TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DISSERTATION

Ubiquitous Animated Agents for Augmented Reality

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

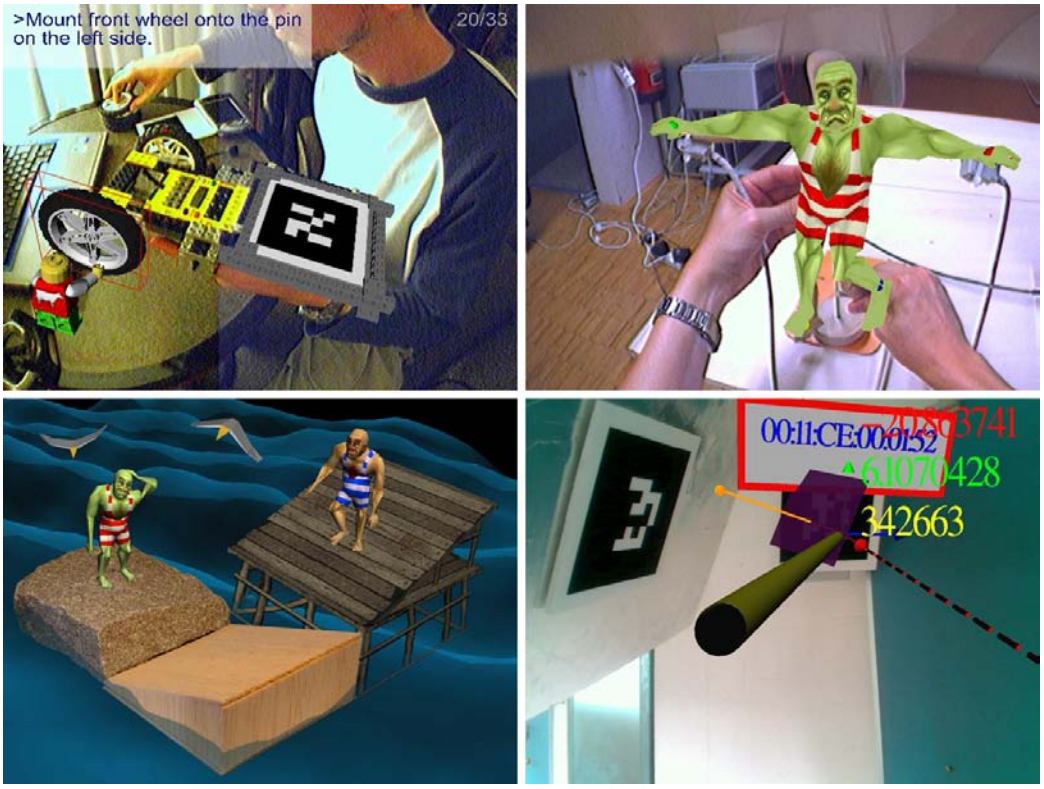
unter der Leitung von
Univ.-Prof. Dr. Dieter Schmalstieg
Institut für Maschinelles Sehen und Darstellen (ICG)
Technische Universität Graz

eingereicht
an der Technischen Universität Wien
Fakultät für Informatik

von
M.Sc. István Barakonyi
Rossauer Lände 41/18
1090 Wien
Matr.-Nr. 0326849

Wien, im Oktober 2006

Ubiquitous Animated Agents for Augmented Reality



István Barakonyi – Dissertation

Reviewers:

Dieter Schmalstieg

Andreas Butz

Abstract

A growing spectrum of Ubiquitous Computing (UbiComp) applications suggests that interaction with computers should be as natural and effortless as using pen, paper and language when writing. Unlike current computer environments that require a considerable amount of adaptation from users for smooth interaction, future digital interfaces are envisioned to act unobtrusively and intelligently in our environment. This dissertation describes a novel user interface approach combining Augmented Reality (AR), UbiComp and Autonomous Animated Agents into a single coherent human-computer interface paradigm that makes steps toward this vision.

A significant challenge for the UbiComp community is to create efficient, natural and user-friendly interfaces since there are no standards and best practices to follow yet. Typical UbiComp scenarios include numerous mobile users roaming a large area while interacting with various stationary and mobile devices. Since the location and behavior of users and devices change rather frequently, an enormous amount of events describing changes gets generated in the environment. Processing such large data sets can be greatly overwhelming for humans, therefore an interface to a UbiComp system is expected to possess certain autonomy in order to filter and interpret relevant events and react proactively without constant user guidance and explicit instructions. By relieving users from dealing with low-level details and allowing computers to make decisions by themselves, these interfaces appear to be "smart".

