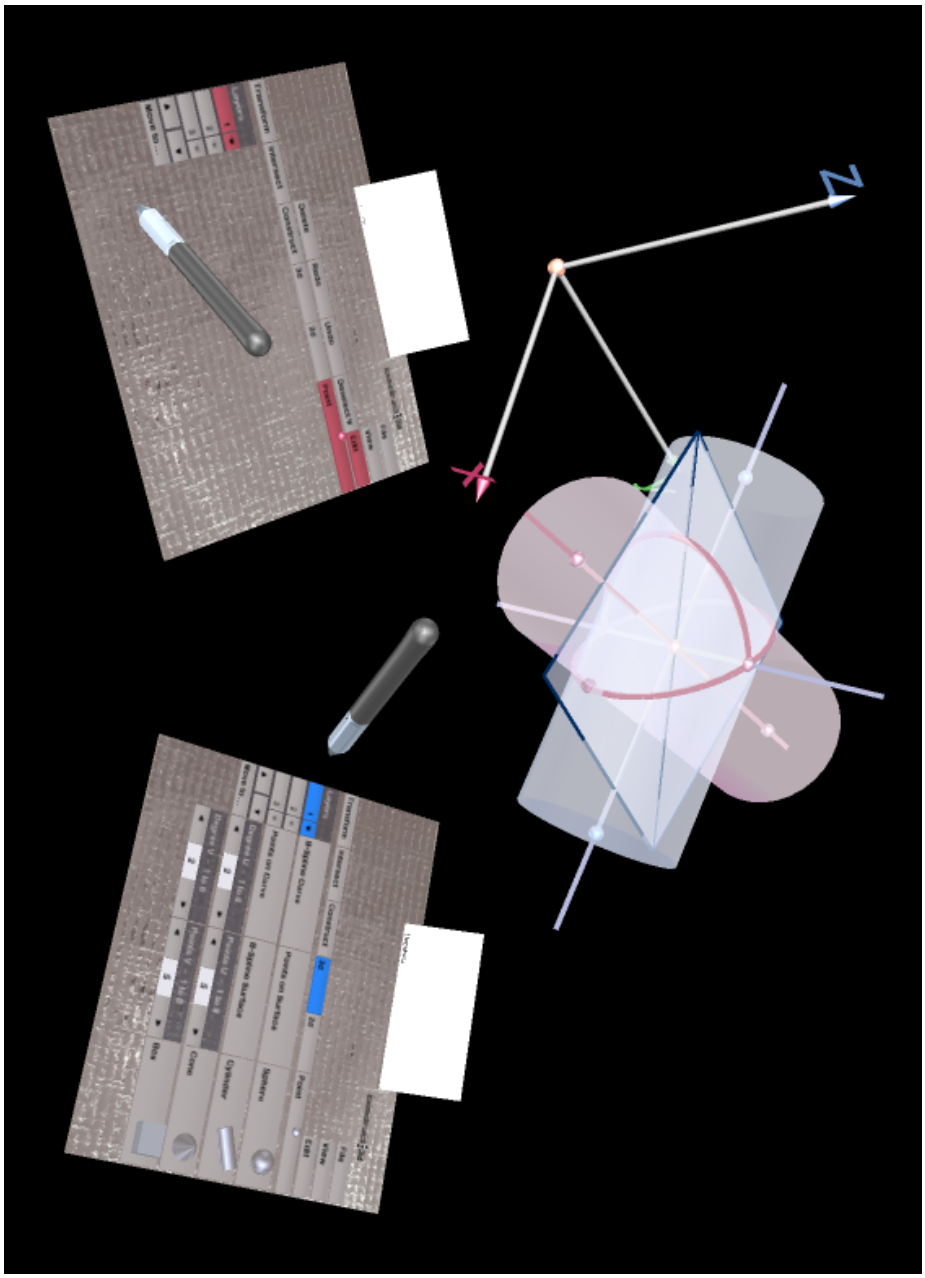

Figure 3.13: Two users collaborating in *Construct3D*, shown on desktop setup

the visible geometry. These command lists and its representation (geometric elements) are implemented by special *node kits*.
These two distinct parts, forming the *scene graph*, are strongly related to each other: Manipulation of geometry causes the creation of new commands in these command history list. On the other hand, the execution of commands in this metadata produces deterministic results on visible geometry. Because of its deterministic and regenerating property, the command list is used for file operations and undo/redo functionality. The latter use case is also the reason, why this command list is called *UndoRedoListKit*.

3.4.3 Increasing distribution efficiency

Especially in long distance distribution, network performance and data throughput are crucial aspects. The idea is to reduce distribution effort and network traffic by keeping the geometry private, as this is only redundant information, which can be regenerated by performing command executions of shared *UndoRedoListKit* data.
By having a *HiddenChildNode* as *parent* of all geometry it is effectively excluded from distribution. Figure 3.14 illustrates the *scene graph* inside the application's *node kit* called *CnDKit*. Geometry resides in a *subtree* below a *HiddenChildGroup*, while the *UndoRedoListKit* is distributed just as the remaining *children* of the *root separator* concerning various projection related *scene graph* parts.

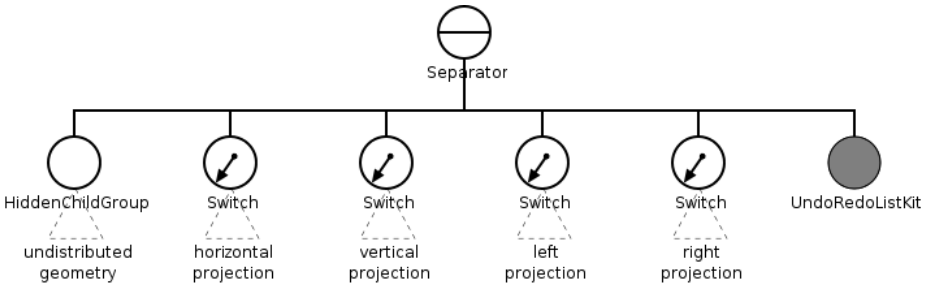

Figure 3.14: Structure of the *scene graph* inside *CnDKit*

But as command lists are not self-executable in the sense that they automatically regenerate geometrics after modification, additional processing has to be done:
On each *slave*, a *sensor* is attached to the *UndoRedoListKit*, observing all changes. Whenever a modification occurs, it has to be determined, what actions have to be taken to keep the geometry synchronized and up-to-date.
Also, each *PIP sheet* is excluded from distribution due to various (partly because of historic, partly because of efficiency) reasons. This requires to keep them in sync by other mechanisms.
Figure 3.15 visualizes that *PIP sheet* geometry is also prevented from distribution by means of a *HiddenChildGroup*. The *parent node*, being part of the *PipSheetKit*, is implemented as a *switch* to allow *context* switching in terms of changing between different *PIP sheets* of multiple running *Studierstube* applications.

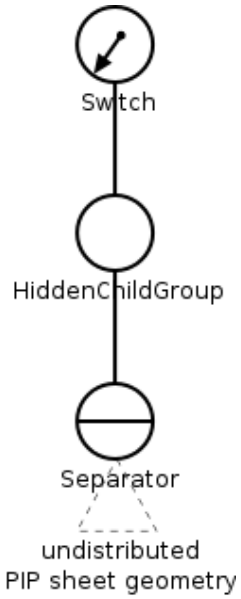

Figure 3.15: Structure of the *scene graph* inside *PipSheetKit* used in *CnDKit*

3.4.4 Geometry update using the undo/redo list

In order to reduce the amount of *scene graph* data to distribute, the *UndoRedoListKit* plays a major role. Geometry update distribution is based on the *UndoRedoListKit* content.

Undo/redo list

UndoRedoListKit is a *node kit* storing a list of all executed application commands in descriptive form. A list entry, a special *node kit*, consists of a command string and associated arguments.
Additionally, the *UndoRedoListKit* maintains a *field*, pointing to the current position in undo/redo list records. This position pointer is used to step through the undo/redo history: When performing such (undo/redo) operations, the value of the position *field* is increased/decreased properly.
When triggering a new application operation, a new command list entry is appended at the current position. In order to clean up the list from outdated records, it is truncated after the pointer position.

Geometry update

As each action causes modification of the *UndoRedoListKit* involving its index pointer, this *field* is the key element in assisting to detect changes caused by distribution.
Unfortunately, updates in *UndoRedoListKit* caused by *DIV* distribution are atomic on a *field* level and not on the *node* level. This means that updates to several *fields* are distributed sequentially. This is a problem, since there are multiple *fields* involved to compose a proper and valid undo/redo state.

Solving this problem requires to ensure stable conditions in *UndoRedoListKit*. A possible solution is the ability to detect and wait for stability and validity. All this is done by keeping a private copy of the undo/redo position pointer, indicating the state of the last successful update. Whenever the shared pointer changes, actions have to be taken. Ensuring stable conditions is done by trial and error: In the absence of stability, update actions will fail, but are causing no damage, as no change actually happens. On success, stability has been achieved so far, as it is needed to perform necessary updates. In this case, the private copy is updated afterwards to reflect the state update.
Since the undo/redo list pointer is an integer number, updates can be broken down into several operations, each of them representing a single atomic entry in the undo/redo list. Updates are performed sequentially to adjust the private pointer copy towards the distributed pointer. Each of the updates can fail due to unstability. So these sequential updates are performed until equality between both pointers is achieved or the first failure is detected. In case of failure, further updates have to be patiently postponed. So, pending updates are delayed until the conditions are stable again. This is likely the case the next time the *sensor* detects changes. So, the algorithm will have another try in performing updates. This strategy can be seen as a somehow lazy, but robust behavior.

3.4.5 Collaboration aspect in context to distribution

Each actively participating user is given his private *PIP* in his own color theme. Seeing the other user's panels (mainly for demonstration purposes) implies that the panel of each user is visible to each other participant and therefore has to be in a system-wide consistent state. Especially educational teacher-student scenarios benefit from this fact.
Application behavior is apparently independent of the *master-slave* property of *DIV*, even though application operations are only possible on the single *master*. This is achieved by performing *tracking* data distribution in *OpenTracker*: Input device data of a user related to a *DIV slave* is transmitted to the *DIV master*, where operations are actually performed. *DIV* distribution ensures that the effects of the operations are also observable by the user, who caused them. The system of transfering *tracking* data to the *master*, triggering proper reactions at this place, whose caused effects are again distributed among all *slaves*, creates the illusion of being completely independent of the *master-slave* property.

