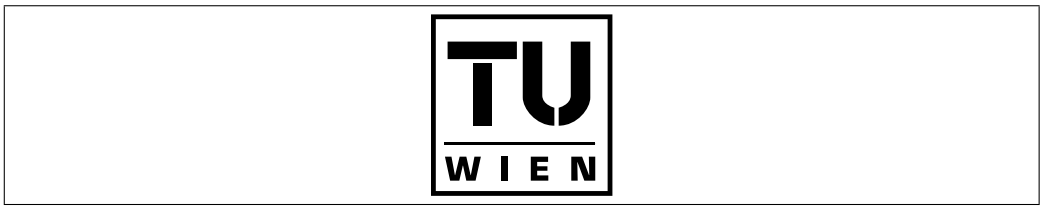


Dissertation

On Software Design for Augmented Reality

ausgeführt
zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften

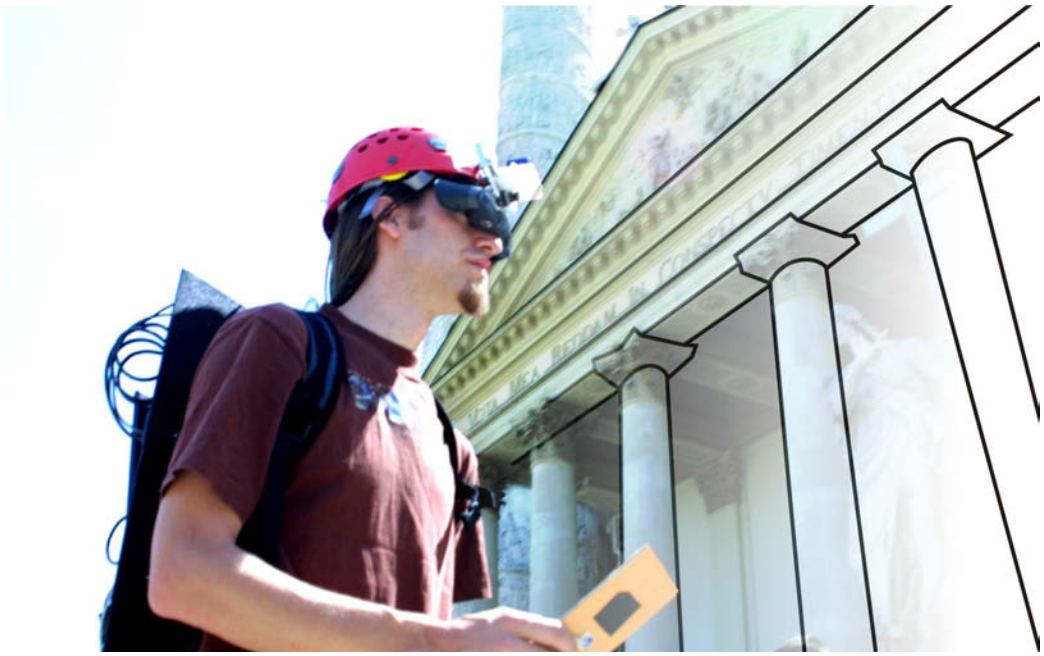
unter der Leitung von
Ao. Prof. Dipl.-Ing. Dr. Dieter Schmalstieg
Institut 188 für Software Technologie und Interaktive Systeme

eingereicht
an der Technischen Universität Wien
Fakultät für Informatik

von
Dipl.-Ing. Gerhard Reitmayr
Bleichergasse 13/9
1090 Wien
Matr.-Nr. 9325167

Wien, im März 2004

On Software Design for Augmented Reality



Gerhard Reitmayr – Dissertation

reviewers:

Dieter Schmalstieg

Gudrun Klinker

Abstract

Augmented reality (AR) is an intriguing user interface technique that combines the properties of the real world with information processed by a computer. By augmenting the experience of the real world with computer-generated sensations, e.g. through the use of head mounted displays with optical combiners, virtual information can behave like a real object and users can leverage their knowledge of the real world to interact with that information.

The development of augmented reality systems encompasses a large number of areas in computer science. Sensor technology is required to measure the state of the real world such as the position and gaze direction of a user. Advanced computer graphics are necessary to render convincing images of virtual information. Mobile applications with a focus on location-based information require large data sets to adequately model the large area they are deployed in. Finally, collaborative applications are implemented using distributed systems to support several users. Therefore, a comprehensive AR system needs to address all of these aspects and build upon a scalable design that combines the individual areas.

The topic of this dissertation is a set of designs each of which address a part of an augmented reality system that combines collaborative and mobile applications. The individual designs focus on flexibility and high-level programmability to allow rapid incremental development of applications. The combination of high-level configurations with optimized implementations can provide the required performance while being flexible and extensible. The areas that have been addressed are configuration and manipulation of tracking inputs; flexible scene graph management combining visual and semantic model properties; a three-tier architecture managing and using large data sets with a data-driven application design; flexible session and space management for complex collaborative applications.

The individual designs are implemented as part of the Studierstube framework and are demonstrated by a set of applications which focus on mobile augmented reality systems that provide location-based navigation aids and information displays to the user. These applications are based on a mobile AR setup developed for indoor and outdoor use. The setup is also shown in a set of collaborative applications supporting multiple mobile users.

The combination of these designs allows for rapid development of flexible and scalable applications. A generic software architecture is presented to illustrate the optimal combination of the individual solutions into a coherent application. The architecture is demonstrated by a tourist guide system which incorporates mobile and collaborative aspects.

Kurzfassung

Augmented Reality (AR) – erweiterte Realität – ist eine User Interface-Technik, welche die Wirklichkeit mit computergenerierten Informationen verbindet. Die Wirklichkeit wird durch den Computer erweitert, z.B. durch Head Mounted Displays mit halbdurchlässigen Spiegeln. Virtuelle Information verhält sich nun wie ein reales Objekt, und der Benutzer kann auf natürliche Weise mit ihr interagieren.

Die Entwicklung von AR-Systemen verbindet verschiedenste Gebiete der Informatik. Sensoren messen Ereignisse in der Wirklichkeit, z.B. die Position und Blickrichtung eines Benutzers. Anspruchsvolle Computergrafik wird benötigt, um glaubwürdige Bilder von virtuellen Informationen zu erzeugen. Mobile Anwendungen bauen auf positionsabhängigen Informationen auf und verwenden daher große Datenmengen, um ein interessantes Gebiet abdecken zu können. Kollaborative Anwendungen setzten verteilte Systeme zur Realisierung ein um auf einfache Weise mehrere Benutzer bedienen zu können. Ein umfassendes AR-System muss alle diese Aspekte vereinen.

Das Thema der vorliegenden Arbeit ist eine Reihe von Software-Designs, welche einzelne Bereiche eines mobilen und kollaborativen AR-Systems abdecken. Die einzelnen Designs streben eine flexible und einfache Programmierung an, um rasche Iterationen bei der Entwicklung von Applikationen zu erlauben. Durch die Kombination von einfachen Konfigurationen mit optimierten Implementation wird die erforderliche Performance ermöglicht, ohne die Flexibilität einzuschränken. Die einzelnen Bereiche sind folgende: die Konfiguration und Verarbeitung von Messdaten von Trackinggeräten; eine flexible Szenegraphen-Architektur, welche visuelle und semantische Aspekte eines Modells verbindet; eine Three-Tier-Architektur für die Verwaltung von großen Datenmengen; ein flexibles Session- und Raum-Management für komplexe kollaborative Anwendungen.

Jedes der Designs wurde als Teil des Studierstube Frameworks implementiert und wird durch einige Anwendungen veranschaulich, welche aus dem Bereich ortsabhängiger Navigations- und Informationsvisualisierung stammen. Ein mobiles AR-System für Indoor- und Outdooreinsatz wurde als Basis dieser Applikationen entwickelt und wird auch in mehreren kollaborativen Anwendungen für mobile Benutzer eingesetzt.

Die entwickelten Lösungen erlauben die schnelle Erstellung von flexiblen und skalierbaren Anwendungen. Eine allgemeine Softwarearchitektur wird beschrieben, welche die optimale Kombination der einzelnen Designs zu einer umfassenden Anwendung erleichtert. Diese Architektur wird schließlich eingesetzt um eine Touristenführer-Anwendung zu implementieren, welche mobile und kollaborative Aspekte vereint.

Acknowledgements

This dissertation would not have been possible without the help of many people: I want to thank foremost my advisor Dieter Schmalstieg for the constant support and guidance during the development of this work. He always suggested new ideas to explore and taught me the principles of scientific work. I also would like to thank Christian Breiteneder for creating a stimulating working environment and being a model of scientific scrutiny.

Many thanks go to past and present members of the virtual reality research group at Vienna University of Technology for extensive discussions and general help with whatever would come up: Istvan Barakonyi, Tamer Fahmy, Anton Fuhrmann, Gerd Hesina, Hannes Kaufmann, Karin Kosina, Florian Ledermann, Joseph Newman, Thomas Pintaric, Jan Prikryl, Thomas Psik, Rainer Splechtna and Daniel Wagner.

The applications described throughout this dissertation are also the result of the dedicated work of our students. Thanks go to Ivan Viola and Matej Mlejnek for their work on the first iteration of the mobile augmented reality setup; to Thomas Lidy and Michael Kalkusch for their work on the original Signpost implementation; and special thanks to Michael Knapp for his outstanding work on both the first and the final Signpost 2 applications.

I also want to thank the WG Bleichergasse and all the members who lived there at some point in time: Ali, Manfred, Markus and Thomas. It was always a great place to live, work and party! Special thanks to Alexandra who had endless patience when I worked through yet another weekend instead of spending more time with her. Finally I'm very grateful to my family who at all times supported my interests in computer science and mathematics and enabled me to reach this point.

Part of this research was supported by the Austrian Science Fund (FWF) contract no. P14470 and Y193, and Vienna University of Technology by Forschungsinfrastrukturvorhaben TUWP16/2002.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Augmented Reality – a ubiquitous user interface

The user interface technique of augmented reality is based on the intriguing idea of merging the presentation of the human computer interface with the real world. Augmented reality (AR) seeks to establish a more natural user interface by giving abstract information properties of the real world or associating it with phenomena encountered within the real world such as space and time. In doing so, AR blurs the distinction between the real world and the user interface in a way similar to the ideas of ubiquitous computing as described by Weiser [115]. While ubiquitous computing focusses on the computer becoming invisible among the objects of everyday life, augmented reality seeks to add to the experience of reality to create new forms of interaction between humans and computers.

The main idea of augmented reality is to superimpose or mix computer generated sensations with stimuli generated by the real world. The distinguishing properties as defined by Azuma [7] are the combination of the real and virtual, real time interaction and registration in 3D. Typically human sense which is augmented is vision as appropriate output devices and techniques are well established. A see-through device that mixes the real image with a computer generated virtual image allows a computer generated image to be superimposed over the user's perception of the real world. As the rendered augmentation is registered in 3D and the display is updated at interactive frame rates, the user will perceive the augmentation in the same way as a real object.

An important part of any AR user interface is 3D interaction. Humans know how to interact with real objects, how to handle and manipulate them.

The augmentation of the real world with artificial objects tries to leverage that knowledge and extend it to the artificial information objects.

Combining mobile (computing using wearable computer setups) with augmented reality can result in a 3D information space around the user [101]. With a global infrastructure to provide location-based information the world itself becomes an interface to a second, overlaid world of information [100].

An important direction of research in augmented reality is the support of collaborative activities. AR can provide a seamless environment for several users simultaneously, thus providing unobtrusive support to their tasks.

1.2 Software design for AR applications

Developing augmented reality applications is still a challenging task, even after years of research and numerous application demonstrators built. Besides the limiting constraints of tracking and display technology, the software complexity involved in producing a convincing application is significant. The difficulties arise directly from the basic properties of augmented reality as defined by Azuma [7]:

1. Combines real and virtual
2. Interactive in real time
3. Registered in 3D

The first property has some impact on the choice of technology to establish a model of the real world. Sensors are deployed to measure individual properties of the real world to create this model. As more accuracy is needed and as more and complex sensors are necessary, the amount of data requiring processing is increasing. Applications require the synthesis of high-level commands and interactions from the raw tracking data by a series of processing steps. Sensor devices also add another level of complexity to the overall system. Hardware devices that require configuration, permanent operation and also raise additional constraints such as hardware interfaces, device drivers for operating systems or distribution of tracking data over several hosts.

The third property not only demands 3D tracking systems, but often leads to 3D output modalities as well, typically 3D graphics. Despite ever increasing graphics throughput, high-quality rendering and realistic effects are still an active area of research and development and can easily bring the most powerful workstation to its knees.

Finally, all of these complexities are aggravated by the second property, real-time interactivity. This property requires solutions that work within short time intervals to provide the necessary minimal frame rate to the user. Even batch processing of input data and computation is often not an option, because during the time it takes to evaluate the data, the user might have changed her opinion about it.

Furthermore, the use of augmented reality in mobile computing applications requires that the software design scales well with respect to the amount of data that needs to be processed and presented. Mobility implies that the user is potentially interested in a large working area and information based on locations within that area.

Collaborative applications require distributed systems that scale well with the number of users. Stationary collaborative applications for same-space work are usually limited in the number of users and therefore do not require complex user and session management. However, the use of AR with mobile systems leads to a potentially large number of participating users who want to collaborate and interact with each other. The sessions of users working together will also be very dynamic as users move about their environment and join or leave existing working groups.

Real mobile collaborative applications in AR will require scalability in both the volume of data and the number of participants in a distributed system. The complexity of these requirements in the context of the basic properties of AR as defined above are non-trivial. Therefore, a good software design will balance the trade-offs between different requirements within an AR application to create a satisfying system.

Additionally, a good software design will try to achieve reusability and independence between application subcomponents. Also, software processes that are well suited to the development of interactive applications require an iterated process to include feedback from the end-users as early and frequently as possible. AR user interfaces are still a new topic and subject much experimentation. Thus, both aspects of developing AR applications require a flexible design that allows testing of a large number of variations within the usually limited time available.

Different areas of changeability can be identified. Tracking devices are replaced by new ones; their location in a room may change and require recalibration; different user interfaces may require new combinations of input devices. Therefore, changes to the tracking setup should be transparent to and independent of the remaining parts of an application to support iterative development.

Development of the graphical user interface will also require frequent changes to test different variations. If the presentation of information and

the interaction therewith can be changed without interfering with the remaining application, the system will provide a more flexible development environment.

Finally, it is certainly desirable to develop application functionality independently of the final user interface or tracking modalities used. Such an approach furthers reuse of the core functions of an application and can lead to a modular approach of building new applications from existing functional blocks. In summary, an efficient development system for augmented reality applications should support experimental and exploratory styles of programming that use an iterative and prototype-based development process.

1.3 Problem statement

All the typical software components in augmented reality systems have been subject to extensive research during the last few years. The work has resulted in a number of frameworks that encapsulate algorithms and implementation expertise in various specialized problems in code, ready for the developer to reuse. However, these frameworks typically provide the developer with a set of library calls or objects that implement the specific algorithms but without illuminating the overall application design. The fine-grained functionality provided by an application programming interface usually requires a large management overhead on the part of the application programmer and therefore still allows inefficient implementations.

The level of abstraction within the frameworks is not sufficiently high to support the exploratory and iterative development process that would be ideal for AR applications. A large number of existing frameworks are based on languages that require compilation and therefore already oppose frequent changes. Others employ interpreted languages but the disatvantages of too fine-grained APIs persist.

Additionally, some software frameworks provide high-level architectural patterns such as means of inter-component communications that are too general to direct the developer towards a working design. The problem is that the trade-offs associated with applying the general architecture to the application are often non-obvious. For example, a generic inter-component communication facility allows the implementation of every single software object as a full fledged component at the cost of increased communication overhead. Aggregating the functionality into appropriate components to improve performance while retaining modularity is usually non-trivial.

1.4 Contribution

The contribution of this dissertation is a set of software designs for recurring subcomponents in augmented reality applications. The designs focus on scalability and performance while providing a high-level configuration approach. The implementations demonstrated offer a declarative programming style for the individual components which permits exploratory and iterative development and simplifies the use of automated tools to generate configurations.

Additionally, it addresses the complex development process of an AR application by providing a design guideline for developing applications within the context of the developed components. Such a guideline can lead the developer to efficient combinations of the individual reusable components. Consequently, the thesis statement is as follows:

> *The following designs for the respective components are efficient implementations which offer high-level programmability:*
>
> - *A pipes-and-filters architecture to implement a data flow approach to the manipulation of tracking data.*
> - *A high-level multi-dispatch design for scene graphs to combine semantic and implementation dependent structures in the scene graphs.*
> - *A data-driven three-tier architecture to provide a scalable data management approach for AR applications.*
> - *A generic session and space management system for distributed augmented reality applications.*
>
> *An integrated software design approach using these components simplifies the implementation of mobile and collaborative augmented reality applications.*

The individual subsystems developed for this dissertation deal with the following aspects of an AR application. Configuration and operations of sensor equipment and appropriate hardware abstractions within the application are described in chapter 3. An extension of the scene graph concept for integrating application and presentation data to simplify application logic and allow scalability in data complexity is presented in chapter 4. A three-tier architecture to handle large scale data sets integrating with the presentation layer is presented in chapter 5. A concept to structure the distributed application configurations of multiple users in a dynamic environment is presented in chapter 6. The true power of these developments can be delivered,

if assembled into an integrated system of patterns. The design guidelines to achieve such integration are described in chapter 7. Finally, the applicability of these guidelines are demonstrated with the example of an outdoor navigation and information browsing application in chapter 8.

The focus of this dissertation lies on augmented reality applications with the following characteristics :

- 3D information and presentation
- 3D interaction
- Location-based mobile AR systems with large databases
- Collaborative applications

Therefore, the individual chapters will present demonstration applications with these properties. While every application will use more then one component, the individual result sections will only focus on aspects relevant to the current chapter. The final tourist guide application described in chapter 8 will combine the different components in a coherent form.

The work described here was developed in the context of the Studierstube software framework for augmented reality applications and therefore extends it in various ways. Figure 1.1 shows an overview of the existing and contributed components of the Studierstube system. A more detailed description of the Studierstube framework is given in section 2.5.

The work presented here contains material previously published in:

- G. Reitmayr and D. Schmalstieg. Mobile collaborative augmented reality. In *Proceedings of the Second International Symposium on Augmented Reality 2001*, pages 114-123, New York, New York, USA, October 29-30 2001. IEEE Press.
- G. Reitmayr and D. Schmalstieg. OpenTracker - an open software architecture for reconfigurable tracking based on XML. In *Proceedings of IEEE Virtual Reality 2001*, pages 285-286, Yokohama, Japan, March 13-17 2001. IEEE Press.
- D. Schmalstieg, G. Reitmayr, and G. Hesina. Distributed applications for collaborative three-dimensional workspaces. *PRESENCE - Teleoperators and Virtual Environments*, 12(1):53-68, February 2003. MIT Press.

- G. Reitmayr and D. Schmalstieg. Location based applications for mobile augmented reality. In R. Biddle and B. Thomas, editors, *Proceedings of the Fourth Australasian User Interface Conference 2003*, volume 25 (3) of *Australian Computer Science Communications*, pages 65 - 73, Adelaide, Australia, February 4 - 7 2003. ACS Inc.

- G. Reitmayr and D. Schmalstieg. Data management strategies for mobile augmented reality. In *Proceedings of the International Workshop on Software Technology for Augmented Reality Systems 2003*, pages 47-52, Tokyo, Japan, October 7 2003.

Details of demonstrations and applications described here were also published in the following works:

- M. Kalkusch, T. Lidy, M. Knapp, G. Reitmayr, H. Kaufmann, and D. Schmalstieg. Structured visual markers for indoor pathfinding. In *Proceedings of the First IEEE International Augmented Reality Toolkit Workshop*, Darmstadt, Germany, September 30 2002. IEEE Press.

- F. Ledermann, G. Reitmayr, and D. Schmalstieg. Dynamically shared optical tracking. In *Proceedings of the First IEEE International Augmented Reality Toolkit Workshop*, Darmstadt, Germany, September 30 2002. IEEE Press.

- D. Schmalstieg, H. Kaufmann, G. Reitmayr, and F. Ledermann. Geometry education in the augmented classroom. In *Proceedings of the IEEE and ACM International Symposium on Mixed and Augmented Reality*, Darmstadt, Germany, September 30 - October 1 2002. IEEE Press.

- G. Reitmayr and D. Schmalstieg. Collaborative augmented reality for outdoor navigation and information browsing. In *Proceedings of the Symposium Location Based Services and TeleCartography - Geowissenschaftliche Mitteilungen*, volume 66, Vienna, Austria, January 28-29 2004.

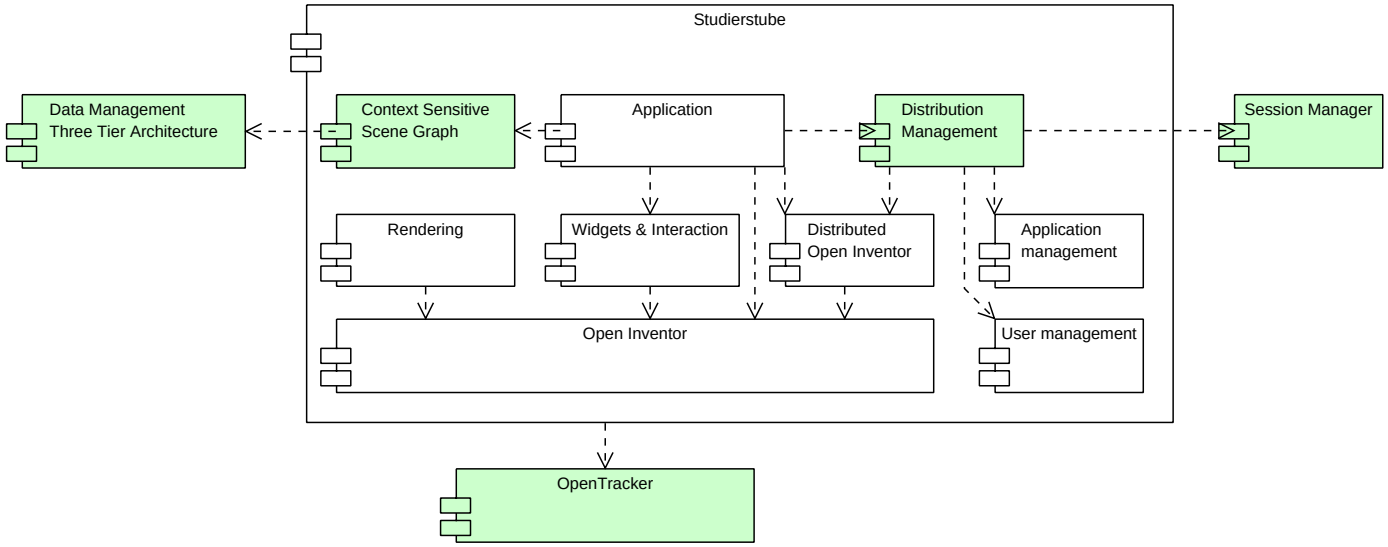


Figure 1.1: Component diagram of the Studierstube system with contributions of this work highlighted.

Chapter 2

Related work

2.1 Mobile augmented reality

Combined with mobile computing using wearable computer setups, augmented reality can create a 3D information space that lives around the user [101]. Together with a global infrastructure to provide such location-based information it uses the world itself as the interface to a second, overlaid world of information [100]. By allowing users to edit the information the world becomes an Augment-able Reality [84] that can be enhanced with digital information by all participants.

The first example of a mobile augmented reality application was the Touring machine [39, 38] demonstrating the possibilities of displaying location-based data in an augmented reality user interface. The followup developments of the Mobile Augmented Reality System (MARS) [46] and situated documentaries [48] further explored the user interface aspects of such systems and potential applications to interactive presentations for tours through a university campus. Further work on presentation and view management issues for heads-up displays [15, 45, 16] are aimed at improving the layout of information within the user's view.

The Battlefield Augmented Reality System (BARS) is a related development based on the Touring machine. First applications demonstrated constructing models with the mobile AR system [8]. However, the main focus lies on efficient information display for mobile users by selective information filtering [52], optimized display techniques [60] and finally combinations of different techniques in an adaptive system [53].

Another application area for mobile augmented reality are maintenance scenarios. Here a worker is equipped with a wearable computer that displays assembly and maintenance instructions directly into the view of the worker.

Early demonstrations involved wire bundling in aircraft manufacture [32]. Other examples involve powerplant maintenance [57] and car design [36].

Applications of augmented reality to navigation were demonstrated early on by Thomas et al. [108, 72]. The further development of the Tinmith system demonstrated the applicability to entertainment [107] and investigated the use of mobile AR for directly constructing models of real objects in place [74, 75].

Other examples of using mobile augmented reality interfaces include the Townwear system [88] which demonstrates simple information overlay for tourists in pre-defined locations. Newman et al. [67] demonstrated mobile AR applications within a sentient environment based on the Bat system [3], a wide area tracking system for buildings.

Recent developments focus on applying mobile AR interfaces to real applications to be deployed to end users. An interesting piece of work is the Geist [59, 58] project that aims to envelope users in an interactive story situated at real places throughout the historic center of Heidelberg.

2.2 Collaborative augmented reality

Early examples of collaborative augmented reality were demonstrated in the Shared space project [21] which demonstrated the use of AR for remote teleconferencing [18, 20] and support of same-space collaboration.

TransVision [82] implemented collaborative AR application for design inspection using hand-held displays that act as lenses through which the combined environment can be seen. The users view and interact with a common model in an intuitive way while being able to see each other without difficulty.

Another piece of work was EMMIE [26] that demonstrated a shared workspace enriched with virtual objects. It focused on managing the environment for different display modalities and users [25]. The combination of mobile AR and remote collaboration between a mobile user and a stationary workspace was also investigated in the MARS project [46] later.

The Tinmith project also investigated remote collaboration between mobile users and a stationary command post in a military setting [71]. Here mobile soldiers would operate in a virtual battlefield and move in the real world while their positions would be displayed in a virtual reality setting back at the command post.

The Studierstube project [91] developed by the author's research group always placed an emphasis on supporting collaboration in shared augmented reality workspaces [92]. Based on an underlying distribution mechanism [44]

Studierstube extends its support to multiple users working with multiple different display techniques in a shared workspace that features multiple applications and management techniques similar to a common 2D desktop [90]. Refer to section 2.5 for a more detailed description.

2.3 Software frameworks for augmented reality

Several research groups having developed a series of augmented reality demonstrators quickly discovered that a software design supporting development increases software reuse and allows quicker iteration of their explorations of possible AR user interfaces.

Coterie The Coterie system [61] is based on Modula3 and supports scene graph based graphics programming and abstractions for tracking devices. It formed the basis for the aforementioned Touring machine and MARS projects. A later extension allowed for networked objects [62] to simplify the implementation of distributed AR applications. The support for distribution was, however, not completely transparent requiring the implementation of various callback functions on the application side. Nevertheless, complex data structures such as scene graphs could be easily shared .

ARToolkit The ARToolkit library [54] is an example of a minimal AR framework. It provides a tracking solution based on fiducial markers and rendering simple graphics using OpenGL on top of the video stream used for tracking. The typical application template only consists of a main loop iterating over the tracking and rendering functions. Despite its simplicity the library has been widely adopted and a large number of AR applications based on ARToolkit have been developed.

Avango is a framework for developing virtual and augmented reality application that also supports transparent distribution [110]. It is based on SGI Performer and extends it with a more powerful scene graph, data fields for objects and a Scheme language binding. By extending a mechanism to connect fields to route data flow between them to support transparent networked operation, distributed applications can be developed that implicitly share data between different instances.

Tinmith The Tinmith system is a full featured software architecture for mobile augmented reality applications [76]. Originating from an older architecture that was built from a network of communicating agent processes [71, 77], it developed into an object-oriented software framework supporting hierarchical scene graph based modelling and generic data flows between objects [73]. Implemented in C++, it features an extensive runtime support system with type information, serialization and persistent storage to the file system.

Rendering support is included via a set of objects that can be composed into a scene graph structure to create geometric models. Tinmith supports advanced modelling features such as CSG operations to simplify the construction of real buildings with a mobile AR system.

DWARF The DWARF project [11] aims for a design concept that differs greatly from traditional AR software designs. The basic units of the DWARF framework are distributed services. A *service* is a piece of software running on a stationary or mobile computer that provides a certain piece of functionality such as optical tracking. Services can be connected to use the functionality of other services establishing a data flow network to achieve a more complex function.

To model what a service can offer to other services and what it needs from other services, DWARF uses the concept of *needs* and *abilities*. A match of one service's need to another service's ability leads to a connection between the services; this is set up by the distributed service managers.

Abilities describe the functionality a service provides, such as position data for optical markers. A service can have several abilities, such as an optical tracker that can track several markers simultaneously. Abilities are typed; an example is *PoseData* for 6D pose.

Needs describe the functionality required of other services. For example, an optical tracker needs a video sequence and descriptions of the markers it should detect. Needs are also typed, and only abilities of the same type can satisfy a need.

CORBA-based middleware manages the services. Each DWARF system network node has one *service manager*; there is no central component. Each service manager controls the node's local services and maintains descriptions of them. The service managers cooperate with each other to set up connections between services. Applications are created by configuring available services into networks connecting their needs and abilities as required [63].

2.4 Software frameworks for tracking

A specialized group of software architectures relate only to the topic of device abstraction and handling of tracking data. As both virtual and augmented reality applications rely heavily on input devices and sensors beyond the basic mouse and keyboard, managing such devices and the data they produce becomes increasingly important to such systems.

Typically, a variety of devices provide the same or similar data, therefore abstraction from an individual device would be beneficial to the reusability and portability of an application. Qualitative properties of input devices such as degrees of freedom may vary but combinations of different devices can make up for the lack of certain features. However, typically the combination of data from different devices is a non-trivial problem due to different measurement modalities, update rates or error properties.

MR Toolkit An early example of a software toolkit dedicated to developing interactive and immersive graphics applications is the MR Toolkit [97]. Historically it focused on virtual reality settings, but can also be applied to augmented and mixed reality applications. It provides device abstraction and network transparency for tracking devices. Therefore, applications are decoupled from the actual tracking devices used and programmers can substitute real devices with virtual ones for debugging and testing purposes.

VRPN The Virtual Reality Peripheral Network (VRPN) [104, 69] is a C++ library implementing device abstraction for a large number of tracking devices and also networking support based on tracking servers and application clients. It defines a small set of data types that can be reported by a device through individual facets. VRPN provides similar features as the MR Toolkit but supports a wider range of tracking devices.

VRCO trackd is a commercial tracking device software framework [113]. A central server process implements device drivers and provides device abstraction and network transparency to applications that connect to the server process. It is extensible through loadable modules that implement the actual device drivers.

VARIO The Virtual and Augmented Reality Input Output (VARIO) framework [96] builds upon the concepts introduced by MR Toolkit and VRPN. Similar to DWARF it implements a generic flow scheduling framework with

distributed components that can reside on different hosts. A central configuration process manages the connections between components and the configuration of individual components. Configurations are made persistent by saving and loading descriptions of the connection and configuration parameters to and from XML files.

VARIO supports multi-modal event data because it does not prescribe the type of data exchanged between components. It only provides for the exchange of byte arrays which need to be interpreted by the components themselves.

2.5 Overview of Studierstube

The context of this work is a research software system for augmented reality applications called *Studierstube* [91]. It provides the foundation and basic software design layers for the contributions developed in the main body. Therefore it is necessary to describe some concepts and features of the system.

The goal of the development of Studierstube is a software framework to support the technical requirements of augmented reality applications. It is a set of extension nodes to the Open Inventor [102] rendering library and an additional layer of objects that provide advanced runtime functions. It includes support for interaction based on 3D tracking events, rendering and output modes for all available virtual and augmented reality output devices, tools for developing distributed applications, and user management functions to support multiple users in a single setup.

2.5.1 Open Inventor

The Open Inventor (OIV) [102] rendering library is the basic software layer upon which Studierstube is build. It is a framework of C++ classes that implement a scene graph based rendering library using OpenGL. The principle architecture is that of an application framework with inversion of control that supports an event driven programming style whereby the application is typically composed as a set of callback functions that react to events issued by the framework.

The basic unit of OIV is a *node*. This is a C++ class type with additional functions to support runtime type system and serialization to and from an ASCII based text format. Nodes aggregate objects called *fields* that store a value of a certain type such as a string, a integer of floating point number, a 2D or 3D vector, or a rotation.

A dedicated node of type SoGroup can also associate a list of other nodes called *children* to form a hierachical structure, the *scene graph*. Such a graph forms a directed acyclical graph and orders a set of nodes into a certain structure. The children of a group node are also ordered and can be accessed in a left-to-right fashion numbering the first child with index 0 up to the last child with index $n - 1$.

The scene graph is traversed recursively by a set of mechanisms called *actions* to compute different data. The actions implement the Visitor pattern [40, p. 331] and call different functions on the nodes to trigger certain behavior. For example, the SoGLRenderAction sends appropriate commands to the OpenGL library to draw the image represented by the scene graph. The SoSearchAction traverses a graph to find nodes of specified type or name. The SoWriteAction serializes a scene graph into the Open Inventor file format.

Each node type can define the behavior for each action separately by registering a function to be called by the action when it traverses a node of this type. Thus a double dispatch mechanism is created to provide a flexible implementation and extension mechanism. For a more detailed discussion of these concepts, see the Inventor Toolmaker [117].

In addition to the scene graph traversal another flow scheduling mechanism is provided by Open Inventor. Fields of nodes can be connected to receive updates from other fields forming a data flow network for small scale stream processing and event handling. A special class of objects called *engines* can be embedded in the data flow graph to process the change events and compute new updates to other fields. These objects are not part of the scene graph but only of the overlaid field network graph.

The Open Inventor API provides methods to construct and operate on the scene and field network graph. Additionally *sensors* observe changes to fields, nodes and the scene graph and report these to callback functions. Time based sensors trigger callback functions after a certain time span, in regular intervals or when the library has CPU time to spare.

Finally, the library provides a text based and a binary file format to serialize a scene graph and field network structure to persistent storage (see the Inventor Mentor [116]). Complex scene graphs can be constructed directly as text files and read in by a client application of the library. Both manual and automatic authoring of such structures becomes possible in an efficient and transparent way.

Studierstube uses all of the above concepts to extend the base library in several ways and provides AR-centered functionality in the form of new nodes, actions and engines.

2.5.2 3D event system

Open Inventor only supports user interface events generated by a standard desktop interface consisting of mouse movements and key and mouse button presses. Such events are propagated into the scene graph via the HandleEventAction and are consumed by different types of nodes. A default set of manipulator nodes exist which implement a set of standard 3D interactions such as translating, scaling and rotating an object with the help of 3D widgets.

Studierstube extends the library to support more generic user input devices, generally 6DOF trackers. It implements a list of individual event channels that supply the scene graph with streams of 6DOF tracking events. These events consist of a channel id referred to as a station number, 3D position, a rotation, button states for up to 8 buttons, a time stamp and an event type discerning between movements or changes to the button state. They are propagated with a dedicated Handle3DEventAction.

An additional node base class Base3D implements the basic methods that the Handle3DEventAction calls during traversal. New node classes that react to 6DOF events use multiple inheritance to inherit from Base3D in addition to the Open Inventor node class and override the action methods to implement their own behavior. A simple example node class is SoStationKit which encapsulates a sub-scene-graph and moves it according to the incoming events. Different configuration fields choose which station channels to listen to for events and geometric offsets for each individual channel.

Nodes can choose to cull the traversal of attached sub-graphs based on containment of the event's position within the bounding box of the node and its children. Traversal can also be limited to events associated with a set of stations or of defined types.

Individual stations are associated with different input devices such as a user's head, a pen or a tablet that are manipulated by the user or with other tracked objects in the environment that are required for an application.

2.5.3 Widget system

A special group of nodes that interact with the 3D event system implement a set of standard widgets. Widgets are graphical objects that react to incoming 3D events and change their state based on a sequence of 3D events. Thus, they implement filters to compute a higher abstracted event from the stream of raw 3D events. Their state is represented by different graphical representations and changes to fields which are picked up by the application in turn.

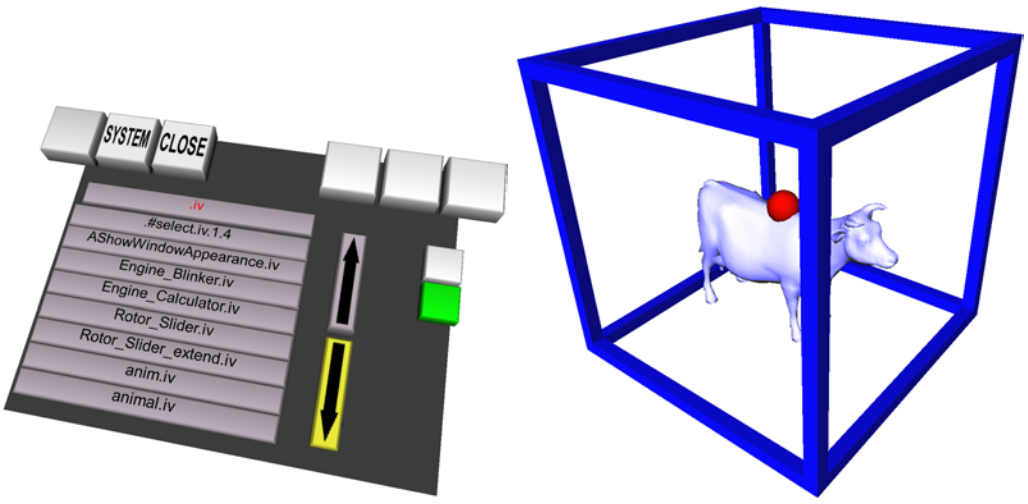


Figure 2.1: (a) A ListBox widget and several buttons to load and save applications. (b) A WindowKit widget contains application data.

2D widgets represent a typical example of such objects. Studierstube implements a standard set consisting of toggle, push and radio buttons, lists, and linear sliders. They are represented by 3D geometry that can be manipulated by the user. For example, buttons are boxes that have different heights for the released and pressed state. Thus, it simulates the look and feel of a real button.

A traditional 2D graphical user interface is usually presented on a tracked tablet called the Personal Interaction Panel (PIP) [103]. The physical representation provides a natural way to interact with the virtual widgets and gives haptic feedback when an interaction device intersects the virtual widget and collides with the real tablet. The PIP supports switching between different sets of widget groups similar to a tabbed dialog. Each pane can represent a user interface associated with a different application.

3D widgets allow simpler but less restricted interaction. They encapsulate geometry that can be dragged and rotated by the user with direct manipulation. The Raypicker widget implements a ray casting interaction that computes the intersection of a ray with given geometry. The ray is controlled by one or more 3D event channels. A more complex 3D widget is the WindowKit node which implements a box shaped container for geometry similar to a window in a 2D graphical user interface. The window's borders act as manipulators to move, rotate and resize it.

2.5.4 Dynamic application loading

Applications are implemented as a sub-scene-graph in a Studierstube process. They are defined by implementing a new application node class that provides the application specific functionality. The application node can use any sub-scene-graph to create the required graphics, user interface elements and interaction methods. Additionally, applications can define their own specialized node types to store dedicated data structures reusing Open Inventor's rich set of field data types. At the same time, such a design enables the use of all Open Inventor operations on the application's data.

Because Open Inventor supports serialization of any scene graph to and from a file, applications can be loaded and saved at runtime. As an application will store all required data structures in fields and/or nodes of a sub-scene-graph, the application's scene graph already represents the application's state. Studierstube supports concurrent execution of several applications and provides an API to start, stop and save applications as well as a user interface for manual control of applications.

2.5.5 Single-host multi-user Studierstube

Collaborative work scenarios are supported by Studierstube by providing the necessary functionality to drive the hardware devices required for multiple simultaneous users and by using an API to model resources for these users. Studierstube supports several independent output windows to drive multi-headed systems that offer a number of video outputs. Thus, several display devices can be connected simultaneously and provide personalized views to their users. Each output window can be configured independently in size, position and rendering method for stereo displays. The virtual cameras for each output window are controlled by independent input devices.

Moreover, the number of input devices is not limited, allowing the use of as many devices and trackers as are necessary to build a multi-user setup. A typical dual user setup for collaborative work will consist of two HMDs, two interaction devices and two PIPs, resulting in a total of six tracked devices and two output windows.

A set of resources consisting of an output window and event channels for head-tracking and input devices are modelled as a user identified by a unique id within the software framework. Applications are notified on startup of the number of users and their configuration. 3D events are associated with a user, if they are representing one of the user's input devices. Therefore, applications can distinguish between users and react differently as required. The output display of each user can also be configured to use a private sub-

scene-graph which is only rendered for this user. This mechanism enables personalized and private views.