This thesis presents software solutions that employ reactive, autonomous and social digital assistants in UbiComp environments. These systems rely on software agent technology tailored to the needs of AR applications, where system behavior is visualized by virtual animated characters appearing on top of the real world. We discuss how autonomous animated agents can be employed to mediate communication between humans and computers in AR environments while exploiting real world attributes as input and output communication channels. The agents maintain a model of the real world by analyzing data coming from the sensors that measure physical properties such as pose, velocity, sound or light, and autonomously react to changes in the environment in accordance with the users' perception. Autonomous, emergent behavior is a novel feature in UbiComp, while awareness of real world attributes is yet unexploited by autonomous agents.

This dissertation explores the requirements for context-aware animated agents concerning visualization, appearance, and behavior as well as associated technologies, application areas, and implementation details. Several application scenarios illustrate design and implementation concepts.

Kurzfassung

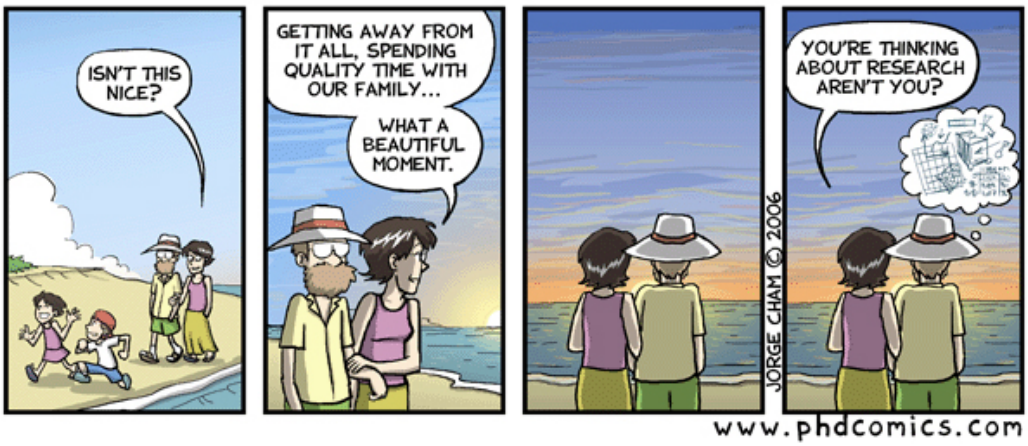
Ubiquitous Computing (UbiComp, ubiquitäre Computertechnik) zielt darauf ab, daß die Interaktion mit Computers so natürlich und mühelos sein soll wie das Schreiben mit einem Stift auf Papier. Im Gegensatz zu derzeitigen Computersystemen, die dem Benutzer Anpassung abverlangen, agieren zukünftige digitale Benutzerschnittstellen unauffällig und intelligent im Hintergrund. Diese Dissertation beschreibt eine neue Art von Benutzerschnittstellen, die die Vorteile von Augmented Reality (AR, erweiterte Realität), UbiComp und autonomen animierten Agenten vereinigt, um eine verbesserte Mensch-Maschine Interaktion realisieren zu können.

Eine signifikante Herausforderung dabei ist die Erschaffung effizienter, natürlicher und benutzerfreundlicher Schnittstellen für UbiComp-Systeme, für die es derzeit keine etablierten Gestaltungsrichtlinien gibt. In typischen UbiComp-Szenarien werden mehrere mobile Benutzer bedient, die sich in einem weitläufigen Bereich frei bewegen und dabei mit verschiedenen stationären und mobilen Geräte interagieren. Da der Standort und das Verhalten der Benutzern sich laufend ändert, wird eine enorme Menge von Statusinformation über den aktuellen Systemstand generiert, die nicht mehr mit manuellen Methoden verwertbar ist. Deshalb wird von zukünftigen UbiComp-Systemen erwartet, daß sie autonom und ohne aktive menschliche Hilfe arbeiten.

Diese Dissertation präsentiert eine Softwarelösung für die Implementierung von autonomen und sozialen computergenerierten Assistenten für UbiComp-Umgebungen. Das vorgestellte System benutzt eine Kombination von Methoden aus den Bereichen Software-Agenten und Augmented Reality, um virtuelle animierte Agenten darzustellen, die mit dem Benutzer sowohl in der virtuellen als auch in der realen Welt interagieren. Die Agenten benutzen ein internes Weltmodell, welches auf der Analyse von Sensordaten für Position, Geschwindigkeit, Audio, Licht und andere Eigenschaften der realen Umgebung beruht, und reagieren selbständig auf die Änderungen. Die Neuigkeit des Ansatzes liegt im autonomen, selbständigen Verhalten der Agenten, welche die Attributen der realen Welt bislang noch nicht vollständig ausgenutzt haben.