3.4.6 Personal Interaction Panel synchronization

Some application states are shared among all users: Active layer selection and layer visibility settings are globally defined. Information about these settings is stored in *fields* in the application's *node kit*. But this data is not distributed directly, as these *fields* are not set up to be part of the *field catalogue* of the *node kit*. This information of temporal nature is shared by integrating it into the undo/redo history, providing special commands. Widget interaction alters dynamically the corresponding *field* and generates the associated command. By creating *field connections* to corresponding panel widgets, visual feedback is guaranteed to be consistent to the global shared state.
Considering user defined states, similar strategies are applied: *Multi-value fields* contain entries for each user's settings, making proper *field connections* to each

panel possible. This ensures, that all replicated panels of a certain user are in sync.

3.4.7 Indirect synchronization mechanisms

Some remaining visual states of the panel do not belong to application states, but should also be consistently displayed throughout the whole system: Menu properties (visibility of submenus) and scrolling states are two examples of such interaction, but non-application states. Keeping these in sync is implicitely performed as pleasent side effect of *tracking* data distribution. Even if *tracking* data networking is not reliable, most of the time each of the participating sites should receive data from all involved *tracking* devices. Although each *PIP* of a *slave* does not directly and locally trigger actions, visual feedback is generated. So if a user presses a button on his panel, it is likely to be the case that data is succesfully distributed to all other participants and the visual representations of this user's panel in the other participant's views are updated according to this.
This mechanism is also used in minor temporary geometric updates: When moving objects, the position of its geometry is constantly updated to give visual feedback. Nevertheless, a corresponding move command is not added to the application history until the object is placed to its final position. Without *tracking* data distribution, all other participating sites would not see the progress of the move operation. Fortunately, distributed *tracking* data allows to perform this preview on each other user's view. Without or on (transient or permanent) failure of *tracking* data distribution, they would have to rely on the reliable fallback mechanism of distributed commands execution.

3.4.8 Conflicts between various synchronization strategies

Paying attention to the synchronization problem described earlier in section 3.3, it is very important to state, that *OpenTracker* and *DIV* messages cannot be put into *total order*. Although causality, order and dependency of messages can be applied to *DIV* and to a certain degree to *tracking* data, this does not work, when mixing both concepts. This introduces the problem how possible inconsistencies can be prevented. An example to such a problematic synchronization aspect is the object move operation, where both strategies (*DIV* as well as *OpenTracker*) are involved.
To resolve this conflict, attention to *data flow* has to be paid to: *Tracking* data is sent to the *master* to allow interaction also substitutionally for each *slave*. So, *OpenTracker* networking features are used to indirectly perform application operations on a central instance, the *master*. The effects of all operations are distributed by *DIV* to all *slaves*. But as stated before, *tracking* data is also used for some minor indirect synchronization features, *DIV* is not dealing with. The solution is fairly easy: *DIV* always overrules *OpenTracker* and *OpenTracker* data is prohibited to cause any transient and permanent application state change on *slaves*. This means that *tracking* data can be used to generate some minor modifications at a certain instant like displaying dragging status while moving objects or updating visual menu appearance. Nevertheless, the final result of a move operation and the visual representation of application states must not be set by *tracking* data on *slaves*. This guarantees consistency by trusting the

reliability and *total message order* property of *DIV*.
Considering data omission failures in *OpenTracker* data, numeric inaccuracies, and late joining users, menu appearance (apart from application states) might be different between several participating sites. Widgets reflecting an application state, distributed by *DIV*, even have to prohibit direct manipulation caused by *tracking* data to maintain consistency.
For similar reasons, it is also possible that different objects seem to be involved in moving operations, not all participants are able to observe the move operation or the positions at a certain instant are not the same. Reverting and resetting the dragging operation on *slaves* afterwards, ensures that the final result is consistent on all participating sites. After all, it is up to *DIV* to update the moved object according to its final position. Even if the *DIV* message is received prematurely (before actually resetting the dragging operation), this strategy works fine: The reset position is updated according to the final destination determined by *DIV*.

Chapter 4

Implementation

According to design considerations presented in chapter 3, some implementation details shall be roughly outlined. Maintaining the same order in the following, *OpenTracker*, *Distributed Open Inventor*, *Studierstube* and *Construct3D* are subject of implementation related issues:

4.1 Tracking data distribution by OpenTracker

Networking functionality in *OpenTracker* is implemented with the help of the *ACE*[1] (*ADAPTIVE Communication Environment*) framework [Sch94]. This is a platform independent toolkit for and written in *C++*. It provides a rich set of components, performing common communication tasks including basic networking.
As mentioned in section 3.1, *datagrams* are sent into and received from the network, interacting with the *data flow graph* of *OpenTracker*.
The *NetworkSource* and *NetworkSink* functionality is managed in the corresponding modules (**NetworkSourceModule** and **NetworkSinkModule**), while (among other structures) the *C++* classes **NetworkSource** and **NetworkSink** serve mainly as data containers.
Internally, the **NetworkSender** class hierarchy with derived **UdpSender**, **MulticastSender** and **UnicastSender** and, on the other hand, **NetworkReceiver**, **UdpReceiver**, **MulticastReceiver** and **UnicastReceiver** are responsible for storing network ressource information and *tracking* data records, as used in the network (**FlexibleTrackerDataRecord**). **Station** is used to demultiplex *tracking* data of different devices (*stations*) on the receiver's side.

4.1.1 XML configuration

Both *NetworkSource* and *NetworkSink* are created at runtime according to the *XML* configuration file via **NetworkSource** and **NetworkSink** elements. These elements (summarized in table 4.1 and table 4.2) allow configuration by attributes:

[1] *http://www.cs.wustl.edu/~schmidt/ACE.html*

- First of all, **mode** specifies the networking mode: The identifiers **"unicast"** and **"multicast"** are possible, where *multicast* mode is the default.
- **number** defines a *station* number, a nonnegative integer, allowing to distinguish *tracking* data of different devices.
- **port** identifies the *UDP port* to send or receive *tracking* data in both networking modes.
- **multicast-address** is used in *multicast* mode, setting the *multicast group* address.
- **address** has to be specified in *unicast* mode of a **NetworkSource** to identify the address of a *server* to contact for *tracking* data.
- By adding the attribute **interface**, **NetworkSink** allows the specification of the *network interface* to use.
- Finally **name** in **NetworkSink** assigns a name to *tracking* data.

attribute	type	required	default
mode	**unicast** or **multicast**	no	**"multicast"**
number	character data	yes	
port	character data	yes	
multicast-address	character data	no	
address	character data	no	
DEF	unique identifier	no	

Table 4.1: **NetworkSource** attributes

attribute	type	required	default
mode	**unicast** or **multicast**	no	**"multicast"**
number	character data	yes	
port	character data	yes	
multicast-address	character data	no	
interface	character data	no	**""**
name	character data	yes	
DEF	unique identifier	no	

Table 4.2: **NetworkSink** attributes

The **name** attribute can also be assigned to the *NetworkSinkModule* in the **NetworkSinkConfig** *XML* element in order to provide additional identification of the *server* (see table 4.3).