2.5.6 Distributed Inventor

Like Studierstube, most distributed virtual environment systems use a scene graph for representing the graphical objects in the application, but many systems separate application state from the graphical objects. To avoid this "dual database" problem [62], Studierstube introduces a distributed shared scene graph using the semantics of distributed shared memory. Distribution is performed implicitly by a mechanism that keeps multiple replicas of a scene graph synchronized without exposing this process to the application programmer or user. Our OIV extension - Distributed Inventor (DIV) [44] - uses OIVs notification mechanism to distribute changes.

The communication between different replicas uses a reliable multicast protocol. All hosts of replicas can send updates but the implementation only provides causally-ordered update semantics. Therefore updates send by different hosts can be executed in different order on individual hosts.

A scene graph to be replicated is denoted by a special group node SoDIVGroup. It is configured with the necessary networking parameters and automatically establishes connection with other peers. Any changes to the sub-scene-graph of such a group node are communicated automatically. It also can be configured to retrieve a current copy of the sub-scene-graph from another peer upon joining a session.

DIV supports a master/slave property for each peer. Only peers with the master property set generate updates. Peers with the master property set to slave only listen for updates and apply them to their local replica. To avoid inconsistencies in the updates of the replicas only one peer is a master within Studierstube and therefore controls the replicated scene graph alone.

As the scene graph can be distributed using DIV, so can be the applications embedded in it. For each application a dedicated SoDIVGroup is created as a parent to the application's scene graph. Then, a newly created application instance will be added to it and will therefore be distributed to all replicas of a scene graph. The programming model of making application instances nodes in the scene graph also implies that all application specific data - i.e. data members of the application instance - are part of the scene graph, and thus are implicitly distributed by DIV.

At any point in time only one replica is a master and therefore controls the application. The other replicas only use the application's scene graph to render the personalized view for their outputs. Moving the master property between replicas implements a simple forms of load distribution [93].

Chapter 3

Data flow engine

Tracking is an indispensable part of any virtual reality and augmented reality application. While the need for quality of tracking, in particular for high performance and fidelity, has led to a large body of past and current research, little attention is typically paid to software engineering aspects of tracking software. Some current systems have a modular approach that allows to substitute one type of tracking device for another. Typically, this is the approach taken by commercial VR products that offer turn-key support for many popular tracking and input devices, but at the cost of a limited amount of extensibility and configuration options. In particular, they make it hard to combine existing features in novel ways.

In contrast, research systems may offer features not found in commercial systems, such as prediction or sensor fusion, but are usually limited to their particular research domain and not intended for the end user. In such systems, replacing a piece of hardware or changing its configuration usually leads to rewriting a significant portion of the tracker software.

In the middle(-ware), there is a lack of tools that allow for a high degree of customization, yet are easy to use and to extend. What is needed is a system that allows mixing and matching of different features, as well as simple creation and maintenance of possibly complex tracker configurations. To address this issue we present a software design and implementation that applies the pipes-and-filter architectural pattern [24, p. 53] to provide a customizable and flexible way of dealing with tracking data and configurations.

3.1 Tracking for Augmented Reality

The contribution of this chapter is the development of a generic data flow network library called *OpenTracker* to deal specifically with tracking data.

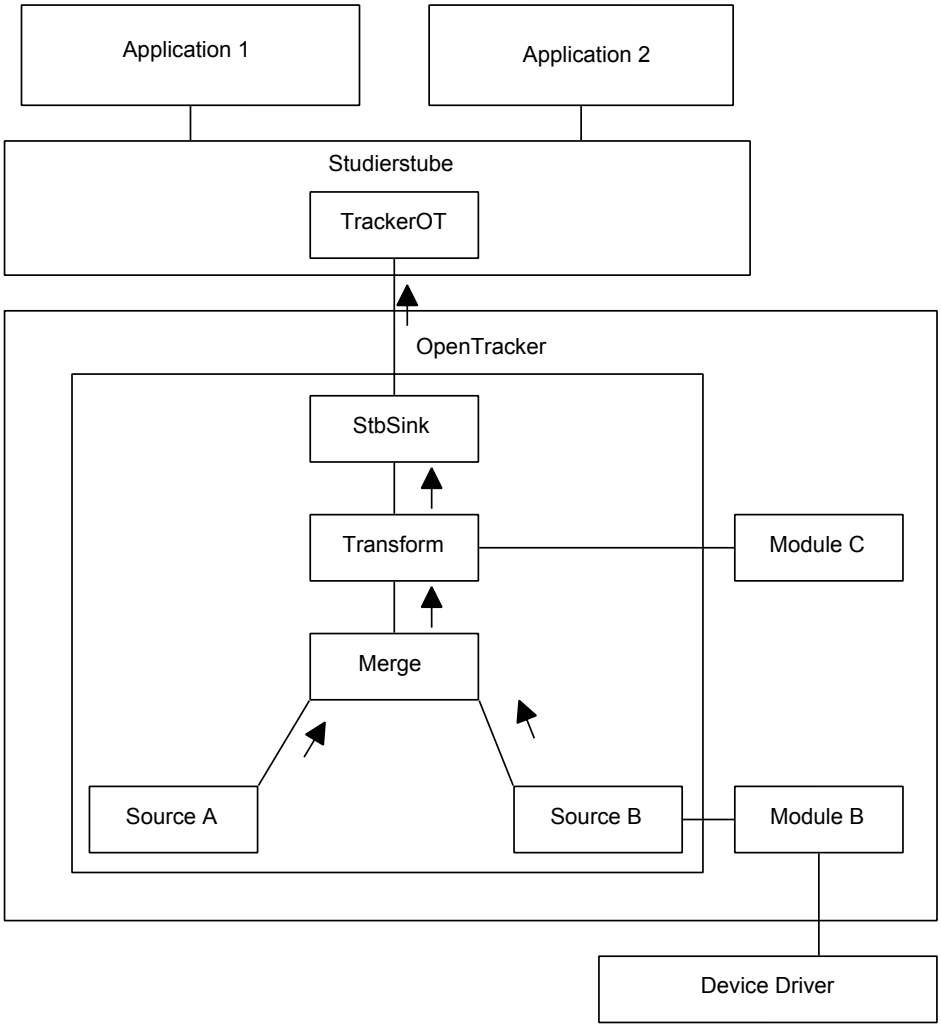


Figure 3.1: The *OpenTracker* library is a new layer abstracting the input devices from applications such as the Studierstube framework. It operates a pipes-and-filter network of nodes that process tracking data. Extensions are implemented by adding new nodes and associated modules that provide services to the nodes.

It is built on two key observations :

- Abstraction of recurring operations on tracking data separates applications from concrete devices and configurations and yields better code reuse.
- Flexible arrangement of such operations simplifies experimentation with and development of VR and AR applications.

In a typical VR or AR application tracking data passes through a series of steps. It is generated by tracking hardware, read by device drivers, transformed to fit the requirements of the application and send over network connections to other hosts. Different setups and applications may require different subsets and combinations of the steps described but the individual steps are common among a wide range of applications. Examples of such invariant steps are geometric transformations, Kalman filters and data fusion of two or more data sources.

The main concept behind OpenTracker is to break up the whole data manipulation into these individual steps and build a data flow network of the transformations. To describe the details of this concept, we will need some theoretical definitions which are discussed in section 3.3. Details of an actual implementation are described in section 3.4. The use of OpenTracker is further simplified by providing a declarative configuration language based on XML. Such an approach allows us to leverage the existing developments in the area of XML and to apply a set of existing tools to the configurations used by OpenTracker.

3.2 Related work

Device abstraction is a standard requirement for 2D graphical user interfaces, (e. g. GKS [51]), and sometimes incorporated into 3D applications [42]. There is a number of libraries such as VRPN [69], MRToolkit [97] implementing device abstraction for input devices typically found in VR and AR systems. Their main goal is to provide a fixed interface to the application for different devices and provide simple services for relaying the data over the network between several hosts. These libraries mostly lack any further means to process the data. Device abstraction is also an important goal of OpenTracker. However, it goes beyond pure abstraction using a static interface in that the data can be re-combined in novel ways.

Many interactive systems employ sophisticated event handling schemes. State changes to attributes of scene objects are either propagated by functional dependencies (e.g. routes in VRML [27], engines in Open Inventor

[102]), or may be handled by user supplied callback functions (e.g. script nodes in VRML [27]). These approaches inspire the architecture of OpenTracker, although none of them deals specifically with tracker configurations.

3.3 Concepts

Each unit of operation in OpenTracker is represented by a node in a data flow graph. Nodes are connected by directed edges to describe the direction of flow. The originating node of a directed edge is called the child whereas the receiving node is called the parent. To allow more than simple linear graphs, we introduce ports, references and edge types as follows.

3.3.1 Multiple Input Ports and References

Each node has one or more input ports and a single output port. A port is a distinguished connection point for an edge, i.e. the node can distinguish between events passing through different node ports. The output port of one node is connected to any of the input ports of another node. This establishes the flow by defining directed edges in the graph. A node receiving a new data event via one of its inputs computes a new update for itself and sends the new data event out via its output port.

Multiple input ports are desirable because computations typically have more than one parameter. Dynamic transformations, for example, are parameterized by the value of another node and thus use the data value received by a child to be transformed differently from the data of the parameterizing child. Merge nodes may select part of the data of an event based on the input port the event used. This allows more complex computational structures.

Additionally, an input port can be connected to several output ports. This enables several children nodes connected to the same input port of a node. Upon receiving an event, the parent node can only distinguish between the input ports, not between the actual children. Fan-In of several events is accomplished by serializing the events and the parent operates on each event in turn.

Conversely, an output port can also be connected to other nodes by using references within the graph. This establishes new edges between a nodes output port and other nodes input ports. However this is transparent to the child node. It cannot selectively send events to only one parent, but all events are distributed equally to all parents.

Figure 3.2 gives some examples of data flow graphs that can be build with OpenTracker. Part (a) shows a simple linear graph applying a geometrical

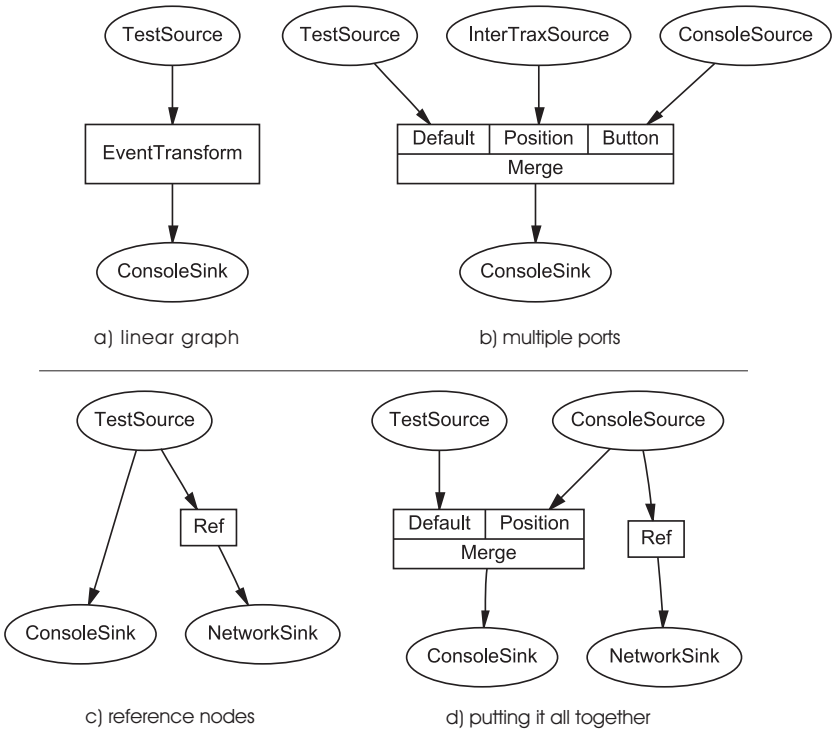


Figure 3.2: Visualizations of a data flow graphs as used in OpenTracker. (a) A linear flow. (b) A node with different input ports. (c) Fan-Out of output ports. (d) A complex example employing all features.

transformation to a data source, (b) shows a node with several input ports, combining the received data. Part (c) is a graph using a reference node to get a copy of the output of a node and (d) combines these features in a more complicated graph.

3.3.2 Edge types

The basic mechanism behind the data flow concept is event passing. Data events are passed from the children nodes upward to their parents. However, not all computations fit well into this model: Algorithms that operate on a list of tracker measurements or that require or compute the tracker state at an arbitrary point in time require different types of input or output interfaces. Examples are smoothing algorithms that take a history of events into account, or prediction algorithms that compute an expected measurement for a given point in time.

Therefore, we also distinguish between different edge types. Edges are typed by typing the ports of the nodes they connect. We establish the rule

Component	Description
position	3 component vector describing a position in space.
orientation	4 component vector describing a rotation as a quaternion.
button	16 bit integer value describing the state of 16 button devices.
confidence	floating point value in the interval $[0, 1]$ describing the quality of the measurement.
time	time stamp giving the time of measurement.

Table 3.1: Components of the OpenTracker event data type.

that only two ports of the same type can be connected and this type is the type of the edge. There are three edge types: *event*, which is implemented by event passing, *event queue* and *time dependent*. The latter two are implemented as interfaces that are polled by the parent node, because the data returned is parameterized. In the case of the *event queue* interface, it is possible to query the number of stored events and retrieve them by index. The *time dependent* interface can be queried by specifying a point in time, for which the appropriate data value is returned.

3.4 Implementation

In an actual implementation we distinguish *source nodes*, which are leaves in the graph and receive their data values from external sources, *filter nodes*, which are intermediate nodes and modify the values received from other nodes, and *sink nodes*, which propagate their data values received from other nodes to external outputs.

The data type passed between nodes is a complex data structure tailored towards the requirements of AR applications and consists of a fixed set of components (see Table 3.1). Although this restriction to a fixed data type appears as an limitation, it can easily be extended or generalized because nothing in the supporting system relies on the type of the event data.

3.4.1 Source Nodes

Most source nodes encapsulate a device driver that directly accesses a particular tracking device, such as a Polhemus or Ascension tracker connected to a serial interface. Other nodes objects form bridges to complex self-contained systems, such as the video tracking library from ARToolkit [55] or implement a DWARF service interface [12]. A third type of source node emulates

a tracker via the keyboard, access network data (see section 3.5.1) or simply responds with constant values (useful for development and debugging).

Some source nodes have a multi-threaded execution model to implement an efficient decoupled simulation model [97] (e. g. , when blocking I/O must be used).

3.4.2 Filter Nodes

Filter nodes receive values from one or more child nodes. Upon receiving an update from one or more of their children, they compute their own state based on the collected data. A non-exhaustive list of filters includes:

- Transformation filters perform geometric transformations of their children's values. These include pre- and post-transformations and may be static or depend on data values received from other children. The latter allows to modify the filtered state relative to another tracker state.
- Button filters perform boolean operations on the button state of different input sources to combine them into a new event value.
- Prediction filters allow to partially compensate for lag in the measuring and processing tracker data.
- Noise and smoothing filters are handy to deal with inherent inaccuracies of trackers.
- Undistortion filter are necessary e.g. to linearize distortions in the magnetic field of a magnetic tracking device.
- Permutation filters are necessary to match data representations from different hardware or software platforms, such as equivalent, but incompatible quaternion representations.
- Merge filters assemble new data values using different parts of the data values of several children. Use cases include the combination of orientation from an inertial tracker with position information from an acoustic tracker, or adding a button device to a closed tracking solution such as Polhemus Ultratrak.
- Conversion filters are able to translate one data type into another. For example, 2D positions from a desktop pointing device can be translated into 3D positions by adding a constant third value.

- Clamp filter are special nonlinear transformation filters that cut off values at user-specified extrema, for example to deliberately limit interaction to a valid range.
- Store-and-forward filters are useful if transient loss of tracking can be expected, for example if occlusion occurs in optical tracking. The last measured value is simply repeated to provide at least a reasonable and valid state.
- Confidence filters select data values from different children based on some measure of confidence in the accuracy of the data.

3.4.3 Sink Nodes

Sink nodes are similar to source nodes but distribute data rather than receive it. They include output to network multicast groups, debugging output to a user interface or thread-safe shared memory output to integrate OpenTracker as a library into other applications.

3.4.4 Time

Time is reflected in several ways in the architecture of OpenTracker. The type system for edges supplies us with different ways of dealing with time, either having an event based approach, with or without queuing of events, or by specifying functions of tracking data as continuous functions of time.

For the event based nodes, each event is time stamped by the individual device driver or node that generated it. Thus nodes can react to the temporal aspects of tracking data. For example, a simple prediction node incorporates the time difference between single events to correctly update its output.

More complex aspects such as a prediction for a changing prediction interval is satisfied by the different edge types. An application that wants to get a calculated value for an arbitrary point in time can query the state at that time from a node supporting *time dependent* output. How this value is calculated depends on the node's implementation.

OpenTracker does not implement any clock synchronization of different hosts working together in a network. There are already well established means to solve this problem such as the NTP protocol [1].

3.4.5 Software architecture

The intent of OpenTracker is to provide an auxiliary library that is to be integrated into VR or AR applications. Therefore it is kept lightweight and

customizable. The library is designed as a class hierarchy of tracker objects, implemented in C++. It is build around a small set of core classes that implement the basic node interfaces, a parser that builds the runtime structure from a configuration file and the main loop driving the event model. Any other functionality is implemented by a set of module classes that can be easily extended or modified.

The module classes create and manage the nodes representing the functionality of the module. In the main loop of the library each module is called to provide new events and after an event is processed to handle results of the data flow. For example, the implementation of a network sink node stores any event data that it received during event propagation. Afterwards the network module checks each network sink node for updated data values, constructs a new network packet and sends it to the configured destination. Modules may be implemented multi-threaded to avoid stalling the main thread during longer computations or polling a device with blocking I/O.

There are also nodes that perform without an underlying module. Examples are filter nodes that implement geometric transformations on incoming events and pass the transformed events to their parents.

There is no fixed interface to the integrating application to maximize flexibility. Application programmers have to either use one of the supplied nodes (such as a generic call back node) or supply their own module implementing sink nodes as interfaces to their application. Moreover, the use of the library main loop is not mandatory. The processing can be integrated with the applications main loop to avoid additional threads and synchronize the tracking data processing more closely with the application. These design decisions ensures that the library can adapted to the special needs of any application.

Figure 3.3 shows a class diagram of the core classes. The class *Context* implements the main loop and keeps reference of all modules and the data flow data structure. It employs an object of class *ConfigurationParser* to parse the configuration files. Actual node implementations are derived from *Node*, for example the *Transformation* or the *TestSource* class. *WrapperNode* and *RefNode* are special nodes that implement the port and reference functionality. *State* is the default event type.

The library is extensible through the use of an abstract *NodeFactory* interface to define the class interface for creating new nodes and through the *Module* class that provides an interface for processing during the main loop. Any extension adds new node types by providing an object that implements the NodeFactory interface. The object is added to a list of factories known by a Context object at startup and can then create nodes of the new type as requested by the parser.

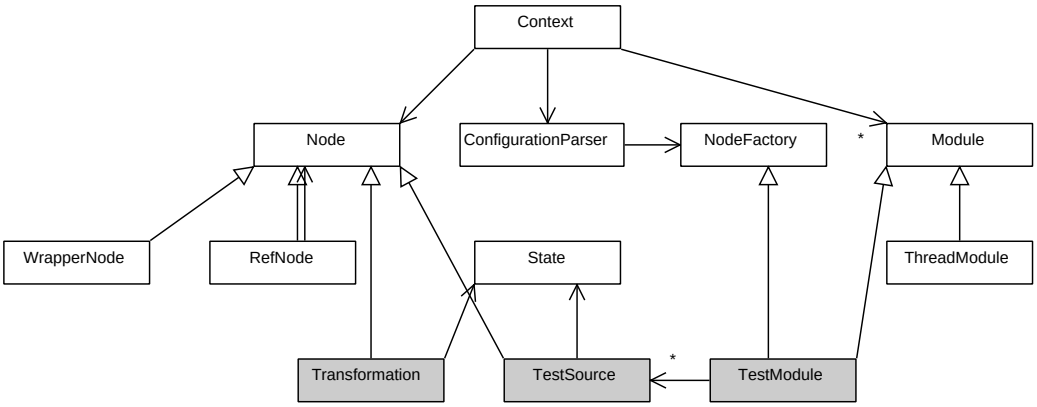


Figure 3.3: Class diagram of the OpenTracker library

To add more complex functionality such as device drivers a subclass of Module is created and added to the list of modules known to a Context object. The modules are called regularly during processing of the main loop. Within these callbacks they can implement any processing and create new events. These events are then propagated into the network by associated nodes. Events can also be read in from the network by these nodes. A module can obtain references to the nodes it is interested in by implementing the NodeFactory interface. It acts thereby as both the creator and the active implementation of the nodes.

3.4.6 Software engineering with XML

XML, the eXtensible Markup Language, is the emerging standard primarily aimed at web-based applications and software systems [23]. XML is a markup definition language that allows to define hierarchical markup languages with so-called document type definitions (DTD). With the appropriate DTD, standard XML tools can be used to conveniently edit, type check, parse, and transform any XML file.

Thus, providing a simple DTD for describing the data flow graphs of tracker nodes opens access to software libraries and tools that simplify several steps of the development cycle:

- A visual DTD editor can be used to design and maintain the DTD.
- An XML parser [106] enforces content format on the tracker configuration file while building the corresponding structure in memory, thus automatically performing many of the consistency checks that otherwise have to be hand-coded.

- The same parser implements an API to manipulate the data structure at runtime and still keep it consistent with the DTD. Such a runtime structure can easily be written out to a valid configuration file again.
- A convenient XML editor such as [50, 49] with a graphical user interface allows the end user to design the tracker configuration without having to master the syntax. It also enforces the correct content format, reducing syntax and semantic errors made by users.
- Integration with high-level software engineering tools that create code or configuration files from specifications is simplified by the use of XML. Even automatic reverse engineering of complex configurations is easier relying on a defined structure than from pure source code.
- Using the eXtensible Stylesheet Language (XSL) [4, 30], automatic textual and even graphical documentation can be created from a tracker configuration file, for example by using the free graph drawing utility dot [6] (see Figure 3.2).

Markup languages are generally used to annotate textual documents with structural information. Thus a general XML document consists of text grouped and structured with tags. Markup languages defined in XML consist of elements, essentially expressed as tags, and a structural model (the content model) of the possible ways these elements may be nested. Moreover, elements are annotated by name-value pairs called attributes.

OpenTracker maps elements to nodes and attributes to members of these nodes. We are not using any textual content but purely rely on the content model provided by the DTD. An open source XML parser [106] builds a tree of elements representing the given configuration file. OpenTracker walks the tree and creates a new node for each element based on the elements name. The string values of the attributes are parsed according to the objects class and the corresponding members are set. Attributes typically describe such data as the parameters of a transformation. The parent - child relationship of the data flow graph is directly mapped onto the parent - child relationship of XML elements.

The content model enforces interface and semantic constraints on the specified graph. As described in section 3.3 edges and the corresponding node ports are typed and therefore restrict the possible combinations in the construction of the graph. These constraints are expressed in the DTD and are checked by an XML parser or enforced by an XML editor. Also restrictions on the number of children are described in the DTD. Source nodes typically do not have any children as they rely on data from external sources

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE OpenTracker SYSTEM "opentracker.dtd">
<OpenTracker>
    <configuration>
        <ARToolKitConfig camera-parameter="camera_para.dat"/>
    </configuration>
    <ConsoleSink comment="Pip">
        <StbSink station="0">
            <EventTransform DEF="Camera" scale="0.001 0.001 0.001">
                <ARToolKitSource tag-file="pip.tag" />
            </EventTransform>
        </StbSink>
    </ConsoleSink>
    <ConsoleSink comment="Pen">
        <StbSink station="1">
            <EventDynamicTransform>
                <TransformBase>
                    <Ref USE="Camera"/>
                </TransformBase>
                <EventTransform scale="-2.1 -2 0" translation="0.14 0.1 -0.01">
                    <WacomGraphireSource device="1"/>
                </EventTransform>
            </EventDynamicTransform>
        </StbSink>
    </ConsoleSink>
    <ConsoleSink comment="Viewpoint">
        <StbSink station="2">
            <EventTransform rotation="1 0 0 0">
                <TestSource frequency="25"/>
            </EventTransform>
        </StbSink>
    </ConsoleSink>
</OpenTracker>
```

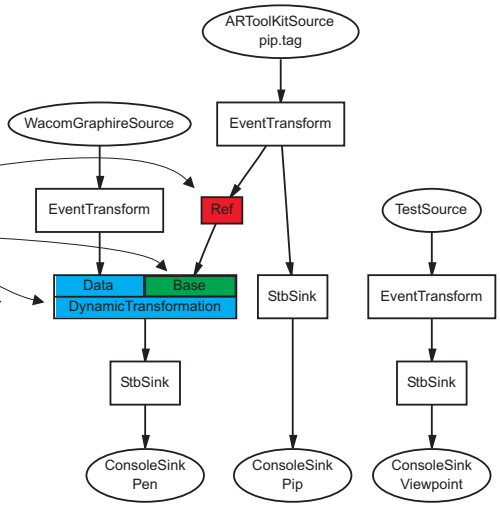


Figure 3.4: An example configuration file and the corresponding data flow graph. The use of Ref nodes and multiple input ports is highlighted.

to compute their own data. A number of filter nodes get the value of a single child node, transform it and pass it on. In contrast, confidence filters use any number of children to compute their data value.

The reference structure is created by using unique ID attributes on elements and referencing these IDs in reference elements. Again XML enforces the uniqueness of these IDs and the parser library simplifies the search for the referenced elements.

While children of nodes with only one input port are directly mapped to children elements in the XML file, children of different input ports need to be addressed differently. This is handled using wrapper elements. Any group of children that is connected to a specific input port is wrapped by an additional XML element. This element in turn is the direct child of the node of interest. These elements are closely related to the node's element and are typically the only possible children elements. They are mapped to special wrapper nodes that can be distinguished by the node implementation. Otherwise they are transparent to the actual data processing.

Figure 3.4 gives an example of such a configuration file, using all of the features described before. The interesting constructs are highlighted and cross linked with the corresponding nodes in the resulting data flow graph.

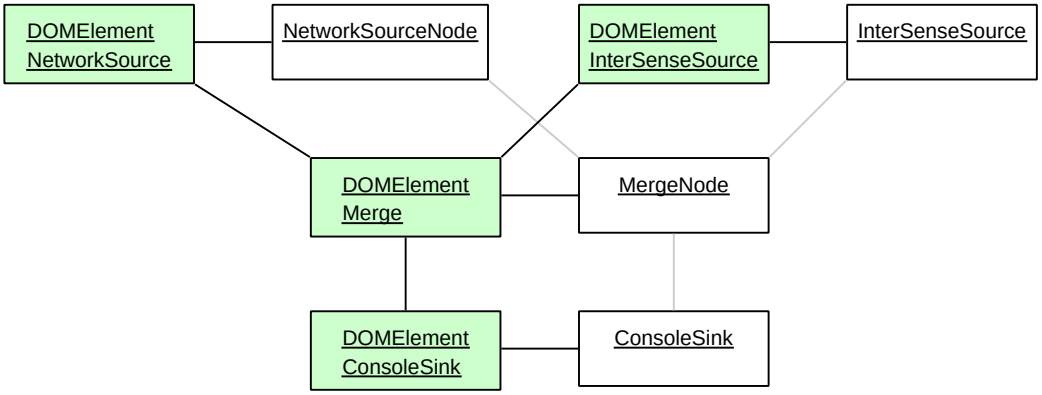


Figure 3.5: The DOM tree elements in green are decorated with the white OpenTracker nodes. The links between the nodes are only virtual and are inferred from the DOM structure.

3.4.7 Data flow implementation

The implementation of the node graph and the data flow of events is directly based on the Document Object Model (DOM, [118]) tree structure provided by the XML parser library Xercex [106]. A configuration file is read in and represented as a data structure in memory. This data structure represents the entities of an XML file such as elements and attributes and the relations between them as structures called DOM nodes. It forms a tree where subelements and attributes are child nodes of other element nodes in the tree. The DOM API provides methods to retrieve the parent node, child nodes and attribute nodes of any element node.

OpenTracker reuses the in-memory DOM tree and decorates it with instances of the node types described above. A DOM node provides a facility to store a mapping of names to pointers to user data and OpenTracker stores the pointer to the node instance associated with a certain element in the configuration file in this map (see Figure 3.5). The Decorator pattern [40, p. 175] describes this construct accuratly and succinctly. Any OpenTracker node implements an interface to retrieve references to its parent or children by recurring to the DOM API of the decorated DOM node.

Tracking events flow through the network via a push and a pull mechanism. The *event* interface uses a push mechanism, that passes the current event from the source nodes via any intermediary nodes to the sink nodes. Every node calls an update method on its parent node which recursively calls its own parent's node update method after processing the event (see Figure 3.6(a)). Thus, the modules associated with the source nodes only have to trigger this event propagation by calling the update method on their source

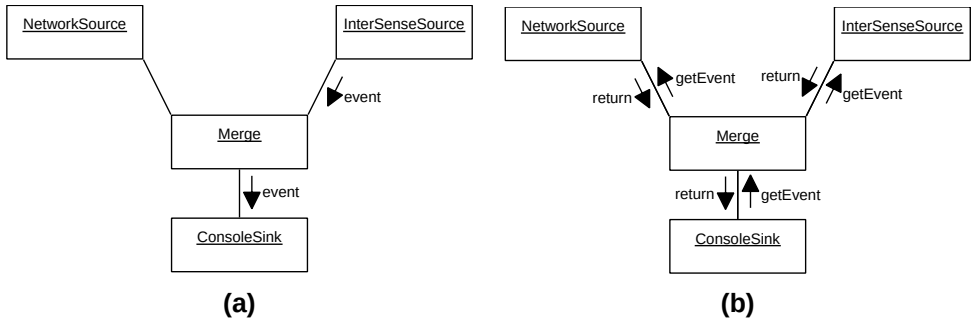


Figure 3.6: Two types of data flow in OpenTracker. (a) Events are pushed by source nodes through the graph. (b) Events are pulled by sink nodes.

nodes. Only a reference to the object storing the event data is passed. A node that changes the event's data has to provide a new instance to avoid changing an instance that may also be used by other nodes. This instance is typically a member of the node reused to avoid frequent allocation and deallocation of an event object on the stack.

The *event queue* and *time dependent* interfaces use a pull mechanism. If a node is queried via one of these interfaces and it requires event information from any children nodes upstream in the network, it recursively calls the children nodes' interfaces with the appropriate parameters (see Figure 3.6(b)). Again all nodes have to provide their own object instances to avoid side effects by shared event instances.

Not all nodes implement all interfaces. The *event* interface is the standard case and is implemented by all nodes. The remaining interfaces are only implemented in a small subset of nodes that use it to implement more complex behaviors. For example, an *EventQueue* node stores a queue of the last events pushed through it and exposes the queue through the *event queue* interface. Another node called *Filter* then uses an EventQueue node to implement a linear filter over the last events that is triggered by any event pushed through it.

3.5 Results

The OpenTracker library has been incorporated into the Studierstube framework and has since become the only interface Studierstube provides for tracking devices. We continue with an overview of some applications that were implemented based on this work.

A dedicated module implements source and sink nodes to allow communication between the OpenTracker data flow network and the Studierstube

applications. A sink node *StbSink* is associated with a single event channel via an attribute denoting the station number of the channel. Tracking events routed to this sink node are picked up by the 3D event distribution mechanism in Studierstube and propagated over the scene graph. A source node *StbSource* generates new events from field values of Open Inventor nodes allowing the generation of virtual tracking events from Studierstube applications.

3.5.1 Distributed tracking

OpenTracker implements a pair of source and sink nodes that allow the transportation of tracking events over the network. Using these nodes larger networks spanning multiple hosts can be created. There are several reasons why it is desirable to share tracker data over a network:

- Using the tracker data at multiple host computers for a distributed virtual environment (local or remote): Input in the form of tracker data becomes readily available through transparent network access via OpenTracker. The scene database still has be to kept consistent by a proprietary application protocol, but the task is much simplified.
- With the same approach, multi-processing based on inexpensive PCs becomes possible with little configuration effort. This is useful to achieve some degree of load balancing. In particular, computationally expensive functions such as filtering or undistortion can be assigned to either sender or receiver, depending on the computational budget.
- Network support makes it easy to span multiple operating systems, in particular if a specific tracking device or service is only available at one particular host.
- Transparent substitution of tracking devices enables to switch devices during run-time or to use virtual devices for testing and playback of prerecorded or generated tracking data.

OpenTracker allows multiple senders and receivers of tracker data to communicate asynchronously by using IP multicast (see Figure 3.7). It is even possible for a single host to operate as a sender and receiver at the same time, by picking up data, then modifying it and re-sending it to the network on another network channel.

While there is a preferred network protocol for OpenTracker, support for additional formats can be easily implemented. In the following, we give some examples as to how a networked setup can be used:

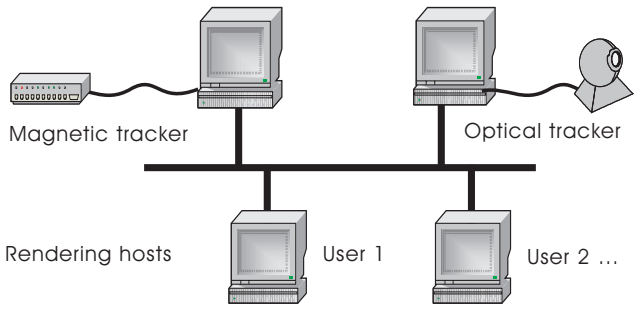


Figure 3.7: A distributed setup using network transparency of OpenTracker. Dedicated hosts drive magnetic and optical tracking systems and provide the data to rendering hosts.

- A tracker server (typically a cheap PC with lots of serial I/O boards running Linux) samples an Ascension Flock of Birds at highest rate and sends the resulting data stream via multicast to several clients using this data to animate a collaborative virtual environment.

- The Polhemus Ultratrak uses a proprietary network format and IP unicast packages. Unfortunately, its closed architecture does not support input devices with buttons such as a stylus or 3D-mouse. Therefore, we added a tracker object to the client that is able to decode the Ultratrak protocol. A button source reads button values from a standard parallel interface, and a merge filter combines these two sources to emulate a complete VR input device.

- Simulation of tracking sources for development. An application can be configured to receive tracking input via multicast and is thus decoupled from any process querying the actual tracking device. Therefore it is possible to substitute the tracking device with a software simulator of the tracking information. This substitution is transparent to the application and can even be performed while the application is being executed.

3.5.2 Mobile Augmented Reality setup

Augmented reality setups often require the integration of various different tracking devices. We built a series of mobile AR setups to investigate mobile collaborative applications [80]. The first iteration consisted of a PC notebook equipped with a NVidia GeForce2Go video chip and a 1GHZ processor and worked under Windows 2000. It is carried by the user in a backpack.

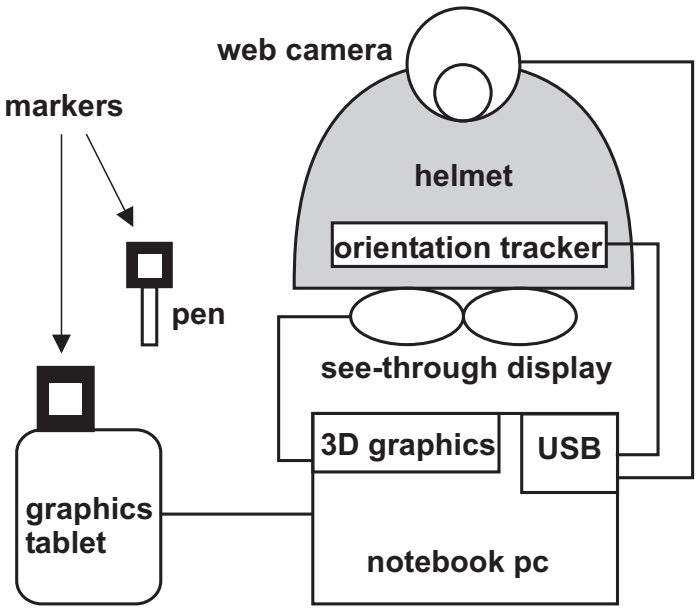


Figure 3.8: Hardware component diagram of the mobile AR setup.

As an output device, we use an Sony Glasstron see-through stereoscopic color HMD. The display is fixed to a helmet worn by the user. Moreover, an InterSense InterTrax^2 orientation sensor and a web camera for fiducial tracking of interaction props are mounted on the helmet.

The main user interface is a pen and pad setup using a Wacom graphics tablet and its pen. Both devices are optically tracked by the camera using markers. The 2D position of the pen (provided by the Wacom tablet) is incorporated into the processing to provide more accurate tracking on the pad itself. Figure 3.8 gives an overview of the setup.

Tracking of the user and the interaction props is achieved by combining data from various sources. The OpenTracker component receives data about the user's head orientation from the InterTrax^2 sensor to provide a coordinate system with body stabilized position and world stabilized orientation.

Within this coordinate system the pen and pad are tracked using the video camera mounted on the helmet and ARToolKit [54] to process the video information. Because the video camera and the HMD are fixed to the helmet the transformation between the cameras and the users coordinate system is fixed and determined in a calibration step.

The pad is equipped with one marker. This is enough for standard operation, where the user holds it within her field of view to interact with 2D user interface elements displayed on the pad. The pen, however, is equipped with a cube featuring a marker on the five sides which are not occluded. This

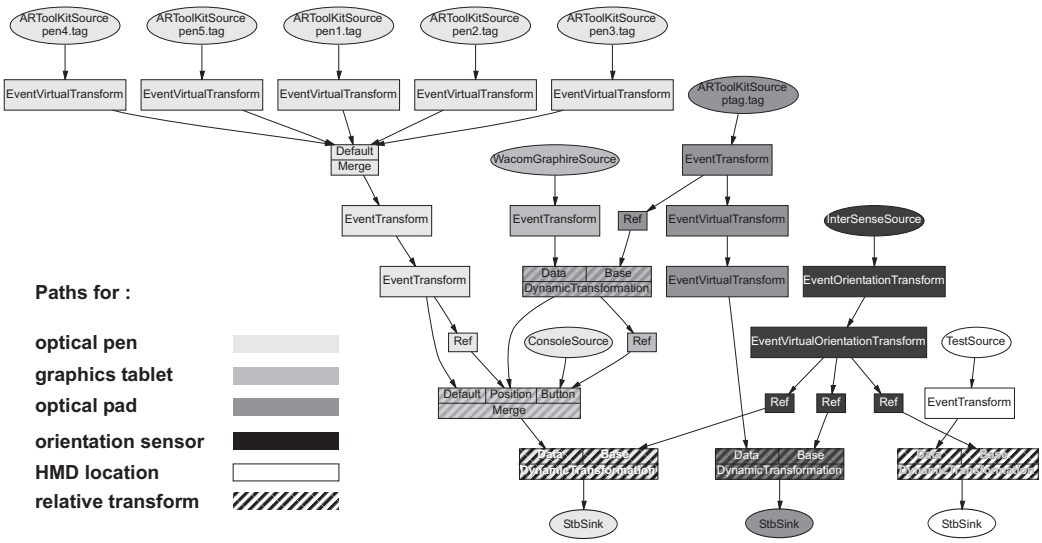


Figure 3.9: The data flow graph of the tracking configuration for the mobile AR setup. Individual flows are indicated per source. The diagram was automatically generated from the XML configuration description.

allows to track the pen in almost any position and orientation. Moreover whenever the user touches the pad with the pen the more accurate information provided by the graphics tablet is used to set the position of the pen with respect to the tablet.

The data flow graph describing the necessary data transformations is shown in Figure 3.9. Round nodes at the top are source nodes that encapsulate device drivers. The round nodes at the bottom are sinks that copy the resulting data to the AR software. Intermediate nodes receive events containing tracking data, transform it and pass it on, downwards. An important type of transformation is the relative transformation that takes input from two different devices and interprets the location of one device relative to the location of the other (called the *base*).

Different colors denote paths through the graph that describe how the tracking data for different devices are processed. Relative transformations are marked by cross stripes in the color of the two paths connecting. For example, the *optical pen* path describes the five markers that are each transformed to relate the pen point location. This information is merged, then further transformed. After another merge with data from the graphics tablet, it is once more transformed to the reference system established by the orientation sensor.

Similarly, the *optical pad* path describes the computation to obtain the location of the pad. As a side effect, the *optical pad* information is used at

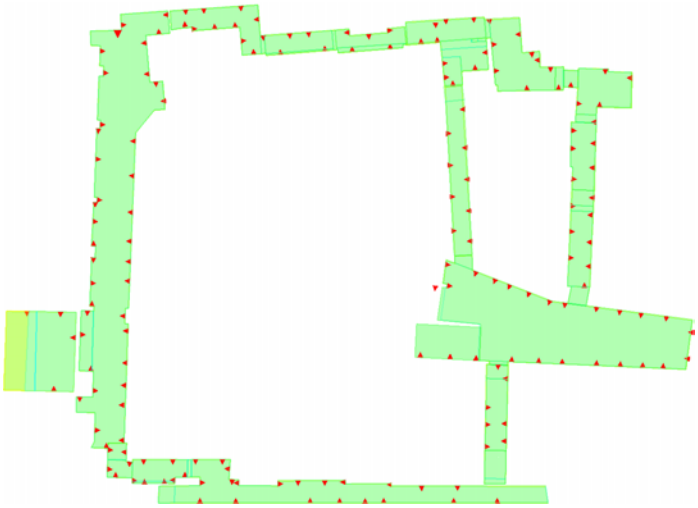


Figure 3.10: This diagram shows the geometric model of the floor the author's research group is located on. The red dots denote the locations of measured markers used to track the user within the environment.

one step to transform the 2D information from the *graphics tablet* path to the actual pen position which is subsequently merged with the pure optical information.

Finally the white *HMD location* path is used to provide information about the head location. The TestSource node's task is to provide a constant value which is then transformed by the orientation sensors.

We would like to note that using a visual XML editor, this complex configuration was created without writing a single line of code.

3.5.3 Indoor wide area tracking

To build an environment where we could test drive our mobile AR kit, we implemented an indoor tracking solution to cover a floor of our building. As we did not have access to a proprietary building-wide positioning infrastructure (such as AT&T Cambridge's BAT system used by Newman et al. [67]), we choose to rely on a hybrid optical/inertial tracking solution. This approach proved very flexible in terms of development of positioning infrastructure, but also pushes the limits of what the used optical tracking library ARToolkit can provide.

To implement a wide area indoor tracking solution we resolved to use a set of well-known markers that were distributed in the environment. Together with a geometric model of the building that includes the location of the well-known markers (see Figure 3.10) we can compute the user's location as

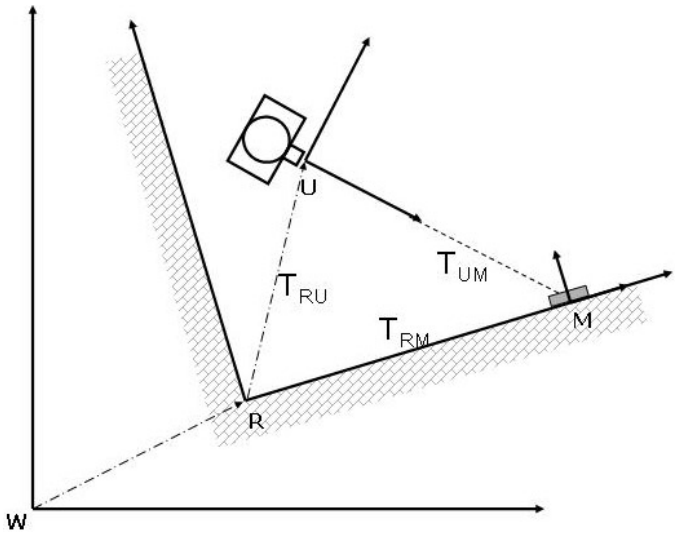


Figure 3.11: This diagram shows the different coordinate systems involved to compute the user's position. U is the user's position and orientation, R the room coordinate system, M a markers position and orientation within that room and W the world or model coordinate system.

soon as a marker is tracked by the optical tracking system. The model is structured into individual rooms and the connections between these rooms called portals. The location of a room within a world coordinate system completes the model. These models and the location of the markers were obtained by manual measurements with a geodetic device.

A measurement of a marker by the optical tracking returns the markers position and orientation T_{UM} within the user's coordinate system U (see Figure 3.11). This is essentially the transformation to convert coordinates from the system U into the system M. Inverting T_{UM} gives the user's position and orientation within the marker's coordinate system M as $T_{MU} = T_{UM}^{-1}$. By combining the fixed and measured transformation T_{RM} between the room coordinate system R and the marker's M with the value T_{MU}, we calculate the user's position and orientation $T_{RU} = T_{RM} \cdot T_{MU}$ within the room coordinate system R.

The implemented tracking approach requires a large set of markers. It is necessary to place a marker about every two meters and to cover each wall of a single room with at least one marker. Deploying it in our floor covering about 20 rooms and long hallways would require several hundred different markers.

However, marking up a large indoor space with unique ARToolKit markers is not feasible for two reasons. The more markers, the higher the degree of similarity of markers will be. Additionally, lighting conditions vary often

from one room to another. All this leads to inferior recognition accuracy. For a larger set of markers this implies a higher number of false recognitions. A large set of markers also enlarges the search space that ARToolKit has to traverse, leading to significant decrease in performance. As a consequence, it is not possible to scale the use of ARToolKit to arbitrary large marker assemblies.

To overcome this restriction, we developed a space partitioning-scheme that allows reusing sets of markers within the given environment. The idea behind this approach is that, if the tracking system knows the user's location, it can rule out large parts of the building because they will not be visible to the camera. Therefore, for two areas, which are not visible to each other, it becomes possible to use the same set of markers. This problem is equivalent to approaches for indoor visibility computation based on potentially visible sets [5].

To compute a possible placement of markers, the space of the model is partitioned into a 3-dimensional cubic grid of a fixed length. Then, marker patterns are assigned to the measured positions such that the patterns are unique within each cubic cell and its 26 direct neighbors. Basically, every cell defines a partial mapping from marker patterns to marker positions in the environment. Varying the grid size allows to tune the tradeoff between the minimal distance of positions of the same pattern and total number of individual patterns necessary. The minimal distance is always twice the grid size. Because the volume of a grid cell and its neighbors increases with grid size, more marker positions lie within the volume and require pair-wise different patterns. We ran calculations for different grid sizes and established a grid size of 4m requiring 30 different patterns to be a good compromise. Altogether we measured 210 marker positions within the building.

The tracking computes the user's location from a known cell and a marker pattern observed by the camera mounted on the helmet. Because the pattern is unique within the cell and it's neighbors the associated marker position can be established and the user's location is computed by concatenating the marker position and the relative measurement computed by the ARToolKit library. These computations are executed within OpenTracker by configuring appropriate transform nodes.

The representation of the mapping from marker patters to marker positions within the cells is implemented with the help of a special node within OpenTracker called *GroupGate*. A GroupGate defines a gate that passes incoming events on, if enabled and stops them otherwise. A set of GroupGate nodes is configured into an directed graph describing a neighborhood relationship. The relationship is typically symmetric but need not be. At any point in time only one GroupGate node is denoted as active. If a GroupGate

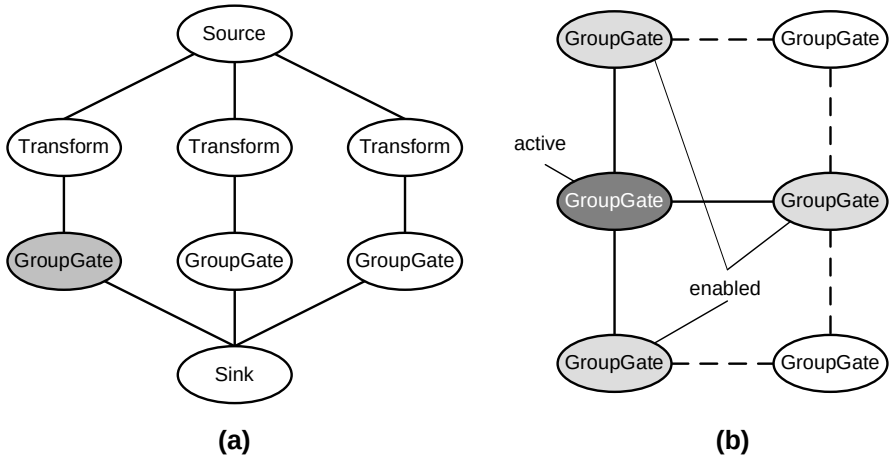


Figure 3.12: The function of the GroupGate node. (a) A set of GroupGates allows to select a single path from a set of paths through the graph. Only the active GroupGate propagates events. (b) The GroupGates are in a neighborhood relation. The active GroupGate's neighbors are enabled to propagate events as well and become active, if an event passes through them.

node is active, it is enabled and all its neighboring GroupGates are enabled as well. All other nodes are disabled and will not pass events. A GroupGate becomes active, if it is enabled and an event passes through it. Additionally, a GroupGate also defines a second input port named override. If an event passes through the override port, the node is also activated.

A single cell is modelled as a GroupGate and its 26 neighboring cells are configured as neighbors. The measured data from a single marker pattern is passed through all possible transformations for the different maker positions it is used at. Then, each transformed data event is passed through the GroupGate of the cell the marker position is associated with. The active GroupGate corresponds to the cell the user is currently in. Because events can only pass through the active GroupGate and its neighbors, data from a marker will be used only once. Moreover the data passes only through the GroupGate associated with the marker position of the last seen marker activating it if necessary. Thus the activation will always shift with the user's movement through the set of GroupGates. Figure 3.12 gives an example of the use of the GroupGate node.

The tracking needs an initial position at startup to set the first active cell and GroupGate. Two solutions were implemented to set the initial cell. The user of the system can select her current location from a list of rooms. For each room a standard cell was selected that reflects the center of the room. Moreover, unique markers can be placed in the environment to identify a

location absolutely. The output of the marker nodes are then connected to the override ports of the GroupGate that corresponds to the cell the unique marker is in. A set of such markers is used to denote the different floors in front of the elevator to allow the system to deduce the user's location automatically, whenever she steps out of the elevator. Because the system cannot measure the position of the elevator, it can not continuously track the active cell the user is in and needs a unique marker to establish the absolute position again.

3.6 Summary

OpenTracker is the first software framework to thoroughly apply the pipes-and-filters architecture to the problem of manipulating tracking data. The resulting advantages are twofold. The high-level language introduced to configure the processing of tracking data simplifies experimental and exploratory programming of data manipulations and also enhances reuse because the effects of an existing configuration are fixed. Describing the configuration in a dedicated language renders it also more accessible to automated methods such as generating a certain configuration.

The layered architecture that OpenTracker enforces on the overall application provides a clear cut interface between the application logic and the functions required to deal with tracking devices. A number of issues appearing in relation with tracking devices such as calibration and registration, network transparency or fusion of input data can be dealt with in a way that is transparent to the application. Decoupling the application specific functionality from the device layer also furthers reuse of the application in the context of different tracking systems.

Chapter 4

Context sensitive scene graph

Scene graph APIs have become an established tool for developing interactive 3D applications. They offer an object-oriented and structured approach to describing the application's graphical needs and interactions with the 3D representation. From a software architecture standpoint they also address a source of complexity in developing graphics applications which Strauss and Carey [102] described as the "dual database" problem.

Strauss and Carey observed that interactive graphical applications were using two separate data structures to compute application state and to render the graphical output. They proposed an object oriented graphics toolkit called Open Inventor that can serve both as a data structure for the application's state and to compute the rendered output image. The unification of two separate data structures into one is a powerful mechanism that greatly simplifies the design of interactive graphical applications. The graphics toolkit provides both a high-level view of the application's data as well as performs the low-level computations necessary to drive a rendering pipeline such as OpenGL.

The idea of a scene graph appears to be contradictory to the established Model-View-Controller (MVC) pattern [24, p. 125] for interactive applications. The separation into the model storing the application's state and the view dealing with the presentation of the state is deliberately given up. The rationale behind the merger of the two databases is the large volume of data that needs to be exchanged otherwise. The resulting performance issues become relevant for interactive applications that have to achieve soft real-time frame rates of 30 Hz and above. Thus, the scene graph is an optimization of the MVC architectural pattern tailored to the need of complex interactive graphics applications. The result is similar to the Document-View variant of the MVC pattern which integrates the View and Controller parts into one component leading to a simpler pattern.

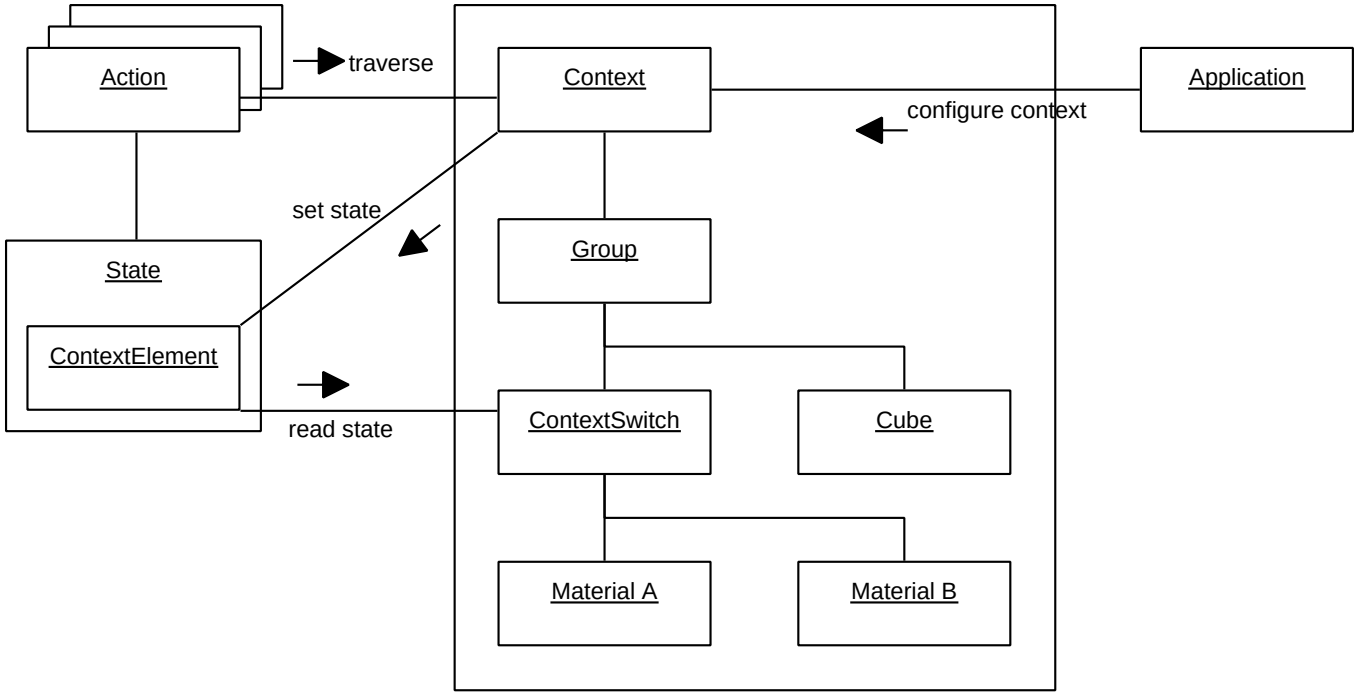


Figure 4.1: The context sensitive scene graph provides a dedicated state during traversal that can be configured by an application. A Context node sets the state and a ContextSwitch reads it out later during traversal. The state allows the decoupling of the context selection and using the context information.

While the general idea is simple to apply, the details of developing a scene graph based application can still be dauntingly complex. Although the scene graph can represent all different visualizations an applications uses, the control over these requires detailed structural knowledge of the scene graph which increases in complexity as the scene graph grows. Therefore, relying on scene graphs to handle everything will not scale well with data size. As a solution, we propose to separate control over the scene graph from the actual structure. Using a new mechanism, the context sensitive scene graph traversal, allows to leverage the advantages of the scene graph while keeping scalability for large data sets.

4.1 Concepts

Traversal is the common method of computing results from a scene graph. An action traverses the scene graph by recursively iterating over all nodes and calling a method on each node. The overall result depends on the structure and content of the scene graph as well as on the performed operations during traversal. The structure is usually static and the traversal has a predefined order that may depend on the action as well.

Actions implement the Visitor pattern [40, p. 331] using a double-dispatch technique. Each type of action manages a table of functions for each type of node. Upon visiting a node, the action looks up the corresponding function and passes the node and itself as parameters. Thus, it allows to vary the operation executed depending both on the type of action and the type of node. The function table technique also allows to extend the framework with new nodes by adding a new entry to the function table of existing actions or with new actions by defining a new function table and appropriate functions.

4.1.1 Scene graph model for data storage

Scene graphs are designed to unify the application's data model with the visual representation. Typically they also offer a runtime object system supporting various levels of introspection, persistence through serialization to file formats and memory management. These additional benefits are exploited to great lengths within Studierstube. Distributed Inventor [44] and dynamic application management depend on these features. The textual representation of the scene graph allow rapid prototyping and simple development by writing scene graphs by hand.

The generality of a typical scene graph API poses a design problem. The structure of the scene graph influences the final outcome of any traversal. For

example, the order of material nodes setting the color of rendered geometry influences the rendered image. Thus, the structure of the scene graph usually reflects the constraints imposed by the desired visual outcome. Complex visual effects may require rendering of the same geometry but with different parameters for the graphics pipeline. Therefore geometry has to be duplicated or at least referenced in distinct places in the scene graph or within several independent graphs.

However, applications typically deal with objects that may have different presentations but that are treated as single entities that are created, manipulated and destroyed as one. Having different presentations scattered about a large scene graph is diametrally opposed to a simple data model for the applications. Even more so, it recreates the "dual database" problem on a higher level of abstraction. Whenever a new object is created, the application has to look up all locations where to put different presentations of it. Similarly, when the object is destroyed all references to it need to be removed.

Affecting changes in the scene graph requires detailed structural knowledge because of its hierarchical nature. Typically, an application needs to establish the path through the scene graph from the root node to the node it wants to change. In the Open Inventor library this is further acerbated by the left-to-right evaluation scheme which implies that also nodes that are not on the path influence the final presentation of the node the application is interested in.

Common solutions to the situation described above are the following:

- Computing an a-priori list of references to the nodes of interest and storing them in the application. Such a method introduces more dependencies between variables and components thereby increasing the complexity of the design.

- Labelling interesting nodes in the scene graph and searching for them at run-time. Following this approach delays the coupling until run-time and is also flexible with respect to changes in the scene graph. However, it incurs a run-time overhead because of the search during execution.

Scene graph APIs usually provide means of influencing the traversal behavior, which are not flexible enough. A static way of switching between sub scene graphs is implemented in a dedicated node that only allows traversal of selected children. Again the application requires knowledge about the identity or location of such a switch node to use the mechanism. Traversal can also depend on the type of action executed. Such an approach is usually only tailored towards the specific requirements and implementation of the action.

4.1.2 Context sensitive scene graph

To overcome the design complexities described above, we propose to add an independent *context state* to any traversal that can influence the traversal order or select from different sub-graphs of the scene graph to be traversed. In effect, the scene graph can be made to "look differently" for each traversal by setting generic parameters that are independent of the traversal itself. By enabling such a selective traversal order independently of the actual action used it becomes an additional orthogonal parameter usable by the application. Because the scene graph can integrate different views in one structure, it unifies the application's requirements into a simpler model. The interface for selecting different views becomes a high-level interface between the functional components of the application and the scene graph for adapting the scene graph to various uses.

The proposed generic context state is a generalization of the concept of state for traversals. The scene graph library tracks a state as a set of stackable data elements during traversal to enable the action to compute its results. Normally, the type and operation of these elements are fixed and tailored to specific nodes and actions. We augment the state with a general purpose element. The element is tracked for all actions and can be set and used by all nodes. Consequently, a set of additional nodes can influence traversal based on the new element but independent of the type of action.

The additional traversal element is called *context* and a scene graph annotated with context is a *context sensitive scene graph*. The context is tracked by an additional state that is processed during traversal. Because it is independent of the operation of any action, it is applicable to all actions in the same way.

The context state is modelled as a partial mapping from $\mathbb{Z}$ onto $\mathbb{Z}$. For each index $z \in \mathbb{Z}$ either a NIL value or a result $r = f(z) \in \mathbb{Z}$ is returned. The NIL value simply describes the fact that the mapping is not defined for this index. The mapping itself is implemented as a simple map data structure, where keys and values are signed integers. The NIL value is returned from the state, if the given index is not stored as a key in the map.

Context state needs to be set and traversals or nodes need to react to the context. These two operations are embedded in special nodes that are part of the scene graph. A dedicated property node allows the modification of the context state during traversal and implements the following operations on the state:

ADD inserts a set of (key, value) pairs into the current context. Older entries with the same keys are overwritten by the new values.

SET sets the context mapping to a given set of (key, value) pairs after deleting all former entries.

REMOVE deletes a set of entries defined by a set of keys from the map. If an entry for a given key is not present, the key is ignored.

CLEAR resets the context to the default state which contains no entries at all.

Other nodes react to the current context during traversal. It would be possible to directly map different states to different behavior such as selecting a color from a given list, based on the value of certain index in the context state. However, this requires implementing a new node for every aspect that the application would like to influence by setting the context.

A more general approach is to use the structure of the scene graph itself to describe the different behaviors associated with the context. A dedicated group node can have children nodes that represent the different options available for a given context. Such a context-aware group node will traverse only children indicated by the context that is active when the traversal reaches the group node. The traversal behavior is generally independent of the action that is executed. The type of action can however be modelled as an entry in the context itself and therefore be taken into account as well.

In effect the context state adds a number of new parameters to the double-dispatch between node type and action type to create a multiple-dispatch invocation. The context-aware group node switches between different sub-scene-graph based on an arbitrary number of states to extend the fixed double-dispatch scheme of the original scene graph.

4.2 Implementation

The implementation of the context sensitive scene graph is straightforward based on Open Inventor's built in extension mechanisms. Open Inventor stores the state during traversal in a set of stacks comprised of individual elements. For each type of element a dedicated stack tracks the current value of the element. The framework provides for extending the traversal state with new elements independently of any action and to enable the use of these elements during all actions. A more detailed exposition on using this mechanism can be found in [117], chapter 2.

A new element *SoContextElement* was implemented that stores the current map describing the context. Accessor methods to add, set, remove

elements to and from the context and to clear it were implemented together with the necessary interfaces of the framework. Moreover, the use of the new element together with every action was enabled.

A dedicated property node *SoContext* was created that modifies the context during traversal. The node uses the element's accessor methods to update it according to its own parameters which are set using a number of fields of the node. The exact specification of the node is given in Figure 4.2. The enumeration field mode can take the values `ADD`, `SET`, `CLEAR` and `CLEAR_ALL` which correspond to the four operations on the context. The multiple value fields index and value denote (key, value) pairs for the `ADD` and `SET` operations while the `CLEAR` operation only uses the index field.

```
SoContext {
    SFEnum  mode     ADD
    MFInt32 index    []
    MFInt32 value    []
}
```

Figure 4.2: Specification of the SoContext node. The first column specifies the type of the field, the second column the name and the third the default value. index and value define the (key, value) pairs and mode the operation to execute.

Two nodes react to the current context during traversal by limiting the traversal to subsets of their children. The *SoContextSwitch* node uses the entries in the context map to compute which children to traverse (see Figure 4.3 for the full specification). The field index sets the key to use in looking up a value in the context map. The returned value is then interpreted as the index of the child to traverse. The values -1 and -3 are interpreted to traverse none or all children. The field defaultChild sets the index of the child to traverse, if the index is not present in the context and is therefore mapped to the NIL value.

```
SoContextSwitch {
    SFInt32 index          INT32_MIN
    SFInt32 defaultChild   -1
}
```

Figure 4.3: Specification of the SoContextSwitch node. The columns are the same as in Figure 4.2. index specifies the entry in the context to read out and defaultChild the behavior if the index is not present.

The *SoContextMultiSwitch* implements a more complex behavior (see Figure 4.4 for an exact specification). Instead of only specifying a single child to traverse, it allows to specify a family of sets of children to traverse further.

These are indexed starting with 0 and increasing the index by 1 for each set. The context state is then used to index into the family of sets and the resulting set of children is traversed.