attribute	type	required	default
name	character data	yes	

Table 4.3: **NetworkSinkConfig** attributes

Putting all that together, an *XML* configuration file can look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE OpenTracker SYSTEM "opentracker.dtd">
<OpenTracker>
    <configuration>
        <NetworkSinkModule name="server name"/>
    </configuration>
    <NetworkSink mode="multicast" name="keyboard" number="0"
                 multicast-address="224.0.0.10" port="12345">
        <StbKeyboardSource number="0"/>
    </NetworkSink>
    <NetworkSink mode="multicast" name="keyboard" number="1"
                 multicast-address="224.0.0.10" port="12345">
        <StbKeyboardSource number="1"/>
    </NetworkSink>
    <NetworkSource mode="multicast" number="0"
                   multicast-address="224.0.0.10" port="12346"/>
    <NetworkSource mode="unicast" number="0"
                   address="localhost" port="12347"/>
</OpenTracker>
```

4.1.2 Node implementation

A *node* (a *C++* class derived from **Node**) has to implement an interface, which definines how *tracking* data is communicated:

- A **Node**, implementing the *event generator* interface, simply returns **1** in the **isEventGenerator** method.
- The method **onEventGenerated** is called whenever *tracking* data, represented by a **State** instance, is passing the **Node** instance, following the path from *sources* to *sinks*.
- By calling **updateObservers**, **State** data is pushed further down the *data flow graph*.

Both **NetworkSource** and **NetworkSink** implement the *event generator* interface. Calling **updateObservers** from within **onEventGenerated** in **NetworkSink** propagates **State** information and extends the *sink* semantics to intermediate *node* type. As *sources* are the origin of *tracking* data, **onEventGenerated** will not be called in **NetworkSource**.

4.1.3 Module implementation

The corresponding *modules* implement the *node factory* interface (deriving from **Module** and **NodeFactory** classes):

- **init** allows *module* configuration as set in the *XML* file.
- The **Module** is actually started by **start**,
- while **close** does final cleanup.
- The method **pullState** pulls passed down **State** data for further actions,

- wheras **pushState** is responsible for pushing **State** data into the *graph*.
- **createNode** is called when reading the *XML* configuration file to create the *tree* of *nodes*.

The **NetworkSinkModule** handles sending *tracking* data in the **pullState** method, while **NetworkSourceModule** implements **pushState** to reinsert received *tracking* data. These methods have to associate between *tracking* data and *UDP* network ressources, paying attention to *station* information and caching **State** instances.

4.1.4 Multicast UDP implementation

In *multicast* mode, the **NetworkSinkModule** simply sends *datagrams* to *multicast groups* as configured by *NetworkSink* elements, while **NetworkSourceModule** (method **runMulticastReceiver**) has to spawn one thread per *multicast group* to listen for incoming *tracking* data packets. This multithreading model is needed to prevent process blocking while waiting for *datagrams* to arrive. On the sender's side no additional thread is needed.

4.1.5 Unicast UDP implementation

In contrast to *multicast* mode and as stated in section 3.1, *unicast* implementation has to maintain network membership manually without the help of a *multicast group*.

NetworkSinkModule

The multithreading approach is also used in the **NetworkSinkModule** (method **runUnicastTransceiver**), due to the fact that membership information has to be received and processed without influencing other tasks. It performs just inspection of the contents of a *datagram* containing a single character and reacts properly. This character distinguishes between *poll* (**'P'**) and *leave* (**'L'**) commands.

NetworkSourceModule

Similar to *multicast* implementation, the threads in **NetworkSourceModule** (method **runUnicastTransceiver**) have to listen for incoming *tracking* data packets. If no *datagrams* arrive (a timeout in receiving packets occurs), *polling* is performed regularly. This is done, because the *tracking* server might have been terminated. These *pPolling* requests reassociate *clients* to the *server* if the latter is restarted. Thread termination causes transmission of a single *leave* command.

4.2 Distributed Open Inventor

The implementation of the *Distributed Open Inventor* design (as mentioned in section 3.2) was also done with the help of *ACE*[2]. Getting notified of *scene graph* changes is achieved by attaching *sensors* to *nodes*.

[2] *http://www.cs.wustl.edu/~schmidt/ACE.html*

4.2.1 Synchronizing scene graphs

A *master* has to observe its shared *scene graph* (see section 3.2): In the method **shareNode** of **CDivMain**, the implementation class of *DIV*, a **SoNodeSensor** is attached to a **SoNode** instance. This allows reaction on changes of the *scene graph subtree* relative to the *node*, where the *sensor* is attached to. A *callback* function is indirectly called by the method **rootChanged**. Even the deletion of this very same *node* can be detected (indirectly by the method **rootDeleted**). **unshareNode** is the counterpart to **shareNode** to disable *scene graph* distribution.
Inside these *notification callbacks*, information about changes in effect can be obtained by retrieval methods of the **SoSensor** parameter. Evaluating this information (in *sensor* scope internally taken from a **SoNotRec**, the *notification* data structure) causes the generation of proper *DIV* network messages. As described earlier by Hesina [Hes01], these messages include:

- Update *field* messages (**DIV_MODIFY**, **DIV_SOMFMODIFY**) to indicate changes of *OIV fields*,
- structural *scene graph* modification messages for *group* operations (**DIV_ADDCHILD**, **DIV_INSERTCHILD**, **DIV_REMOVEALLCHILDREN**, **DIV_REMOVECHILD**, **DIV_REPLACECHILD**),
- not further specifiable updates (**DIV_TOUCHMODIFIED**) as well as
- *node* naming messages, also used in *lazy naming* (**DIV_SETNAME**, **DIV_SETSOSFNAME**).

On the receiver's side, *DIV* messages coming from the network have to be interpreted. **processMessage** of **CDivMain** is indirectly called to evaluate the contents of these messages and react properly as indicated by the message type.

4.2.2 Additional control messages

Some additional *DIV* messages are available for other useful features:

- *Node transfer* permits initial transmission of *scene graphs* and is handled by a *request-response mechanism* (**DIV_REQUEST_NODETRANSFER**, **DIV_TRANSFERNODE**).
- *DIV* message compression (**DIV_COMPRESSED**) tries to make network traffic more efficient by compressing data of other *DIV* messages. This is especially targeted on the *node transfer* feature.
- *Master* and *slave* property can be switched conforming to another *request-response mechanism*, as described in section 3.2 (**DIV_REQUEST_MASTER**, **DIV_TRANSFER_MASTER**).

4.2.3 Network interface

ControlModule defines the interface to implement an underlying network:

- **dataReceived** should be called within the implementation, when payload data has been received to notify the associated **NetworkProcessor**, a utility base class instance, which has been supplied via constructor.
- **sendData** is usually called by the **NetworkProcessor** and has to transmit payload data stored in a buffer of supplied length.
- **close** should terminate access to the network, which should have been established on construction time.

Targeted on *ACE*, **ACE_ControlModule** (derived from **ControlModule**) introduces some useful features: Sending and receiving is done in an extra thread and managed by synchronized queues. Providing a thread allows to perform network tasks asynchronously without blocking *OIV* and the rendering process. Data exceeding a threshold of about 1 KByte can be optionally compressed (**DIV_COMPRESSED** message).

- **sendQueuedData** has to actually transmit payload data, as **sendData** is already implemented to preprocess (compress) and queue data. It is called by **OutputMessageHandler**, working on the output queue.
- **queueReceivedData** should be called to queue received payload data. **dataReceived** is automatically called by means of **InputMessageHandler**, processing the input queue uncompressing data, if necessary.