```
SoContextMultiSwitch {
    SFBool  ordered         TRUE
    SFInt32 index           INT32_MIN
    MFInt32 whichChildren   []
    MFInt32 numChildren     []
    MFInt32 defaultChildren [-1]
}
```

Figure 4.4: Specification of the SoContextMultiSwitch node. The columns are the same as in Figure 4.2. whichChildren and numChildren specify different subsets of children. index again defines the entry in the context to specify the subset to traverse.

The field index again specifies the index into the context map to use. The field whichChildren specifies the sets as a concatenated list of the children's indices for each set. To delimit the individual sets in the list the field numChildren contains a list of the size of each set. The field defaultChildren specifies the subset of children to traverse, if the given index is not set in the context map. Finally, the field ordered specifies if the given indices in result set should be traversed in the order they are specified in the fields or should be sorted to be traversed in the left-to-right order of the scene graph.

4.3 Decoupling of model and control

The nodes described above allow the construction of scene graphs that can be controlled without detailed knowledge of their structure.

In a simple setup SoContextSwitch nodes are embedded throughout the scene graph to only traverse partial sub-graphs. For example, a scene graph might store different representations of objects locally for each object separately but as children of SoContextSwitch nodes (see Figure 4.5). The nodes are configured to use the same index and the order of the children corresponding to different presentations is the same for each object. Then an application can control the presentation by manipulating a single SoContext node above the scene graph to set the common index to the desired value.

Several of such indices can be overlaid to produce a matrix like structure of options. Typically combinations can be arranged by simply serializing SoContextSwitch nodes configured with different indices for different aspects (see Figure 4.6). For example, one index could select the color of an object and a second one the render style. Because they can be set independently,

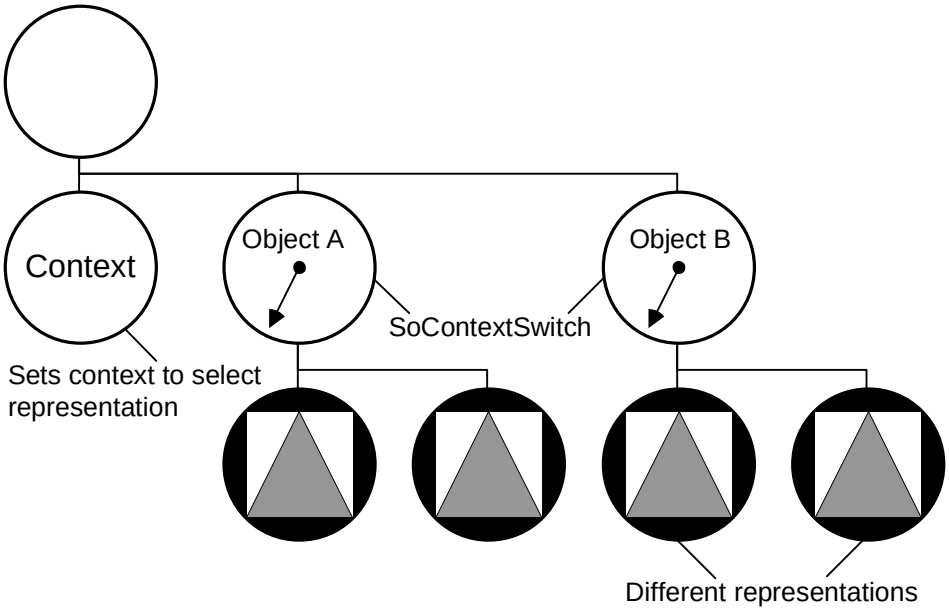


Figure 4.5: A context sensitive scene graph to select between differen representations of a set of objects.

the switches can be arranged in any order and do not depend on each other. Another approach is to combine the switches in a hierarchical structure. The top switch is controlled by one index and selects among different sub-switches which are controlled by a second index. Sub-switches can also be reused between different top switches or branches of top switches to reduce the number of nodes in the scene graph. Such an approach is enabled by the DAG nature of scene graphs.

More complex computations can be implemented by alternating switches and SoContext nodes in the scene graph. Then, the choice of one or more indices can influence the setting of another set of indices, implementing a general mapping from a tuple of indices to another tuple. Recursively building such mappings can lead to powerful computations by simply arranging scene graphs in the appropriate way.

The proposed mechanism shifts the complexity of managing multiple presentations from the application code to the scene graph data structure. However, it also enforces a unified approach to dealing with multiple representations and dynamic switches between them. Automatic methods of generating the graph can be applied to cope with the added complexity in the scene graph. Refer to chapter 5 for an in-depth discussion of such an approach.

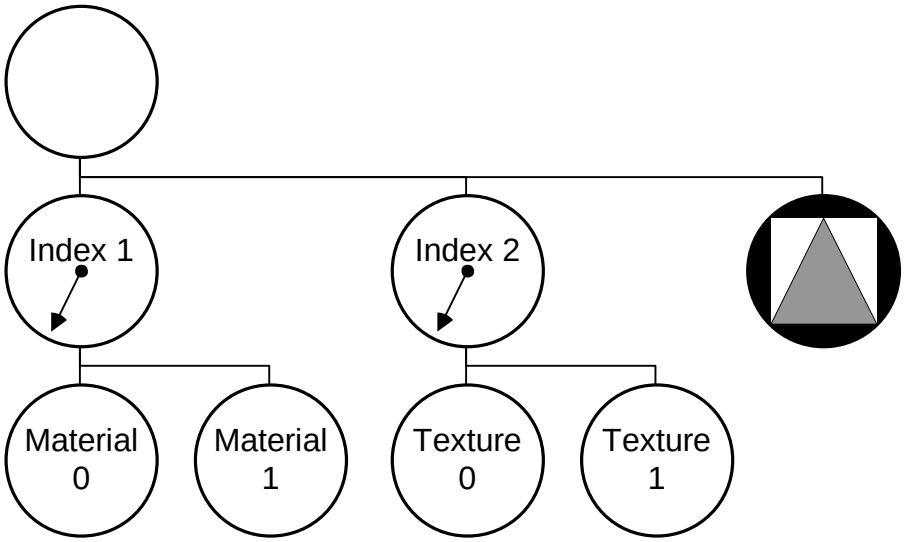


Figure 4.6: A context sensitive scene graph to select between multiple options at the same time to allow different combinations.

4.4 Results

The general mechanism provided by context sensitive scene graphs is used in the Studierstube framework to implement functions ranging from providing system level information to supporting the needs of individual applications. Some demonstrated uses are described in the following sections.

4.4.1 System management in Studierstube

Applications within Studierstube need different system level information to implement their behavior. Displayed information can depend on the user, left or right eye, the DIV group used to distribute the information and many more parameters. Such information can be transported in a traversal independent manner using the context of a traversal.

Studierstube defines a set of well-known context indices as parameters (see Table 4.1). The information about the user id and the selected buffer is only available during a render traversal, because it is not applicable for other actions.

Several nodes in the Studierstube framework use the system specific context information to implement adaptive presentations. The *SoWindowKit* node draws window borders in different colors for users depending on each user's window focus. The focus color can be configured for each user individually. The SoWindowKit node uses the user id information during a render

Index name	Value	Description
User	-1	user id associated with the current output window.
Application	-2	application id of the application that contains the current node.
Window	-3	window id of the window widget that contains the current node.
Eye	-4	a value LEFT or RIGHT specifying into which buffer the current render traversal writes.
DivMode	-5	the value MASTER or SLAVE specifying the mode of the application the current node is in.

Table 4.1: Context information provided by the Studierstube framework.

traversal to index into an array of color values storing these user dependent colors and sets the window borders' color to the user's focus color. The context information separates the control for which user is rendered from all the instances of windows that might be present in a scene graph.

Another optimization concerns the use of the application id. Widgets need to implement special behavior in a distributed setting and need to change field values without the propagation of these changes over DIV. Fields are selectively put into and removed from filters in the DIV mechanism to disable or enable propagating changes. Therefore, a widget needs to know the instance of the DIV object it is subject to. The instance can be retrieved from the framework based on the application id provided via the context mechanism with a simple look-up. This is much more efficient than the alternative, namely to search the scene graph from a common root node for the widget node and then walk the computed path to look for any DIV objects that are ancestors of the widget node.

4.4.2 Signpost - attributing of a general model tree

The Signpost application described in [81] is an indoor navigation application based on a wide area tracking solution (see also section 3.5.3). It provides navigational hints to a user roaming a building. The user interface provides a number of graphical representations of the buildings geometry:

World in Miniature A world model with the current location of the user floats in a head stabilized position in front of the user.

Augmentation of room geometry The room geometry of the building can be augmented by a wire frame representation.

Navigation aides The navigation system highlights doors the user has to pass through.

A core requirement of the software architecture was to build a modular, extensible system that allows to easily add application features. To address this issue we separated the building model from the components that are responsible for different presentations and interactions. A dedicated *model server* component holds the scene graph of the building model and presents an interface for client components to reuse it in their own scene graph for rendering. The interface also provides hooks into the model scene graph, so that client components can add their own rendering style nodes to the model scene graph in a controlled manner. The model scene graph itself relies on the context sensitive features to implement the control mechanisms.

The model server's scene graph relies on two dedicated nodes, *SoBAUBuiling* to model a whole building and *SoBAURoom* to model a single room contained within a building. The scene graph encapsulated by an SoBAUBuilding node contains a switch node holding all the rooms of the building. The switch node allows to traverse only a subset of all rooms to limit the rendering to a certain selection. The SoBAURoom node contains a scene graph describing the geometry of a room separated into different sets of polygons for wall, ceiling, floor and other surfaces. Also portal geometry representing doors and connections to other rooms are modelled separately. Every set of polygons is rendering individually with its own render parameters. These parameters are set by dedicated sub scene graphs that are controlled by a SoContextSwitch node (see Figure 4.7).

A client of the model server obtains a reference to the top level SoBAUBuilding node and a unique index for the context sensitive traversal. Then it inserts its own rendering styles into the SoBAURoom nodes as desired. Finally, the client's scene graph reuses the SoBAUBuilding node but sets the context so that only its own rendering styles are used. The described mechanism is again encapsulated in a dedicated node SoBAUClient which takes care of the details of obtaining the index and inserting the style nodes. It is configured using an SoBAUStyle node that holds the different styles for walls, ceiling, floor and portal polygons. Moreover, it also provides an interface to set the top level SoMultiSwitch node to only traverse a subset of the rooms.

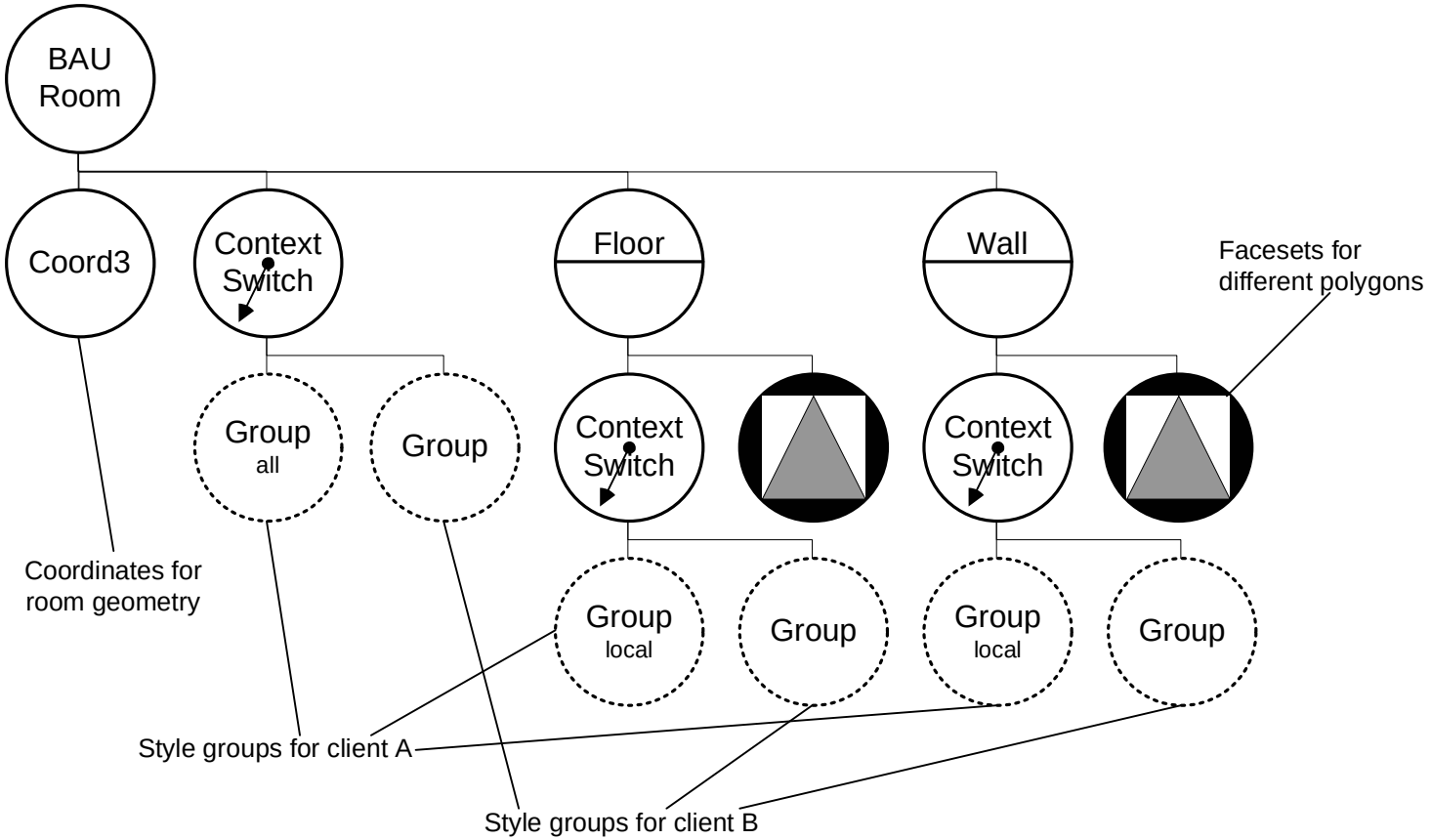


Figure 4.7: The scene graph contained in an SoBAURoom instance uses context sensitive traversal to select between different rendering styles. Clients of the SoBAURoom node reserve a certain index value and add their own rendering style to the style groups below the SoContextSwitch nodes. By setting their reserved value as the parameter of the SoBAURoom context index, the instance of SoBAURoom will use their customized rendering style.

The different client components implementing the WIM, augmentation or navigational hints all use one or more SoBAUClient nodes in their scene graph. They all can select different presentations via the SoBAUStyle parameter node while being independent of the actual building scene graph. For example, the augmentation component uses an SoBAUClient node with styles to render the polygons in wire frame mode with different colors for walls, ceiling, floor and portals. A second SoBAUClient node can be switched into the scene graph to render filled polygons into the Z-buffer only before rendering the wire frame to achieve a hidden line effect.

4.5 Summary

The context sensitive scene graph traversal introduces aspects of declarative programming using the structural aspects of the scene graph. While the original scene graph approach already represents a powerful application of the Composite pattern [40, p. 163], the declarative programming possible by constructing a scene graph was limited to a fixed visual appearance dictated by the structure. By adding an orthogonal method to vary the structure another dimension of programmability was added.

The context implements a generic multiple-dispatch function for scene graph traversal which can be accessed via simple declaration of additional scene graph nodes and subgraphs. Therefore, it allows a declarative style of programming which easies the implementation work for applications.

The structural complexity of a context sensitive scene graph can increase dramatically with the number of objects and number of features that are varied. This development can be countered with automated methods. A data-driven program architecture will allow to generate the required scene graph from given data and a template pattern to apply to the set of objects passed in. The programmer then only needs to create the template instead of manually applying the same structure to all objects. A possible architecture that realizes the proposed method is presented in the next chapter.

Chapter 5

Data management

Mobile augmented reality applications require a detailed model of the user's environment including semantic and contextual elements related to the (potentially dynamic) real environment. Therefore AR models are more difficult to produce and maintain than typical virtual reality and visual simulation applications, which concentrate on visual fidelity of a purely virtual model, even if produced from the measures taken from a real object. Also, detailed information on the environment is often only available in legacy systems and needs to be extracted and transformed to be useful for the AR application.

However, AR not only requires integration of a wider variety of data sources to build interesting applications, it also creates new types of content. For example, in the virtual showcase project [22], light maps are used to provide detailed lighting effects on real objects. In AR, geometrical models of real objects are frequently not used for visualization purposes, but for dealing with occlusions, rendering of shadows, interaction, and vision-based tracking of real objects.

In an ubiquitous computing environment, multiple applications and users need to share the same environment model. These applications should be based on a common database which should also be capable of shared access for modification and updates. Since different applications will potentially not work with the same abstraction or representation of the model data, it may be difficult to keep the model data consistent if changes cannot be uniquely traced back to the data source.

To address the complex modelling and data handling needs of ubiquitous augmented reality applications, we present the concept of a 3-tier application architecture based on XML [23] as the enabling technology. A central XML-based database stores a common model used for all applications. Required data is transformed and imported from different sources using common XML tools. Once in the database, the data can be maintained more easily and

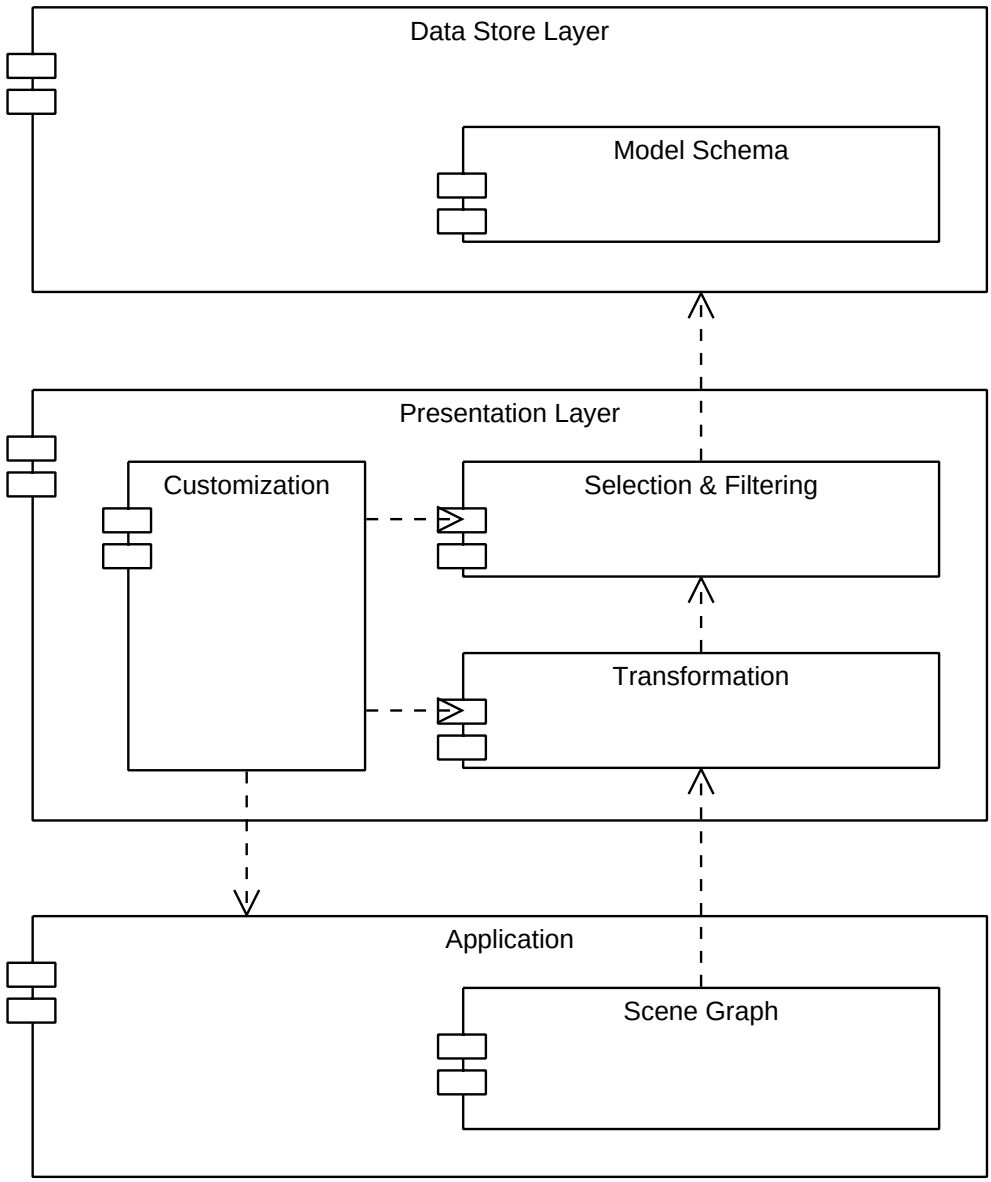


Figure 5.1: The proposed three-tier architecture. The data store layer stores data in the model format. The presentation layer contains transformations that mediate between the applications and the data store and can be customized to fit different output requirements.

application or domain specific preprocessing operations can be applied. At run time, the client applications query the central store for relevant pieces of information (e.g. based on the current location). Before the data is delivered to the client, it is transformed from its original form to one directly useful to the client application. These transformations will often cull the model to return only those aspects of the data relevant to the application.

5.1 Related work

Typical AR demonstrations work with small data sets that have been entered manually and do not require data warehousing. To our knowledge, there has been little work done on data management techniques for large AR models. One piece of related work by Julier et al. [52] addresses the issue of selecting appropriate data for display, but from a user's point of view rather than that of the application. Höllerer et. al [47] describe the use of a database and description logic based meta-data to store a model of a building floor which is annotated with meta-data for navigation target selection. The sentient computing project [3] uses a CORBA run-time infrastructure to model a live environment as distributed software objects where locations and attributes of objects are updated permanently. Newman et al. [67] describe a set of AR applications based on this infrastructure.

The Nexus project [87] is unique in that is specifically deals with the software architecture required for ubiquitous location-based applications and providing abstract interfaces to position data to such applications. Although the project does describe some preliminary augmented reality applications [41], it does not focus on AR applications interacting with complex information structures. Glonass [33] also describes a software architecture to distribute context information but does not provide the extensive models that advanced AR applications require.

The geographic information systems (GIS) community has a lot of experience with storing and manipulating large scale geometric data [98]. However, the current data sets are still mostly dealing with 2D features without complex interrelations. A current trend towards 3D models for communities such as the City of Vienna [35] will hopefully provide a better basis for mobile AR applications.

5.2 Concepts

The architectural concept is based on a 3-tier model [2]. The first tier is a central database storing the overall model (see Figure 5.1). The second tier mediates between database and application by converting the general model from the database into data structures native to the respective application. The applications themselves are the third tier and only deal with data structures in their own native format.

The architecture provides the usual advantages of the 3-tier model. A common data model and data store reduces the amount of redundancy in the stored data required for different applications and allows centralized and efficient management of this data. The middle layer separates the presentation issues from the actual data storage and the applications. Thus the applications can be developed using efficient data structures independent of the actual storage format. Moreover, the transformation can be adapted to changing data formats or processes without touching either the application or the storage back-end, because it is a distinct entity separated from both.

Our architecture differs from to the traditional 3-tier model for client-server applications in some aspects. The second tier is rather passive and focuses only on data translation and will not provide extensive application functionality. Because our third tier on the client itself needs to provide interactive realtime feedback, the argument for a thin client offering only user interface functions and no application processing does not hold anymore. Calculations by the application need to be available to the user interface instantaneously, which was also a driving force in the unification of application data and graphics data in the scene graph data structure.

We propose to use such an architecture and build upon XML technology, leveraging recent developments in the web development community. The proposed architecture is very common in this area and directly supported in a number of products, either open-source or developed commercially. The use of XML has a number of advantages for our task:

- A hierarchical data model fits well to our general spatial model. Rather than using a flat enumeration of building representations, a hierarchical model can represent several levels of a spatial hierarchy, from districts and streets down to rooms within buildings and other detailed geometrical data.

- While a document and file-oriented approach is generally sufficient for research prototyping, it obviously lacks scalability. More powerful storage solutions are required for real applications. Some of these exist in

the form of XML databases [105, 66, 99]. As XML technology is generally aimed at compatibility, the tools and APIs used for prototyping are directly supported by commercial XML products, and the transfer to a production system is greatly simplified.

- XML tools such as XSLT [30] allow rapid prototyping and development of import, transformation and export tools to and from the data model. Such tools focus on the functional aspect of the transformations and reduce the overhead work to implement parsing and generation of data structures.

- Parsers and generators exist for a wide range of programming languages, and allow applications and tools to use the most appropriate language for the task.

- Standards for meaningful descriptions of data exist, on a syntactic level such as RDF [64] as well as on a semantic level for meta-data such as the Web Ontology Language [65] or the Dublin Core [114]. This allows to define and use ontologies to support semantically rich queries and interactions.

5.2.1 Modelling

At the heart of our architecture lies a data model that encompasses the requirements of all applications. Care was taken in keeping the model extensible so that new data could be integrated during the development. This data model is described by an XML schema [37].

The model should fulfill a number of key requirements:

- Geometric representations and hierarchies need to be stored in the model.

- Interaction with other data schemas should be possible to maximize reuse of already established knowledge presented in the form of these schemas.

- Extensibility for new applications and data types with fall-back options for generic processing is important.

The model is based on an object-oriented approach using a type hierarchy to define types of objects. The root type is called *ObjectType* and contains an id and a generic annotation subelement that can be used to store any XML tree. All data types defined in the model are derived from this type. The

SpatialObjectType adds pose information and a geometrical representation to the super class. We further derive the *SpatialContainerType* that adds a children subelement to aggregate entities of type *ObjectType* for hierarchical composition.

From the three base types, we derive a number of application specific types that are used in the actual data files. The *Object, SpatialObject* and *SpatialContainer* elements are used for general purpose data and correspond directly to the base types (see Figure 5.2). Applications can define additional types derived from the base types to provide more specific information. For example, we define a special *Waypoint* element used by an outdoor navigation application which has a specific subelement to define neighboring waypoints connected by a path. Because elements refer back to their base type, an application can always provide a reasonable fall back behavior if it encounters an unknown derived application element. The Nexus project uses a similar structure to model their data types.

The XML tree is interpreted in the standard geometrical way, by defining a child's pose relative to its parent. We chose this mapping to support conventional modelling of visual data as directed acyclic graphs or trees. However, the open XML based format is not bound to any particular visualization tool or platform, and affords the definition of other than spatial relations by using relational techniques such as referring to object ids.

The annotation subelement of the abstract root type can be used to model free form data or to augment pre-existing types with extra information. This allows us to use more flexible technologies to annotate the objects in our model.

5.2.2 Data handling

Having defined a model and data format, there are a number of tasks and tools necessary to fill the database, transform and manipulate the data and finally make it available to the user by developing appropriate applications. The typical tasks include the following:

Import Extract information from source data formats based on XML or other formats and map it to the data model. Non-XML source formats also require the combination of an appropriate parser with an XML generator to map the foreign format to XML.

Maintenance Maintaining a model requires the application of filters and transformations on data stored in the model format.

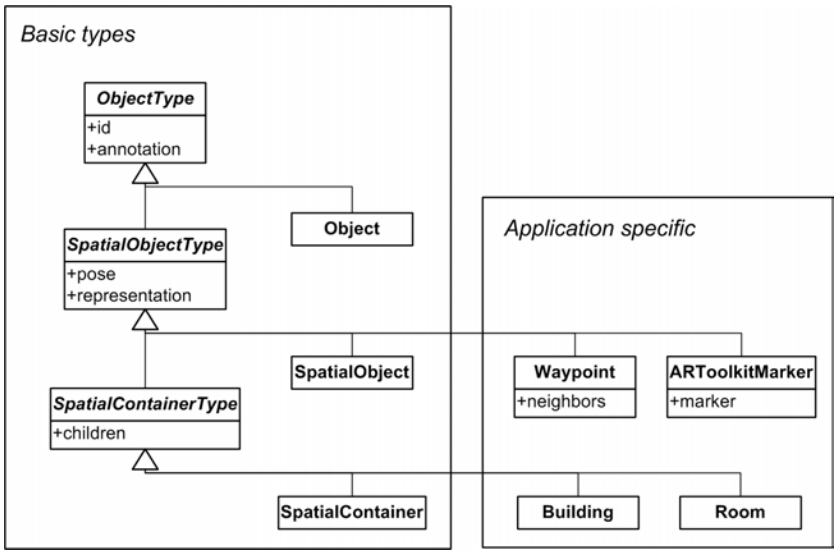


Figure 5.2: Overview of the type hierarchy in the model schema. A set of basic types can be used for general modelling. Applications may derive additional types for specific requirements.

Export In the last step transformations are applied to retrieve relevant application data from storage and generate data structures for the applications. As described in section 2.5.4, applications are implemented as Open Inventor scene graphs. Each application uses a custom XML stylesheet to directly generate the required scene graph and additional data structures from the general model.

The use of a separate step to transform the data into the application format has a number of advantages. It separates the general data format from the application specific data structures and provides an interface between both. It also provides a point for customizing the presentation independently of the application, similar to the way traditional cascading stylesheets work for HTML. As the stylesheet generates the actual graphical content, it can adapt it to different output requirements or profiles.

5.3 Implementation

We will now describe the implementation of the model schema and typical data transformations in more detail covering the first and second tier of the architecture. The third tier itself is already the unmodified AR application that directly processes only the generated data structure and is therefore of no further interest at this point.

5.3.1 Schema definition

The definition of the model schema is based on the structural XML Schema definition [109]. The implementation of the type system described in section 5.2.1 follows the concepts of the XML Schema definition and tries to leverage the expressive power of schemas. We defined an XML language called *BAUML* (Building AUgmentation Modelling Language) to describe our model data.

Both our BAUML model and the XML Schema definition contains the concept of a type as a class of data structures that conform to a certain schema. To avoid confusion in the following discussion we will call types contained in the BAUML model *model types* and XML schema types only *types* or *complex types* as appropriate.

The basic model types are implemented as complex types of the Schema language. Such a definition allows the derivation of several element tags that share the common attributes and subelements of the basic model types. The complex types allow a hierarchy of types which is used to model the relationships between the basic model types. Any new general model types would be added on this level to the modelling language.

There are only three basic model types defined so far: *ObjectType*, *SpatialObjectType* and *SpatialContainerType*. Figure 5.3 shows their detailed definition. The SpatialObjectType is derived from the ObjectType and the SpatialContainerType is derived in turn from the SpatialObjectType forming a simple linear derivation hierarchy.

The ObjectType only models general properties applicable to all objects in the model. The only properties defined are a unique identifier and a subelement called *annotation* which is defined to allow inclusion of arbitrary subelements to allow extension of the model by combining different schemas.

The SpatialObjectType further specializes the ObjectType to include properties applicable to an object with a spatial footprint. They consist of two parts, a *representation* element storing a geometric description of the object and a *pose* storing a location in three-dimensional space. Both elements are typed to specific helper types that describe the format of the contained information.

The SpatialContainerType is a further specialization of the SpatialObjectType to add hierarchial composition of complex geometric structures. It implicitly defines a coordinate system and adds a *children* element that may contain in turn elements derived from ObjectType. All geometric information contained in such child elements is interpreted relative to the coordinate system of the parent SpatialContainerType. The coordinate system is transformed by any pose defined in the pose element.

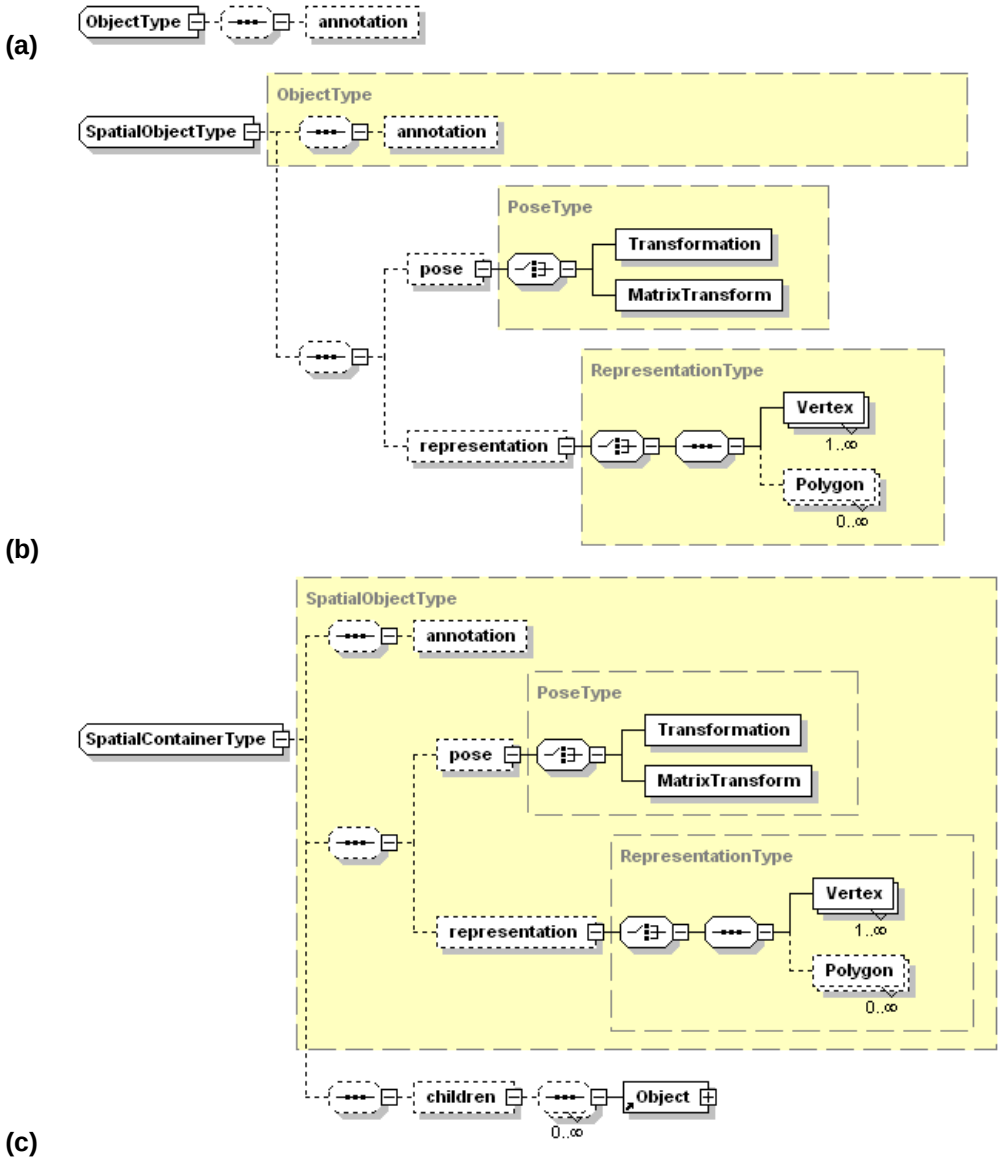


Figure 5.3: The structure of the three basic model types. Subelements describing complex properties are modelled again as dedicated helper types which are not part of the type hierarchy.

Application developers need to create instances of the basic model types. Therefore, the BAUML language defines elements derived from each of the complex types defined for the basic model types. In addition to that, the application specific elements are also derived from the most appropriate complex types and augmented with further attributes or subelements to afford their application's requirements. To explicitly model the derivation relationship between the basic types and the instance elements, a compulsory attribute is added that stores the name of the complex type the element is derived from. This mechanism allows XML processors to work on the information modelled by the generic basic types alone as a fall back solution for unknown element types. Refer to appendix A for a complete definition of the BAUML language.

5.3.2 Transformations

The second tier is the presentation layer of our architecture and provides the transformations from the generic model data into application specific data structures. In our implementation the target data format were scene graphs described in the ASCII file format of Open Inventor. An application loads the generated scene graph as a parameter and operates on it.

The actual transformations where implemented in XSLT [30] which is a language for transforming XML documents into other XML documents. Any text file can be generated as a special case of an XML document. XSLT is a fully functional language that is used to describe the actions an XSLT processor will execute on an input document to generate an output document. XSLT represents a powerful implementation of the concept of data driven programming because the structure and content of the input file drives the control flow of the XSLT processor.

The basic item of operation is a *template* which specifies a pattern to match on the input document and a template XML fragment which is written to the output file. The XML fragment can be customized with parameters and data taken from the matched location in the input document. In addition to that, templates can be called with parameters to provide a fixed control flow as required. Libraries of templates can be included from external style sheets to provide reuse and modularity in XSLT development.

As an example for the typical transformations implemented in XSLT we will describe a generic style sheet that generates a basic scene graph describing the hierarchy and geometry of all objects in the given BAUML input file that have a geometric representation.

The style sheet creates an Open Inventor scene graph that represents the hierarchical structure of a model stored in a BAUML file. It transforms

all elements derived from the SpatialObjectType to a sub-scene-graph consisting of the local transformation stored in the pose sub-element and an IndexedFaceSet describing the geometry stored in the representation subelement. Elements derived from SpatialContainerType are transformed to Group nodes containing the above information and the transformations of all child elements. The result is an Open Inventor file which can be displayed and reused in Studierstube applications.

The starting point of the style sheet is a template which matches the root node of the input XML file. The template generates the necessary header information for an Open Inventor file and then applies template matching again to all its child elements. Child elements do not match the root template anymore but they can match one of three templates which operate on elements derived from ObjectType, SpatialObjectType and SpatialContainerType. Each of these templates generates the Open Inventor representation of the stored elements.

These templates reuse named templates to generate the scene graph description for the pose and representation subelements if present. Figure 5.4 shows the structure of the template acting on elements derived from SpatialObjectType. The template itself only generates a Separator enclosing the actual representation which is generated by the named templates that are called inside. The current element is passed implicitly as a parameter to the called templates.

```
<xsl:template name="TransformSpatialObject"
  match="*[@baseType='SpatialObjectType']">
<xsl:call-template name="GenerateDefName"/>
Separator {
    <xsl:call-template name="GenerateTransformation"/>
    <xsl:call-template name="GenerateFaceSet"/>
}
</xsl:template>
```

Figure 5.4: The basic template to transform elements derived from SpatialObjectType. Some technical details are omitted for clarity.

Figure 5.5 and 5.6 give two examples of such named templates that generate the representation. The first template *GenerateDefName* shown in Figure 5.5 simply looks for the *id* attribute of the current element and if present will generate an Open Inventor node name. Such a name consists of the keyword `DEF` and the name stored in the id attribute.

The template *GenerateFaceSet* shown in Figure 5.6 generates an IndexedFaceSet to render the geometry of an element stored in the representation subelement. If the representation subelement is present, it will output an In-

```
<xsl:template name="GenerateDefName">
  <xsl:if test="@id">DEF <xsl:value-of select="@id"/>
    <xsl:text> </xsl:text>
  </xsl:if>
</xsl:template>
```

Figure 5.5: The basic template to generate DEF names for the grouping Separator nodes.

dexedFaceSet node. The vertices of the model are stored in the vertextProperty field and are generated by iterating over the *Vertex* subelements of the representation element. The attribute *position* contains the 3D coordinates of a vertex and its content is simply copied to the output. In a second step the polygons are described as set of indices into the vertex array. Again the template iterates over all *Polygon* subelements and copies the indices over to the output. Between processing individual polygons it also writes out a stop value to signify the end of an index set.

The template also takes care of creating the correct syntax for the final Open Inventor format by adding quotes and commas as appropriate. Therefore, both the structure as well as the syntactical considerations of a transformation are captured by the XSLT language.

```
<xsl:template name="GenerateFaceSet">
  <xsl:if test="representation">
  IndexedFaceSet {
      vertexProperty VertexProperty {
          vertex [
         <xsl:for-each select="representation/Vertex">
           <xsl:value-of select="@position"/>
           <xsl:if test="not(position()=last())">, </xsl:if>
         </xsl:for-each>
          ]
      }
      coordIndex [
      <xsl:for-each select="representation/Polygon">
        <xsl:value-of select="translate(
           normalize-space(@vertices),' ',',')"/>, -1
        <xsl:if test="not(position()=last())">, </xsl:if>
      </xsl:for-each>
      ]
  }</xsl:if>
</xsl:template>
```

Figure 5.6: The basic template to generate an IndexedFaceSet representing the geometry stored in the representation element.

The following example demonstrates the use of the described templates. A simple SpatialObject is transformed into a scene graph using an IndexedFaceSet to visualize the geometry. Figure 5.7 gives the BAUML representation of the object, a simple cube, and the resulting Open Inventor scene graph created by the transformation. The overall scene graph is created by the TransformSpatialObject template that writes out the encapsulating Separator and then calls the helper templates at the appropriate positions.

The GenerateDefName template is called before the Separator string is written out to generate an appropriate DEF name, if an id attribute is present. Then another helper template is called to create the Transform node representing the transformation stored in the pose subelement. Finally, the GenerateFaceSet template uses the geometry information in the representation subelement to create an IndexedFaceSet node. The subnode VertexProperty contains the vertex positions and the face set the polygon definitions.

5.4 Results

In this section, we will describe results from a number of mobile augmented reality applications we implemented. All applications are based on the model language *BAUML* which was described in more detail in the former sections.

5.4.1 Indoor navigation

The Signpost indoor navigation application [81] heavily uses the described architecture to leverage the advantages of data-driven programming. At the heart of the application lies a building model that is reused between different components of the application (see section 4.4.2). In addition to that it uses a tracking system based on recurring fiducial markers that are distributed throughout the environment (see section 3.5.3). Therefore the application requires two different data sets, one for rendering and one for tracking configuration, both of which are based on the same model.

A number of advanced tasks need to be supported by the data management besides these basic data requirements. The tracking system is based on associating recurring markers with measured positions according to a certain scheme. To support extending and changing the model, it is necessary to be able to compute the associations based on a partially fixed set and only for a subset of new markers.

```
<?xml version="1.0" encoding="UTF-8"?>                          #Inventor V2.1 ascii
<SpatialObject baseType="SpatialObjectType" id="MyObject">  ⟷   DEF MyObject Separator {
  <pose>                                                            Transform {
    <Transformation translation="1 0 0"/>                               translation 1 0 0
  </pose>                                                               }
  <representation>                                          ⟷        IndexedFaceSet {
    <Vertex position=" -0.5 0.5 0.5"/>                                  vertexProperty VertexProperty {
    <Vertex position="-0.5 -0.5 0.5"/>                                      vertex [  -0.5 0.5 0.5,
    <Vertex position="0.5 0.5 0.5"/>                                                  -0.5 -0.5 0.5,
    <Vertex position="0.5 -0.5 0.5"/>                                                 0.5 0.5 0.5,
    <Vertex position="0.5 0.5 -0.5"/>                                                 0.5 -0.5 0.5,
    <Vertex position="0.5 -0.5 -0.5"/>                                                0.5 0.5 -0.5,
    <Vertex position="-0.5 0.5 -0.5"/>                                                0.5 -0.5 -0.5,
    <Vertex position="-0.5 -0.5 -0.5"/>                                               -0.5 0.5 -0.5,
    <Polygon vertices="0 1 3 2" type="wall"/>                                         -0.5 -0.5 -0.5
    <Polygon vertices="4 5 7 6" type="wall"/>                                   ]
    <Polygon vertices="6 7 1 0" type="wall"/>                               }
    <Polygon vertices="2 3 5 4" type="wall"/>                               coordIndex [ 0,1,3,2,-1,
    <Polygon vertices="6 0 2 4" type="wall"/>                                            4,5,7,6,-1,
    <Polygon vertices="1 7 5 3" type="wall"/>                                            6,7,1,0,-1,
  </representation>                                                                      2,3,5,4,-1,
</SpatialObject>                                                                         6,0,2,4,-1,
                                                                                         1,7,5,3,-1
                                                                        ]
                                                                    }
                                                                }
```

Figure 5.7: An example for applying XSLT transformations to a BAUML model. The SpatialObject on the left is transformed into a scene graph with the appropriate transformation and geometry. The first arrow indicates the application of the GenerateDefName template and the second the application of the GenerateFaceSet template.

Consequently, we have identified the following tasks related to the data management:

- Compute a marker pattern distribution scheme based on the position of markers and already associated marker patterns.
- Generate a scene graph based model of the environment for rendering and interaction.
- Generate a tracking configuration based on the pattern distribution stored in the model.

Figure 5.8 gives an overview of the work flow involved in maintaining the model and generating the required data. Input to the model results in a modification of the building model. After such a manipulation the marker allocator tool updates the model itself by computing the marker pattern distribution and storing the model again. The resulting improved model is then the basis for any application relying on the model. In case of the Signpost application, the building converter tool takes the current model and generates the OpenTracker configuration file and the Open Inventor file for the final applications.

The Signpost applications themselves rely on the Open Inventor file generated by the building converter tool. The model server component reads in the scene graph described in the file and manages it for the other components. The required specialized scene graph structure described in section 4.4.2 was already generated by the building converter tool. Therefore the application code itself need not concern itself with the original generic model, but operates directly on the dedicated data structure. The generation of the specialized data structure is factored out into a dedicated tool. Therefore, the architecture decouples the application from the general model.

The tracking configuration is described by an OpenTracker config file. Again, the required file for a given building stored in the model is generated by the building converter tool. The tool creates the necessary structure of ARToolkit source nodes, transformation nodes and GroupGate nodes as described in section 3.5.3.

5.4.2 Information browsing

To implement a simple information browsing application for mobile AR systems we combined the BAUML language with a simple XML language to describe various aspects of objects stored in the BAUML language. The secondary language defines a set of elements storing various attributes such as

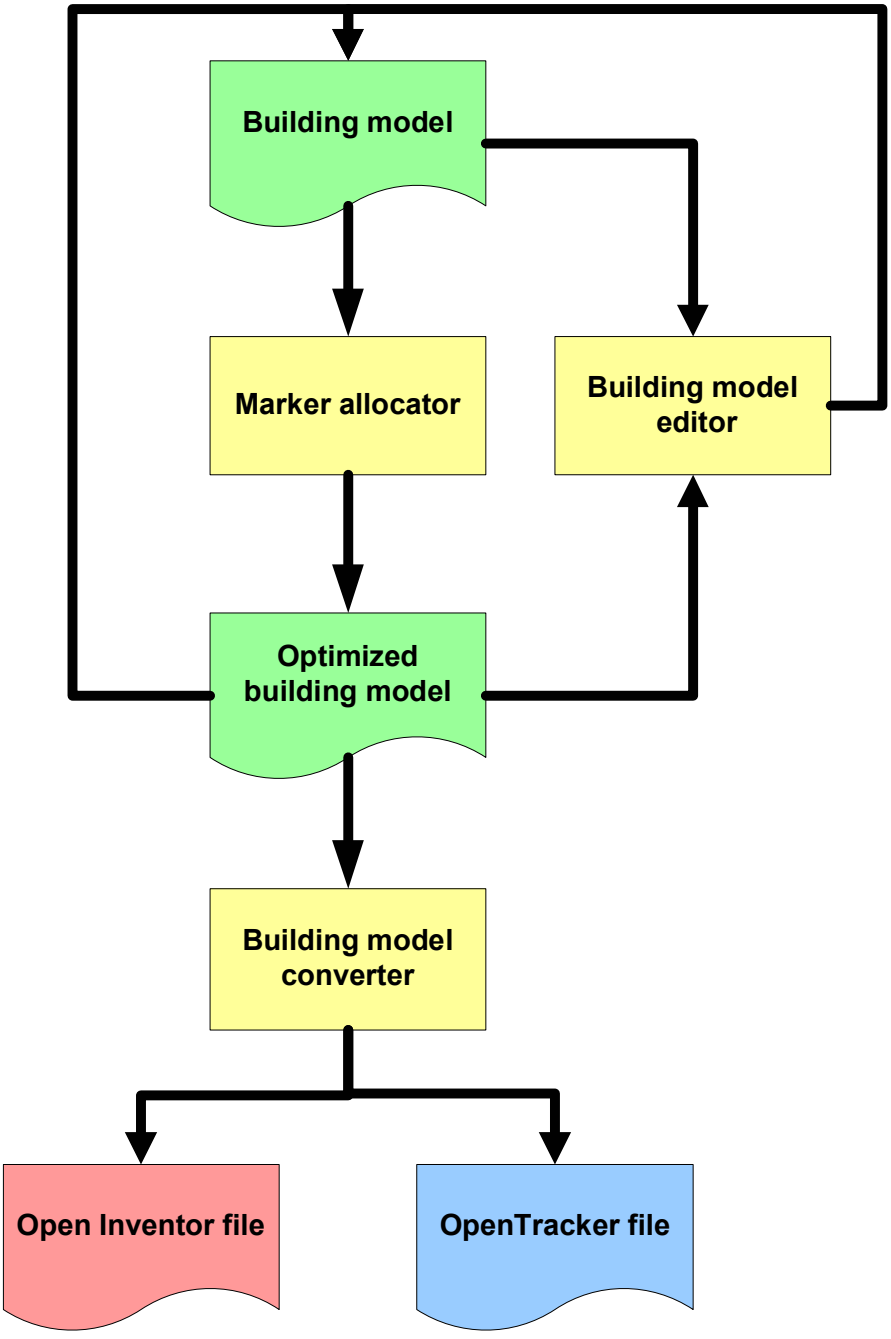


Figure 5.8: The work flow for data management in the Signpost application. The green shapes denote the BAUML model before and after the maintenance operation of associating marker patterns with positions. The output data sets are shown at the bottom. Different tools to process the data are shown as yellow boxes.

owner, a telephone number and a general description. Then we annotated the already existing Room objects within our building model with data formatted in the new language by adding the elements to the annotation element of the respective BAUML entries.

Furthermore dedicated transformation scripts extract the information and position it with respect to the containing BAUML object in the environment. The scripts generate a scene graph containing visual representations of the information. The final application itself simply renders the scene graph using the user's current view.

5.5 Summary

The proposed architecture and set of tools are based on established techniques in current computer science. However, we have found little evidence of using scalable data management techniques for large scale AR. There is a small set of related areas where indirectly applicable techniques are used: GIS databases are primarily aimed at 2D information and static, stationary use cases. Location-based services are aimed at abstract, text-oriented information and low-end devices. Visual simulation focuses on high-fidelity graphical representation and stationary displays. In this work, we have attempted to explore the combination of aspects of these areas into a scalable mobile AR database system.

Today's most common technology for storing data are relational database management systems (RDMS). However, we did not base our approach on a relational data model for several reasons. The hierarchical structure inherent in 3D data models is more directly mapped to an XML structure. One of the great advantages of XML technology is its self-description and self-organization by using schemas and namespaces. AR applications benefit from this flexibility, because data definitions can be decoupled by domain and developed independently without breaking application compatibility.

The basic extensibility of BAUML allowed us to merge new data with the general model without breaking any existing applications but still create meaningful associations within the model. The ability of XML technology to mix various orthogonal languages within one data set simplifies such local extensions of the data model.

In principle, these aspects can also be addressed with a relational model, but at the expense of imposing the organizational workload on the developer rather than the underlying tools. Nevertheless, an RDBMS provides a solid performance basis for implementing an XML storage system and will play an important part in any production system following the proposed architecture.

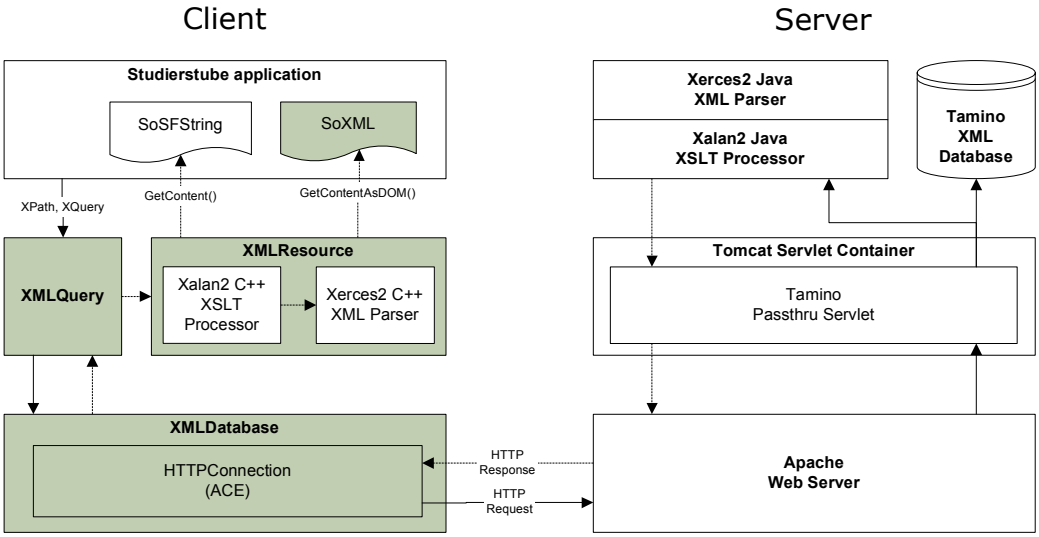


Figure 5.9: Design of the Studierstube XML database components. The server side provides a transformation layer on top of a native XML database. The client side provides an API to query and update the database and to transform result sets.

During the writing of this dissertation we have begun to develop a storage solution based on a native XML database. The basic system design uses a simple client/server architecture (see Figure 5.9). Transformations can be provided on both the server and client side. In the former case they are part of the servlet providing the connection to the XML database backend. In the later case, the client API allows to perform arbitrary transformations on the result sets obtained from the server.

Chapter 6

Managing collaboration

All but the most simple collaborative applications will require some level of distribution between different processes. A typical collaborative augmented reality application will consist of one workstation per user which drives the rendering and interaction for this user. A distributed system will then implement some sort of data sharing between the workstations to provide the collaborative experience for several users.

Studierstube applications deal with very different scenarios ranging from large virtual workspaces that are operated by several hosts to mobile and location-based applications and collaboration between disjoint workspaces. A system that is able to accommodate all such situations needs to be flexible and provide a general model to describe these situations. However, the complexity should not extend to the user or the application programmer. Therefore, the distributed system should work as transparently to the end user or application programmer as possible. Moreover, a generic model and API allows to develop different user interfaces for the required user intervention which are appropriate for the usage scenario.

A workgroup of hosts executing a collaborative session should be able to accommodate changes to its configuration, for example, to provide the current state of applications to late-comers. For more complex scenarios such as mobile augmented reality, ubiquitous computing and large or remote shared workspaces whole sets of applications need to be managed by appropriate forms of migration. Application migration requires that a running application instance moves from one host to another, while user interface and internal state are kept intact.

We found that most of these situations deal with the relationship between the location of one or more workspaces and a set of application objects contained in these workspaces. To accurately model these relationships we implemented a locale concept inspired by Barrus et al. [10] to explic-

itly model the relationship between locations and applications. The locale concept is supported by a runtime system management infrastructure that implements the basic operations to make the locale management transparent to the application developer.

6.1 Related work

Building collaboration aware applications that have true multi-user interface elements should be only slightly harder than building conventional applications or application programmers will be reluctant to do so [86]. In object oriented frameworks, a feasible approach is therefore to provide components (widgets) that have built-in collaboration facilities, and can readily be (re-) used by application programmers or even retro-fitted to legacy applications [13]. The Studierstube framework offers similar possibilities by using transparent distribution and multi-user 3D widgets.

Besides static workgroup topology, some research also considers dynamic changes to the workgroup and client migration [17, 28], in particular accommodation of late-comers that need to be updated on the current state of the session. Two competing solutions are replaying all previous events to the newcomer vs. transmitting a current image of application state. Because the history of previous events can become arbitrarily large despite potential for compression [29], recent work favors the image copy approach [110]. This is partly due to novel architectures that make it easy to marshal complex runtime structures [14], and is also the foundation for our application migration facility. It should be noted, however, that this kind of migration in a constrained runtime environment is not comparable to full operating system level process migration.

Finally, several projects on collaborative user interfaces inspired our work. SharedSpace [19] features collaborative augmented reality, but is limited by its lack of an underlying distributed system. CRYSTAL introduces multitasking to virtual environments [111]. The closest relative to our approach is EMMIE [26], which provides a similar platform, but does not include dedicated application management. Other collaborative user interfaces, such as mediaBlocks [112] or multi-computer interaction [85, 83] anticipate many of our goals, but do not incorporate stereoscopic 3D graphics.

The work described here builds on the developments described in the dissertation of Gerd Hesina [43]. He extended the basic Studierstube framework with a distributed scene graph called Distributed Inventor (DIV). DIV supports transparent replication of an Open Inventor scene graph among a group of peers using a reliable multi-cast protocol. To avoid concurrent changes a

single host is assigned to be master and has sole control over the scene graph, while the other hosts become slaves and only replicate the changes. A more detailed discussion of DIV can be found in section 2.5.6.

6.2 Locale framework

Extending Studierstube from a simple distributed framework supporting fixed configurations of shared scene graphs to a dynamic and ubiquitous application environment requires a basic model of the relations between the basic components of users, applications and places. The *locale framework* is this model and will be described in more detail in the following.

6.2.1 Requirements

To clarify the discussion of the requirements and solutions we start with a short description of the most important concepts.

An *application* is a single Studierstube application which can be started and stopped as one entity. Using the dynamic application features of Studierstube, such applications can be written to and loaded from files for persistent storage and later retrieval of a certain application state (see section 2.5.4).

A *workspace* is usually a single coherent coordinate system that one or more users interact in. A single workspace can contain more than one application and conversely, an application can be member of one or more workspaces to support remote collaboration.

A *user* within Studierstube is a set of configuration items that represent a user to the framework and any applications executed. It encompasses the the output display and format, and a set of interaction channels that are associated with a single user. Applications adjust their behavior to different users because they will receive information on the different users that are working with these applications.

A *host* is generally a single Studierstube process that participates in a distributed session to provide a collaborative workspace. While several processes can be executed on a single workstation, usually only one is used to maximize the performance. Therefore, we simply identify the single process with the host.

Within the Studierstube framework we want to support a simple but flexible way to implement various collaboration scenarios. These should include collaborations between users in a fixed workspace, mobile users that roam between workspaces and join and leave them, mobile users that carry

their own mobile workspace and finally remote collaboration between different workspaces. Still, the actual collaboration situation should be transparent to the application. Also the management of applications within these workspaces should not be a burden on the user but rather happen as transparent and seamless as possible. The appropriate user interface for these tasks still depends on the actual setup. We can identify the following requirements from the given description:

Data sharing We require a data sharing mechanism to implement collaborative applications supporting a variety of usage scenarios and setups. It should fit to the typical application programming model in the Studierstube framework and avoid additional overhead for the application programmer.

Shared workspace Collaboration within Studierstube should support a workspace metaphor where multiple users can interact with multiple applications and have control to start and stop, save and restore applications and move them between different workspaces.

Transparent to user Details of the application management should be transparent to the user. The systems developer should provide appropriate user interfaces that are employed by the user to control the applications.

Transparent to application programmer Also, application programmers need not to worry about the workspace and management user interfaces in which their applications will be deployed. Therefore, details of workspaces, such as their geometric configurations, should be transparent to the application where possible.

6.2.2 Concepts

To address the requirements we deploy the following concepts: a distributed shared scene graph mechanism implements the basic data sharing; dynamic session management across all hosts in a distributed setup organizes running applications by appropriate forms of migration; dynamic spatial management models the relations between workspaces and applications.

Distributed shared scene graph The basic data sharing requirements are already addressed by Distributed Inventor. It provides a transparent way to distribute scene graphs and therefore allows programmers to develop distributed and collaborative applications in a similar manner to stand-alone

applications. However, DIV only takes care of fine grained data distribution and does not address the system management issues of sharing and moving whole applications between users and workspaces.

Dynamic session management To support the complex workspace scenarios, a dynamic session management similar to a window manager on a modern desktop operating system is required. It also needs to incorporate different users and their operations on the overall session. A user may join a shared workspace at any time which requires that running applications are migrated to the user's setup. Leaving a session might require to keep a copy in the user's disjoint workspace or to remove the application instance. A central component called *session manager* keeps the track of the relations between workspaces, applications, hosts and users and updates hosts of any changes to the overall configuration.

Dynamic spatial management Besides the generic session management facilities provided by the session manager, a spatial model of the workspaces is required. This is provided by the *locale framework* which provides the necessary representations for workspaces as locales. These locales also incorporate the spatial properties such as locations of workspaces and applications within workspaces and manage transformations of tracking events as required.

6.2.3 Definition of locales

We define a locale as an independent coordinate system that is used to group application objects based on geometric or semantic properties. A locale defines a pose (position and orientation) for each contained application - more precisely, the root of the application's scene graph. There is no global coordinate system for applications; every application's pose is only meaningful with respect to a specific locale. Locales themselves are defined with respect to a global coordinate system. In a more formal way we define a locale as a set M of objects o and a function f that maps each object o to a pose represented by a transformation matrix t:

$$L = (M, f) \quad \text{where} \quad f : \begin{cases} M \rightarrow \mathbb{R}^{4\times4} \\ o \rightarrow t = f(o) \end{cases}$$

Unlike Barrus et el. [10], we allow an application to be a member of several locales at the same time. Any application that is member of two locales defines already a relative transformation between the two locales. If

the application o is member of two locales $L_1 = (M_1, f_1)$ and $L_2 = (M_2, f_2)$, we can define the relative transformation from L_1 to L_2 as $t = f_2(o) \cdot f_1(o)^{-1}$. If there is more than one application common to both locales, we can define a relative transformation for each of them.

We allow an application to be a member of several locales even if its global pose (taking into account the relative pose of the application in each locale, and the absolute pose of the locales) is inconsistent. Such a configuration is not meaningful in most cases where two locales overlap in the same physical space, and share an absolute input device (tracking system). In these cases, an application pose is configured to be coordinated across locales. However, in remote collaboration situations, a more relaxed pose control can be useful.

6.2.4 Managing applications with locales

Within the distributed system we use locales to organize applications into sets. Each host subscribes an arbitrary collection of locales and replicates all applications that are members of these locales. Because an application may be a member of several locales at the same time, there are two situations when an application is shared between two hosts:

- Two hosts use the same locale. In this case they share all applications of that locale. As clients sharing a locale will usually be physically close (and on the same network segment), their sharing of input data can be implemented efficiently.

- Two hosts use two different locales containing the same application. The application can have different positions in each locale but is still shared between the two hosts. This allows remote collaboration between disjoined workspaces.

Information on the status of individual hosts, users, locales, etc. is kept by the session manager (see section 6.3.4). Any Studierstube host notifies the session manager about the locales it is working with, the applications it runs and the users it knows about. The session manager keeps track of the relationship between locales, applications and users. It notifies other hosts about any changes in the configuration, updates newcomers of the current status and performs cleanup operations after a client disconnects. The session manager is not involved in propagating any input events or updates to the actual application; this is exclusively done with fast peer-to-peer multi-cast networking among the hosts.

Instead, its purpose is to coordinate the hosts in performing the following book-keeping operations:

Join a locale A host has to notify the session manager of all locales it is working with. Thereby is subscribes to be updated on any changes to the set of applications contained in the locale.

Leave a locale Again a host also has to notify the session manager, if it is not interested in a locale anymore. In this case applications contained in the locale will not be shared anymore with the departing host.

Start an application This tells the session manager that a certain application is loaded and started in a certain locale. All hosts that use the same locale will be notified of the new application and become slaves for the newly shared application via application migration. The starting host becomes the master for this application.

Stop an application This notifies the session manager that a certain application is stopped and unloaded. The server notifies all hosts that are using locales the application appears in of the event and all hosts unload the application as well.

Share an application In this case an already existing application instance is also added to another locale. Any hosts that use this locale will share the application as well.

We impose some restrictions on the way a host may work with locales. Firstly, if a host joins a locale it shares all applications that are members of this locale. This allows us to use locales to control which hosts share which applications. Secondly, a host can only execute one instance of a shared application. That is, the host must not join two locales which contain the same application. This restriction guarantees that a single application and its content will not appear in two different places to a single user, if its poses in two different locales do not correspond to the relative transformation between these locales (see section 6.2).

6.3 Implementation

The implementation of the locales framework is based on three parts: the basic distribution mechanism for applications based on the Distributed In-

ventor described in section 2.5.6; the setup in the Studierstube host to model locales, the application's containment in locales and to deal with the correct transformations of tracking data between locales; the management of the distributed system itself which is located in the session manager component. Each of these parts is described in further detail in the following sections.

6.3.1 Using Distributed Inventor for applications

Extensions in Studierstube are created by OIV subclassing, and can be loaded and registered with the system on the fly. Using this mechanism, we can take the scene graph based approach that avoids a dual database (graphical + application data) to its logical consequence by embedding applications as nodes in the scene graph. Applications in Studierstube are written as new application classes that derive from a base application node (see section 2.5.4). Multiple instances of application objects can be present in the scene graph simultaneously for multitasking.

Therefore, to distribute an application it is sufficient to distribute the scene graph that represents that application using DIV (see section 2.5.6). A SoDIVGroup is created at all hosts interested in sharing the application and its scene graph is added to that group. The new application node is added to all replicas of the scene graph, and therefore is distributed. With the application node all data contained in attributes will be replicated, not only the scene subgraph of graphical objects, but also attributes that are not visible objects but represent other application data. Non-graphical attributes are simply added as additional fields of the application node that do not directly contribute to rendering. We have found this unified treatment of graphical and non-graphical data to drastically simplify application development.

Application specific computations, such as callbacks triggered by events created from user input, need not be repeated at every host. Instead, for every application instance, a master host is determined, which is responsible for performing all execution of application code. The updates to the application state resulting from these computations are then replicated in the slaves replicas of the application instance. At the same time, the master host can be determined for every application instance separately. Coarse grained parallelism is introduced by distributing the master responsibilities over the hosts.

6.3.2 Shared applications

A workgroup of hosts executing a collaborative session should be able to accommodate changes to its configuration, for example, to provide the current

state of applications to late-comers. For more complex scenarios such as mobile augmented reality, ubiquitous computing and large or remote shared workspaces whole sets of applications need to be managed by appropriate forms of migration.

Studierstube keeps each application contained in an SoDIVGroup to enable sharing of the application. The attributes of the SoDIVGroup are set according to commands received from the session manager and in turn implement the actions required for starting, sharing and migrating applications.

At startup an application is simply instantiated in the scene graph with its associated SoDIVGroup disabled, so that the application runs in standalone mode. As a next step an application would be shared among different hosts using *application migration*. In an already running assemble of applications distributed among various hosts, *activation migration* can distribute the master token between the different instances of an application.

Application migration requires that a running application instance moves from one host to another, while user interface and internal state are kept intact. Therefore, complete transportation of the live application to a remote host is necessary. Since all application state is encoded in the scene graph, marshalling an arbitrary application into a memory buffer becomes a standard operation of OIV (SoWriteAction). The application's complete live state, both graphical and internal, is captured in a buffer, and can be transmitted over the network to the target host where it is demarshaled (SoDB::readAll) and added to the local scene graph, so it can resume operation. Next, the application's binary object module is loaded at the destination, and callbacks from scene graph objects are adjusted.

To trigger application migration the session manager sends commands to the receiving host to create an empty SoDIVGroup for the new application and to configure it for the multi-cast channel of the application. The group is also configured to receive a copy of the application as described above. If the new instance is the first slave instance, then the session manager also configures the sending host to activate the dormant SoDIVGroup and to configure it as a master in the resulting DIV communication.

Activation migration is a simple procedure to change an application instance's master from one host to another similar to [9]: The master application node and its contained subgraph recursively deregister their event callbacks at the old master host, and the instance at the new master host registers its callbacks. The hosts swap roles in a way transparent to other hosts, the user and even the application itself. A possible application for this lightweight procedure is load balancing as described in [93].

The described function does not require any interference from the session manager and is directly implemented in the SoDIVGroup. The application

triggers a field on its encapsulating group to request master mode. When the mode is transferred both instances update the session manager to the new state.

6.3.3 Locales in the scene graph

The implementation of locales is straightforward by mapping the logical structure discussed above to a scene graph. For every locale that a Studierstube host subscribes, a special locale group node with a transformation reflecting the locale's pose is created. Application objects are inserted below the locale node based on locale membership, again with a pose transformation applied relative to the locale. Both application objects and locales themselves can be stationary or tracked. Tracked locales allow dynamically changing relationships among locales.

Tracking data needs to be associated with a locale to provide correct transformations into other locales if necessary. Every locale stores a set of station ids of event channels that are associated with the locale. Events generated by these stations are transformed with the inverse of the locale's pose before they are propagated through the scene graph. When the events pass a locale group node the embedded transformation is applied and they are correctly transformed from their originating locate to the current locale. In the case of the locale the event originates from the transformations cancel each other out and the event is processed unchanged.

6.3.4 Session manager

The session manager is a simple server process that is central to organizing a distributed session of several Studierstube hosts. Acting as a central contact and configuration management point, the session manager provides the basic needs of discovering peers and managing the configuration of distributed applications. All Studierstube hosts in a distributed session connect to the session manager for the duration of their participation. The communication between the hosts and the manager is based on exchanging asynchronous messages. Reacting to such a message can create new messages that are send to and from the session manager.

A distributed session is characterized by a set of locales, hosts and applications and their relations: A host can join a set of locales and a locale can contain a set of applications, while applications can be contained again in a set of locales. This information is maintained in a set of classes seen in Figure 6.1. Additionally, any configuration of users attached to hosts are stored with the host and the locales they are associated with. Joining hosts

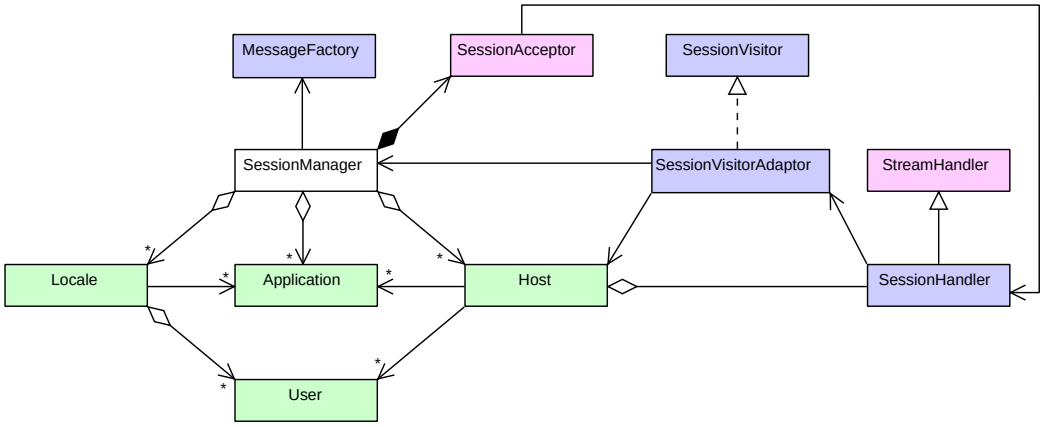


Figure 6.1: A class diagram detailing the implementation classes of the session manager. Green classes form the data model, pink classes deal with the low level network connections and violet classes perform parsing and dispatching of messages.

are then updated with the configuration of any other users that are contained in the locale.

A connected host is always kept up-to-date on the part of the model that it requires to know. The scope of the visible subset are all locales it joins and the applications and users contained therein. The session manager also sends configuration parameters for the SoDIVGroups of individual applications to start and stop sharing an application based on the number of hosts that share the application. The required information can exceed the scope of a single host and therefore can only be computed by the session manager. Hosts in turn create and join locales, and start and stop and share applications within joined locales.

The design of the session manager itself is oriented after a single-threaded server working in a reactive mode. Within a single threaded event loop a singleton object [40, p. 127] of type SessionManager waits for incoming messages from any of the connected hosts. Upon receiving a message it performs the necessary computations, updates its internal state and sends new messages to hosts notifying them of any necessary changes. Then the event loop will sleep again waiting for the next incoming event.

The implementation uses the Reactor pattern [95, p. 179] to process incoming events and connections. If a host establishes a new connection, a new SessionHandler object is created that stores the object representing the connection and a new Host object that represents the information on the host itself in the data model. The SessionHandler class also implements the handle callbacks that are called by the Reactor to notify the object of incom-

ing data. Incoming data is parsed into messages and then forwarded to the SessionManager singleton that interprets the message and acts on it.

Parsing of incoming messages is implemented in terms of the Abstract Factory pattern [40, p. 87] and acting on messages as a Visitor pattern [40, p. 331]. The SessionManager provides a singleton object of type MessageFactory that stores construction functions for all known message types. These functions parse a data block and decide whether they can create a message object of a certain type or not. The SessionHandler passes incoming data blocks to the MessageFactory singleton and receives a new message object. Next, it executes a visit method on the message object passing in a SessionVisitorAdaptor object. The message object itselft then dispatches the right method on the SessionVisitor interface of the adapter object to implement the double dispatch functionality. The adaptor object's methods then call the correct methods of the SessionManager distinguishing between different message types and passing in additional parameters identifying the connection.

The SessionManager object itself implements a set of callback methods that get message objects of different types passed in. Acting on the different message types and content, it updates the data model and sends out any new messages to hosts as required.

6.4 Results

In this section we will describe some example configurations using the locale framework. The examples use increasingly complex setups of hosts and applications. The locale framework can be used nicely to organize the set of shared applications into subsets that are meaningful to the user.

6.4.1 Basic stationary multi user setup

The simplest configuration for a distributed Studierstube session is a two user setup where a dedicated host renders the view of each user. Thus, the setup consists of two hosts, each running a Studierstube process and possibly some additional computers driving tracking hardware. Such a configuration is typical for a shared workspace where two users work collaboratively with a set of applications.

Such a setup is supported by configuring a single locale that both hosts join. The two users are created to be objects in that locale as well. All applications that are started by either user will be created in the joint locale and therefore be shared between the two hosts and available to both

Figure 6.2: A user moves an application across three displays by manipulating the associated marker. The application migrates from host to host in the process.

users. The master copy will typically reside on the host of the user who started the application. By activation migration the master property can be switched between the two hosts for each application individually, for example to distribute the load evenly to achieve load balancing.

The setup can be augmented with private spaces for each user where applications appear that are only visible to the user the private space belongs to. For each user an additional private locale is created that is joined only by the host associated with that user. Then any application that should be accessible only for this user is started in the private locale. Because the second host does not join the private locale of the first host, the applications contained in the locale are not shared with the second host and therefore are not accessible to the second user.

6.4.2 Application migration

A more complex configuration demonstrating the use of application migration is a tiled display wall where each display is driven by a dedicated host. Applications are moved throughout the whole display area and migrate between tiles based on spatial containment. Again the locale framework is used to implement the necessary migration management.

A recent trend in visualization is the construction of tiled displays from inexpensive projectors driven by clustered PC workstations [70]. A basic test setup was used to simulate such a display wall. A Studierstube cluster works as a scalable panorama display using conventional displays placed side by side or in arbitrary configurations. Each display provides a window into a portion of the virtual workspace by depicting the content of a locale associated with that sub volume. Overhead cameras pick up how users move markers in the workspace, and move the corresponding application instances accordingly. Applications migrate between displays, if they cross from one volume into an adjacent volume (see Figure 6.2).

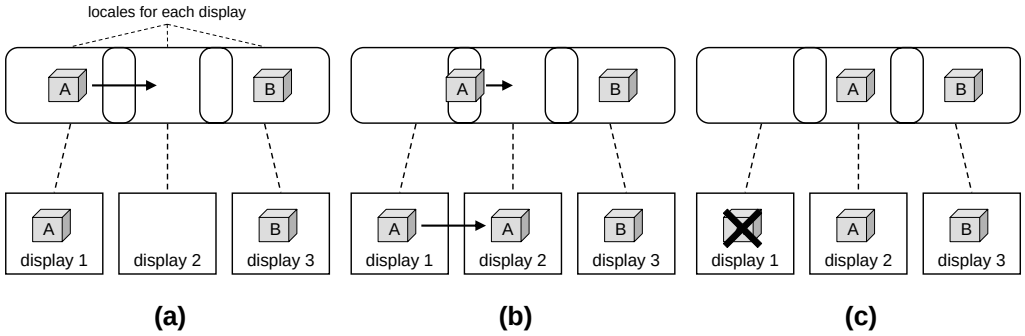


Figure 6.3: Locales organizing application migration for a multi display setup. (a) Each display uses an assigned locale. It only executes the applications contained in its locale. (b) As an application is moved across the combined display, it enters another locale and the application migrates to the associated display. (c) As soon as the application leaves its former locale, it is removed from it and the display.

For each host a locale is defined and joined by that host. A dedicated control application is started in the locale and checks for any markers that enter a spatial volume defined as the extend of the locale. The volume describes the subset of space that a single display tile renders. Whenever a marker enters such a volume, the control application triggers the migration of the corresponding application to the associated locale. If a marker leaves the volume again, the application is removed from the locale.

Since each host/display combination displays a volume of finite extent, it need not know about the content of other locales. Hence as long as application instances remain stationary, there is no need to communicate with other hosts about the interaction regarding application instances contained in the locale, thus preserving network bandwidth and improving scalability by exploiting locality (see Figure 6.3(a)). An application instance need only be shared if it spans multiple locales because it happens to lie on the border (see Figure 6.3(b)) or is very large. If the application is moved from one locale to another, it will migrate from display host to display host. As soon as it leaves a locale completely it will be removed from that host (see Figure 6.3(c)).

Using the tangible marker objects, the panorama setup can accommodate interaction in the style of Rekimoto's multi computer drag and drop operations [83]. A user can move an application instance across the workspace, and the corresponding application will migrate from locale to locale, thus preserving the principle of locality.

Later this basic setup was extended to a fully functional display wall where multiple projectors were used to provide a seamless tiled display [89] (see Figure 6.4). The basic distribution setup, however, remains the same.

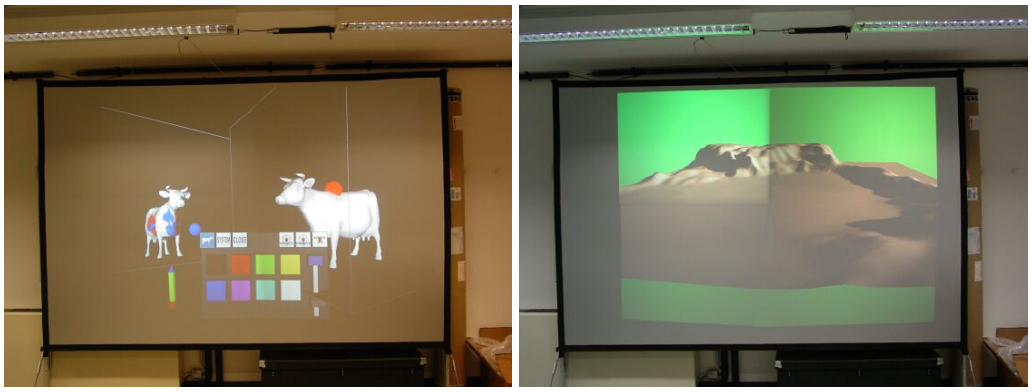


Figure 6.4: Two images from the tiled display wall. Accurate correction for oblique projection and edge blending provide a single seamless image.

6.4.3 Augmented Classroom

A more complex setup demonstrates the full power of the locale framework. In the Augmented Classroom [94] a visionary scenario is created that demonstrates the integration of augmented reality into a classroom situation. We designed an environment that blends mobile augmented reality, collaboration, and tangible user interfaces.

The Augmented Classroom allows two users equipped with wearable mobile augmented reality systems to directly interact in a shared workspace similar to the configuration described in section 6.4.1. They use direct manipulation and tangible user interfaces to interact with a 3D geometry education application called Construct3D [56]. In this setup a student and an instructor can work collaboratively to understand and solve geometrical problems.

To let a larger audience participate in the results of the ongoing instruction, a projection screen presents the application's output as well. The geometrical objects the mobile users interact with are manipulated with tangible markers independently from the mobile users' view. Therefore, the remainder of the class can interactively explore the results of the current or a past collaborative session.

Figure 6.5 gives an impression of the setup. The two mobile users interact at a table with the application. Individual constructions are attached to tangible markers and can be inspected from any side by manipulating the markers. Tracked gloves allow direct manipulation of the constructions such as moving points and selecting objects for operations (see Figure 6.6). Both users share the same workspace and can interact with each other to help with a construction and point out interesting features in the models.

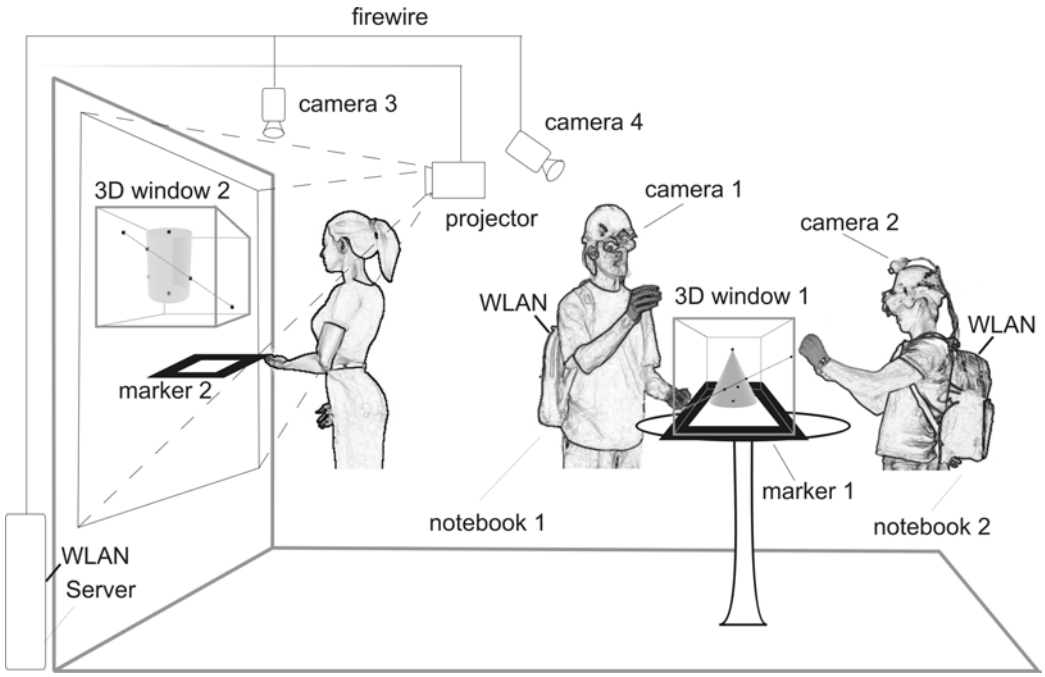


Figure 6.5: A sketch of the Augmented Classroom setup. The shared workspace on the right is used by two mobile users. The display wall on the left allows inspection of the mobile users' work.

Another user inspects the ongoing work on a construction on the projection screen to the left. She can manipulate the same construction in her workspace with another tangible marker. Note, that the same construction is presented at two independent locations in the real world. However, only a mobile user can perceive both presentations simultaneously.

The supporting locale configuration for such a complex setup is shown in Figure 6.7 and is as follows: A workspace locale is created for the collaborative workspace of the two mobile users. Both mobile hosts join the workspace locale and start Construct3D in it, thereby sharing it between both hosts to allow both users to interact with the application simultaneously. A second locale, the screen locale, is created for the space visible in the projection screen, and the host dedicated to rendering the view on the screen joins this locale. The application is also distributed into the second locale and therefore shared by the render host as well.

However, as the host does not join the same locale as the two mobile systems, the application's geometry can be tracked and moved in space independently from the workspace locale. The tracking configuration splits tracking information into two subsets, one for the workspace locale and one for the screen locale. The constructions of the application receive their po-

Figure 6.6: Users in the Augmented Classroom. To the left a mobile user is interacting with a construction. To the right two users are inspecting a construction on the display wall.

sition depending on the host that renders them. Thus, the mobile setups render the objects at their position in the workspace locale and the render host at the position in the screen locale.

The application is still shared between all three hosts and therefore any change and manipulation to the constructions by the mobile users is shared among all hosts and can be perceived instantly on the screen as well. Therefore, the spectators will always see an up-to-date version of the current work on the screen.

The setup could be augmented with additional features from the configuration described in the former sections. Private locales for each mobile user could be added to support private applications only accessible by and visible to that user.

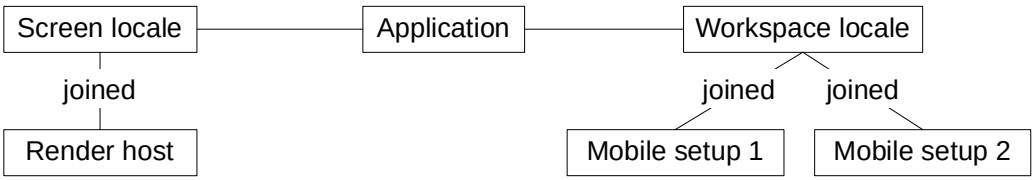


Figure 6.7: The locales configuration for the Augmented Classroom setup. The application is contained in both locales. The hosts join different locales to support independent positions of the application within their workspace.

6.5 Summary

The goal of Studierstube to provide a ubiquitous workspace where users interact with multiple applications simultaneously requires the management of users, applications and places where interaction happens. The described locale framework achieves this by modelling the required interactions accurately. The technical requirements of application migration and distribution were already present in the Studierstube framework. However, more complex scenarios than a fixed set of users and hosts require the additional management of the basic migration and distribution functions.

The locale framework does not provide a high-level configuration interface. In the described form it is still an application programming interface within the Studierstube framework. Additional components such as dedicated applications that provide a user interface to control the API need to be implemented to give the user control over the functionality. For example, the panorama screen setup used dedicated control applications running in each locale that would trigger the application migration based on hand-held tiles entering an associated volume. Therefore, an important direction of future work is to develop appropriate user interfaces for managing applications in changing collaborative work scenarios.

Chapter 7

AR application design

Complex application benefit from a strong framework such as Studierstube, but overtime our experience has shown that this is not sufficient for efficient development. In addition, we find it necessary that application development is supported with best-practice guidelines and a work-flow to guide the developer. This chapter develops a common design for Studierstube based applications as a set of sub-components interacting with each other using a system of patterns. A work-flow is described that can help the developer to factor different aspects of the application design into the correct sub-components and to arrive at a complete implementation and configuration of each component.

7.1 Principles

The first guideline for a developer is a reference architecture of a Studierstube application. Figure 7.1 gives an overview of the different sub-components that a typical application assembles. These sub-components each focus on a subset of the problem space and address the configuration and implementation issues surrounding the areas of tracking, control user interfaces, 3D interaction, graphical representation, data management for large scale data repositories, distribution and the core application logic.

To fill the reference architecture with an implementation a work-flow is provided as another guideline that assists with two recurring development tasks. Firstly, it provides a separation of the problem description into concerns matching the sub-components. Secondly, it outlines a path to the consecutive implementation of the sub-components. The actual implementation work is supported by a number of patterns for internally controlling the sub-components and organizing the communication between these. A

catalog of design issues to take into consideration helps with exploring the solution space for each sub-component.

A Studierstube application typically consists of the following components:

Tracking Tracking encapsulated in an OpenTracker configuration.

Application core The application core consists of application specific functionality.

Control UI A control user interface built from standard 2D widgets and other high-level interaction.

3D Interaction 3D user interfaces to select and manipulate 3D information directly.

3D presentation 3D presentation containing models and rendering information.

Data Management The data management works outside of the actual runtime environment to provide large models.

Collaboration Different forms of data distribution to implement collaborative applications.

Each of these components and a set of issues related to implementing these will be discussed in the following sections. Moreover their relations and possible communication patterns between them will be described. Finally an overall work-flow tying together the individual pieces into a larger net is described.

Within the following sections we also describe some patterns that represent reusable solutions to typical design problems. They appear at appropriate places within the discussion of the components they relate to. In the language of Buschmann et al. [24] they are similar to idioms, because they are usually tailored towards the Studierstube framework and do not always make sense in another context.

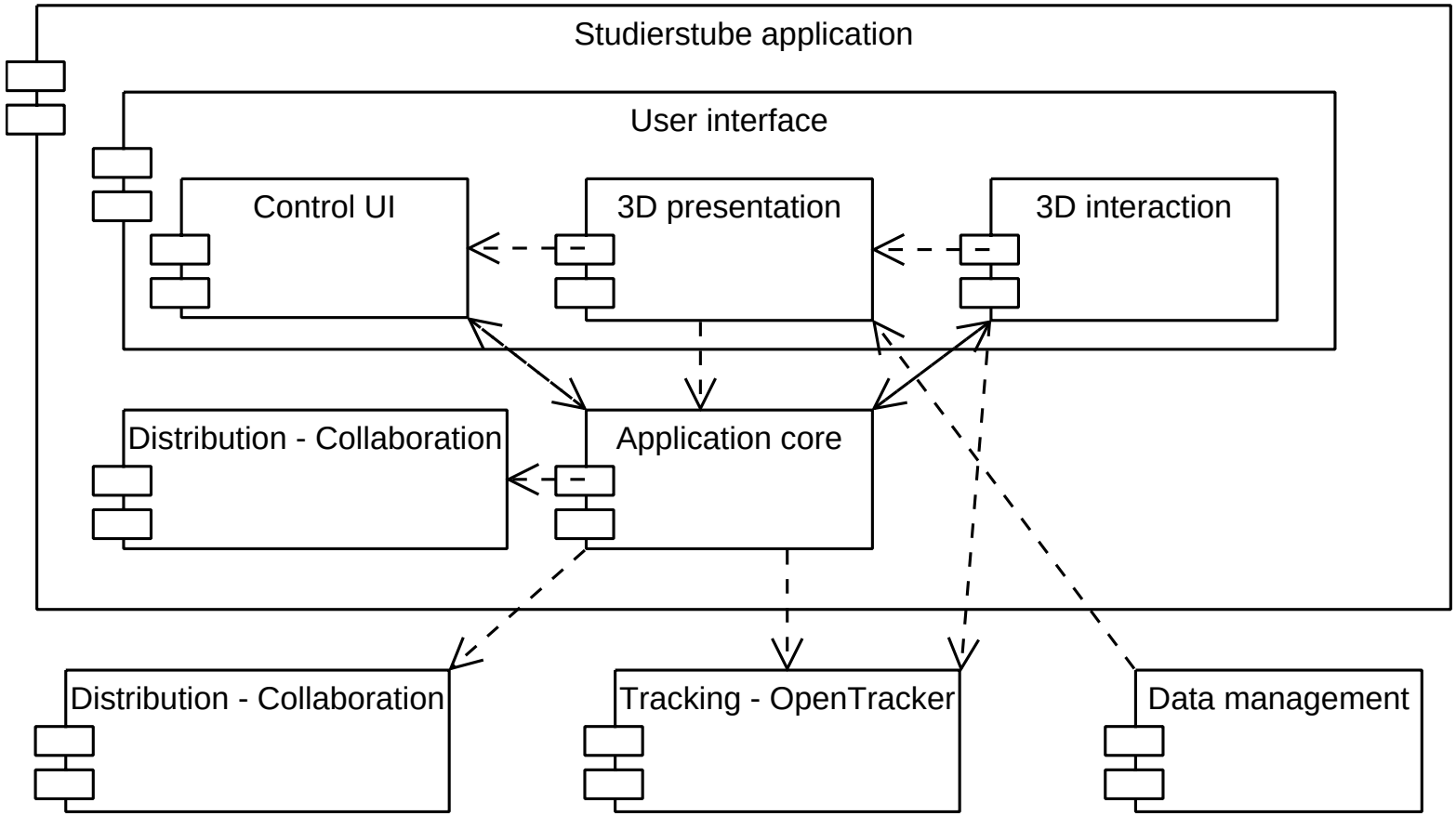


Figure 7.1: The central components of a Studierstube application.

Reicher et al. [79, 78] have developed a reference software architecture similar to the one presented here to investigate and compare different architectures deployed in AR applications. It is an interesting exercise to match the sub-components as given above to the subsystems of the reference architecture (see Figure 7.2). Some components are straightforward to associate: the tracking component is contained in the tracking subsystem; the application core is in the application subsystem; the data management is in the world model subsystem. The control user interface and 3D interaction components are both contained in the interaction subsystem. They are separated in our model because Studierstube provides different highly specialized objects that allow and fit such a separation. Finally, the 3D presentation component is certainly contained in the presentation subsystem but can also be part of the world model subsystem. The collaboration component does not appear explicitly in the reference architecture. However, it can be mapped as part of the context subsystem where collaborative designs such as blackboard for sharing information are placed.

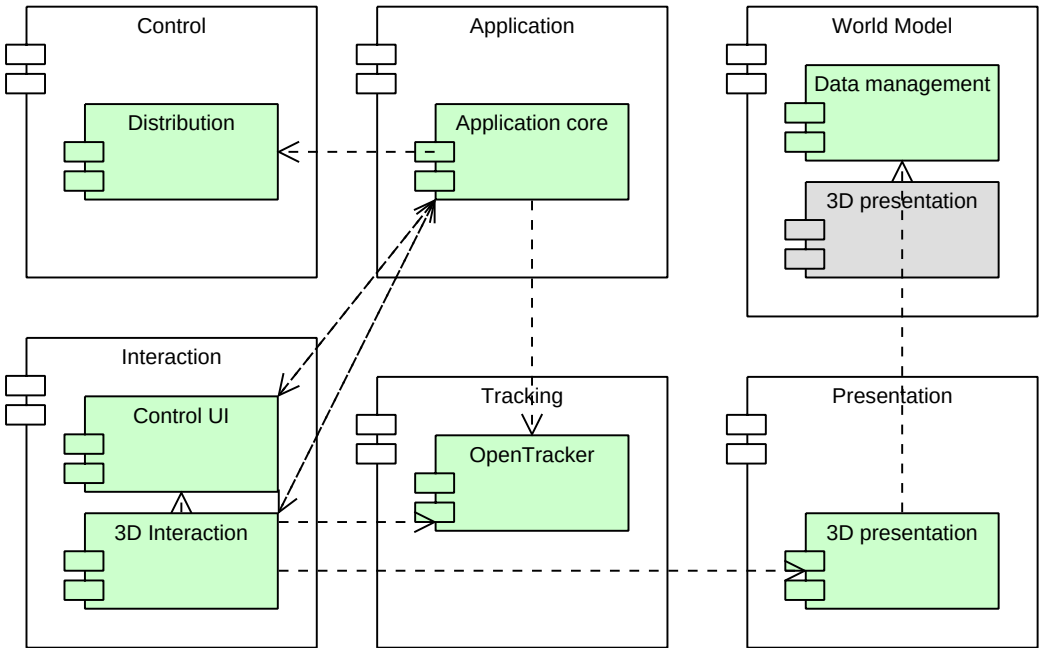


Figure 7.2: Mapping of Studierstube application components in green to the subsystems of the reference architecture. The grey components mark alternative mappings for a component.

7.1.1 Tracking

Tracking is an indispensable part of any AR application and therefore plays a predominant role in most current software designs. Implementing tracking specific functionality in a dedicated software component allows for code reuse and changeability of an application. Indeed, these were some of the main motivations for the development of a data flow library for tracking data in chapter 3. However, it is also necessary to make good use of the provided functionality. In this section we will explore some principles that lead to optimal use of the OpenTracker sub-component.

A first step is to identify the abstract tracking requirements of the application. The requirements should specify the quality of the tracking data in terms of degree of freedom, update rate, accuracy and also variability of these features. The number and use of several tracking channels needs to be established, for example knowledge that a user's head and one or two interaction props need to be tracked. The type of reference system should be defined as well, e.g. to distinguish between a world-stabilized, body-stabilized or head-stabilized coordinate system [18]. These considerations should not involve the actual tracking systems used or any necessary combinations of such systems. The goal is to establish the final tracking input to the application itself.

In a next step, the actual configuration of the individual trackers is created. Typical considerations include the availability of the trackers. Does the data require any post-processing such as filtering? Do we have to combine and fuse different trackers to create the required tracking data, for example by fusing a position tracker and and orientation tracker to provide full 6 degrees-of-freedom information? And finally, does the tracking data need to be transformed to register the tracker's reference system to the application's system or to calibrate any markers within the tracker's system?

Another aspect of any tracking configuration is the location of the tracking resources. Typically not all trackers will be connected to the host running an application, but will provide the necessary data via network transmission. Refer to section 3.5.1 for an in-depth discussion of this topic. The location of possible transformations of the data is usually associated with the distribution of tracking data. Should such transformations be located at the source so that they are visible for every user of the data or should they only appear in the local configuration of the application?

A final issue is feedback from the application itself into the tracking system. Calibration procedures, switching between different tracking systems or tuning filters, all require the application to manipulate the tracking configuration in some way. By providing prepared entry points in the configuration

the developer can define a well-known interface to the application. Such a separation decouples the application itself from the tracking configuration because it does not require any detailed structural knowledge about the configuration anymore.

The following pattern describes an effective way of implementing such a feedback loop with the help of OpenTracker (described in chapter 3) which maintains the separation between application and tracking configuration while providing the required feedback channel.

Tracking feedback loop

Problem Some situations require the application core to provide feedback into the tracking subsystem. For example a calibration procedure needs to set an offset value to correct incoming tracking data of an interaction prop. The following issues need to be considered:

- Changes to the tracking configuration should not influence the application directly as long as the semantics of the feedback do not change.
- Conversely, changes to the application's side should not propagate to the tracking configuration.
- The interface should use existing designs in both the application and the tracking domain to avoid a close coupling to the feedback mechanism.

Solution Use an application-driven OpenTracker source node (in our case a StbSource node) to provide transformation information to the tracking subsystem. Configure the node to use output from an Open Inventor node that generates the required transformation data. The transformation data is then combined within OpenTracker to correct incoming tracking data using any dynamic transformation filter node.

Structure The StbSource node acts as a subscriber to an arbitrary Open Inventor node. The Open Inventor node acts as the interface to the scene graph and field connection network. Both together form a bridge between the event flow of Open Inventor and the event flow of OpenTracker (see Figure 7.3). The dynamic transformation filter node acts as the combining element which applies the changing transformation to incoming data. Both the OpenTracker node and the Open Inventor node act as proxies to their respective event system and encapsulate the underlying connection allowing changes on both ends that do not propagate to the other end.

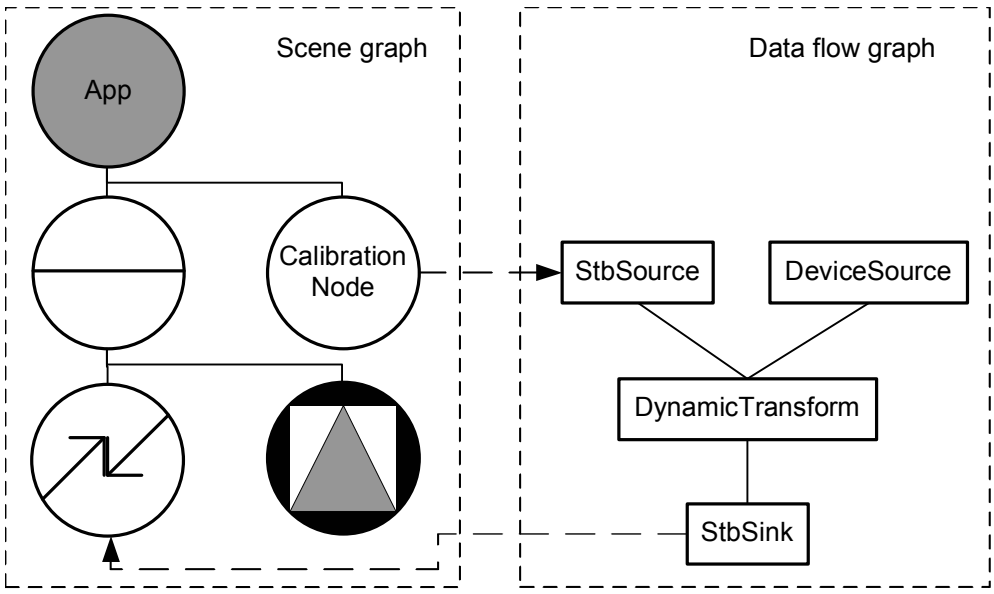


Figure 7.3: The structure of the tracking feedback loop pattern.

Dynamics Whenever field data stored in the Open Inventor node associated with the StbSource changes, the StbSource node generates new tracking events based on the new data. These events are then forwarded to the filter node which applies the changing transformation to incoming data. The fields can depend on other fields from other nodes and engines in turn.

Consequences The Feedback loop pattern offers important benefits:

Transparency Because each proxy node acts in the standard way of the associated event system the feedback loop conforms to the usual semantics of that system. The individual interfaces are well known and should not hold any surprises to the developer.

Flexibility Changing the source of the feedback data can be accomplished by changing the tracking configuration in a well known location or simply reconnecting the source field on the Open Inventor node. The first option lends itself more to hand-tuning a configuration, while the latter offers changeability at runtime as well. □

7.1.2 User Interface

User interface components in an augmented reality application usually fall into one of two categories. Traditional control user interfaces are required to set the mode of an application, select from a set of given textual values,

set abstract numerical values or trigger actions. The interactive and three-dimensional nature of AR also requires the use of 3D direct manipulation user interfaces to provide the natural interaction with real or virtual objects. The Studierstube framework provides dedicated APIs for both kind of user interfaces and allows the developer to assemble the required mix of interface modalities. Still, a number of issues should be taken care of.

A first step is to identify the states of the application that will be controlled by the user interface. A list of these states consisting of their name, type and possible values or ranges of values can be helpful. Now it is possible to decide which states should be represented and modified using traditional 2D widgets and which information and input should be provided by 3D interaction. Within Studierstube the Personal Interaction Panel is a standard way of presenting a 2D GUI (see section 2.5.3). Other possibilities is the PUC framework [68] that allows the application to interface with a graphical user interface presented on a hand held device.

Any presentation of the user interface should be separated from the application itself. On the one hand, the application should expose the current state as a set of variables that describe the state as succinct and fitting as possible. The user interface, on the other hand, will typically consist of a set of widgets with less abstract state information that needs to be filtered and aggregated to arrive at the high-level information that matches the application's state. The translation between both can be achieved by a layer of glue logic that implements the necessary steps. If any change to the user interface or the application's logic is necessary, only this intermediate layer needs to be adapted. The following pattern describes this approach in detail.

Widget adaption layer

Problem User interface widgets are software components for standard interface actions tailored towards reuse. Therefore, they typically model the state appropriate to the widget. However, application state usually differs from the abstract widget state and dedicated functions are required to translate from the input data from a set of widgets to the high-level state of the application. For example, a 3-valued state variable in the application can be represented by a single selection listbox, a group of three radio buttons or some other widget. While the listbox will be able to directly represent the selected value in it's state, the radio buttons require an additional layer that infers the selected value from the button state and identity.

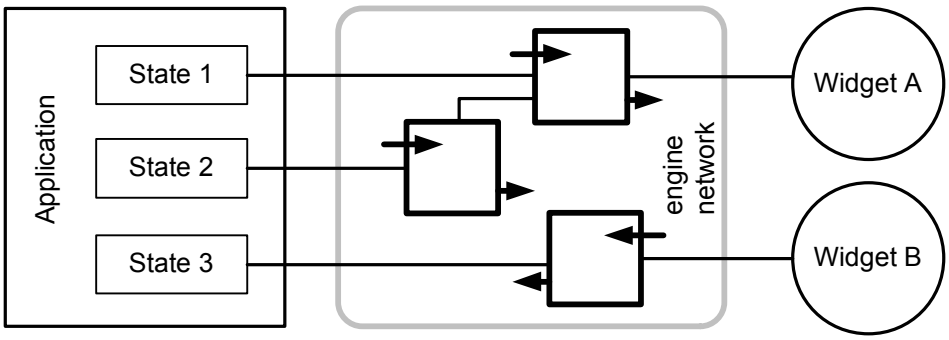


Figure 7.4: The structure of the widget adaption layer pattern. Data between the application object on the left and the widgets on the right is transformed by an engine network in the middle.

The following issues need to be considered:

- To preserve changeability, the application state should not be tightly coupled to the widget state but rather be represented as required by the application.
- Conversely, standard widgets should be reused instead of developing an expanding set of custom widgets.

Solution Add a dedicated layer to separate widget state and application state and translate between different representations. Within Studierstube both widgets and applications are represented as Open Inventor nodes and state is represented as fields. Therefore, the layer should be implemented as a network of field connections and engines that automatically performs the required translations. This layer implements the Bridge pattern [40, p. 151].

Structure Model your application state as fields of one or more application nodes. Create the appropriate widget set that should be used in the application. Then author the required network of field connections and engines that perform the necessary translations.

Dynamics The pattern itself is fairly static. It implements a data flow network that performs the computations implicitly. Any changes to the widget or application state triggers updates in the field connection network that evaluates the connections and performs the configured calculations. The necessary notifications are handled by the Open Inventor framework itself.

Consequences The Widget adaption layer offers the following benefits:

Accurate state representation The application state is represented as fit for the application and not as required by the widget set. Therefore, the application functionality is decoupled from the actual widgets used.

Transparency Both widget set and application state can be changed without propagating changes into the other component. All necessary transformations are captured within the engine and field connection network and therefore can be modified at a single point. □

For 3D interaction some support from the presentation system is required because the user will interact with visible objects that are part of the 3D registered experience, either real or virtual. In each case the interaction method usually requires a model of the object in order to compute intersections with interaction devices and the results of the operation. Within Studierstube, such a model can be used cooperatively with the 3D presentation component by relying on the scene graph. To address differences in usage but still allow an organized structure of the scene graph the context sensitive traversal described in chapter 4 can be deployed. The following pattern describes such a solution in detail.

Reuse scene graph for both rendering and interaction

Problem Scene graphs used for rendering and interaction computations may require different geometrical representations of application objects which are represented as scene graphs. However, managing different representations increases the complexity of the application implementation. Two main forces need to be resolved:

- Different functions require different representations of objects.
- Data management should operate on single objects without having to deal with their representations.

Solution Combine the different representations in a single context sensitive scene graph. The scene graph is organized in such a way, that object management only deals with a single sub-scene-graph per object. The different functions select via associated context values the appropriate representation during traversal.

Structure The required scene graph structure is presented in Figure 7.5. Each object contains an SoContextSwitch node which contains the two different representations as separate sub scene graphs. The switch node is

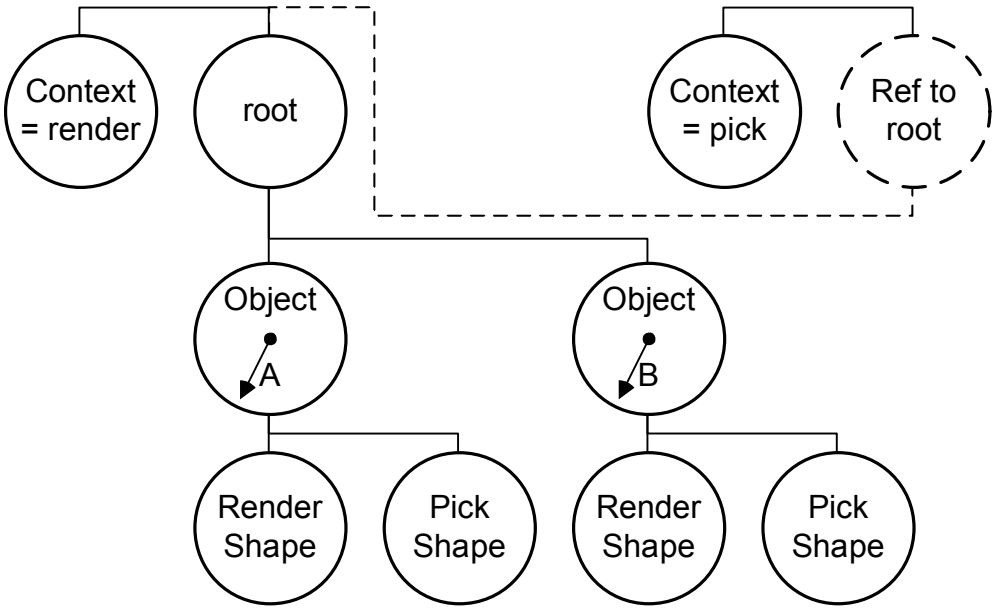


Figure 7.5: The structure of a scene graph combining representations for rendering and interaction. The scene graph is reused through a second reference to the root node.

configured to use a well-known index whose value chooses between the representations during traversal. An SoContext node sets the context value to select which version is used for each reference to the scene graph containing all objects.

Dynamics The pattern acts during traversal by presenting a different scene graph to different traversal functions. Object management operations such as creation, insertion and deletion are used at a single point in the scene graph.

Consequences The pattern has the following consequences:

Simple control The management of the application objects is simplified because different uses are merged into a single sub-scene-graph.

Increased object complexity The simplified management structure is countered with increased complexity within the objects's scene graph itself. □

Additionally the 3D interaction requirements can have influences on the tracking requirements of the application. Typically they will create new interaction props or require information derived from other tracking sources but transformed to fit the interaction model.

7.1.3 3D presentation

In a visual augmented reality application the presentation of visual data plays a key role. The rendering style of virtual objects is varied to provide visual feedback to the user in selection and interaction tasks. Thus, the application programmer has to deal with variations on rendering and selections of data to be presented within an environment.

Within Studierstube all visual data is stored and presented by using a scene graph. Therefore, the two operations of varying the presentation style and selecting a subset of objects to present should be translated into operations on the scene graph. By combining both operations, any variation required becomes possible. In the extreme case, each variation could be combined with the selection of a single object, resulting in each object being rendered differently to one another. Both operations can be implemented by building a scene graph that supports context sensitive traversal.

The methods described in chapter 4 can be applied to the scene graph containing the visual data. Two dimensions need to be investigated: identify the individual objects the application uses; list the different presentation styles that should be possible for all or a subset of objects. Then one can derive the structure of a context sensitive scene graph that allows the application to switch between different presentations per object.

The resulting template scene graph structure is also important input for the data management step. This template needs to be applied to the data objects retrieved from the data storage to generate the scene graph the application will use.

7.1.4 Application core

In the design of the tracking configuration, the user interface and the presentation component, a number of state variables and input and output channels have been defined. These need to be connected with additional functions to provide the final application functionality. The required algorithms can now be implemented in a dedicated software object called application core.

Within Studierstube the core can be represented within the application class that is created to relate the application to the framework's features. Such an application class is an Open Inventor object derived from the basic SoContextKit object. Any state variable, input and output channels should be represented as Open Inventor fields to leverage the properties of the Open Inventor framework such as serialization, distribution and introspection. Additional static configuration information is also best represented in this way to allow simple modifications to the configuration.

Directly using the Open Inventor model to represent all the necessary information is a powerful paradigm. The framework's features such as observers for field changes and time flow directly help with the implementation of the application's core by inverting the control flow from the application to the framework. The developer only has to implement callback functions that react to events and changes in the application's state.

Also, the strict separation of the functional aspects from the presentational aspects and the interface of fields allow the reuse of the functional or the presentational components independently of each other. For example, the application object could be directly reused within another project by omitting the user interface and presentation component.

The separation between the user interface component, the 3D presentation and the application core again follows the traditional Model-View-Controller pattern for interactive applications. However, it is applied at a more abstract level above the pure graphical data and raw input devices. The user interface acts as a controller that provides already filtered events as output from widgets. The 3D presentation is a view that can also be reused for interaction computations or general application model. Finally, the application core represents the functional aspects of the model.

Not all aspects of an application need to be implemented in one SoContextKit object. A good design guideline is to distribute independent functionalities between different Studierstube applications that may interact as necessary. Such a separation allows to iteratively develop the complete application by implementing only a coherent subset of functionality at one point in time. Also, testing and reuse of such smaller functional components is simplified.

7.1.5 Data management

For applications that present large amounts of data some organized form of data management is required. In chapter 5 we presented an approach to organize and reuse complex data sets for AR applications. The developed architecture will now be tied into the process of developing such an application.

In section 7.1.3 we have seen that the scene graph can be organized to allow for simple reuse and changes without complex operations on the application's behalf. However, the structures that are created within the scene graph to support these operations can become too complex to be created by manual means for a larger set of data. Therefore, the automated transformations from a general data model into the required scene graph enable the use of the techniques developed in the former sections.

If the scene graph structure has been defined, the following three steps are required to reuse the data management architecture.

Firstly, identify possible mappings of your required data to the model schema BAUML. If required, add new types to BAUML or embed additional dedicated information into existing elements. The annotation element is the default place for such low-key extensions. Additional information can be put into a separate XML namespace to avoid name clashes between different applications that reuse the same model.

Secondly, the original data needs to be imported into the general model. Depending on the source format, this can be a simple transformation between different XML dialects or require a dedicated tool that reads the source format and creates an equivalent model.

Finally, a set of tools needs to be developed to generate the required scene graph structure and any additional control information from the stored data. These tools take the model as input and create a serialized Open Inventor scene graph for use in the final application's scene graph. The result is embedded into the application's scene graph by including a File node or direct preprocessing of the different scene graph files to create a unified application file.

7.1.6 Collaboration

A focus of Studierstube lies on collaborative applications to support several users in a shared workspace. To support the required data distribution the Distributed Inventor (DIV) component allows the transparent sharing of scene graphs between different hosts. A number of different design patterns can be built upon this basic function which cater to different requirements in coordination and consistency between the different hosts.

The basic unit is a scene graph below a SoDIVGroup node which is shared between other instances of SoDIVGroup that are configured to use the same multicast channel to communicate. Any group node configured as a master can send updates and all group nodes receive and apply updates to their copy of the scene graph. The updates are causally-ordered, that is they may be applied in different orders at different locations but updates from one source are always applied in the same relative order they were generated.

Simple sharing of information between two or more application instances can be accomplished by sharing a dedicated sub-scene-graph. The restrictions of the update ordering allow only certain applications to be able to use such a scheme. Typically only incremental changes like constantly adding new nodes to a scene graph or operating on distinct parts of the scene graph can use such a simple model.