4.2.4 Multicast UDP implementation

ACE_RMCast_ControlModule inherits all features from **ACE_ControlModule** and implements missing functionality to actually join and leave *multicast groups* as well as send and receive data. The *RMCast* library used in this implementation ensures *reliability* of the otherwise *unreliable* nature of the *UDP* protocol.
Multicast UDP networks can be configured by specifying the *multicast group* and *port*.

4.2.5 TCP implementation

ACE_TCP_ControlModule is also derived from **ACE_ControlModule** and conducts all *TCP* related actions: It establishes *server* functionality via **ACE_TCP_Acceptor** and tries to discover the network actively via **ACE_TCP_Connector**.
Both classes are derived from **ACE_Event_Handler**, a well known *ACE* concept, to perform asynchronous actions. In the case of **ACE_TCP_Acceptor**, the *TCP* connection accept functionality is implemented by this mechanism at the same time.
Established connections are represented by **ACE_TCP_Connection** instances and are stored in **ACE_TCP_ControlModule**. This class also utilizes **ACE_Event_Handler** as callback mechanism to react on events concerning

TCP connections: Whenever *TCP* is ready to read from or write to a connection, a certain method is called.

TCPCommand

All message processing is done inside classes derived from base **TCPCommand**, each class representing a particular messages type:

- **TCPDataCommand** contains payload data and interfaces with **ACE_TCP_ControlModule** for sending and receiving.
- Control message **TCPIdentifyPeerCommand** is sent, whenever a *peer* actively establishes a connection. So, **ACE_TCP_Connector** is responsible for creation and transmission of such messages. Receiving such a message causes **ACE_TCP_ControlModule** to examine the given identification, closing redundant connections.
- **TCPNewPeerCommand** replaces **TCPIdentifyPeerCommand**, when an initial connection is established. Apart of the reaction receiving a **TCPIdentifyPeerCommand**, distribution of the new *peer* also has to be performed.
- **TCPDistributeNewPeerCommand** contains information of new *peers* including *TTL* lifetime information. This message is sent as reaction to **TCPNewPeerCommand** or **TCPDistributeNewPeerCommand** (if lifetime information is not expired) to all connections but the one which triggered message processing. As mentioned in section 3.2, *peer* identification and *TTL* value has to be examined, optionally establishing nonexisting connections via **ACE_TCP_Connector** and message forwarding.

TCP networks are configurable by *server port*, a list of potential *peers* running a *TCP* network (for initial discovery) and an initial *TTL* value used for new *peer* distribution, if directly contacted by a newly participating *peer*.

4.2.6 TCP multicast UDP hybrid implementation

ACE_Bridging_TCP_RMCast_ControlModule derives from **ControlModule** and **NetworkProcessor**. As described in section 3.2, it is in an experimental state of implementation and stores instance pointers to a single **ACE_RMCast_ControlModule** and one **TCP_ControlModule**. Data is sent to both networks simultaneously. On data reception, it is processed and resent to the network, where it was not received.
Configuration options include all *multicast UDP* and *TCP* settings.

4.2.7 Open Inventor extension

The *OIV* implementation, used throuhout this work, is *Coin3D*[3]. Its functionality had to be extended to obtain additional information for efficient distribution:

- More precise information on changes of *multiple-value fields* (**SoMField** and derived) is needed to identify parts that have actually changed.

[3] *http://www.coin3d.org/*

- Information on structural *scene graph* changes (**SoGroup** and derived) has to be retrieved.

A **SoNotRec** object is essential to the core *notification* feature and accessible from within any *sensor*. So this is the place to extend *OIV* by adding new members and methods dealing with information of *multiple-value field* operations and structural *scene graph* modifications.
A **SoNotRec** is mainly constructed in basic *OIV* classes to initiate the *notification* process. Modifying this framework, **SoBase** and **SoField** were extended by a virtual method (**createNotRec**), transferring **SoNotRec** construction. In subclasses, this allows to append additional information, obtained by various manipulation methods. For example, **SoGroup** overloads this method to store data reflecting the structural *scene graph* change, **SoMField** (and derived) add information about modified *multi-value field* slots. Accidently, this information is present only in manipulation methods and cannot be passed down by arguments easily. So this data is temporarily cached and can be accessed, when the update framework calls the overridden method somewhere from within the manipulation method. From this point on, the information resides in **SoNotRec** and the data cache is not needed anymore.

4.2.8 DivGroup

For convenience, *DIV* can be easily used by inclusion of *DivGroup*: Configuration und usage is much simpler than directly performing operations on **CDivMain**.
SoDivGroup is a *group node*, enhanced by *DIV* capabilities for its *subtree*. Configuration effort is reduced to set up *OIV fields* properly (summarized in table 4.4):

- First of all, **networkLayer** is an **SoSFEnum** indicating the network to run *DIV* on: **MULTICAST** is the default, other options are **TCP** and **TCPMULTICAST** (*hybrid* mode).

- **multicastGroup** (of type **SoSFString**) contains the *multicast group* address to use in *multicast UDP* or *hybrid* mode.

- **tcpPeers** is the counterpart in *TCP* and *hybrid* mode, storing a number of *peers* to contact initially. Being of **SoMFString** type, the list can contain zero or more combinations of *host* and *port* address information. This list is succesively processed, trying to establish a *TCP* connection. On successfull connection to a *peer*, no further action is performed with the list tail.
  Initially contacted *peers* figure out the addresses of newly arriving *peers* by inspecting the *TCP* connection properties. So it is important to emphasize that each of these addresses should be globally defined and uniquely valid, to guarantee that the figured out addresses also comply to this requirements. This is needed, because these addresses are distributed among other *peers* in **TCPNewPeerCommand** messages. If these requirements are ignored, other *peers* might fail to contact the desired new *peer*.
  Firewalls might also deny establishing *TCP* connections. This also can lead to incomplete networks, lacking of some vital connections.

- The **SoSFInt32 port** is the *port number* of the *multicast group* in *multicast* and *hybrid* mode as well as the *server port* in *TCP* and *hybrid* mode.
- **nic**, a **SoSFString**, allows to set the *network interface* for *multicast UDP* in *multicast* and *hybrid* mode.
- **tcpDistributePeersTTL** (of **SoSFInt32** type) controls the *TTL* value used in distribution messages of new arriving *peers* in *TCP* and *hybrid* mode and defaults to **2**.
- The **SoSFBool multicastAutoUniqueNaming** is enabled by default (**TRUE**), indicating that an automatically generated prefix created by *multicast group* information should be used in *multicast* mode.
- **manualUniqueNamingPostfix** is used in *TCP* and *hybrid* mode as well as in *multicast* mode, when **multicastAutoUniqueNaming** is disabled. This **SoSFString** value (defaulting to **""**) defines a prefix appended to an unnamed *node*.
  Naming is crucial due to the fact that *nodes* are identified in *DIV* by their names. This implies that names of replicated *nodes* have to match.
- The flag (**SoSFBool**) **active** is used to enable **DIV** support. Initially, this is disabled (**FALSE**).
- **isMaster**, a **SoSFBool** flag, allows to determine the *master* property.
- Triggering (**SoSFTrigger**) **getMaster** initiates the *request-response mechanism* to become a *master*.
- **initFromMaster** (of type **SoSFBool**) defaults to **TRUE**, enabling the *node transfer* feature.
- Finally the flag (**SoSFBool**) **transferTextures** allows disabling transmission of textures and images to reduce network load. It defaults to **TRUE**.

Putting all this together, a *DivGroup* can be represented like this in an *OIV* file:

```
SoDivGroup
{
    networkLayer TCP
    port 12345
    tcpPeers [ "localhost:12346" ]
    active TRUE
    manualUniqueNamingPostfix "Div_TCP"
    initFromMaster TRUE
}
```

4.3 Studierstube

Stbapi is the core library in *Studierstube* written in *C++*. Apart from *DIV* implementation (see section 4.2) it contains also *distribution management* functionality as described in section 3.3.

field	type	default
networkLayer	**SoSFEnum**	**MULTICAST**
multicastGroup	**SoSFString**	""
tcpPeers	**SoMFString**	[""]
port	**SoSFInt32**	**0**
nic	**SoSFString**	""
tcpDistributePeersTTL	**SoSFInt32**	**2**
multicastAutoUniqueNaming	**SoSFBool**	**TRUE**
manualUniqueNamingPostfix	**SoSFString**	""
active	**SoSFBool**	**FALSE**
isMaster	**SoSFBool**	**FALSE**
getMaster	**SoSFTrigger**	
initFromMaster	**SoSFBool**	**TRUE**
transferTextures	**SoSFBool**	**FALSE**

Table 4.4: **SoDivGroup** *fields*

4.3.1 Distribution management

Client functions of *distribution management* are part of *Stbapi*, while the *server* resides in the *session manager*, a stand-alone tool program (**sman2**).
workspace, the executable program to start *Studierstube* applications, is configurable by a new additional option to indicate the desired network mode for an application host: **-dtcp** (**-dt** for short) enables *TCP* mode instead of *multicast UDP* mode.

SessionManager

SessionManager opens a *server socket* for all *clients* listening for requests. **Message** is the base class of all messages of various types and exchanged between *Studierstube* and *session manager*. It defines some basic mechanisms:

- **readMessage** is used to inspect the contents of a message, setting member data accordingly on success.
- **writeMessage** converts message data to proper network representation.

SessionMessage extends **Message** by the additional **execute** method, a placeholder to perform actions depending on message data.
All messages are derived from **SessionMessage**, implementing **readMessage**, **writeMessage** and **execute**. Additionally, a static method for interfacing with **MessageFactory** is included to make all **Message** derived classes known to distribution management and being able to construct the proper **Message** instance from network representation.
Apart from messages carrying mainly user ressource data (**UserKitMessage** derived) and those modelling user (**AddUserMessage**, **RemoveUserMessage**) and *locale* (**JoinLocaleMessage**, **LeaveLocaleMessage**) concepts, there are only few messages remaining:

- **StartApplicationMessage** as counterpart to **StopApplicationMessage** includes data about the desired *DIV* networking mode (**DIVMode**).

- **SetDIVParamMessage** is perhaps of highest importance for *DIV*. This message contains all parameters needed for *DivGroup* configuration: *DIV* is activated according to a flag, the *master-slave* property is set and **DIVParam** covers information about networking mode and associated settings.

So, *session manager* does all *DIV* reconfiguration by sending a **SetDIVParamMessage** to each participating site. With the help of *locale*, user, and application messages, it keeps track of participant's membership, distributed applications and user presence: **Locale**, **Application**, **User** and **Host** contain information about related concepts.
Application stores *DIV* configuration used by **SetDIVParamMessage**, which contains **DIVMode**, application-wide *multicast* parameters (**DIVMulticastParam**) and *TCP* information (**DIVTcpParam**) on a per **Host** basis. These network settings are created by **DIVAddressFactory**, a utility class, which generates unique address and *port* information by maintaining counters.
When running *DIV* in *TCP* mode, it is important to contact the *session manager* by passing a **-smo** option with globally defined and uniquely valid network address to **workspace**. The reason of this is easily explained: The address of each participant is figured out by inspecting the *TCP* connection between *session manager* and *Studierstube*. On default, the *session manager* (program **sman2**) is trying to be contacted on **localhost**, an address not conforming to the network address requirements, mentioned before: The address of the participating site, figured out and used by *DIV* in *TCP* mode, might be also **localhost**. Usually, this causes failures in building up the network, when not only a single hosts is participating. This is a result of transmitting network addresses (without any translation effort) to all *peers* in each **SetDIVParamMessage**.

StbDistrManager

StbDistrManager forms the *client* counterpart to distribution management in *Stbapi*. It establishes a *TCP* connection to *session manager* and delegates execution of commands to a **DistributionStrategy** instance, where **LocalDistribution** is a dummy implementation without distribution abilities. Derived **RemoteDistribution** does message sending and retrieval.

SoContextManagerKit

SoContextManagerKit is called from within **RemoteDistribution** while execution of received application and *DIV* related messages. On the other hand, it causes also transmission of those messages by calling **StbDistrManager** methods:

- **loadApplication** causes the creation of the **SoDivGroup** (indirectly by calling **loadApps**), used as *parent* of the application's *scene graph*. After building the application *scene graph* from stream representation, a **StartApplication** message is generated and sent to the *session manager*.
- **setDivParam** is called as reaction to the reception of a

SetDIVParamMessage and does all *DIV* reconfiguration. So this is where distribution is enabled.

4.3.2 Distribution exclusion

SoHiddenChildGroup is the implementation of the *HiddenChildGroup* concept (as mentioned in section 3.3). Even if this implementation seems so small, it suffices to do all work well. Basically, methods of **SoGroup** are overridden:

- **notify** inspects a **SoNotRec** instance. If the origin of the *notification* is not the current *node* instance, the *notification* is retriggered, omitting propagation of the original *notification* record.
  This smart retrigger strategy is needed to indicate that something has happened in the *subtree* below, even if the actual origin is hidden by pretending being the origin. Without this, some basic mechanisms including rendering would likely fail.
- **write** modifies the behavior of the *write action*, blocking *scene graph traversal* to exclude *children* from being written to stream.
- For consistency reasons, **readChildren** disallows *children* to be read from stream.

These few modifications to original **SoGroup** behavior are sufficient to effectively prevent distribution. But changing reading from and writing to streams impacts also file operations: No *children* of a **SoHiddenChildGroup** are possible in file representation. Compared to the *sensor* used by *DIV*, breaking *notification* propagation has the same effect on any other *sensor* attached somewhere else than in any *subtree* of the *children*. *Field connections* including *engine* concept work fine, as no *notification* propagation upwards the *scene graph* takes place.

4.4 Construct3D

Construct3D is implemented in **SoCnDKit**. According to the *scene graph* structure, depicted in section 3.4, a *root separator* contains all application specific *nodes*, while some additional data is present as *fields* inside the *CnDKit*.

4.4.1 Application startup

On application startup, a *CnDKit* is usually created by reading data from file, if not obtained by distribution:
workspace is given a program argument like **-a c3d.iv** to supply a *OIV* file containing a **SoApplicationKit**. A **SoApplicationKit** aggregates a **SoContextKit** to define its *context*. By means of a **SoClassLoader** *node*, *Studierstube* is able to locate the library, where custom classes are defined. This dynamic loading mechanism, as described in section 3.3, is used to get access to **SoCndKit**, a custom *ContextKit* implementation.
User configuration is happening in a similar manner:
The **workspace** option **-uk** allows the specification of an *OIV* file containing at least one *UserKit*. Apart from assigning a unique *user id*, a **SoUserKit**

instance consists of *nodes* storing viewing and *PIP* information: *DisplayKit*, *WindowKit*, *PipKit* and *PenKit* are part of this data collection to define user ressources. Although a *PIP sheet* can be statically assigned via *PipSheetKit*, this is done dynamically in *Construct3D*.
These ressources directly determine rendering output mode, configure users and associated *PIP* ressources, if necessary.
OpenTracker is configured by parsing an *XML* file, followed by the **-tr** program option: **StbSink** elements are used in this configuration file to interface with *Studierstube*, inserting *tracking* data as 3D events into the *scene graph*.
In distributed operation mode, a *locale* has to be specified to join to by supplying an argument like **-jl default**. All but one *Studierstube* instances must not specify the **-a** argument (unless users intend to load multiple applications), as the application *scene graph* is obtained performing *node transfer*.

4.4.2 Dynamic initialization

The *ContextKit* interface defines useful methods to dynamically initialize a **SoContextKit** instance like **SoCnDKit**:

- **checkWindowGeometry** is perhaps one of the most important methods:
  In case of absence of a *scene graph*, it is built according to the structure shown in section 3.4 including the *HiddenChildGroup*.
  If the method is called and the structure is already present, it is likely to be the case that it was initialized by distribution. If the *slave* property was set, the implementation has to find the **SoHiddenChildGroup** instance and add its undistributed *subtrees* manually. (Recalling the behavior of a *HiddenChildGroup* in section 3.3, distribution of *subtrees* is effectively blocked, but the *parent* is actually replicated.) Finally the **SoUndoRedoListKit** instance has to be searched for to attach the *sensor* triggering geometry regeneration.

- **checkPipGeometry** is the place to focus on *PIP sheet* creation, used as template for all users.
  In *Construct3D*, the *PIP sheet* layout is dynamically loaded from a separate file (**pipSheet.iv**) and added as *child* to a **SoHiddenChildGroup** instance, previously defined as trunk in **c3d.iv** and found by applying a *search action* to attach the *PIP sheet* geometry to.

- **checkPipConnections** allows initial *PIP sheet* configuration.
  SoCnDKit performs operations on its database (**pipLayouts**), associating *user id* to *PIP sheet* geometry. This is needed to distinguish each user's *PIP* and being capable of determining user information when dealing with interface actions.

- **checkPipMasterMode** is used for additional processing depending on the *master-slave* property. As this method is called, whenever this property changes, this is the central position to set up interfaces between the application and each *PIP* properly.
  In *master* mode, callbacks are registered to react on user's *PIP* actions. On *slaves*, most of these callbacks are undesired, as the *master* has to

perform all actions, relying on *tracking* data distribution as emphasized in section 3.4. Some callbacks have to be present also for *slaves* to prevent direct widget interaction.
Also, *field connections* are set up between **SoCnDKit** *fields* and *PIP* interface elements. These *fields* (summarized in table 4.5) automatically keep *PIP sheet* appearance in sync to various application states. Instead of adding these *fields* to the *catalogue* to perform direct synchronization, application state is distributed by special commands.
undoAvail and **redoAvail** (both **SoSFBool**) determine, if undo and redo operations are allowed. As there exists a single shared undo/redo history, no distinction between each user is made. In fact, simple *field connections* to interface elements are set up to enable and disable them properly.
The **activeLayer** *field* (of type **SoSFInt32**) is connected via *engines* to interface elements visualizing the current, globally defined active layer.
Each element of the also globally for all users defined **layerOn** *field* array (of type **SoSFBool**[]) is connected to the layer related widget indicating its visibility.
Finally, **drawPoints** contains user related settings of type **SoMFBool**. This *field* specifies, if point setting or selection mode is active. An *engine* is responsible for selecting the proper user value of this *multi-value field*. This *field connection* controls the appropriate widget on the user's *PIP*.

- **setMasterMode** is called, whenever the *master-slave* property changes. Consequently, this method is also executed on startup.
A *master* tries to read from a special file called **loadOnStart.iv**, looking for an *UndoRedoListKit* exactly in the same manner, as saved files are treated: On success, its content is copied to the instance being part in the *scene graph* and undo/redo list commands are executed.
Slaves must also execute undo/redo list commands until the actual index position, but instead of reading from file, the *UndoRedoListKit* is already up to date by *node transfer* caused by *DIV*. An initial execution is necessary, because the *sensor* cannot be attached to the *UndoRedoListKit* instance before *node transfer* takes place: Program execution in **SoCnDKit** is not started, until an instance has been previously created by processing *OIV* file format interpretation obtained from the *node transfer* stream.
Also, some minor differences between *masters* and *slaves* are dealt with in this method to allow smoothely transitions in this property.

field	type	initialization
undoAvail	**SoSFBool**	**FALSE**
redoAvail	**SoSFBool**	**FALSE**
activeLayer	**SoSFInt32**	**0**
layerOn	**SoSFBool[]**	**{TRUE, FALSE, FALSE, ...}**
drawPoints	**SoMFBool**	[]
pipLayouts	**SoMFNode**	[]

Table 4.5: Important **SoCnDKit** *fields* (not being part of the *field catalogue*)

4.4.3 Undo/redo list

UndoRedoListKit is implemented in **SoUndoRedoListKit**. **SoCnDKit** instantiates a single **SoUndoRedoListKit** to operate on. Undo/redo functionality is done in a few methods:

- **findUndoRedoList** is the recommended way to search for and retrieve the **SoUndoRedoListKit** instance included in the *scene graph*.
- **undo** performs a single undo operation,
- while **redo** executes a redo.
- **redoAll** is used in file loading to perform several redo operations as a whole. Commands are executed until the index pointer is reached.
- **undoRedo** does a similar thing and is called on *slaves* as reaction on changes caused by *DIV*. The algorithm executes several undo/redo commands, processing the working queue.

SoUndoRedoListKit

To the core, **SoUndoRedoListKit** aggregates **SoCommandKit** items, organized in an ordered list. This information is stored in *fields* of restricted accessibility level, also summarized in table 4.6:

- **undo_redo_List** (of type **SoMFNode**) contains all **SoCommandKit** instances,
- whereas **undo_redo_List_pos** (**SoSFInt32**) resembles the information, pointing to the current position in that list.

startTime is present only for informational purposes.

field	type	default
undo_redo_List	**SoMFNode**	[]
undo_redo_List_pos	**SoSFInt32**	**-1**
startTime	**SoSFTime**	**0.0**

Table 4.6: **SoUndoRedoListKit** *fields*

To access and manipulate these *fields*, the **SoUndoRedoListKit** provides powerful methods:

- **getNumOfItems** retrieves the number of undo/redo list entries,
- while **getListPos** returns the index pointer, ranging from **-1** to the position of the last entry.
- **getCommandLine** can be used to return a particular **SoCommandKit** instance. It ensures maximum flexiblitity, whether the undo/redo list record at the current position is of interest, or random access to any command in the list by supplying an index as parameter prefered.
- **incListPos** increments the undo/redo list position (used in redo operations),

- while **decListPos** performs decrementation (used when undoing operations).
- Moreover, **setListPos** directly manipulates the index, pointing to the current undo/redo list record.
- **truncateList** flushes and truncates the undo/redo list after the current position. This can be also used to clean the whole list on reinitialization.
- **copyList** sets all internal *fields* according to the settings of another **SoUndoRedoListKit** instance. This is useful in the case of file loading.
- Finally, **add** is used to add a new undo/redo list record (**SoCommandKit**) at the current index position, flushing the list beyond this position pointer before and incrementing the pointer afterwards to guarantee a consistent state.

In the following, a small example of an *UndoRedoListKit* in *OIV* file format is given. It represents two operations, adding a point and selecting it afterwards:

```
DEF undo_redo_List SoUndoRedoListKit
{
    undo_redo_List [
        SoCommandKit {
            command "add"
            commandPos 0
            objectName "P_0"
            objectType "Point"
            selectedObjectNames [  ]
            position 0.1 0.2 0.1
            userID 0
        },

        SoCommandKit {
            command "select"
            commandPos 1
            objectName "P_0"
            selectedObjectNames [  ]
        } ]
    undo_redo_List_pos 1
}
```

SoCommandKit

As it was already observable in the small example before, **SoCommandKit** contains many *fields* to cover information about a single operation. The most important *fields* of table 4.7 shall be explained:

- The **SoSFName command** determines the carried out command.
- **objectName** specifies the name (**SoSFName**) of a single object an operation is associated to,

- while **selectedObjectNames** contains the names (**SoMFName**) of several objects, if the operation relies on multiple objects.
- **objectType** (of type **SoSFName**) is the type of the object in object creation operations.
- The *user id* of any operation, related to a certain user, is stored in **userID** of type **SoSFInt32**.

field	type	default
command	**SoSFName**	""
commandPos	**SoSFInt32**	**0**
objectName	**SoSFName**	""
objectType	**SoSFName**	""
filePath	**SoSFName**	""
selectedObjectNames	**SoMFName**	[""]
position	**SoSFVec3f**	**0.0 0.0 0.0**
startPosition	**SoSFVec3f**	**0.0 0.0 0.0**
endPosition	**SoSFVec3f**	**0.0 0.0 0.0**
degreeNew	**SoSFInt32**	**0**
degreeOld	**SoSFInt32**	**0**
sliderValueNew	**SoSFFloat**	**0.0**
sliderValueOld	**SoSFFloat**	**0.0**
layer	**SoSFInt32**	**0**
activeLayerNew	**SoSFInt32**	**0**
activeLayerOld	**SoSFInt32**	**0**
layerOn	**SoSFBool**	**FALSE**
userID	**SoSFInt32**	**-1**
appID	**SoSFInt32**	**-1**
time	**SoSFTime**	**0.0**

Table 4.7: **SoCommandKit** *fields*

None of these *fields* can be directly accessed, but numerous retrieval methods are provided.
SoCommandKit instances are not created directly. Instead, a wide range of derived classes are present, offering a convenient way to construct new undo/redo list records. All of these subclasses are not known to *OIV* as separate *node* types. Nevertheless, the mechanism is fairly simple:

- By directly creating a specific subclass, all relevant parameters of the more general base class are set and the instance can be treated like any ordinary *node*, as the base class is a complete *node* implementation. Especially, adding to **SoUndoRedoListKit** and writing to stream are possible.
- When reading from stream, **SoCommandKit** instances are created, as *OIV* cannot distinguish between subclasses and even does not know anything about them. But this suffices, as internal *fields* of the general **SoCommandKit** define command and associated parameters, providing an interface to retrieve data. Subclasses offer additional convenience methods, mainly to classify the **SoCommandKit** instance and also for further information.

All **SoCommandKit** subclasses are closely related to *Construct3D* application features:

- **SoAddCommandKit** is not intended for direct use and base for all creation commands:
  - A **SoAddPointKit** represents a point generation operation.
  - **SoAdd2DPrimitiveKit** covers major 2D primitives (line, circle, ellipse and triangle),
  - while **SoAdd3DPrimitiveKit** is the 3D counterpart (sphere, cylinder, cone, box).
  - **SoAddCurveKit** is related to free-form curves (of approximation or bezier type),
  - whereas **SoAddSurfaceKit** represents surfaces (of approximation or b-spline type).
  - Planes are added to the undo/redo history by **SoAddPlaneKit**.
  - Parameters for boolean or construct solid geometry operations are stored by instantiating a **SoAddBoolKit**.
  - Sweeps are represented in the undo/redo list by **SoAddSweepKit**.
  - Data about slicing is added to history by creating a **SoAddSliceKit**.
  - **SoAddIntersectionKit** covers information about intersection curves.
  - **SoAddAngleBisectorKit** is related to a bisector angle.
  - Finally, **SoAddTextKit** concludes the list of creation commands adding textual measurement information.
- Selection changes are generated by instantiation of a **SoSelectKit**,
- and **SoDeselectAllKit** is a special command to clean selection.
- Whenever an object moves, a corresponding **SoMoveKit** is constructed.
- **SoDeleteKit** represents deletion of certain objects,
- removing a single object can be done by adding **SoDeleteOneKit** to undo/redo history.
- Curve creation parameters can be altered by means of a **SoRebuildCurveKit**.
- General transformations (translation, rotation, mirroring) are represented by **SoTransformKit**.
- With **SoChangeLayerKit**, objects can be moved between different layers.
- Changing the active layer causes instantiation of **SoActiveLayerKit**,
- while information about layer visibility is added via **SoLayerOnKit** to the undo/redo list.

- **SoDrawPointsKit** represents draw vs. selection mode property.
- Various slider value changes used for curve and surface creation are distributed by instantiating a **SoSliderValueKit**.
- **SoLoadFileKit** corresponds to the *VRML* file loading mechanism.

Some operations, particularly geometric construction functions, generate a preview to give visual feedback before actual execution. This preview is also implemented as single **SoCommandKit**, but not added to the **SoUndoRedoListKit**. Furthermore, this preview is not distributed. So, before undoing or redoing, the effects of any preview command have to be reverted.

4.4.4 Geometry update detection and execution

As described in section 3.4, a *sensor* is used to detect updates in the **SoUndoRedoListKit**. **SoUndoRedoListSensor** is attached to this *node kit*, observing changes. It maintains pointers to **SoCnDKit** and **SoUndoRedoListKit** and triggers updates in the **callback** method:
On *slaves* and if the origin of the detected update is the **SoUndoRedoListKit**, **undoRedo** of **SoCnDKit** is called. This method conducts undo/redo operations, inspects the index pointing to the last successful executed undo/redo operation and determines the direction of adjustment to the updated index. Undo/redo operations are performed step by step, calling **undo** or **redo**, as appropriate and monitoring success and failure: The private index pointer is updated every time, after an atomic operation is actually carried out. An unsuccessful attempt is characterized by returning **false** without altering anything. In this case, the loop is prematurely terminated, waiting patiently for the next distribution update to retry also the remaining pending operations.
Apart from this important undo/redo execution, the same *sensor* is used to check, if undo/redo is possible in the current instant, setting availability *field* information properly to give visual feedback to the user. This minor task has to be performed independently of the *master-slave* property.

4.4.5 Personal Interaction Panel

As mentioned before, the panel is loaded from **pipSheet.iv**. This *OIV* file references other subfiles, performing further structuring. For instance, **pipLayout.iv** defines all *PIP sheet* geometry, while **routings.iv** adds a bunch of *engines*, used to define and constrain interaction possibilities.
The statically defined interface behavior, residing in **routings.iv** and implemented by *engines*, includes (for details see figure 4.1):

- simulating drop-down menus by displaying submenus according to the toggle state of its parent menu button,
- simulating menu bar behavior (*File*, *View* and *Edit* menu bar as well as the *Transform*, *Intersect*, *Construct*, *2D* and *3D* menu bar) by constraining that at any instant only a single submenu is exclusively visible,
- simulating scrolling through a set of widgets by arrow buttons (*Layers* widget group),

- simulating radiobutton behavior in active layer selection buttons (*Layers* widget group),
- constraining interaction between layer visibility and active layer selection, so that an active layer cannot be made invisible and, on the other hand, activating a layer implies turning its visibility on.

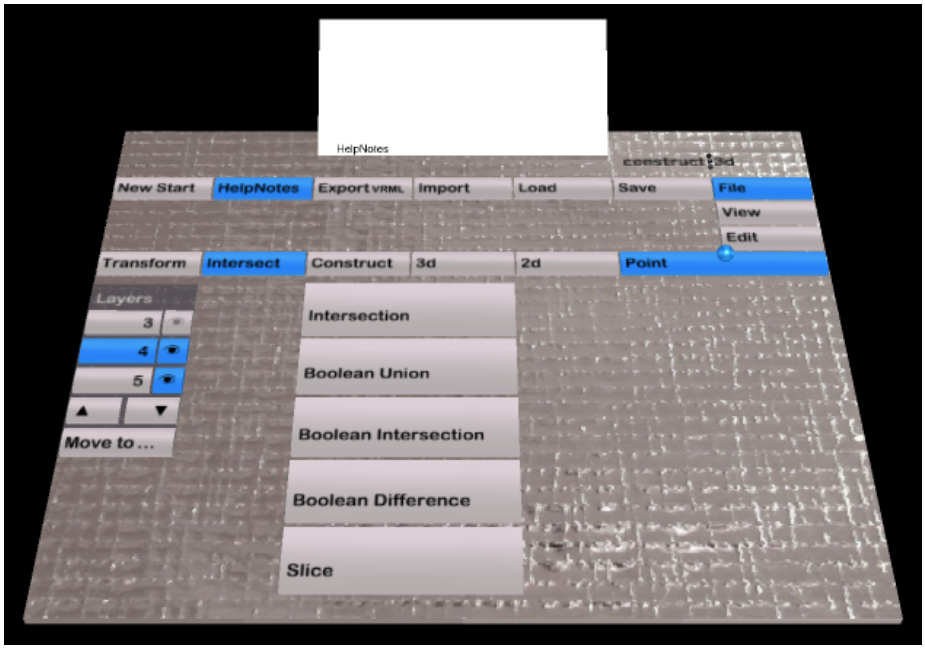

Figure 4.1: Interface panel in *Construct3D*

The interface behavior is implemented statically in *OIV* file. Application code focuses solely on interfacing with the panels, not needing to pay attention to the interaction logic. So, each panel is a self-contained system, being completely independent of code in terms of performing and constraining user interaction functionality. The strict separation of interaction logic and interface between application and panel is proven to be very useful in distribution.
To visually distinguish each user's panel, a color theme is applied to each of them. Hence, **pipSheet.iv** does appearance altering performed by applying characteristic *material* colors and *textures* (as shown in figure 3.13). As each user is given the same set of *PIP sheet* geometry definitions with all possible color schemes, selecting the proper appearance is done in a similar manner to ordinary *switches* in *OIV*. In contrast to them, the index of the *child* to render is not specified as *field* located within the *node*, but defined elsewhere in the *scene graph* by utilizing *context sensitive scene graph traversal* [RS05] functionality: A **SoContext** instance in the *scene database* defines the index of the *child* to render (via *context* property **"Stb.Owner"**), when traversing the **SoContextSwitch** containing all *PIP sheets*.

4.4.6 Dynamic user management and master-slave property

Studierstube supports users dynamically joining and leaving the workspace. So *Construct3D* has to handle the arrival of late joining users. The number of users in a single *Studierstube* instance can be freely defined.
Each *PIP* assigned to a user is automatically activated in the *scene graph* by *context sensitive* functionality described earlier. *Construct3D* has to do setup on all emerging panels, including creation of *field connections*, attaching *sensors* and registering callbacks depending on the *master-slave* property. As mentioned before, **checkPipMasterMode** is responsible for this task.
The majority of application operations are related to a certain user and displayed in his own color theme. Each *user id* is uniquely associated to a theme. This implies a *field* in **SoCommandKit** storing the associated *user id*. Recalling table 4.5, all settings privately defined for each user are basically of **SoMField** type. This grants access to user settings by indexing with the *user id*, while global settings to be shared among all users are of **SoSField** type.
Distribution features of *Studierstube* automatically assign *master-slave* property defined by the *session manager*. Terminating the *master* results in transfering this property to one of the remaining *slaves*. This property change is also handled in **checkPipMasterMode** (for *PIP sheet* related tasks) and **setMasterMode**.

4.4.7 Slave specifics

Roughly speaking, a *master* behaves very similar to a stand-alone application instance. As depicted earlier, it has only to ensure that distribution can succesfully take place by creating an appropriate *scene graph* and implementing custom *nodes* properly. Besides, it does not actually take care of the fact, if distribution is actually enabled or stand-alone operation mode is on.
In contrast to that, the functionality of a *slave* is in several aspects different. Apart from the requirement to inspect undo/redo list changes and react properly, they have to pay also attention to other issues: On *slaves*, all operations causing transient and permanent application state changes are prohibited.
Interfacing with any of the widgets in the *PIP* causes not a single effect in the *slave* application instance. In contrast to a *master*, the majority of callbacks are simply not activated.
Executing operations on the *master* virtually creates the illusion of being able to directly perform interface operations in the local application instance. *Tracking* data distribution is responsible for this behavior. Furthermore, visual feedback of application state is caused solely by *DIV* distribution. Uncritical interface states are excluded from this.
In move operations, local modifications are allowed until the instant, when the manipulation is going to be persistent. At this moment, the previously saved position is recalled. *DIV* updates the actual as well as the saved position. This makes the resetting mechanism completely independent of the processing order of *DIV* and *OpenTracker* data updates.
Also autoloading of a startup file (**loadOnStart.iv**) is disabled on *slaves*. Instead, the distributed undo/redo list is executed, which includes the contents of the startup file previously read on the *master*.

The *master-slave* property is obtained by the *context* of *Studierstube*. An associated member variable of **SoContextKit** is accessible via method **getMasterMode**. The proper *context* of any *node* can be determined by **getAnyContextFromNode** in **SoContextManagerKit**. As **SoCnDKit** is derived from **SoContextKit**, this is not necessary from code within this class.

Chapter 5

Conclusion

In application of all previously presented implementation efforts, some results could be gained. This work is concluded by giving an outlook to future improvements.

5.1 Results

Running *DIV* on *TCP* offers for the first time long distance distribution possibilities without the effort for tunnelling or relying on special infrastructure (*MBONE* [Eri94]).
Platfrom independence is no problem, as long as a port for any involved software framework exists. *Coin3D*[1], the *OIV* implementation from *Systems in Motion*, is available for *Microsoft® Windows®*, a wide range of *UNIX*, *Linux* and *BSD* platforms as well as for *Mac OS X* (from *Apple*). Several *GUI* bindings plug seamlessly into *Coin3D*: *Microsoft® Windows®*, *Xt/Motif* and native *GUI* for *Mac OS X* as well as cross-platform *Qt*[2] (from *Trolltech*) are supported. Utilizing *ACE*[3], ported also on a wide range of platforms, *DIV* is also capable of running on many *operating systems*. All these software frameworks are implemented in *C++*.

5.1.1 Distributed Open Inventor

Plain *DIV* was tested on *Microsoft® Windows®* machines at *Vienna University of Technology* and at *Graz University of Technology* hosting *Linux* computers.
The test setup included a simple viewer application. This application allows distribution of the contents of an *OIV* file by utilizing a *DivGroup*. Program arguments thoroughly specify distribution configuration options: The *master-slave* property is set, the network layer can be specified as well as associated network configuration data. *Scene graph* content is transferred by using the *node transfer* feature.
Figure 5.1 shows a typical test case, distributing a quite complex 3D model

[1] *http://www.coin3d.org/*
[2] *http://www.trolltech.com/products/qt/index.html*
[3] *http://www.cs.wustl.edu/~schmidt/ACE.html*

with simple interaction functionality across different platforms with individual viewpoint selection. *DIV* is running on established *TCP* connections between Vienna and Graz. Depending on the amount and complexity of *scene graph* data, initial *node transfer* takes some time. But after this initialization state, interaction is fast and responsive.

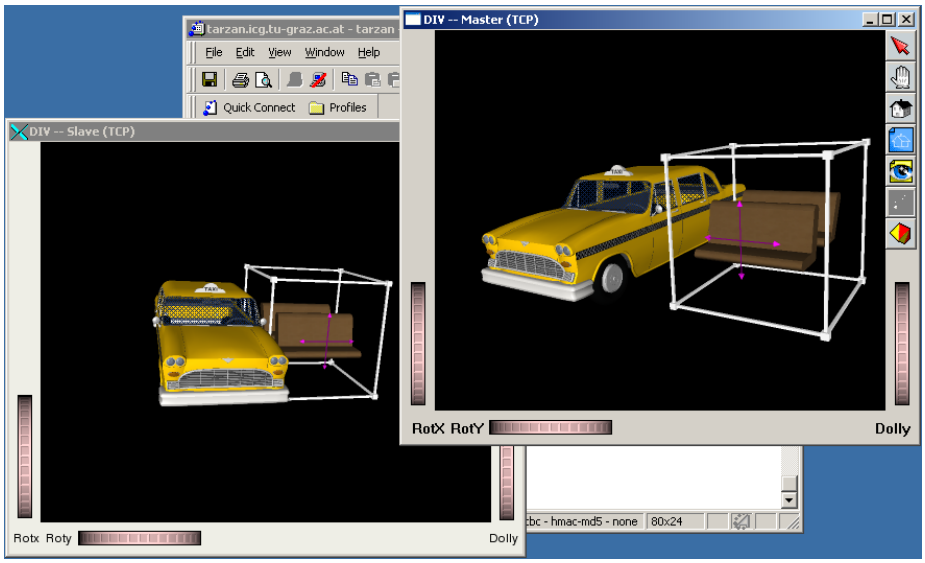

Figure 5.1: Cross-platform, long distance *DIV* distribution using *TCP*: *Master* on top right and remote *slave* on bottom left

Comparing the *multicast UDP* and *TCP* implementation, it is easily observable that *TCP* performance is basically overtopping *multicast UDP*, especially in small networks: Generating huge amounts of *DIV* updates by heavily manipulating the *scene graph* contents with *draggers*, network data throughput in the *TCP* implementation seems to be much better. On *multicast UDP*, the send queue gets full comparatively fast, which causes the render thread blocking the *master*. Consequently, interactive manipulation is not possible while having the render thread waiting for dequeuing to take place. Maintaining the same conditions (queue size) while running these massive stress tests, this blocking phenomenon could not be achieved on *TCP*. This congestion behavior is already caused in the buffering queue used for a single purpose: synchronizing data packets between rendering and network thread. Data packets are fetched from this queue by the *TCP* network thread obviously faster than in the *multicast UDP* implementation. This leads to the assumption that the add-on library to *ACE*, guaranteeing *reliable multicast UDP* is sub-optimal in terms of performance and data throughput. *TCP* and its inherent *reliability* feature seems to be the better choice even for local networks, offering the possibility to run *multicast UDP* directly.
Of course, more than two participating sites can be involved in distribution. Further cross-platform tests included more computer systems and up to 5 application instances. Participating sites were dynamically joining and leaving the distribution network, application instance start order is irrelevant. Deficiencies in performance and responsiveness were not observable.

Hybrid networking was also tested, interconnecting *network* fragments running *TCP* and *multicast UDP*. Due to its early stage of development, attention had to be paid to avoid any *network redundancy*.
As reworked *Distributed Open Inventor* was proven to be stable, its functionality will be included as an add-on patch in the next release of *Coin3D*.

5.1.2 Distribution within Studierstube

Studierstube[4] is also a cross-platform framework, primarily running on *Microsoft*® *Windows*® and *Linux* machines.
First tests were performed in a scenario with a simple painting application, shown in figure 2.6. This application bases on *DIV*, distributing *PIP sheet* configuration options and object modifications caused by painting actions. *Slaves* are completely passive, visualizing interaction results of the *master*. So, no multi-user interaction mode is implemented in this test case and *tracking* data distribution is not necessary. However, it is enabled for cross-platform testing, as illustrated in figure 5.2.

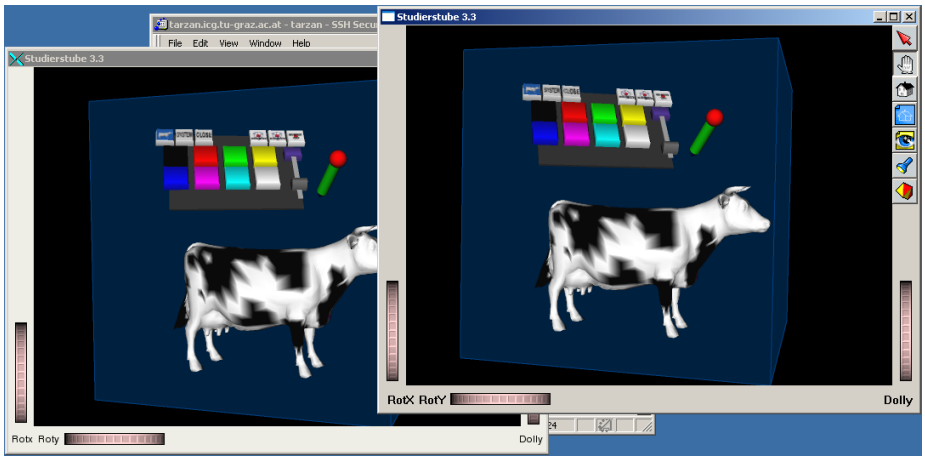

Figure 5.2: Cross-platform, long distance *DIV* (using *TCP*) and *tracking* data (using *unicast UDP*) distribution in *Studierstube*: *Master* (top right) and remote *slave* (bottom left)

Similar to plain *Distributed Open Inventor* tests, further test cases involved several computers and more *Studierstube* instances. Since networking was effectively decoupled from rendering in *DIV*, each application instance seems to contain robustness to a very high degree. So network latencies were not able to cause an additional decrease of performance on other participants of the distribution system.

5.1.3 Distributed Construct3D

Construct3D is finally fully benefiting from all distribution features: Multi-user functionality raises the demand for *tracking* data distribution. Supporting

[4] *http://www.studierstube.org/*

master transfer ability is desired for more flexible use cases.
Generally speaking, a very high degree of flexibility is ensured by three orthogonal aspects:

- User configuration and its associated ressources (output devices, panels and pens) can be freely specified.
- The participating site originally hosting the application (*DIV master*) can be selected without restriction and is completely independent of associated users and their ressources. Startup order is completely insignificant and the *master* automatically migrates by *session management* on termination.