A more complex approach is to combine the above design with a master/slave mode where only one instance is in master mode at any given point in time. An application then has to request the master token before it can proceed to update the shared scene graph.

Studierstube itself provides a strong application model based on the functionality described above. Within this model a full application scene graph is mirrored between hosts sharing that application. One host acts as a master instance and executes the application's functions that are triggered by callbacks and timers in the event based execution model of Open Inventor. There are no conflicts between different hosts, because only the master host will affect changes to the scene graph while the remaining slave hosts only render the replicated scene graph. The application sharing model can be applied to the overall complete application or to only one functional part implemented as a SoContextKit (see section 7.1.4). Another design is presented in the following Application proxy pattern. It describes how to create a dedicated application object that deals with the distribution aspects of an application.

Application proxy

Problem Not all collaborative applications can be modelled with a symmetric data distribution scheme such as provided by Distributed Inventor. Distributed application instances might only have to coordinate some data such as a common target or exchange commands. Here the full replication of the application's data is unnecessary and increases the complexity of the software design because local variations need modelled explicitly. However, the collaborative functions should still be able to reuse the collaboration management features provided by the Studierstube. To summarize, the following forces need to be resolved:

- An application can not be shared using the default model of the distributed scene graph.
- High-level application migration and management features of Studierstube should still be accessible to the application.

Solution The communication between different application instances is accomplished via dedicated proxy applications that are replicated using the standard distributed scene graph approach. On each host local communication mechanisms based on field connections interface the application and its proxy. The proxy applications are created in one or more dedicated locales

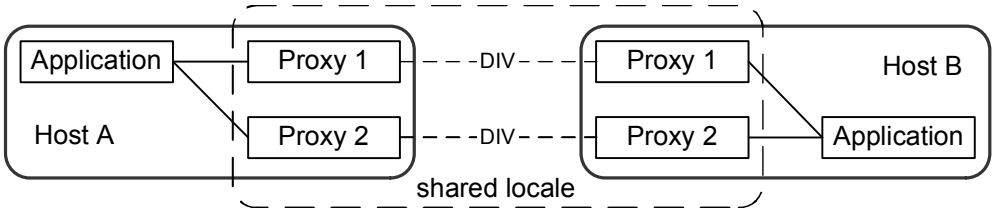


Figure 7.6: The relations between an application and its application proxy. Proxies exist an all hosts and communicate locally with the application. Proxy instances communicate remotely via DIV.

which are joined by all hosts participating in the collaboration. Therefore the automatic application and locale management mechanisms are still used to setup communication between the application instances. The application proxies are an example of the Remote Proxy pattern [24, p. 263].

Structure The structure consists of the separate applications per host. One application implements the actual functionality, while the other acts as a communication proxy. The application only communicates with its local proxy and is not aware of the other instances within the distributed system. The replicated proxy instances then act on the remote side for the application and communicate again via local mechanisms with the remote instances. The shared nature of each proxy application provides the remote communication mechanism. Figure 7.6 gives an overview of the communication structure.

Dynamics The application proxies decouple the functional application part from the dynamics of the distributed system. The proxies themselves act only on events requiring distribution of information. Whenever a new replica of a proxy is created, it establishes new connections on the local host with the resident applications. Any other communication is dependent on the applications function itself.

Consequences The pattern has the following consequences:

Transparency The functional part of the application is separated from the communicating part. Often the whole collaborative functionality can be implemented with the proxy relying on the basic functionality offered by the basic application.

Added complexity The overall configuration of a collaborative system is increased as the required locale configuration becomes more complex. Therefore the pattern is more applicable in a configuration that already requires a detailed locale model. □

Applications that require complex manipulations of their scene graph but do not have elevated data communication requirements can directly be shared as such. Also simple applications within a symmetric environment where each instance receive full information on tracking data will benefit from the simpler structural model. The implicit sharing renders the slave copies as pure replicates to provide the same view. The master copy holds the sole responsibility for computations and can act as a stand-alone application.

7.2 Design work-flow

The different sub-components do not stand alone but are interconnected with various dependencies and need to conform to common interfaces to enable communication between them. A structured approach to investigating into the different sub-components and tracking the dependencies between them simplifies the problem of resolving these dependencies and providing the necessary interfaces. Such an approach is presented as a work-flow for the design of augmented reality applications. The overall work-flow is shown in Figure 7.7.

Starting from a problem description as a set of use cases, a general description of the user experience or similar material, we can proceed to analyze the requirements with respect to four problem areas: tracking, user interface, presentation and data model. For each of these areas we have discussed a set of issues and typical implementation patterns that can be applied.

A result of the analysis of the above sub-components is a set of parameters, additional and specialized functionality, scene graph structures for the presentation part and a data model that provides the basic data for the generation of the scene graph structures. Based on such a collection of structured information it is now the final task of implementing specialized functionality in the form of new software components and providing custom data transformations between the data model and the appropriate data structures used in the application itself.

The proposed work-flow is applicable to the development of Augmented Reality applications because it addresses the complete set of problem areas encountered in such applications. Moreover, it attempts to address the dependencies between different sub-components in an appropriate way by constructing the depending components after any components providing services to avoid premature design decisions based on incomplete information.

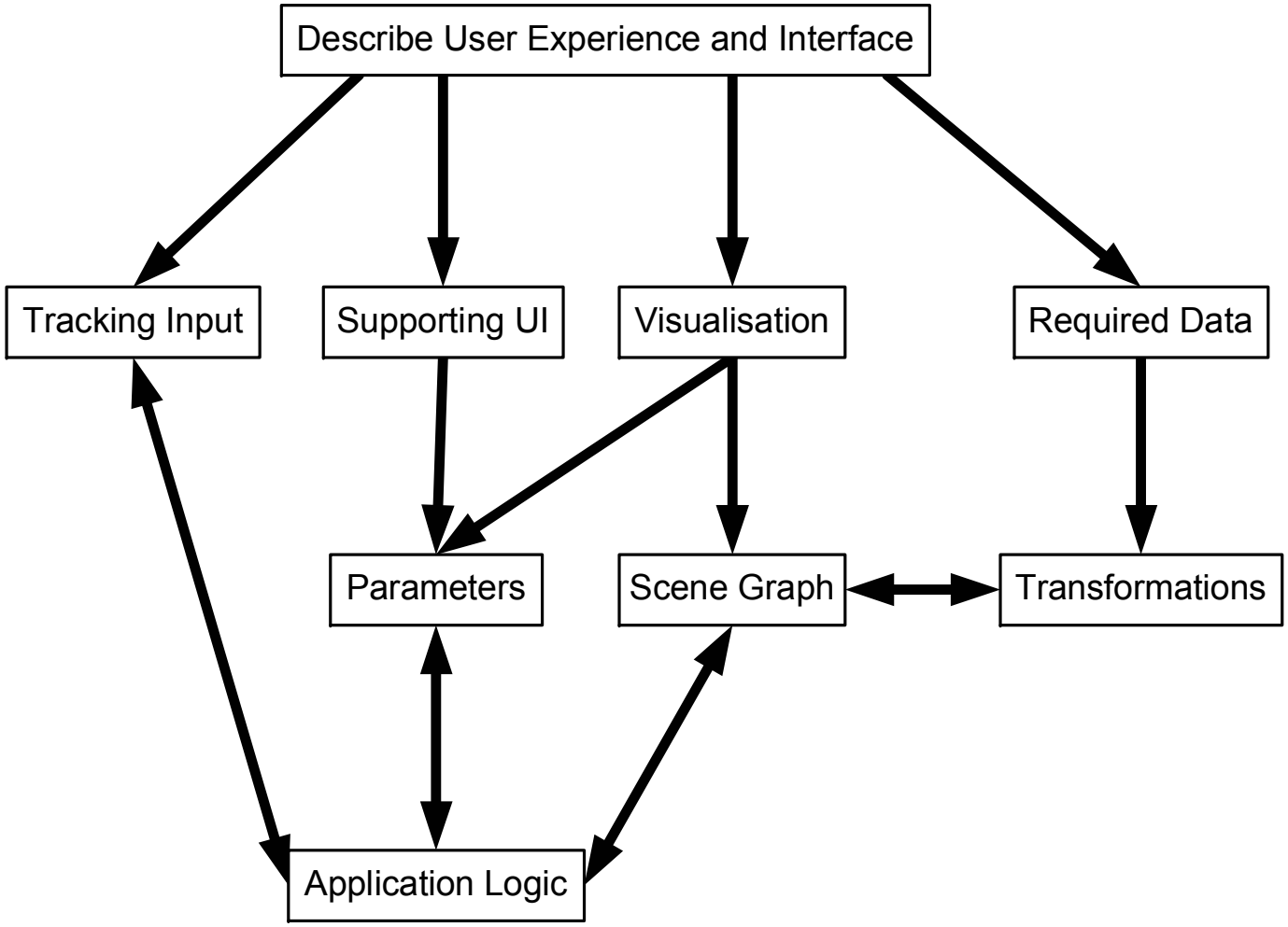


Figure 7.7: Work flow for designing an augmented reality application in the Studierstube framework.

Note, that the design work-flow does not try to impose a certain software engineering methodology but can be used within any deployed methodology. For example it can be employed within every iteration of an iterated prototyping process or can be applied to a limited feature of an application within a feature driven process. Its role is to provide a guideline for the design of an augmented reality application by actively supporting the investigation of different problem areas and structuring the interactions between these areas.

7.3 Summary

The approach taken here to describe efficient application development for the Studierstube framework tries to follow the approach of providing a system of patterns for AR applications. While it does list a number of individual patterns for certain problem areas, the main impetus lies in the sensible separation of concerns into appropriate subcomponents and then applying tested methods of connecting them to create the final application.

The design principles and the work-flow discussed in this chapter evolved over a period of 4 years of working intensively with the Studierstube framework to create a number of augmented reality systems. Preliminary versions and subsets of the presented ideas were taught in the lab on virtual reality where students built small augmented reality demonstrators. These groups of newcomers to the Studierstube framework provided valuable feedback, either by consciously reporting on issues and insights they had, or unconsciously by the errors and mistakes they made.

In order to demonstrate the developed concepts and patterns at work, a larger example will be developed in the next chapter. It describes the design of a fairly complex augmented reality application in the area of outdoor navigation and information browsing.

Chapter 8

A collaborative tourist guide application

After we have developed a design process for augmented reality applications within the Studierstube framework, we will demonstrate the applicability of the process with the example of a mobile collaborative AR application for outdoor use. The application scenario revolves around a group of tourists visiting the City of Vienna.

The needs and requirements of a tourist are a suitable starting point for testing location-based applications. A tourist is typically a person with little or no knowledge of the environment. However, tourists have a strong interest in their environment and also want to navigate through their surroundings to visit different locations. Guided tours are a common practice for tourists. In such a situation a single person navigates a group of people and presents information.

Consequently, we have chosen a tourist guide application for the City of Vienna as an example scenario for an augmented reality application that integrates a large amount of data from different sources. It provides a navigation aid that directs the user to a target location and an information browser that displays location referenced information icons that can be selected to present more detailed information in a variety of formats. Both functions support collaboration between multiple mobile users.

The tourist guide application will demonstrate the use of the various software components developed in the former chapters. We will develop it as an example of the design work-flow outlined in the last chapter and show how the individual design patterns are applied to arrive at a complete system that is open for future extensions.

8.1 Requirements

Three tasks should be supported by the tourist guide application: Navigation, information browsing and annotation in the form of simple virtual graffiti. All functions should be useable in a stand-alone mode by a single user. Additionally, they should support collaborative work with the roles of expert and novice users. Here the tour guide of a group of tourists acts as an expert and should be able to control the functions of the tourists devices. The individual tasks are described in more detail in the following.

Navigation A navigation component should provide support for the following tasks: The user selects a destination by name and the system will present markers that direct her to the selected destination. These markers are based on path that is computed to the destination. A variant of the selection process is to select a type of destination such as a shop, church or police station and navigate towards the nearest instance of such a destination.

The presentation of the guides should be intuitively understandable and require no further skills to understand besides the everyday knowledge of how real objects behave. The system should also adapt to the user's actual movement and provide always a correct route to the selected destination.

A number of collaborative navigation tasks should be supported. A user can choose to follow another user and will be continuously directed to the location of her partner. The converse operation of guiding another user to a selected destination should be possible. And finally, a simple meeting functionality should provide directions for both users to meet at a location halfway between them.

Information browsing A user should be able to view information in different media forms which describes interesting facts on certain locations or objects. For example, frescos on a building facade could carry information on the sculptor who created them and the topic of the picture. Statues could give information on the background of the person they are representing.

The information itself should consist of text, images and 3D objects or animations that appear in the view of the user or at fixed locations as defined by the content author. The display of the information itself should be triggered by the system without any manual action by the user. It should provide an intuitive browsing experience where information is presented on the object in the focus of the user's attention and vanish as soon as the focus shifts to another object.

A user might get overwhelmed by a large number of locations that trigger information displays or not be interested in every type of information. Therefore, the objects and content needs to be described with a set of keywords and the user can select interesting information by selecting a subset of these keywords. Only information matching the selected keywords will be displayed.

Support for multiple users should allow a tour guide to select the currently visible information and trigger it on all members of the tour group. The guide should also be able to set the keywords to select the information items which are relevant to the current tour.

Annotation Users should also be able to contribute in a simple way to the available information. Annotation allows users to place graphical icons in the environment with different colors and shapes. To simplify the interaction for determining the position, icons are always placed on the surface of existing buildings which reduces the required manipulation to a simple direction from the user's current position.

A group of users should be able to share the different icons that each participant creates, so that every user can see all icons. Again, to reduce possible clutter, a filtering option should allow users to restrict the visible icons. Possible criteria are the icon's creator, it's color or shape.

8.1.1 The mobile augmented reality setup

The mobile AR setup uses a notebook computer with a 2GHz processor and an NVidia Quadro4Go graphics accelerator operating under Windows XP. It includes a wireless LAN network adapter to enable communication with a second mobile unit. A Trimble Pathfinder Pocket differential GPS receiver is used to determine the position of the system in outdoor applications. All the equipment is mounted to a backpack worn by the user. We use a Sony Glasstron optical-see-through stereoscopic color HMD fixed to a helmet as an output device. An InterSense InertiaCube2 orientation sensor provides information on the direction the user is looking in while a PointGrey Research Firefly camera mounted above the HMD is used for fiducial tracking and video see-through configurations. Both devices are mounted on a helmet worn by the user (see Figure 8.1).

The system presents information to the user on the head mounted display. Such information is either presented as graphical objects rendered to fit into the natural environment or as text, images and 3D objects providing a heads-up display. The graphical objects are drawn to enhance and complement the user's perception of the natural environment. They can represent abstract

(a) (b)

Figure 8.1: The mobile AR setup: (a) A laptop computer, GPS receiver and battery packs are mounted on a backpack. (b) The HMD, inertial sensor and a miniature camera are mounted on a helmet worn by the user.

information, alternative representations of real objects or highlighted real structures. The heads-up display is used to provide a graphical user interface consisting of typical 2D widgets such as buttons, lists, and text fields and to provide other status information. Figure 8.2 shows a typical view through the users display.

The user can control a cursor within the 2D user interface part in the upper right corner with a touchpad that is either worn on the belt or handheld. She can switch between different modes of the application such as navigation, information browsing and annotation. Each mode presents a number of individual panes to provide control of parameters and other options related to the current task. A general heads-up display at the bottom of the view presents generic information such as the current location, selected target location, distance to the target and an orientation widget modeled after a compass.

8.2 Applying the work-flow

Applying the work-flow we will start with investigating into each of the four main aspects and derive a specification from the given requirements and application descriptions. We will start with the tracking aspect and walk through the decisions made to arrive at the final configuration of the tracking component.

(a) (b)

Figure 8.2: (a) An overlay of the building model over the real world. (b) 2D user interface components and heads-up display.

Next, we analyze the user interface requirements and presentation aspects of each application and then derive also the application core from that. To simplify the analysis and the development we chose the approach outlined in section 7.1.4 and split the overall application into three components which are implemented as stand-alone Studierstube applications: Navigation, information browsing and annotation. Navigation addresses the navigational aspects of the tourist guide application, information browsing the display and interaction with pre-authored information on sights in the environment and annotation provides the interactive placement and display of user supplied information icons.

For each component we will discuss the user interface implemented, the presentation aspects and the derived state variables. Then we describe the resulting application logic and the overall implementation of the component. Having identified the data structures used by the applications we will then describe the data management schemes put in place to provide the content for the applications from a general model. Finally, we apply one of the collaboration patterns to implement an appropriate form of collaboration for each of the application components.

A last section will deal with the creation the general model. Because the three-tier architecture presented in chapter 5 separates the applications from the model itself, the applications are not affected anymore by the necessary transformations.

8.3 Tracking configuration

The central tracking requirement for the described application is to localize the user in the city environment. Moreover to accurately overlay information and highlight structures full 6DOF tracking of the user's view is required. A world-stabilized reference system was chosen as the overall reference system over other possibilities such as a body-stabilized coordinate system because it allows for simple interpretation of coordinates involved in building the system. It is more intuitive to think about a moving user than to transform the world geometry to fit a user centric reference frame.

In addition to the user's pose a ray picking interface is required to allow interaction with 3D registered information. Generally, this would require a second 6DOF input channel to control the ray. Thus, for the abstract tracking input requirements we arrived at two 6DOF input channels that would generate data in a world-stabilized reference system.

The actual devices chosen to track the user's pose were a DGPS location device and an 3DOF orientation tracker based on inertial sensors. The required tracking configuration contains a number of steps to generate the final pose: DGPS data requires transformation from the global WGS84 reference frame to the local map datum with the additional offset used to create a reasonable origin for our application. The resulting local coordinates are merged with the rotation data from the inertial device. To improve the user's experience a smoothing filter was also applied to the position data to avoid hard jumps between updates.

Moreover, we chose to separate the above operations into a dedicated process and only provide the final pose data via network transport to the application. Thus, we could exchange the tracking process with another configuration that provides only simulated pose data in a manner that is transparent to the overall application simplifying debugging and testing.

After experimenting with various tracking approaches to track an additional user interface prop for providing the ray pick interaction, we decided to settle on a simpler gaze derived ray pick. Therefore, the pose required for the ray can be derived from the user's pose by a simple transformation which was added to the overall tracking configuration.

Because the inertial sensor does not operate in a fixed reference frame and also exhibits drift we chose to add a calibration feedback loop as described in section 7.1.1 to the user's pose. The required addition was implemented in the tracking configuration of the application itself, instead of at the dedicated process driving the devices. Such an arrangement allows us to debug the calibration function with simulated pose data as well.

(a) (b)

Figure 8.3: (a) A visualization of the path to the selected target without clipping on known objects. (b) The same path clipped at an object.

8.4 Navigation application

As the first aspect of the application we want to examine the navigation interface. In navigation mode the user selects a specific target address or a desired target location of a certain type such as a supermarket or a pharmacy. The system then computes the shortest path in a known network of possible routes. It is interactive and reacts to the user's movements. It continuously re-computes the shortest path to the target if the user goes astray or decides to take another route.

The information is displayed as a series of waypoints that are visualized as cylinders standing in the environment. These cylinders are connected by arrows to show the direction the user should move along between the waypoints. Together they become a visible line through the environment that is easy to follow (see Figure 8.3(a)). The user can enable an additional arrow that points directly from her current position to the next waypoint. Buildings can clip the displayed geometry to enable additional depth perception cues between the virtual information and the real world (see Figure 8.3(b)). Finally, simple directional information is displayed, if the user is not able to perceive the next waypoint because she is looking into the wrong direction.

8.4.1 User interface

The user interface consists of selection of a destination, either from a list of available destinations, or by selecting a number of keywords that describe a certain type of destination. A number of Boolean state flags are also controlled by the user to customize the presentation of the path to the

Name	Widget type	Description
Targets	Listbox	a list of possible target addresses
Keywords	Listbox	a list of possible keywords to filter targets
Active	Togglebutton	a control to enable or disable display of navigation information
Clipping	Togglebutton	a control to enable or disable clipping of navigation information on buildings
Arrow	Togglebutton	a control to enable or disable display of an arrow pointing the user to the next waypoint
HUD	Togglebutton	a control to enable or disable the heads-up display

Table 8.1: States controlled by a 2D user interface in the navigation application.

computed destination. There is no further 3D interaction involved besides moving through the environment. The inputs described above are presented to the user in a traditional 2D GUI using a set of standard widgets provided by the Studierstube framework. Table 8.1 denotes the parameters that are controlled by the user interface and their representation as widgets.

The 3D presentation needs to display any path throughout the waypoint network that is computed by the navigation application. A simple solution is to represent all waypoints and directed edges with geometry in the scene graph and only traverse the subset of waypoints and edges that create the computed path. Thus, we created a scene graph that contains two large parts, one for the set of waypoints and one for the set of edges. Each of these parts is controlled by a switch node that allows to select a subset of children to traverse by setting the indices of the children in a field of the switch node. Then selecting and displaying a path is a simple matter of creating two lists of indices that correspond to the waypoints and edges to draw.

Some more components are required to implement the additional visual effects described above. To implement the clipping on buildings another sub-scene-graph was created that contains the buildings geometry but is only rendered into the Z-buffer and not into the frame buffer. It is traversed before the scene graph containing the path geometry and therefore obstructs any path geometry behind a building geometry because the Z-test for fragments from the path geometry fails. An additional switch allows to select whether the building geometry is actually drawn or not.

The additional arrow drawn between the user's current position and the first waypoint is created by adding a sub-scene-graph that contains a shape drawn between two given points. The points are set by the navigation ap-

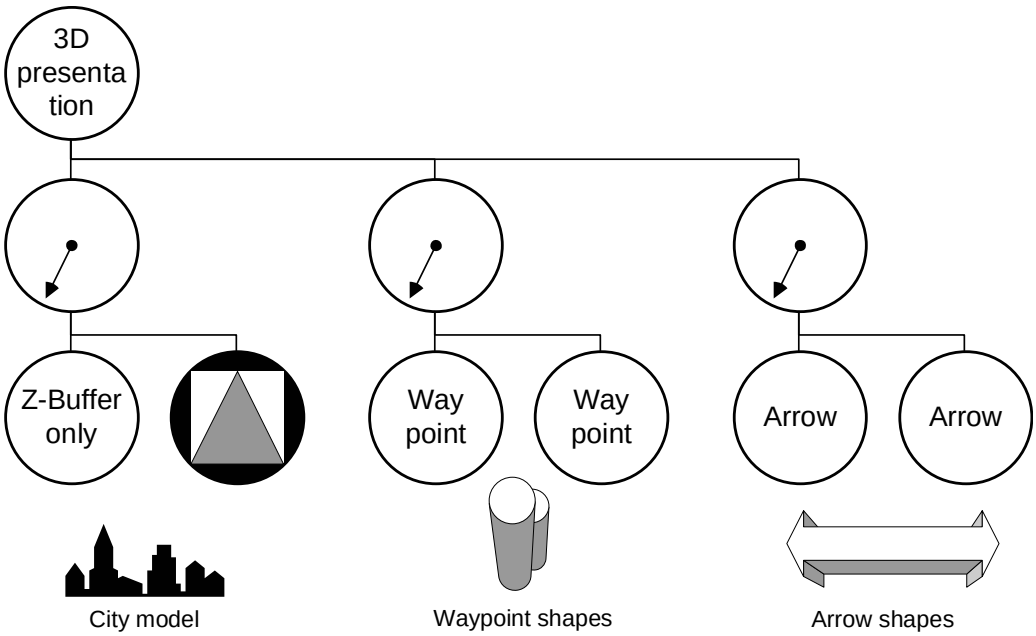


Figure 8.4: Schematics of the scene graph for the navigation application.

plication to the user's current position and the position of the first waypoint on the path. Again, a switch controls the display of this feature. Figure 8.4 gives an overview of the complete resulting scene graph.

Another part of the user interface output is the heads-up display. The display is implemented as a scene graph that renders with a fixed orthogonal camera into the user's view, thereby creating a head-stabilized display. The following set of generic information is shown in the heads-up display: The current position as textual values, the current orientation represented with a compass-like widget, the address of the currently active target, the distance to the target computed along the path, and an indication if the target is reached.

8.4.2 Application core

The application's functional core needs to deal only with computing the path from the current position to the selected destination within the available network. All other aspects are already taken care of by the components described above. We model the core as a single Open Inventor node called NavigationContext that implements the path finding algorithm. The node has a number of fields which are interpreted as configuration fields, input fields, and output fields. It simply computes the value of the output fields

Name	Type	Description
Configuration fields		
waypoint	MFString	a list of all waypoint identifiers
waypointPosition	MFVec3f	a list of the positions of all waypoints
edge	MFInt32	a list of edges as neighborhood list for each waypoint
numEdge	MFInt32	the number of edges for each waypoint in the edge list
Input fields		
destination	MFString	list of possible destinations
userPosition	SFVec3f	current position of the user
userOrientation	SFRotation	current orientation of the user
Output fields		
currentWaypoint	SFString	the waypoint the user is closest to
currentIndex	SFString	index of the currentWaypoint
path	MFString	the list of waypoints making up the path
edgeIndex	MFInt32	list of edge indices for display
nodeIndex	MFInt32	list of node indices for display
computedTarget	SFString	the closest target found
distance	SFFloat	distance to target
direction	SFRotation	direction from the user to the next waypoint
relativeDirection	SFFloat	relative azimuth between users direction and direction to the next waypoint

Table 8.2: Fields of the NavigationContext node. Configuration fields hold static data describing the waypoint network, input fields are incoming interfaces from the user interface and output fields set the presentation parameters and heads-up display information.

based on the current value of all configuration and input fields. The distinction between configuration fields and input fields is only minimal: input fields are expected to be changed by the user interface while configuration fields contain static information that is read upon startup. The node also implements the application API of the Studierstube framework. Table 8.2 shows the different fields of the NavigationContext node.

The configuration fields define the information on the network of waypoints and connecting edges. Waypoints are identified by textual ids and have a position in space both of which are stored in two lists where items at the same indices correspond to one waypoint. A list of all connected waypoints is given for each waypoint joined into one large list of indices. To

be able to separate the joint list, a second list containing only the number of items per list for each waypoint is given. Such a method of storage is similar to the one used by indexed face sets. From the given information the node builds an internal representation of the waypoint network as a directed graph with the edges weighted with the distance between two waypoints. All connections are represented as two directed edges going into both directions.

The input fields describe the input to the wayfinding algorithm. The basic information is the user's position as the starting point of the path and one or more destinations described by the textual ids for the corresponding waypoints. The algorithm then computes the shortest path using Dijkstra's algorithm [34] from the waypoint that is closest to the user's position to one of the destination waypoints. If a list of possible destinations is given, it computes the path to the closest destination along the network. Whenever either the user's position or the destination field changes, the wayfinding algorithm is executed again to provide the dynamic behaviour of the application.

The results of the computation are written to the output fields. The computed closest destination is written to the computedTarget field. The lists of waypoints and edges to traverse are output in the nodeIndex and edgeIndex fields and the textual ids in the path field. The waypoint closest to the user is output as textual id in the currentWaypoint field and as index in the currentIndex field. The overall distance along the path is written to the distance field. The additional ouputs direction and relativeDirection give information about the direction of the next waypoint relative to the user's current orientation to provide some additional displays.

The application's functional core is separated completely from the input and presentation aspects of the overall application. The connection between these components is achieved by using Open Inventor mechanisms to hook up and transform field values from the user interface widgets to the NavigationContext node and back again to the scene graph containing the presentational aspects. Some connections are also directly between the widgets and the scene graph circumventing the core functionality (see Figure 8.5).

8.4.3 Data management

The application's implementation is based on an intricately choreographed and interconnected set of components consisting of the user interface widgets, the NavigationContext node for wayfinding and the complex scene graph to display the resulting graphs. They each store parts of the information as lists of waypoint ids and position, edge lists or list of geometrical objects representing the waypoints and edges. All these lists are connected by indentical indexing, that is, the first entry in all of these lists correspond to

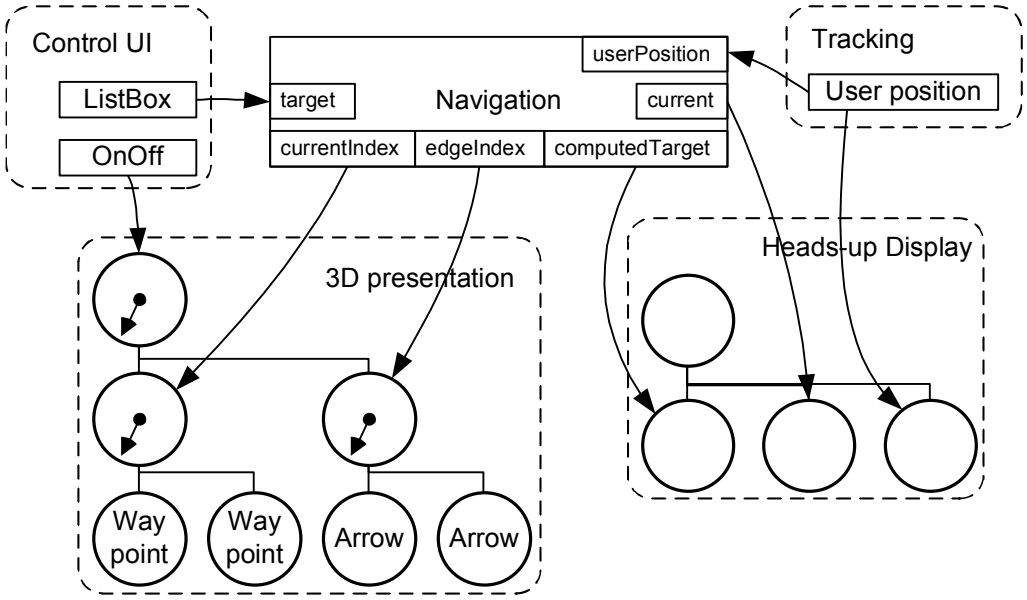


Figure 8.5: Schematics of the field connection and engine network for the navigation application. Tracking and Control UI structures provide input events to the application core object which in turn outputs data to configure the scene graph and heads-up display.

one waypoint the second to the next and so on. As the complexity to create these lists for larger networks increases dramatically, manual authoring will become increasingly prone to error and an automated method of creating the lists is required. Therefore we apply the mechanisms for data management described in chapter 5 to address this issue.

We define a new element in the model schema to explicitly represent a waypoint. The new element is called *Waypoint* and is derived from the SpatialObjectType because a waypoint does have spatial properties such as a position. The id attribute stores a unique id for each waypoint and the pose element is used to represent the waypoints position. The network is described by an additional attribute called neighbors that holds the ids of all the connected waypoints forming a directed graph. If a segment of the network is transmissible in both directions, it is represented by two edges.

The described representation allows to edit the network on a local basis for each waypoint without destroying intricate relationships between long lists and other data structures. Because it is part of our general model now, it can be reused by all kinds of applications and is not limited to the navigation application.

A dedicated transformation style sheet reads in a model and generates the lists described above automatically from the data stored in the Waypoint

elements of the model. The implementation simply needs to iterate over all Waypoint elements it encounters and build the lists by appending the extracted information to the individual lists. The geometric representations are also built by writing a scene graph corresponding to the structure described in the sections before. The presentation can be customized at this point by writing out different geometry.

Finally, the lists and the scene graph are written out in the Open Inventor ASCII file format to be used as configuration parameters in the resulting file describing the full application. The application file is then read into a Studierstube process describing the application's structure and scene graph and executing it by passing events along the connections between fields of different components.

8.4.4 Collaboration

If two or more users are present, a number of collaborative interactions are possible. The interface will present a list of all users that have joined the collaboration session. Every user can select another user and specify an interaction mode:

Follow The user can decide to follow the selected user. The navigation display will update the target location to always coincide with the waypoint closest to the selected user.

Guide The user can guide the selected user by setting the destination point of the selected user. The navigation system of the selected user will then behave as if that user had selected the target herself.

Meet This mode supports to meet halfway with the selected user. The navigation system calculates the meeting point to be halfway between the two waypoints the users are closest to. Then the destinations of both users are set to this new target. Each user can still change the common target to a more suitable location if desired.

The described functionality requires direct communication between to designed systems out of a set of participating systems. Moreover, it should also allow for disjunct pairs from the same set to work in parallel. To achieve such a direct communication we chose the model of the application proxy described in section 7.1.6. Each system has an additional Studierstube ap-

plication called UserContext that implements the collaborative navigation by using the local NavigationContext application nodes on both user's systems.

The UserContext application is running on each mobile system in a shared locale that is joined by all systems. Therefore in the shared locale there is one such application for each user. The application only present a user interface to the user on the master system where they originated from. The slaves act as proxies on behalf of the master and send events to fields of the NavigationContext application on the local system they are residing on. The communication between the master and the slave is by setting field values on the UserContext node which are propagated transparently by the DIV mechanism. Therefore the UserContext node has another set of fields dedicated to the communication between the master and the slave instances (see Table 8.3). The master and slaves simply implement different behaviors in reacting to changes of these fields.

Name	Type	Description
Configuration fields		
name	SFString	the name of the user
Input fields		
activeNeighbor	SFString	the name of the user to interact with
mode	SFInt32	variable describing the mode of operation
Output fields		
destination	SFString	the destination resulting from the operation and user given.
Communication fields		
currentWaypoint	SFString	the current waypoint of the user
currentPath	MFString	the current path of the user
currentIndex	SFInt32	the index of the current waypoint

Table 8.3: Fields of the UserContext node. The additional communication fields are used to transport data from the master to the slave instances.

The UserContext allows the selection of another user to interact with by setting the value of the input field activeNeighbor. Moreover, the user selects the mode of interaction through the mode field. As the values of these fields are also communicated to the slave instances, they have the required information to act on the master's behalf. Typically only the slave instance on the system of the user denoted in activeNeighbor acts while the other instances ignore all updates. General information about a user's state are transported in the fields currentWaypoint, currentPath and currentIndex that reflect the output of the user's NavigationContext.

In *Follow* mode the master instance acts by setting the destination of the local NavigationContext to the currentWaypoint field value of the local slave UserContext instance of the active neighbor. Therefore, the collaboration depends only on the information represented by the currentWaypoint field.

To implement the *Guide* mode the master simply sets the destination field to the selected destination for the active neighbor. The slave instance that resides on the active neighbor's system then updates the local NavigationContext to guide the neighbor to the selected destination. Here the slave acts as a proxy for the master component.

The *Meet* mode combines both functionalities. First the master uses the information of the local slave copy of the active neighbor to determine a route to the current waypoint of the neighbor. Then it uses its own slave to set the remote and the local NavigationContext to a waypoint halfway along the computed path. The result is that both the local user and the active neighbor user are guided to the same meeting waypoint.

8.5 Information browsing

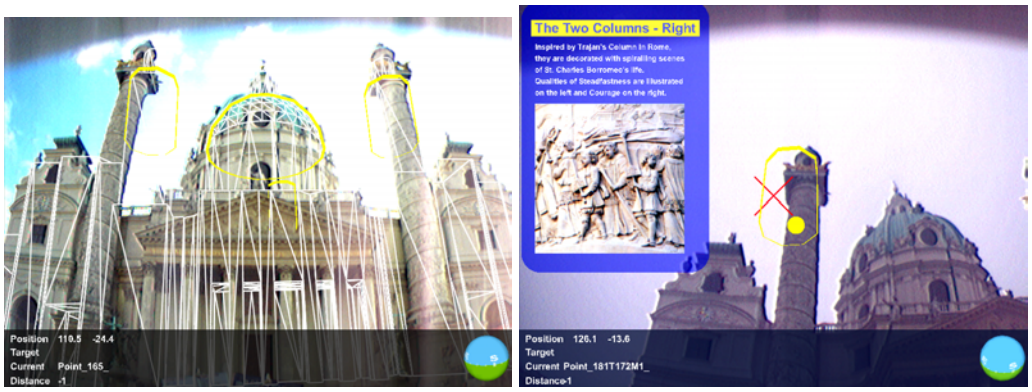
The information browsing mode presents the user with location-based information. Location referenced information icons appear in her view and are selected by looking at them. They then present additional associated information. The application conveys historical and cultural information about sights in the City of Vienna.

The information icons can have any shape for display as well as for ray intersection. In the current application we use geometric representations of parts of buildings to annotate these with cultural information. The icons appear to the user as outlines of the building parts. A virtual ray is cast through the center of the display and intersected with the information icons. The first icon that is hit triggers the display of information that is associated with it (see Figure 8.6).

The information consists of images and text which were taken from a guide book. These are shown to the user in the heads-up-display. The user can also select a subset of the icons to be active and visible by choosing a set of keywords the icons should relate to. The reduction of visible icons avoids visual clutter.

8.5.1 User interface

The information browsing component uses a 2D GUI to control the overall application states (see Table 8.4). A set of toggle buttons allows to switch



(a) (b)

Figure 8.6: (a) Different parts of the building are highlighted to show possible additional information. (b) The user selects the column by looking at it and the content is displayed.

on the application and to enable interaction with the presented targets or choose the presentation of the information icons. A list of keywords allows to select only a subset of active icons. Two additional buttons set the state for collaborative use and will be explained in the following collaboration section. In addition to the control user interface there is also 3D interaction in the form of a gaze directed ray picking operation to trigger the display of the information represented by the individual icons.

Name	Widget type	Description
On	Togglebutton	Turns application on or off
Raypicking	Togglebutton	to enable interaction by looking at targets
Show	Togglebutton	to enable highlighting of possible targets
Keywords	Listbox	a list of keywords to filter active targets for
Listen	Togglebutton	to enable receiving mode in collaborative use
Guide	Togglebutton	to enable guiding mode in collaborative use

Table 8.4: States controlled by a 2D user interface in the information browsing application.

The presentation part incorporates two different visualization requirements. The information icons need to be rendered registered to the real world depending on the selected keywords and state of the application. The information associated with a certain icon needs to be rendered for an activated icon. The latter consists of a different and independent scene graph either containing head-stabilized textual information and images or 3D models at world- or head-stabilized locations. Additionally there is also the requirement

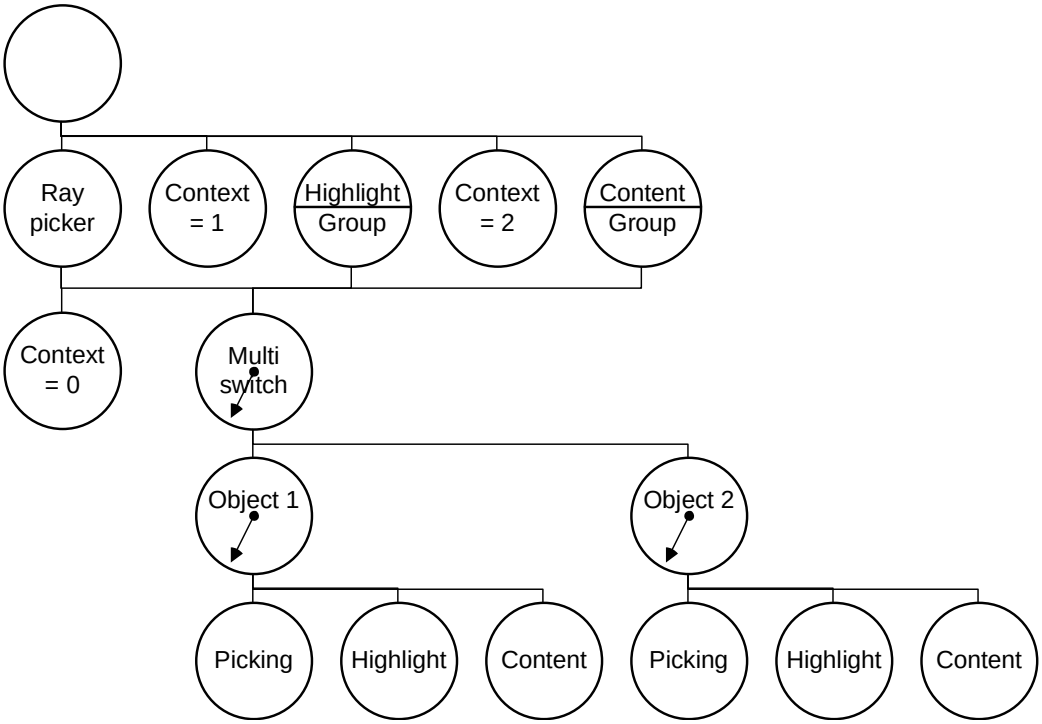


Figure 8.7: Schematics of the scene graph for the information browsing application.

of a geometrical representation of the pickable areas that is appropriate for the 3D interaction and therefore should only consist of a simple geometrical description without advanced visual effects that might require a more complex scene graph.

However, these different representations are all associated to one object, a single piece of information registered with a certain location. An organizational structure that does not loose this association simplifies selection of subsets of objects and provides for a more dynamic extension to add or remove icons at runtime. The necessary mechanism is provided by the context sensitive scene graph described in chapter 4. Each object is individually broken down into three sub scene graphs, each containing one of the above representations. An application defined well-known context index is associated to select between the representations and each representation is in turn associated with a well-known value of the context index. The resulting scene graph is visualized in Figure 8.7.

The constructed context sensitive scene graph is then reused for visualizing possible icons, to visualize a currently active icon's content and as picking geometry for the gaze directed raypicker. An SoContext node inserted before

the scene graph selects the actual representation to traverse for each object by setting the well-known context index to the appropriate value. A general switch containing the objects selects which icons to display or to pick.

The different reuse location of the context sensitive scene graph are further contained in switches that enable or disable traversal of the respective part of the scene graph in dependence of the widget states in the control user interface. For example, the keyword selection listbox outputs a list of keywords in a state field. The list of keywords is translated into a list of indices of objects in the scene graph using a map engine that uses a generic string to string map to store an arbitrary relation. The relation contains the association between the individual objects and the keywords they are associated with. Thus, the map engine outputs a list of indices into the top switch node containing only the objects that match all of the selected keywords. The output of the engine is used for the visualization of the icons as a well as for the picking geometry.

The selection of the activated icon is achieved in a similar manner. The raypicking interaction widget outputs a path into the scene graph to the basic geometry node it hit. Along this path there exists an SoContextSwitch node that contains the different representations for one object. This node is named with a unique id corresponding to the object. A converter engine takes the path as input and outputs the names of the objects along the path omitting all anonymous objects emitting only the unique id of the object. The output is again routed into another map engine that translates the unique id into the corresponding index in the top switch node of the scene graph part that renders the associated information (see Figure 8.8).

Because all of the required functionality is implemented via field connections, engines and dedicated switch nodes, no further application functionality was required. Therefore no specialized node was implemented filling the role of the application core.

8.5.2 Data management

As with the navigation component the required scene graph exhibits a complexity that prohibits extensive manual authoring. Therefore, again a data-driven approach was taken by storing the required information in the general model and then creating the scene graph structures in an automated transformation step.

The information objects are represented by another dedicated element in the model schema called *Annotation* which is derived from the SpatialObject-Type. The spatial representation is used to provide the geometry for picking which allows to author individual regions for the object. The visible geome-

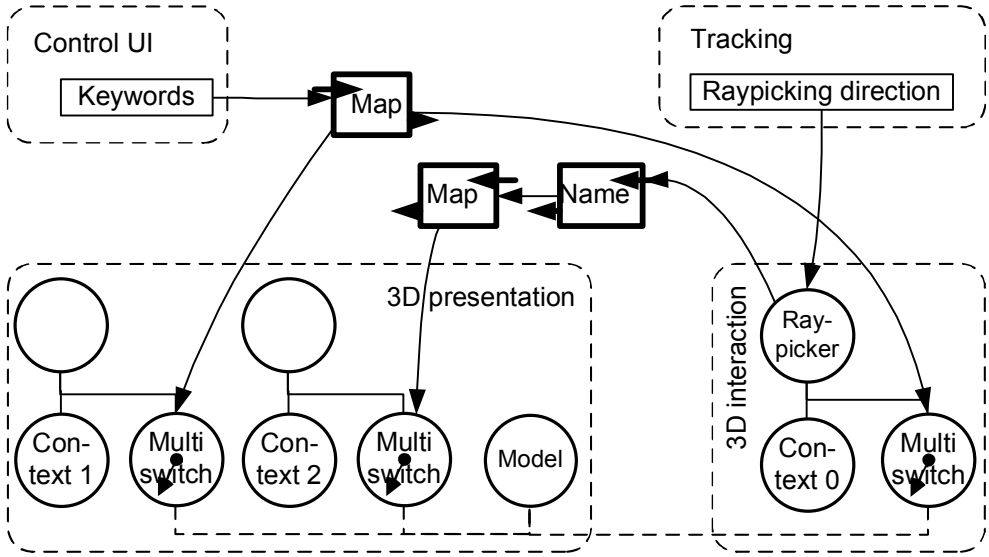


Figure 8.8: Schematics of the field connection and engine network for the information browsing application. Control user interface inputs are mapped to indices in the scene graph for display and interaction. The selected scene graph part is then mapped to indices for highlighted display.

try is derived from the picking geometry by using a stencil buffer operation to render a silhouette of the picking geometry. The information content is stored in a set of additional elements stored in the generic annotation subelement. The possible elements are a single *Title* element, a set of *Content* elements storing different information items and a *Keywords* element storing the keyword associations of the information object. The Content element stores an attribute giving the mime type of the stored content and then the content or a URI of the content file as CDATA.

The information stored in the Annotation elements is transformed by a dedicated style sheet into the scene graph described above. In addition to that, the different map engines that relate keywords and object ids to indices are also created by the style sheet. Because the style sheet creates also the scene graph representing the information content, it is the single point of customization for the presentation of the information. For example it computes the necessary arrangement of image and text representations as seen in the information overlay in Figure 8.6(b). The actual Studierstube application file only contains a reference to the created scene graph file.

8.5.3 Collaboration

The information browsing mode supports multiple users. Users can choose to share their selection of topics, or alternatively, tour guides can control the selection for a group of guided users. A user can also trigger the highlighted information on another user's display. The two user interface widgets Listen and Guide select the user's mode. If the Guide mode is active the user's keyword selection and active icon will be shared with the group of participating users. If the Listen mode is active the shared keyword and active icon information of another user will be used to set the state of the local user's display. Therefore, the local user will see the same information content that the guiding user has selected and will also use the same keyword selection, if she decides to still browse the icons on her own.

The functionality described above is implemented by using a dedicated shared scene graph that stores the currently selected keywords and the currently active icon. All information browsing applications are set as masters for the shared scene graph and can therefore change the state as required. At the same time they also receive any updates initiated by another instance.

The widgets then simply enable or disable routing of the current users selection to the shared state or vice versa. If Guide mode is active, any change to the keyword list or active icon state are routed to the shared state and in turn propagated to all other instances. If Listen mode is active, any change received from another instance is used to update the local user interface and application state.

As a result of the simple group communication protocol, there are no mechanisms to guarantee consistency or allow one user full control over the other users. However, such strict constraints were not required for the desired functionality and can be left to social protocols in a user group.

8.6 Annotation

In addition to pure browsing, users can also annotate the environment by placing virtual icons of different shapes and colors on structures and buildings in their surroundings (see Figure 8.9). Again, a virtual ray is cast through the cross hair in the heads-up display and intersected with the geometry of the buildings. If an intersection is detected a yellow sphere is placed at the intersection point to visualize it. Then the user can place a predefined 3D icon at the intersection point. The icon is oriented parallel to the tangent surface in the intersection point. The user can select between different predefined shapes and colors to use and can also choose which kinds of icons to display.

(a) (b)

Figure 8.9: (a) Annotation options and icons. (b) The building geometry used for ray intersection is overlaid.

The virtual icons are shared between different users in a collaborative session and are annotated with the name of the user who created them. In this case a user can also include the name of the icons' creator into the selection of visible icons.

The virtual markers can help to point out features on distant structures such as building facades. Users can attach and discern different meanings associated with markers by assigning different styles. They support collaborative work styles because they are shared information and can help users to communicate information about individual locations in the surrounding.

8.6.1 User interface

The user interface is again uses 3D interaction with a gaze directed ray picking operation to select the positions of 3D icons. The component is supported with a 2D GUI to enable display of 3D icons, to enable the ray picking operation and to control what kind of 3D icon to place in the environment. For collaborative use an additional filter operation also allows to filter for certain subsets depending on the author and the color of icons. Table 8.5 gives a complete list of user interface states presented in the 2D GUI.

The presentation part is a simple scene graph containing all icons set by the user. At the start of the application the scene graph is empty. As the user creates new icons the application core adds these to the scene graph. A multi switch is controlled by an engine network that maps the selected keywords to the indices in the switch to allow the list in the user interface to filter the displayed icons. The configuration of the engine network is updated by the application core to represent the current state of the scene graph.

Name	Widget type	Description
On	Togglebutton	turns the application on or off
Raypicking	Togglebutton	to enable interaction by looking at targets
Icons	Listbox	a list of keywords to filter icons
Shapes	Radiobuttons	a radiobutton group selecting one of three shapes
Colors	Radiobuttons	a radiobutton group selecting one of three colors

Table 8.5: States controlled by a 2D user interface in the annotation application.

8.6.2 Application core

The application core is the central component of the annotation component. The Open Inventor node AnnotationContext implements the core and also the Studierstube application API to provide a full featured application. See Table 8.6 for the configuration and input fields of the core component. Its main functionality is to create a new instance of a 3D icon from the selected shape and color and place it in the environment at the location determined by the ray picking interaction widget.

The basic operation of the core is to create a new instance of the scene graph configured in the field markerTemplate, substitute certain nodes in the template scene graph with the content of the input fields iconTemplate and colorTemplate to customize the template scene graph with the currently selected shape and color and add it to the presentation scene graph referenced in the field icons. The instantiated template scene graph is also configured to place its geometry at the position passed in by the input field cursorPosition and to be oriented according to the surface normal described in pickNormal. The input fields isPicking and pickButton communicate the necessary interaction information to allow the core to detect a button press during a valid picking operation to trigger the creation of a new icon.

The core also needs to update the engine network that filters the displayed icons for certain keywords. The necessary engines are referenced in the configuration fields indexMap and markerMap and are updated with the associations for a new icon when it is created. The markerKeywords field is updated to represent new color or user name keywords as required and in turn updates the listbox in the user interface with possible values to select.

Note, that the input fields for the shape and color options are not simple values derived from the 2D user interface widgets but references to sub scene graphs which are inserted into the template scene graph. The necessary translation between the widget state of the radio buttons and the final node

Name	Type	Description
Configuration fields		
icons	SFNode	the scene graph containing created icons
markerTemplate	SFNode	a template scene graph for all markers
name	SFString	local user name
indexMap	SFEngine	reference to the engine mapping ids to indices
markerKeywords	MFString	available marker and color names
markerMap	SFEngine	reference to the engine mapping user and color names to ids
Input fields		
cursorPosition	SFVec3f	position of the pick point
pickNormal	SFVec3f	surface normal in the pick point
isPicking	SFBool	flag to denote if picking is valid
iconTemplate	SFNode	current icon template node
colorTemplate	SFNode	current color template node
pickButton	SFInt32	button state of the ray pick widget

Table 8.6: Fields of the AnnotationContext node. Configuration fields hold references to engines and the scene graph updated by the node while input fields are incoming interfaces from the user interface.

reference is again accomplished with a configurable engine network. Such a mechanism decouples the application core from the user interface component and allows simple and quick changes to the user interface.

The annotation component makes no further use of data management because it does not require large and complex data sets to begin with. Therefore no data besides the building geometry for picking is derived from the general model. The building geometry however is reused from the navigation component which already uses it for clipping the path geometry against real buildings.

8.6.3 Collaboration

The implementation of the collaborative aspects makes use of the shared scene graph model similar to the information browsing component. Here each instance of the annotation component updates a shared scene graph with the new sub scene graphs representing the individual icons as they are created by the user. All other instances observe the shared scene graph for new children added to it. If a new child is received it is also added to the local presentation scene graph and engine network. As the icon scene graph

carries information on the authoring user, a new keyword entry can be added to the user interface as well using the markerKeywords field. Because only additions to the scene graph are interesting events, the lack of global ordering of these events does not matter as there exists no dependency between the individual icons. The result to the user will be the same for different orderings of updates.

The scheme does also provide for late coming users because the shared scene graph is configured to be updated from another instance. Therefore any late coming system will automatically receive the current state as soon as it joins the Distributed Inventor session. The application core will then simply add all the existing icons one after the other in a rapid iteration. Because it checks for already existing icons before adding them to the presentation scene graph, it can also deal with network failures and reconnection and will ignore the resulting replay of all existing icons.

8.7 Data acquisition

A number of aspects of the overall model were not discussed yet. An example is the general building geometry required for clipping paths in the navigation component or for picking in the annotation component. Another aspect is the incorporation of the waypoint network into the general model. Because the applications start with data derived from the general model, they need not be concerned with the creation and maintenance of the model itself. These topics will be further described in the following.

A general model was build to serve as the basis for all tourist guide applications. A single source would allow simpler editing and maintenance of the model as changes to the geometry and content would propagate appropriately to each subcomponent automatically. The general model was stored in the BAUML format described in appendix A. A 3D model of a part of Vienna was obtained from the cartography department of the city administration (see Figure 8.10). This model was created as a reference model of Vienna, and is part of the official general map of the city [35]. The model itself was delivered in the VRML format which we converted to the Open Inventor ASCII file format. Then we developed a dedicated import tool that read the Open Inventor file as a scene graph and constructed a BAUML representation. The created 3D model was used as the base for the general model.

The department of Geoinformatics at Vienna University of Technology supplied us with a network of accessible routes for pedestrians, delivered in GML2 format [31], an XML based file format used in the geographic

Figure 8.10: A subset of the 3D model of the City of Vienna. It includes a digital elevation model, building blocks and roofs. 3D model curtesy of Vienna City Administration.

information systems (GIS) community. This model was derived from the general map of Vienna and is represented as an undirected graph. Each node in this graph is geo-referenced and used as a way point in the navigation model. For each building, a so-called address point was defined and included into the path network to be able to construct a path to this address. As navigation graph data was available in an XML based format, a simple XSLT transformation script was sufficient to incorporate this data into the model.

Furthermore, annotation information such as businesses located at certain addresses was derived from the general map of Vienna. This information is connected to address points in the spatial database.

It was necessary to compute the intersection of the 3D model data and the navigation graph, as the relevant input data was derived from two overlapping sections of the city map. This was achieved by computing a subset of the model within a given bounding box and then repairing the internal structures of the navigation graph to make sure the data is still coherent after the trimming. The maintenance tool directly reads from and writes to the common data model.

Finally, we placed the icons as spatial representations of interesting information into our model. Cultural information taken from a guide book was

included at various places to provide the detailed data for the information browsing component.

8.8 Summary

This chapter demonstrated the augmented reality application design principles outlined in chapter 7 at a larger application for outdoor navigation and information browsing. The application of a generic AR software framework such as Studierstube allowed us to leverage the graphical and interaction possibilities of modern 3D rendering libraries and simplified development to a great extent. Using an iterative development process we could enhance the user interface rapidly because changes to the presentation and interaction could be implemented and tested within short turn around cycles. The separation of concerns with the described software components captured the areas requiring change accurately. Together with the clear-cut interfaces between them, any modification could be tracked and applied to a well-defined set of components without touching other parts of the application.

The described system was outfitted with data surrounding the area of Karlsplatz and adjacent building blocks down to Gußhausstraße in Vienna. The area of Karlsplatz proved to be a good testing ground because it is open enough to allow reception of GPS signals for positioning, has a somewhat complex network of foot paths through Resselpark and a number of famous tourist attractions such as Karlskirche are situated at its border. Finally, it is close enough to our institute to allow frequent visits for development and testing purposes. The images throughout this chapter where captured during such test runs. Also, the user's general appearance would usually add to the interest in the location demonstrated by traditional tourists.

The presented application tries to give an outlook into possible future user interfaces for location-based services. Although many of the implemented user interface features have been demonstrated before, our work exceeds former work in two areas. Firstly, the collaboration features add another dimension to the possibilities of such systems by supporting groups of users in their tasks. While the features of the system are implemented, we need more tests of the collaborative aspects to determine useful and interesting extensions. Secondly, the integrated approach to handling the data required by a location-based service allows the system to scale to environments of realistic size. The use of a flexible data model also simplifies extending the system with future applications that will have new requirements for data to be stored in the repository.

Chapter 9

Conclusions

The presented work focused on reusable and flexible software designs for a number of software components in a typical augmented reality application. The demonstration applications showed how to apply the described designs in common AR application areas. The high-level programmability of the individual solutions allowed for rapid development using an iterative process by small groups of one or two developers. We attribute the ease of development and flexibility of the application designs to a large degree to this feature. Therefore, we believe that such advanced designs can add to the efficiency of the development process while increasing the quality of the resulting applications.

The main point that we learned, is to apply just the right level of abstraction to the individual problem areas. It is important to preserve the flexibility of a design while freeing the developer from the mundane task of creating the targeted functionality by building on a low-level API. Therefore each design encapsulates certain low-level operations and only provides configurability at a carefully chosen level.

The validity of a generic application design is harder to prove. The described design and guidelines to develop applications by it, are an attempt to formalize and describe design knowledge accumulated in the process of implementing a substantial number of applications within the Studierstube framework. The results from teaching the generic design to beginners and from refactoring existing applications confirm the design.

Beginners to the Studierstube framework find it easier focusing on the component they have to implement than having to solve everything at once. The high-level programmability made complex AR applications more accessible because it invites learning by experimentation. Refactoring existing applications such as Construct3D usually provided for more flexibility within the existing application design. As the design work progressed, the imple-

mentations became less likely to break with the slightest change to the user interface or tracking configurations.

Augmented reality is evolving from the state of early research that needs to address fundamental problems such as improving tracking accuracy, input devices, rendering performance and appropriate output devices to a state that focuses more on the applications and their content. Usability of applications in real work environments and efficient ways to develop these come into focus.

Knowledge about developing augmented reality applications and appropriate user interfaces will become more important in the future. The current state of development tools requires extensive knowledge in software design, operation of equipment and creativity for and insight into the user interface aspects. Therefore, important research questions are: How can the development of AR applications be simplified? What tools allow a wider audience of developers and designers to craft AR experiences?

A number of research directions can be deduced from this question:

- How would a tool look like that allows simple development of an AR application? It needs to integrate tracking configuration and operation, graphics design and interaction. At the same time it should operate on a level above software development to reach a wider group of people.

- Automated configuration and calibration tools for tracking equipment. It should be possible to build tools that allow us to assemble an AR setup consisting of several trackers and configure it with a series of mouse clicks similar to installing an application on a modern operating system.

- Content generation for AR applications requires additional types of content beyond the visible models. Models of real objects are required for interaction, occlusion and or display management, all of which are common techniques in AR applications. Thus, AR data structures appear to model properties that are common between different applications. If this is true, it should be possible to extract these common properties into a generic AR related ontology that can form the basis for more intelligent and communicative AR applications.

- Reusable patterns for interaction in AR applications. Basic research is able to provide us with different interaction forms. The next step is to evaluate for which tasks they are appropriate and codify the knowledge in well documented interaction patterns.

More specialized research directions for the work presented are manifold. Each of the sub topics offers new directions that are interesting to pursue. A short discussion of each topic will be presented in the following sections.

9.1 Data flow network

The current OpenTracker implementation has a set of shortcomings. The data type processed is a fixed structure tailored towards a specific application. An extension to different data structures will enable multi-modal processing of input data and expand application area of the OpenTracker concept.

Runtime reconfiguration of the tracking graph would allow a number of interesting applications. A dedicated tracking configuration tool can build up a configuration based on user input and simplify setting up an AR system. Auto-calibration of a running system becomes possible and would improve the registration errors of the computer generated images transparently and without intervention by a human operator.

A more ambitions research direction is Ubiquitous Tracking which aims to provide an ubiquitous infrastructure service to AR applications. An application can register with a UbiTrack service and request tracking information on objects it is interested in. The service would then automatically compute a configuration based on the available tracking devices and send it to the application. A reconfigurable OpenTracker layer can use the configuration to provide the required tracking data to the application. The actual tracking devices and required configuration would be transparent to the application and could change at runtime as required.

9.2 Context sensitive scene graph

A promising direction of future work is to extend the test operations on the context with more complex expressions based on basic set theory. Such an approach will extend the expressiveness of the structural variations. However, a good trade-off between the possible expressiveness and the complexity of the expressions has to be found. An overly complex language will only add the available implementation options without providing a clear design to the application programmer.

Allowing other data types such as floating point values or using names to index the context could simplify the use of the context sensitive scene graph. Floating point values would allow for continuous metrics such as distance to be used in expressions of switch tests. A number of user interface

techniques such as information filtering based on distance could be added at the declarative scene graph level without further implementation work.

By recursively combining changes to the context and switches based on the context a simple declarative programming framework can be created. It is interesting to investigate the theoretical limits of the resulting language.

9.3 Data management

To provide a ubiquitous AR experience a distributed system similar to the World Wide Web will be required. Any such system will need a common language to describe data and exchange between information providers and client applications. The GIS community has experience with large scale geographic data stores but has sofar focused mostly on 2D information that is fairly static. Mobile AR applications will require more 3D information and a dynamic model that captures the current state of the environment of the user. Therefore, a common model language to describe AR specific models and data in a ubiquitous environment is required.

A complete scalable production system for mobile AR will require a server-based persistent database, which can be accessed by clients over a network. The work described here used a simple file based approach and executed the transformations off-line.

The transformation layer can either be implemented at the data store or as part part of the client application that filters all communications with the data store through the transformation layers. An advanced design could split the transformations layer into functions running on the server such as filtering and selections and pure data structure generators at the client side, for optimal use of network bandwidth.

We have begun to develop such a service facility based on an XML database to store the model as XML fragments combined with a transformation engine and network access. A client library provides access via the HTTP protocol and allows additional XSLT transformations on the received result sets.

9.4 Managing Collaboration

The technical foundations for collaborative AR applications are now well understood and available by using existing software frameworks such as the Studierstube. But a plethora of applications that are either static or move with users in an ubiquitous computing environment pose the problem of

providing efficient and scalable user interfaces for managing visibility, access or simply distribution of these applications.

Users should not be overwhelmed by complex interactions or large number of applications that need to be manually controlled. An intelligent system should provide intuitive functions to manage the participation in joint work sessions with other users. Besides parameters used in current communication systems such a lists of known participants or searches for common interests, parameters specific to AR applications could be used such as location or visibility of real objects.

Application development itself should also not be constrained or become more complex because of the required control. A sufficiently large ubiquitous application environment cannot assume that all active applications are tested to integrate and present the same coherent interface to the user. Therefore, the supporting software infrastructure should provide the required functions to allow integration of applications that were developed independently.

Appendix A

BAUML definition

This chapter gives a complete definition and description of the BAUML XML language that is used to describe building models and tracking infrastructure.

A.1 Global simple types

Simple types define data types that can be used in attributes of XML elements or in the CDATA section of an element with a simple production only. They allow to describe finite valued or numerical types and restrictions or repetitions of such types. The following simple types are defined in the BAUML language as support types to provide some restriction for attributes of later defined complex types.

UnitSphereValue

Double values in the interval $[-1, 1]$ which is a basic type for quaternion representation.

Definition

```
<simpleType name="UnitSphereValue">
  <restriction base="xs:double">
    <maxInclusive value="1" fixed="false"/>
    <minInclusive value="-1" fixed="false"/>
  </restriction>
</simpleType>
```

DoubleList

A list of double values, base type for restricted data types like vectors, rotations, etc.

Definition

```
<simpleType name="DoubleList">
  <list itemType="xs:double"/>
</simpleType>
```

IntegerList

A list of integer values, useful for list of indices.

Definition

```
<simpleType name="IntegerList">
  <list itemType="xs:integer"/>
</simpleType>
```

UnitSphereValueList

A list of UnitSphereValue, base type for restricted data types like Quaternions etc.

Definition

```
<simpleType name="UnitSphereValueList">
  <list itemType="xs:double"/>
</simpleType>
```

Vec3

A simple type storing three double values separated by spaces.

Definition

```
<simpleType name="Vec3">
  <restriction base="DoubleList">
    <length value="3" fixed="false"/>
  </restriction>
</simpleType>
```

Quaternion

A simple type storing four double values in the interval $[-1, 1]$ separated by spaces.

Definition

```
<simpleType name="Quaternion">
  <restriction base="UnitSphereValueList">
    <length value="4" fixed="false"/>
  </restriction>
</simpleType>
```

A.2 Global complex types

Global complex types are used to model various parts of the model types. By representing these structures independently from the model types, they can be reused and extended easily.

RepresentationType

Geometrical representation of SpatialObjects. Any kind of representation could be possible. So far we support the simple vertex list and polygon model for BAUML which are modelled by the subelements `Vertex` and `Polygon`.

Definition