- Finally, by configuring *OpenTracker* properly, *tracking* data distribution, is independent of all other aspects.

This means each *Construct3D* instance can be configured in multiple ways by defining the number of users, its associated ressources, specifying application retrieval method (by distribution as *slave* or by file input as *master*) and *tracking* data obtaining strategy:

- As already mentioned in section 3.4, some kind of central and persistent *Construct3D* service can be established without any active user and rendering output associated. In contrast to this, dynamically migrating *Construct3D* application hosts with actively collaborating users directly associated, is also easily possible. The configuration effort is not difficult, as contacting the *session manager* performs all bootstrapping.
- User configuration is quite unrestrictive: It suffices that each user has a unique *user id*, although selecting sequential numbers starting from 0, is recommended. (In addition to that, a color theme has to be present.)
- As *OpenTracker* data distribution network is completely independent of the *Distributed Open Inventor* network, a separate stand-alone *tracking* server, even without *Studierstube* running, can be realized, distributing *tracking* data to all participants. On the other hand, multiple *tracking* data distributors, whether stand-alone *OpenTracker* instances or integrated into *Studierstube*, can be deployed.

Figure 5.3 shows three distributed *Construct3D* instances running on desktop setup. In this non-immersive setup, each of the two actively collaborating users is associated with one *Studierstube* instance. The remaining instance on the top belongs to a passive user without associated interaction devices.
In this test case, it became clearly evident that geometric recalculation was the limiting factor of performance. Obviously the goal of all efforts was achieved: Distribution of application commands is performed very quickly, but geometry updates can take some time, if complex geometry dependencies and operations are pending. In favor of unburdening the *master*, each participating site became autonomous in performing geometric recalculation. This is applied decentralization in terms of geometric operations.

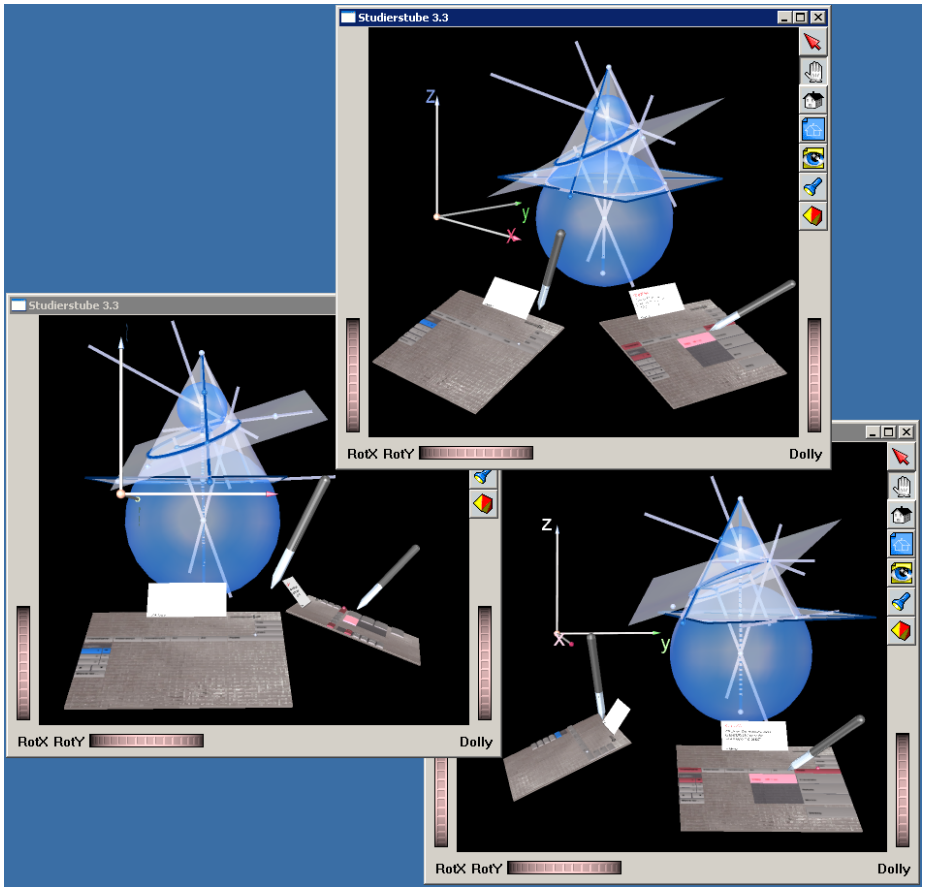

Figure 5.3: Long distance *DIV* (using *TCP*) and *tracking* data (using *unicast UDP*) distribution in *Studierstube* running *Construct3D*: The *master* (associated with first user in blue color theme) is on bottom left, a *slave* (associated with second user in red color theme) is displayed on bottom right and an additional *slave* on top (a passive viewer) is also present.

5.2 Future work

5.2.1 Network connection establishment issues

As described in section 3.2, *TCP* connections to a newly arrived *peer* are actively established by all but one other *peers* forming the network at this instant. (A single *peer* is excluded, as a connection from the initially contacted *peer* already persists.) This can cause trouble, when the *peer* is located behind a *firewall*. A mechanism to notify the *peer* of all network participants, to whom connections could not be established, would resolve this problem.
Currently, even if *Studierstube* is combined with the *session manager*, this problem persists with late joining participants located behind a *firewall*. Although the *session manager* notifies all other participants indirectly by reconfiguring the list of potential initial *peers*, this causes no reestablishment of *TCP* connections, if they are already part of a distribution network. The list of all *peers* is also transferred to the late joining participant, but according to its connection establishment strategy, no further action is performed, if a *TCP* connection could be successfully established.

5.2.2 Hybrid networks

As stated in section 3.2, *hybrid networking* should be extended to run a protocol making use of potential *redundancy* in *network topology*. The purpose of this protocol is to detect *network redundancy* in advance and react properly by determining those *network nodes*, where *bridging* functionality has to be disabled to prevent packets from endlessly travelling in the network. On the other hand, disabled *bridging* functionality has to be immediately reactivated, if the *network* is separated under any circumstances. With other words, the protocol has to ensure that at any instant the network conforms to *spanning tree* property in actual network traffic *data flow*.
This would put *hybrid networking* out of its experimental stage and could be integrated in *Studierstube* to allow *hybrid networking* for more effective distribution capabilities. *Hybrid networks* are expected to be suitable for common application scenarios: Running *multicast UDP* on local networks of participating organizations (schools, universities, ...) is usually possible, but several of these networks have to be interconnected by *TCP*. Choosing heterogenous network protocols helps keeping the number of *TCP* connections as low as possible, especially in large-scale environments.

5.2.3 Large-scale and extensive performance tests

Up to now, distribution functionality was tested only for a relatively small number of participating sites (about 5). In the future, large-scale and comprehensive *scalability* tests should take place, paying also attention to performance and data throughput. Bottlenecks that might arise, could be possibly prevented by reconsidering the *network topology*, as this has great influence on *scalability* issues. Ideally, these tests should be coupled with improved *hybrid network* support, as a result of the capability to treat network ressources more efficiently.

Bibliography

[Azu97] Ronald T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, aug 1997.

[BK99] Mark Billinghurst and Hirokazu Kato. Collaborative mixed reality. In *Proceedings of International Symposium on Mixed Reality (ISMR '99). Mixed Reality–Merging Real and Virtual Worlds*, pages 261–284, 1999.

[CD88] George F. Coulouris and Jean Dollimore. *Distributed systems: concepts and design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[Eri94] Hans Eriksson. MBONE: the multicast backbone. *Commun. ACM*, 37(8):54–60, 1994.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing.* MIT Press, Cambridge, MA, USA, 1994.

[Hay98] Mark G. Hayden. *The ensemble system.* PhD thesis, Cornell University, 1998.

[Hes01] Gerd Hesina. *Distributed Collaborative Augmented Reality.* PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2001.

[KSDG05] Hannes Kaufmann, Karin Steinbügl, Andreas Dünser, and Judith Glück. General training of spatial abilities by geometry education in augmented reality. *Annual Review of CyberTherapy and Telemedicine: A Decade of VR*, 3:65–76, 2005.

[KSW00] Hannes Kaufmann, Dieter Schmalstieg, and Michael Wagner. Construct3D: A Virtual Reality Application for Mathematics and Geometry Education. *Education and Information Technologies*, 5(4):263–276, 2000.

[MF98] Blair MacIntyre and Steven Feiner. A distributed 3D graphics library. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 361–370, New York, NY, USA, 1998. ACM Press.

[MK94] Paul Milgram and Fumio Kishino. A taxonomy of mixed reality visual displays. *IEICE Trans. Information Systems*, E77-D(12):1321–1329, dec 1994.

[NLSG03] Martin Naef, Edouard Lamboray, Oliver Staadt, and Markus Gross. The blue-c distributed scene graph. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, pages 125–133, New York, NY, USA, 2003. ACM Press.

[Peč02] Jan Pečiva. Development of a distributed scene toolkit based on open inventor. Master's thesis, Department of Computer Graphics and Multimedia, Brno University of Technology, 2002.

[Rhe91] Howard Rheingold. *Virtual reality*. Simon & Schuster, Inc., New York, NY, USA, 1991.

[RS01] Gerhard Reitmayr and Dieter Schmalstieg. An open software architecture for virtual reality interaction. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 47–54, New York, NY, USA, 2001. ACM Press.

[RS05] Gerhard Reitmayr and Dieter Schmalstieg. Flexible Parametrization of Scene Graphs. In *VR '05: Proceedings of the 2005 IEEE Conference 2005 on Virtual Reality*, pages 51–58, Washington, DC, USA, 2005. IEEE Computer Society.

[SC92] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 341–349, New York, NY, USA, 1992. ACM Press.

[Sch94] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. Technical report, Comp. Science Department, Washington University, 1994.

[SFH+02] Dieter Schmalstieg, Anton Fuhrmann, Gerd Hesina, Zsolt Szalavári, L. Miguel Encarnação, Michael Gervautz, and Werner Purgathofer. The Studierstube augmented reality project. *Presence: Teleoperators and Virtual Environments*, 11(1):33–54, 2002.

[SG96] Zsolt Szalavári and Michael Gervautz. The Personal Interaction Panel - a two-handed interface for augmented reality. Technical Report TR-186-2-96-20, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, sep 1996.

[SRH03] Dieter Schmalstieg, Gerhard Reitmayr, and Gerd Hesina. Distributed applications for collaborative three-dimensional workspaces. *Presence: Teleoperators and Virtual Environments*, 12(1):52–67, 2003.

[Str93] Paul S. Strauss. IRIS Inventor, a 3D graphics toolkit. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 192–200, New York, NY, USA, 1993. ACM Press.

[THS+01] Russell M. Taylor, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T. Helser. VRPN: a device-independent, network-transparent VR peripheral system. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 55–61, New York, NY, USA, 2001. ACM Press.

[Tra99] Henrik Tramberend. Avocado: A Distributed Virtual Reality Framework. In *VR '99: Proceedings of the IEEE Virtual Reality*, page 14, Washington, DC, USA, 1999. IEEE Computer Society.

[Wer93] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.