```
<complexType name="RepresentationType" mixed="false" abstract="false">
  <choice minOccurs="1" maxOccurs="1">
    <sequence minOccurs="1" maxOccurs="1">
      <element name="Vertex">
      ...
      </element>
      <element name="Polygon">
      ...
      </element>
    </sequence>
  </choice>
</complexType>
```

Subelement	Description
Vertex	A single vertex in the representation.
Polygon	A polygon indexing into the list of vertices.

Example

```
<representation>
  <Vertex position="0 1 0" />
  <Vertex position="0 1 1" />
  ...
  <Polygon vertices="0 1 3 2" type="portal" name="toHallway" />
  ...
</representation>
```

Vertex

This element type stores a single vertex of a polygon based geometrical representation. At least one point is necessary for a non empty representation.

Definition

```
<element name="Vertex" maxOccurs="unbounded" minOccurs="1"
         nillable="false">
  <complexType mixed="false" abstract="false">
    <attribute name="position" type="Vec3" use="optional"/>
    <attribute name="id"/>
  </complexType>
</element>
```

Attribute	Description
position	The position attribute specifies the vertex position as x y z.
id	a unique id for each vertex to enumerate them

Example

```
<Vertex position="0 1 0" id="1" />
```

Polygon

A polygon specifies a list of vertices it is built from. In addition to that it can specify a certain type.

Definition

```
<element name="Polygon" minOccurs="0" maxOccurs="unbounded"
         nillable="false">
  <complexType mixed="false" abstract="false">
    <attribute name="vertices" type="IntegerList" use="required"/>
    <attribute name="type" use="required">
      <simpleType>
        <restriction base="xs:string">
          <enumeration value="wall"/>
          <enumeration value="floor"/>
          <enumeration value="portal"/>
          <enumeration value="ceiling"/>
          <enumeration value="special"/>
        </restriction>
      </simpleType>
    </attribute>
    <attribute name="name" type="xs:NCName" use="optional"/>
  </complexType>
</element>
```

Attribute	Description
vertices	list of indices of vertices the polygon is built from. The indices use the implicit numbering as the vertices appear and not the id attribute of the Vertex element.
type	The polygon can be of one of the following types: wall, floor, portal, ceiling, special.
name	A polygon can also have a name. This is used to relate the portal polygons to certain portals, for example.

Example

```
<Polygon vertices="0 1 3 2" type="portal" name="toHallway" />
```

PoseType

This complex type specifies the pose of SpatialObjects. It provides different ways to specify the pose using either the Transformation or the MatrixTransform sub-elements.

Definition

```
<complexType name="PoseType" mixed="false" abstract="false">
  <choice minOccurs="1" maxOccurs="1">
    <element name="Transformation">
    ...
    </element>
    <element name="MatrixTransform">
    ...
    </element>
  </choice>
</complexType>
```

Subelement	Description
Transformation	an element describing a transformation composed from a translation, rotation and scale.
MatrixTransform	an element describing a 4×4 transformation matrix.

Example

See Transformation or MatrixTransform for examples on the contents of an element of this type.

Transformation

This specifies a simple affine transformation consisting of a scale, a rotation and a translation in that order. The rotational part can be specified in different ways.

Definition

```
<element name="Transformation" minOccurs="1" maxOccurs="1"
         nillable="false">
  <complexType mixed="false" abstract="false">
    <attribute name="translation" type="Vec3" use="optional"
               default="0 0 0"/>
    <attribute name="rotation" type="DoubleList" use="optional"
               default="0 0 1 0"/>
    <attribute name="rotationType" use="optional"
               default="axisangle">
      <simpleType>
        <restriction base="xs:string">
          <enumeration value="axisangle"/>
          <enumeration value="quaternion"/>
          <enumeration value="matrix"/>
          <enumeration value="euler"/>
        </restriction>
      </simpleType>
    </attribute>
    <attribute name="scale" type="Vec3" use="optional"
               default="1 1 1"/>
  </complexType>
</element>
```

Attribute	Description
translation	the translational component of the Transformation.
rotation	gives the rotational part of the transformation. The format itself is specified by the rotationType attribute.
rotationType	this attribute specifies the format of the rotation attribute. It allows the following choices :
	axisangle gives the rotation as four doubles. The first three describe the axis of rotation and the fourth the angle in radians.
	quaternion gives the rotation as a quaternion. The first three entries specify the x, y, z components of the vectorial part. The fourth entry is the homogenous part of the quaternion.
	matrix gives the rotation as a 3×3 orthogonal matrix with determinant 1.
	euler gives the rotation as Euler angles around x, y and z.
scale	gives the non-uniform scale of the transformation.

Example

```
<Transformation translation="1 0 0" scale="1 1 0.5"
                rotationType="quaternion" rotation="1 0 0 0" />
```

MatrixTransform

This specifies a general 4×4 transformation matrix in the usual manner. The upper-left 3×3 submatrix specifies the linear transformation on the 3 dimensional space and the 4^{th} column vector the translation.

Definition

```
<element name="MatrixTransform" nillable="false" abstract="false">
  <complexType mixed="false" abstract="false">
    <attribute name="matrix" use="optional">
      <simpleType>
        <restriction base="DoubleList">
          <length value="16" fixed="false"/>
        </restriction>
      </simpleType>
    </attribute>
  </complexType>
</element>
```

Attribute	Description
matrix	an arbitrary 4×4 transformation matrix.

Example

```
<MatrixTransform matrix=" 0  1  3  2
                          4  5  6  7
                          8  9 10 11
                         12 13 14 15" />
```

A.3 Basic types

The basic types are the XML schema types that all model elements are derived from. Therefore, they describe the basic properties of the most general entitites. The simple and complex types described before are used to build up the basic types. There are no examples given for these types because they are implemented using the global elements Object, SpatialObject and SpatialContainer where examples are given.

ObjectType

Everything is represented as an object. Objects generally will follow this type.

Definition

```
<complexType name="ObjectType" mixed="false" abstract="false">
  <sequence minOccurs="0" maxOccurs="1">
```

```
    <element name="annotation" minOccurs="0" maxOccurs="1"
             nillable="false"/>
  </sequence>
  <attribute name="id" type="xs:ID" use="optional"/>
</complexType>
```

Attribute	Description
id	The id attribute allows to define a unique identifier for any object of this type.

Subelement	Description
annotation	Annotation data for all objects. The idea is to add any metadata of objects here. This could also included application specific data. Therefore we specify so far no further content model, but allow any content here. RDF might be a good way to specify annotation data.

SpatialObjectType

Any object that represents something at a location. It includes a pose and a geometric representation. Abstract objects at a location simply have no representation.

Definition

```
<complexType name="SpatialObjectType" mixed="false"
             abstract="false">
  <complexContent>
    <extension base="ObjectType">
      <sequence minOccurs="0" maxOccurs="1">
        <element name="pose" type="PoseType" minOccurs="0"
                 maxOccurs="1" nillable="false"/>
        <element name="representation" type="RepresentationType"
                 minOccurs="0" nillable="false" abstract="false"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Subelement	Description
pose	The subelement pose is of type PoseType and represents the location of the object with respect to its containing object.
representation	The subelement representation is of type RepresentationType and contains the shape of the object.

SpatialContainerType

An object that is also a container of other spatial objects. It adds a children element to the general SpatialObject. The childrens pose is interpreted relative to the SpatialContainerType parent object. This allows us to build the ususal hierarchical spatial models. So far this is the only concept that we codify in our ontology. We also use the hierarchical structure of the XML format to express the relationship of containment implicitly.

Definition

```
<complexType name="SpatialContainerType" mixed="false"
             abstract="false">
  <complexContent>
    <extension base="SpatialObjectType">
      <sequence minOccurs="0" maxOccurs="1">
        <element name="children" minOccurs="0" maxOccurs="1"
                 nillable="false">
          <complexType mixed="false" abstract="false">
            <sequence minOccurs="0" maxOccurs="unbounded">
              <element ref="Object" minOccurs="1"
                       maxOccurs="1" nillable="false"/>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Subelement	Description
children	This subelement stores the list of children of the container type node. It allows all elements in the same substitution group as Object.

A.4 Global elements

The global elements are the definitions of the actual elements used in model files that use the BAUML schema. Each of the elements described here corresponds to a single entity in the model. The elements are based and derived from on of the basic types described in the last section. To make the derivation hierarchy accessible to automatic tools every element carries an attribute baseType that identifies the basic type it is derived from. This allows applications that do not know anything about the element type to revert to some default behavior.

Object

This element is derived from the basic type ObjectType and can represent any object. It can be used everywhere.

Definition

```
<element name="Object" nillable="false" abstract="false">
  <complexType mixed="false" abstract="false">
    <complexContent>
      <extension base="ObjectType">
        <attribute name="baseType" type="xs:NCName" use="required"
                   fixed="ObjectType"/>
      </extension>
    </complexContent>
  </complexType>
</element>
```

Attribute	description
baseType	Fixed value set to ObjectType

Example

```
<Object id="myObject" baseType="ObjectType">
  <annotation>
    a simple object without much more information.
  </annotation>
</Object>
```

SpatialObject

This element describes a spatially located object with or without a spatial representation. It can be used to represent any object with a shape or location without any further information attached to it. It implements the basic type SpatialObjectType.

Definition

```
<element name="SpatialObject" substitutionGroup="Object" nillable="false"
         abstract="false">
  <complexType mixed="false" abstract="false">
    <complexContent>
      <extension base="SpatialObjectType">
        <attribute name="baseType" type="xs:NCName" use="required"
                   fixed="SpatialObjectType"/>
      </extension>
    </complexContent>
  </complexType>
</element>
```

Attribute	description
baseType	Fixed value set to SpatialObjectType

Example

```
<SpatialObject  baseType="SpatialObjectType">
  <representation>
    <Vertex position="-16 -17 -22"/>
    ...
    <Polygon vertices="0 6 7 12"/>
    ...
  </representation>
  <pose>
    <Transformation translation="0 1 2" scale="0.5 0.5 0.5" />
  </pose>
</SpatialObject>
```

SpatialContainer

This element describes a spatially located object that contains other objects and is derived from SpatialContainerType. If these are spatial objects their location is relative to this object. Thus, it allows to build models using spatial hierarchies.

Definition

```
<element name="SpatialContainer" substitutionGroup="Object"
         nillable="false" abstract="false">
  <complexType mixed="false" abstract="false">
    <complexContent>
      <extension base="SpatialContainerType">
        <attribute name="baseType" type="xs:NCName" use="required"
                   fixed="SpatialContainerType"/>
      </extension>
    </complexContent>
  </complexType>
</element>
```

Attribute	Description
baseType	Fixed value set to SpatialContainerType.

Example

```
<SpatialContainer  baseType="SpatialContainerType">
  <representation>
    <Vertex position="-16 -17 -22"/>
    ...
  </representation>
  <pose>
    <Transformation ... />
  </pose>
```

```
  <children>
    ...
  </children>
</SpatialContainer>
```

Building

A building in the model. This element is a simple derivation from the SpatialContainerType to only mark a certain entity as a building. It does not add any further information and acts as a SpatialContainer element in every other aspect.

Definition

```
<element name="Building" substitutionGroup="Object" nillable="false"
         abstract="false">
  <complexType mixed="false" abstract="false">
    <complexContent>
      <extension base="SpatialContainerType">
        <attribute name="baseType" type="xs:NCName" use="required"
                   fixed="SpatialContainerType"/>
      </extension>
    </complexContent>
  </complexType>
</element>
```

Attribute	Description
baseType	Fixed value set to SpatialContainerType.

Example

```
<Building id="csbuilding" baseType="SpatialContainerType">
  <representation>
    <Vertex position="-16 -17 -22"/>
    ...
  </representation>
  <children>
    <Room id="Floor0">
      ...
    </Room>
    ...
  </children>
</Building>
```

Room

A room in the BAUML part of the format. It is also derived from SpatialContainerType and in addition specifies portals. These are special polygons that are connected to other polygons in other rooms. The portals are specified in

a special Portal child element and reference polygons of the representation by name.

Definition

```
<element name="Room" substitutionGroup="Object" nillable="false"
         abstract="false">
  <complexType mixed="false" abstract="false">
    <complexContent>
      <extension base="SpatialContainerType">
        <sequence minOccurs="1" maxOccurs="1">
          <element name="portals" minOccurs="0" maxOccurs="1"
                   nillable="false">
            <complexType mixed="false" abstract="false">
              <sequence minOccurs="1" maxOccurs="1">
                <element name="Portal">
                ...
                </element>
              </sequence>
            </complexType>
          </element>
        </sequence>
        <attribute name="baseType" type="xs:NCName" use="required"
                   fixed="SpatialContainerType"/>
      </extension>
    </complexContent>
  </complexType>
</element>
```

Attribute	Description
baseType	Fixed value set to SpatialContainerType.

Subelement	Description
portals	This subelement lists the portals of the room. It contains Portal elements which are described next.

Example

```
<Room baseType="SpatialContainerType" id="HE0446">
  <representation>
    <Vertex position="-3 -1 -0"/>
    ...
    <Polygon type="floor" vertices="0 6 7 12"/>
    <Polygon name="HE0435" type="portal" vertices="14 15 16 17"/>
    ...
  </representation>
  <children>
    <ARToolkitMarker baseType="SpatialObjectType" pattern="pattern12">
      ...
    </ARToolkitMarker>
    ...
```

```
  </children>
  <portals>
    <Portal polygon="HE0435" polygonnb="HE0446" room="HE0435"/>
  </portals>
</Room>
```

Portal

A Portal defines directed link to another room and a portal therein.

Definition

```
<element name="Portal" maxOccurs="unbounded" minOccurs="1"
         nillable="false">
  <complexType mixed="false" abstract="false">
    <attribute name="room" type="xs:string" use="required"/>
    <attribute name="polygon" type="xs:string" use="required"/>
    <attribute name="polygonnb" type="xs:string" use="optional"/>
  </complexType>
</element>
```

Attribute	Description
room	the id of the target room.
polygon	the name of the polygon in this room.
polygonnb	The name of the target polygon in the target room. This is not required.

Example

See the Room element for an example of the use of the Portal element.

ARToolkitMarker

An ARToolkitMarker element represents explicitly a fiducial marker for optical tracking using the ARToolkit library. It is specified by either a set of corners or its pose and the size of the marker. If it is represented by a set of corners, it must contain 3 or 4 vertices in this order: top-left, top-right, bottom-left, and optional: bottom-right. In addition, it stores bookkeeping information such as an identifying number and a pattern identifier.

Definition

```
<element name="ARToolkitMarker" substitutionGroup="Object"
         nillable="false" abstract="false">
  <complexType mixed="false" abstract="false">
    <complexContent>
      <extension base="SpatialObjectType">
        <attribute name="number" type="xs:integer" use="required"/>
        <attribute name="baseType" type="xs:NCName" use="required"
                   fixed="SpatialObjectType"/>
```

```
        <attribute name="pattern" type="xs:string" use="optional"/>
        <attribute name="size" type="xs:float" use="required"/>
        <attribute name="overrideTrigger" type="xs:boolean" use="optional"
                   default="false"/>
      </extension>
    </complexContent>
  </complexType>
</element>
```

Attribute	Description
baseType	Fixed value set to SpatialObjectType.
number	the number of the marker similar to the id but only for ARToolkitMarker elements.
pattern	an identifier of the pattern used. The interpretation of the identifier depends on the implementation
size	the size of the square pattern in meters.
overrideTrigger	a flag to denote whether the marker can override any relative computation, if tracked by the system.

Example

```
<ARToolkitMarker baseType="SpatialObjectType" number="41"
                 pattern="pattern12" size="0.19">
  <pose>
    <Transformation rotation="-0.942051 0.008294 -0.335096 0.022247
                     0.999770 -0.015871 0.334732 -0.019795 -0.942018"
                     rotationType="matrix"
                     translation="-17.581115 -15.350913 -24.893107"/>
  </pose>
</ARToolkitMarker>
```

Waypoint

A waypoint is a node in a street skeleton graph for outdoor navigation. It has a pose and probably an empty representation. In addition it adds links to all its neighbor nodes to build a navigation graph.

Definition

```
<element name="Waypoint" substitutionGroup="Object" nillable="false"
         abstract="false">
  <complexType mixed="false" abstract="false">
    <complexContent>
      <extension base="SpatialObjectType">
        <attribute name="baseType" type="xs:NCName" use="required"
                   fixed="SpatialObjectType"/>
        <attribute name="neighbors" type="xs:string" use="required"/>
      </extension>
    </complexContent>
```

```
  </complexType>
</element>
```

Attribute	Description
baseType	Fixed value set to SpatialObjectType.
neighbors	This attribute stores a list of ids of the neighbor waypoints.

Example

```
<Waypoint id="P100" baseType="SpatialObjectType" neighbors="P12 P14 P68">
  <pose>
    <Transformation translation="197 0 -216"/>
  </pose>
</Waypoint>
```

Bibliography

[1] NTP: The network time protocol. http://www.ntp.org/, February 16 2004.

[2] A. Aarsten, D. Brugali, and G. Menga. Patterns for three-tier client/server applications. In *Proc. PLoP'96*, Allerton Park, Illinois, USA, September 4–6 1996. Addison-Wesley.

[3] M. Addlesee, R. Curwen, S. Hodges, A. Hopper, J. Newman, P. Steggles, and A. Ward. A sentient computing system. *IEEE Computer: Location-Aware Computing*, August 2001.

[4] S. Adler et al. Extensible stylesheet language (XSL) 1.0. http://www.w3.org/TR/xsl/, October 15 2001.

[5] J. Airey, J. Rohlf, and F. P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, pages 41–50, 1990. not read.

[6] AT&T. Graphviz. http://www.research.att.com/sw/tools/graphviz/, visited February 20 2004.

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

[8] Y. Baillot, D. Brown, and S. Julier. Authoring of physical models using mobile computers. In *Proc. ISWC 2001*, pages 39–46.

[9] H. Bal, M. Kasshoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1990.

[10] J. W. Barrus, R. C. Waters, and D. B. Anderson. Locales and beacons: Efficient and precise support for large multi-user virtual environments. Technical Report TR-95-16a, MERL, August 1995.

[11] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reichner, S. Riss, C. Sandor, and M. Wagner. Design of a component-based augmented reality framework. In *Proc. ISAR 2001*, pages 45–54, New York, New York, USA, October 29–30 2001. IEEE and ACM.

[12] M. Bauer, O. Hilliges, A. MacWilliams, C. Sandor, M. Wagner, G. Klinker, J. Newman, G. Reitmayr, T. Fahmy, T. Pintaric, and D. Schmalstieg. Integrating Studierstube and DWARF. In *Proc. STARS 2003*, pages 1–5, Tokyo, Japan, October 7 2003.

[13] J. Begole, M. Rosson, and C. Shaffer. Flexible collaboration transparency: Supporting worker independence in replicated application-sharing systems. *ACM Transactions on Computer-Human Interaction*, 6(2):95–132, 1999.

[14] J. Begole, C. Struble, C. Shaffer, and R. Smith. Transparent sharing of java applets: A replicated approach. In *Proc. ACM User Interface Software and Technology (UIST'97)*, pages 55–64. ACM, 1997.

[15] B. Bell, S. Feiner, and T. Höllerer. View management for virtual and augmented reality. In *Proc. UIST'01*, pages 101–110, Orlando, Florida, USA, November 11–14 2001. ACM.

[16] B. Bell, T. Höllerer, and S. Feiner. An annotated situation-awareness aid for augmented reality. In *Proc. UIST'02*, pages 213–216, Paris, France, October 27–30 2002. ACM.

[17] K. A. Bharat and L. Cardelli. Migratory applications. In *Proc. ACM User Interface Software and Technology (UIST95)*, pages 133–142. ACM, 1995.

[18] M. Billinghurst, J. Bowskill, J. Morphett, and M. Jessop. A wearable spatial conferencing space. In *Proc. ISWC'98*, Pittsburgh, Penn., USA, October 19–20 1998. IEEE and ACM.

[19] M. Billinghurst and H. Kato. Collaborative mixed reality. In *Proc. ISMR'99*, pages 261–284, Yokohama, Japan, 1999. Springer Verlag.

[20] M. Billinghurst and H. Kato. Real world teleconferencing. In *Proc. CHI'99*, Pittsburgh, PA, USA, May 15-20 1999. ACM.

[21] M. Billinghurst, S. Weghorst, and T. Furness. Shared space: An augmented reality interface for computer supported collaborative work. In *Proc. of Collaborative Virtual Environments Workshop '96*, Nottingham, Great Britain, September 19–20 1996.

[22] O. Bimber and B. Fröhlich. Occlusion shadows: Using projected light to generate realistic occlusion effects for view-dependent optical see-through displays. In *Proc. ISMAR 2002*, pages 186–195, Darmstadt, Germany, September 30 – October 1 2002. ACM and IEEE.

[23] T. Bray, J. Paoli, C. M. Sperberg-McQueen, et al. Extensible markup language (XML) 1.0. http://www.w3.org/TR/REC-xml/, October 6 2000.

[24] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*, volume 1. Wiley, Great Britain, 1996.

[25] A. Butz, C. Beshers, and StevenFeiner. Of vampire mirrors and privacy lamps: Privacy management in multi-user augmented environments. In *Proc. UIST98*, pages 171–172, San Francisco, CA, USA, November 2–4 1998. ACM.

[26] A. Butz, T. Höllerer, S. Feiner, B. MacIntyre, and C. Beshers. Enveloping users and computers in a collaborative 3d augmented reality. In *Proc. IWAR'99*, pages 35–44, San Francisco, CA, USA, October 20–21 1999. IEEE.

[27] R. Carey and G. Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley, 1997.

[28] G. Chung and P. Dewan. A mechanism for supporting client migration in a shared window system. In *Proc. ACM User Interface Software and Technology (UIST96)*, pages 11–20. ACM, 1996.

[29] G. Chung, K. Jeffay, and H. Abdel-Wahab. Accommodating latecomers in shared window systems. *IEEE Computer*, 26(1):72–74, 1993.

[30] J. Clark. XSL transformations (XSLT) version 1.0. http://www.w3.org/TR/xslt, 1999.

[31] S. Cox, A. Cuthbert, R. Lake, and R. Martell. Geography markup language (GML) 2.0. http://opengis.net/gml/01-029/GML2.html, February 20 2001.

[32] D. Curtis, D. Mizell, P. Gruenbaum, and A. Janin. Several devils in the details: Making an AR app work in the airplane factory. In R. Behringer and G. Klinker, editors, *Augmented Reality - Placing Artificial Objects in a Real Scene*, pages 47–60. A.K. Peters, 1998.

[33] A. Dearle, G. N. Kirby, R. Morrison, A. McCarthy, K. Mullen, Y. Yang, R. C. Connor, P. Welen, and A. Wilson. Architectural support for global smart spaces. In *Lecture Notes in Computer Science 2574*, pages 153–164. Springer, 2003.

[34] E. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[35] L. Dorffner and A. Zöchling. Das 3D modell von wien - erzeugung und Fortführung auf basis der wiener mehrzweckkarte. In *Proc. CORP 2003*, pages 161 – 166, Vienna, Austria, February 25 – March 1 2003.

[36] F. Echtler, F. Sturm, K. Kindermann, G. Klinker, J. Stilla, J. Trilk, and H. Najafi. The intelligent welding gun: Augmented reality for experimental vehicle construction. In S. Ong and A. Nee, editors, *Virtual and Augmented Reality Applications in Manufacturing, Chapter 17*, chapter 17. Springer Verlag, 2003.

[37] D. C. Fallside. XML schema part 0: Primer. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, January 5th 2004.

[38] S. Feiner, B. MacIntyre, and T. Höllerer. Wearing it out: First steps toward mobile augmented reality systems. In *Proc. ISMR'99*, 1999.

[39] S. Feiner, B. MacIntyre, T. Höllerer, and A. Webster. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban enviroment. In *Proc. ISWC'97*, pages 74–81, Cambridge, MA, USA, October 13–14 1997.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[41] N. Haala, J. Böhm, and M. Kada. Processing of 3d building models for location aware applications. In *International Archives on Photogrammetry and Remote Sensing IAPRS*, volume XXXIV, pages 157–162, Xian, August 2002. ISPRS Commision II Symposium.

[42] T. He and A. Kaufman. Virtual input devices for 3D systems. In *Proc. IEEE Visualization'93*, pages 142–148. IEEE, 1993.

[43] G. Hesina. *Distributed Collaborative Augmented Reality*. PhD thesis, Vienna University of Technology, May 2001.

[44] G. Hesina, D. Schmalstieg, and W. Purgathofer. Distributed open inventor : A practical approach to distributed 3D graphics. In *Proc. ACM VRST'99*, pages 74–81, London, UK, December 1999.

[45] T. Höllerer, S. Feiner, D. Hallaway, B. Bell, M. Lanzagorta, D. Brown, S. Julier, Y. Baillot, and L. Rosenblum. User interface management techniques for collaborative mobile augmented reality. *Computer & Graphics*, 25(25):799–810, 2001.

[46] T. Höllerer, S. Feiner, T. Terauchi, G. Rashid, and D. Hallaway. Exploring MARS: developing indoor and outdoor user interfaces to a mobile augmented reality system. *Computer & Graphics*, 23(6):779–785, 1999.

[47] T. Höllerer, D. Hallaway, N. Tinna, and S. Feiner. Steps toward accommodating variable position tracking accuracy in a mobile augmented reality system. In *Proc. AIMS'01*, pages 31–37, Seattle, WA, USA, Aug 4. 2001.

[48] T. Höllerer and J. Pavlik. Situated documentaries: Embedding multimedia presentations in the real world. In *Proc. ISWC'99*, pages 79–86, San Francisco, CA, USA, October 18–19 1999.

[49] IBM. Xeena XML editor. http://www.alphaworks.ibm.com/tech/xeena, visited February 20 2004.

[50] Icon Information Systems GmbH. XMLSpy. http://www.xmlspy.com, visited February 20 2004.

[51] ISO. Graphical kernel system (GKS). IS 7942, 1985.

[52] S. Julier, M. Lanzagorta, Y. Baillot, L. Rosenblum, S. Feiner, and T. Höllerer. Information filtering for mobile augmented reality. In *Proc. ISAR 2000*, pages 3–11, Munich, Germany, October 5–6 2000. IEEE and ACM.

[53] S. Julier, M. Livingston, D. Brown, Y. Baillot, and E. Swan. Adaptive user interfaces for augmented reality. In *Proc. STARS 2003*, pages 35–42, Tokyo, Japan, October 7 2003.

[54] H. Kato and M. Billinghurst. Marker tracking and HMD calibration for a video-based augmented reality conferencing system. In *Proc. IWAR 99*, San Francisco, USA, October 1999.

[55] H. Kato and M. Billinghurst. Marker tracking and HMD calibration for a video-based augmented reality conferenencing system. In *Proc. IWAR'99*, pages 85–94, San Francisco, CA, USA, October 21–22 1999. IEEE CS.

[56] H. Kaufmann and D. Schmalstieg. Mathematics and geometry education with collaborative augmented reality. *Computer & Graphics*, 27(3):339–345, June 2003.

[57] G. Klinker, O. Creighton, A. H. Dutoit, R. Kobylinski, C. Vilsmeier, and B. Brgge. Augmented maintenace of powerplants: A prototyping case study of a mobile AR system. In *Proc. ISAR 2001*, pages 124–133, New York, New York, USA, October 29–30 2001. IEEE.

[58] U. Kretschmer. Using mobile systems to transmit location based information. In *Proc. of the ISPRS Commission III Symposium*, Graz, Austria, 2002.

[59] U. Kretschmer, V. Coors, U. Spierling, D. Grasbon, K. Schneider, I. Rojas, and R. Malaka. Meeting the spririt of history. In *Proc. VAST 2001*, Glyfada, Athens, Greece, November 28–30 2001. Eurographics.

[60] M. A. Livingston, J. E. Swan II, J. L. Gabbard, T. H. Höllerer, D. Hix, S. J. Julier, Y. Baillot, and D. Brown. Resolving multiple occluded layers in augmented reality. In *Proc. ISMAR 2003*, pages 56–65, Tokyo, Japan, October 7–10 2003. IEEE.

[61] B. MacIntyre and S. Feiner. Language-level support for exploratory programming of distributed virtual environments. In *Proc ACM UIST'96*, pages 83–94, Seattle, WA, USA, Nov. 6–8 1996. ACM.

[62] B. MacIntyre and S. Feiner. A distributed 3D graphics library. In *Proc. ACM SIGGRAPH '98*, pages 361–370, Orlando, Florida, USA, July 19–24 1998.

[63] A. MacWilliams, C. Sandor, M. Wagner, M. Bauer, G. Klinker, and B. Bruegge. Herding sheep: Live system development for distributed augmented reality. In *Proc. ISMAR 2003*, pages 123–132, Tokyo, Japan, October 7–10 2003. IEEE.

[64] F. Manola and E. Miller. Rdf primer. http://www.w3c.org/TR/rdf-primer/, May 3 2003.

[65] D. L. McGuinness and F. van Harmelen. OWL web ontology language overview. http://www.w3.org/TR/2004/REC-owl-features-20040210/, February 11th 2004.

[66] W. Meier. eXist: An open source native XML database. In A. B. Chaudri, M. Jeckle, E. Rahm, and R. Unland, editors, *Web, Web-Services, and Database Systems. NODe 2002 Web- and Database-Related Workshops*, Springer LNCS Series, page 2593, Erfurt, Germany, October 2002. Springer.

[67] J. Newman, D. Ingram, and A. Hopper. Augmented reality in a wide area sentient environment. In *Proc. ISAR 2001*, pages 77–86, New York, New York, USA, October 29–30 2001. IEEE and ACM.

[68] J. Nichols, B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol. Generating remote control interfaces for complex appliances. In *Proc. UIST 2002*, pages 161–170, Paris, France, October 27–30 2002. ACM.

[69] U. of North Carolina at Chapel Hill. VRPN - virtual reality peripheral network. http://www.cs.unc.edu/Research/vrpn/, visited February 20 2004.

[70] C. Pavlakos, R. Frank, A. McPherson, G. Humphreys, M. Eldridge, A. Finkelstein, and A. Heirich. Commodity-based scalable visualization. SIGGRAPH 2001 course notes # 37, 2001.

[71] W. Piekarski, B. Gunther, and B. H. Thomas. Integrating virtual and augmented realities in an outdoor application. In *Proc. IWAR'99*, pages 45–54, San Francisco, CA, USA, October 21–22 1999. IEEE CS. early tinmith II, nice description of the system comprised of individual components connected via tcp. almost like dwarf.

[72] W. Piekarski, D. Hepworth, V. Demczuk, B. H. Thomas, and B. Gunther. A mobile augmented reality user interface for terrestrial navigation. In *Proc. of the 22nd Australasian Computer Science Conference*, pages 122–133, Auckland, NZ, January 18–21 1999.

[73] W. Piekarski and B. H. Thomas. Tinmith-ev5 - an architecture for supporting mobile augmented reality environments. In *Proc. ISAR 2001*, pages 177–178, New York, New York, USA, October 29–30 2001. IEEE and ACM. short description of new system design, not corba like but more data flow graph of between objects.

[74] W. Piekarski and B. H. Thomas. Tinmith-metro: New outdoor techniques for creating city models with an augmented reality wearable computer. In *Proc. ISWC 2001*, pages 31–38, Zurich, Switzerland, 8–9 October 2001. IEEE.

[75] W. Piekarski and B. H. Thomas. The tinmith system - demonstrating new techniques for mobile augmented reality modelling. In *Proc. AUIC2002*, Melbournce, Vic, Australia, January 2002. ACS.

[76] W. Piekarski and B. H. Thomas. An object-oriented software architecture for 3D mixed reality applications. In *Proc. ISMAR 2003*, pages 247–256, Tokyo, Japan, October 7–10 2003. IEEE.

[77] W. Piekarski, B. H. Thomas, D. Hepworth, B. Gunther, and V. Demczuk. An architecture for outdoor wearable computers to support augmented reality and multimedia applications. In *Proc. Third International Conference on Knowledge-Based Intelligent Information Engineering Systems*, Adelaide, Australia, August 1999. IEEE.

[78] T. Reicher, A. MacWilliams, and B. Bruegge. Towards a system of patterns for augmented reality systems. In *Proc. STARS 2003*, pages 6–11, Tokyo, Japan, October 7 2003.

[79] T. Reicher, A. MacWilliams, B. Bruegge, and G. Klinker. Results of a study on software architectures for augmented reality systems. In *Proc. ISMAR 2003*, Tokyo, Japan, October 7–10 2003. IEEE.

[80] G. Reitmayr and D. Schmalstieg. Mobile collaborative augmented reality. In *Proc. ISAR 2001*, pages 114–123, New York, New York, USA, October 29–30 2001. IEEE.

[81] G. Reitmayr and D. Schmalstieg. Location based applications for mobile augmented reality. In R. Biddle and B. Thomas, editors, *Proc. AUIC 2003*, volume 25 (3) of *Australian Computer Science Communications*, pages 65 – 73, Adelaide, Australia, February 4 – 7 2003. ACS.

[82] J. Rekimoto. Transvision: A hand-held augmented reality system for collaborative design. In *Proc. VSMM'96*, pages 18–20, Gifu, Japan, September 1996.

[83] J. Rekimoto. Pick-and-drop: A direct manipulation technique for multiple computer environments. In *Proc. UIST'97*, pages 31–39. ACM, 1997.

[84] J. Rekimoto. Matrix: A realtime object identification and registration method for augmented reality. In *Proc. APCHI'98*, 1998.

[85] J. Rekimoto and M. Saitoh. Augmented surfaces: A spatially continuous workspace for hybrid computing. In *Proc. CHI'99*. ACM, 1999.

[86] M. Roseman and S. Greenberg. Building real-time groupware with groupkit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, 1996.

[87] K. Rothermel and A. Leonhardi. Maintaining world models for context-aware applications. In W. R. Wells, editor, *Proceedings of the 8th International Conference on Advances in Communication and Control (COMCON-8)*, 2001.

[88] K. Satoh, K. Hara, M. Anabuki, H. Yamamoto, and H. Tamura. TOWNWEAR: An outdoor wearable MR system with high-precision registration. In *Proc. IEEE Virtual Reality 2001*, pages 210–211, Yokohama, Japan, March 13–17 2001.

[89] D. Schmalstieg and G. Eibner. Hybrid user interfaces using seamless tiled displays. submitted for publication, 2004.

[90] D. Schmalstieg, A. Fuhrmann, and G. Hesina. Bridging multiple user interface dimensions with augmented reality. In *Proc. ISAR 2000*, pages 20–29, Munich, Germany, October 5–6 2000. IEEE and ACM.

[91] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavari, L. M. Encarnao, M. Gervautz, and W. Purgathofer. The Studierstube augmented reality project. *PRESENCE - Teleoperators and Virtual Environments*, 11(1), 2002.

[92] D. Schmalstieg, A. Fuhrmann, Z. Szalavari, and M. Gervautz. Studierstube – an environment for collaboration in augmented reality. In *Proc. of Collaborative Virtual Environments Workshop '96*, Nottingham, UK, September 19–20 1996.

[93] D. Schmalstieg and G. Hesina. Distributed applications for collaborative augmented reality. In *Proc. IEEE VR 2002*, pages 59–66, Orlando, Florida, USA, March 24–28 2002. IEEE.

[94] D. Schmalstieg, H. Kaufmann, G. Reitmayr, and F. Ledermann. Geometry education in the augmented classroom. In *Proc. ISMAR 2002*, Darmstadt, Germany, September 30 – October 1 2002. IEEE.

[95] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, volume 2. Wiley, Great Britain, 2000.

[96] R. Schönfelder, G. Wolf, M. Reeßing, R. Krüger, and B. Brüderlin. A pragmatic approach to a VR/AR component integration framework for rapid system setup. In *Proceedings of the 1. Paderborner Workshop "Augmented und Virtual Reality in der Produktentstehung"*, pages 67–79, Paderborn, Germany, June 11–12 2002. Heinz Nixdorf Institut.

[97] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, July 1993.

[98] S. Shekhar, M. Coyle, B. Goyal, D.-R. Liu, and S. Sarkar. Data models in geographic information systems. *Communications of the ACM*, 40(4):103–111, 1997.

[99] Software AG. Tamino XML server. http://www.softwareag.com/tamino/, January 5th 2004.

[100] J. C. Spohrer. Information in places. *IBM Systems Journal*, 38(4):602–628, 1999.

[101] T. Starner, S. Mann, B. Rhodes, J. Levine, J. Healey, D. Kirsch, R. Picard, and A. Pentland. Augmented reality through wearable computing. *Presence*, Special Issue on Augmented Reality(4):386–398, 1997.

[102] P. Strauss and R. Carey. An object oriented 3D graphics toolkit. In *Proc, ACM SIGGRAPH'92*. ACM, 1992.

[103] Z. Szalavári and M. Gervautz. The personal interaction panel — A two-handed interface for augmented reality. *Computer Graphics Forum*, 6(13):335–346, 1997.

[104] R. M. Taylor II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. VRPN: A device-independent, network-transparent VR peripheral system. In *Proc. VRST 2001*, pages 55–61, Banff, Alberta, Canada, November 15–17 2001. ACM.

[105] The Apache Software Foundation. Apache Xindice. http://xml.apache.org/xindice/, January 5th 2004.

[106] The Apache Software Foundation. Xerces XML parser. http://xml.apache.org/xerces-c/index.html, visited February 20 2004.

[107] B. Thomas, B. Close, J. Donoghue, J. Squires, P. D. Bondi, M. Morris, and W. Piekarski. Arquake: An outdoor/indoor augmented reality first person application. In *Proc. ISWC2000*, pages 139–146, Atlanta, Georgia, USA, October 16–17 2000. IEEE.

[108] B. H. Thomas, V. Demczuk, W. Piekarski, D. H. epworth, and B. Gunther. A wearable computer system with augmented reality to support terrestrial navigation. In *Proc. ISWC'98*, pages 168–171, Pittsburgh, USA, October 19–20 1998. IEEE and ACM.

[109] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/, January 8th 2004.

[110] H. Tramberend. Avocado: A distributed virtual reality framework. In *Proc. IEEE VR 1999*, pages 14 – 21, Houston, Texas, USA, March 13 – 17 1999. IEEE.

[111] J. Tsao and C. Lumsden. CRYSTAL: Building multicontext virtual environments. *PRESENCE - Teleoperators and Virtual Environments*, 6(1):57–72, 1997.

[112] B. Ullmer, H. Ishii, and D. Glas. mediablocks: Physical containers, transports, and controls for online media. In *Proc. SIGGRAPH98*, pages 379–386, 1998.

[113] VRCO. Trackd. http://www.vrco.com/products/trackd/trackd.html, visited March 2nd 2004.

[114] S. L. Weibel and T. Koch. The dublin core metadata initiative: Mission, current activities, and future directions. *D-Lib Magazine*, 6(12), December 2000.

[115] M. Weiser. The computer of the twenty-first century. *Scientific American*, 1991.

[116] J. Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*. Addison-Wesley, 2nd edition, November 1993.

[117] J. Wernecke. *The Inventor Toolmaker: Extending Open Inventor*. Addison-Wesley, 2nd edition, April 1994.

[118] L. Wood, A. L. Hors, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, G. Nicol, J. Robie, R. Sutor, and C. Wilson. Document object model (DOM) level 1 specification (second edition). http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/, January 19th 2004.

Curriculum Vitae

Gerhard Reitmayr

Bleichergasse 13/9
A-1090 Vienna
Austria
reitmayr@ims.tuwien.ac.at

1974	Born on the 25^{th} of November in Vienna, Austria
1981 – 1985	Primary School (Volksschule) at the Schulzentrum Maria Enzersdorf (SMZ), Maria Enzersdorf, Austria
1985 – 1989	Secondary School at the Bundesgymnasium Bachgasse, Mödling, Austria
1989 – 1993	Secondary School at the Bundesrealgymnasium Keimgasse, Mödling, Austria
June 1993	Graduation (Matura) from the Bundesrealgymnasium Keimgasse
1993 – 1999	Studies in engineering mathematics at the Vienna University of Technology
January 2000	Graduation "Diplom-Ingenieur der Technischen Mathematik" from the Vienna University of Technology Thesis "Changing Stability of the Closed loop System by Modifying the Cost Function"
2000 – 2004	Doctoral program in computer science at the Vienna University of Technology
March 2004	Dissertation "On Software Design for Augmented Reality"