Acknowledgements

The true winner of this dissertation is my wife, Rita. She was the one who shared my joy at the "ups" and gave me energy when I felt deflated at the "downs". Without her I would have not succeeded and therefore I dedicate this thesis to her. The image below is for everybody whose partner is working on a PhD.



Dieter Schmalstieg, my PhD supervisor provided me with great ideas to improve my research and a stable financial background to let me worry only about scientific results. His continuous attention and guidance gave me steady motivation for my work. I also want to thank him for accepting my stubbornness in my choice of research topics. While working at the Vienna University of Technology, I had a chance and great pleasure to work with Christian Breiteneder, whose professional attitude to scientific work and social competence have made a lasting impact on me. I also would like to thank Mitsuru Ishizuka at the Tokyo University and Helmut Prendinger at the National Institute of Technology, Japan for giving me the opportunity to cooperate with them and for opening up the exciting domain of autonomous agent research for me.

One of the most important values gained from my PhD studies was that I was able to share the misery of tight deadlines and the delight of success with numerous colleagues in the *Studierstube* team at the Graz and Vienna University of Technology. A special honorary mention goes to Joseph Newman (*we has it, precious!*), thanks, Joe, for being a perfect office and roommate for such a long time! I am grateful to numerous people for their continuous support (in alphabetical order): Alexander Bornik, Tamer Fahmy, Markus Grabner, Denis Kalkofen, Michael Kalkusch, Hannes Kaufmann,

Karin Kosina, Florian Ledermann, Erick Mendez, Judith Mühl, Thomas Pintaric, Thomas Psik, Bernhard Reitinger, Gerhard Reitmayr, Markus Sareika, Gerhard Schall, Eduardo Veas, Daniel Wagner, and Albert Walzer. I will miss the great group atmosphere! Circled Cube, Ulrich Krispel, Christoph Schinko, and Markus Weilguny contributed precious work to some demo applications of mine.

The long way that led me to completing my PhD would have been impossible without the continuous and unconditional emotional and financial support of my parents. I thank them for always being there for me.

Contents

List of Figures

Chapter 1

Introduction

1.1 The World as User Interface

More than half a century ago the first large-scale computers were created to improve human work performance by accelerating and automating tasks previously carried out manually. The interface between human users and these room-sized computing devices was designed to be manipulated by specifically trained operators.

Several decades later the computational power of heretofore room-sized computers has been largely surpassed by that of palm-sized computers, and the previously many-to-one user-computer ratio has gradually become one-to-many with the boom of the embedded computer industry [96]. High-resolution displays and a wide range of novel input devices have gradually replaced formerly physical man-machine interfaces relying on buttons, sliders and levers by virtual interfaces composed of 2D and 3D graphical elements.

Although the computational power offered by commercial off-the-shelf devices and the steep decrease in the price of digital devices have opened up new horizons in user interface technology, a new set of problems has emerged:

- Despite significant advances in display technology, a steep increase in processing power, and a dramatic decrease in the size of computing devices, the seam between the physical and digital world has not been mitigated to a degree desirable for a more widespread use of computers. Most human-computer interfaces have been so far computer-centered instead of human-centered, requiring users to map their intention to explicit commands easily understandable by computers. These virtual interfaces demand a significant amount of user training, preventing computers from becoming simple enough to penetrate our everyday life beyond the domain of office automation tasks.

- Users are getting overwhelmed by the growing number and bewildering complexity of computing devices they need to operate. Computers demand more and more user attention, which renders former interfaces solely based on direct manipulation techniques [88] unsustainable.

This thesis describes steps made towards solving the aforementioned problems by applying design principles and interaction techniques from the following three research areas:

- Augmented Reality
- Ubiquitous Computing
- Software Agents

The framework presented in the thesis demonstrates that an effective combination of advantageous features of the above three research domains yields a closer integration of computers into the physical user environment, which makes the seam between human and computer less apparent. This chapter makes a brief overview of the individual domains and summarizes contributions made by the thesis.

1.1.1 Augmented Reality

Paul Milgram introduces the Virtuality Continuum [61] between the real and the virtual world. Augmented Reality (AR) [6] and Augmented Virtuality (AV) user interfaces lie in the middle of this continuum as they rely on a mixture of real and virtual interface elements and thus are jointly referred to as Mixed Reality. This thesis focuses on AR applications, where virtual objects are aligned with and superimposed onto the real world. AR applications enable the preservation of the real user environment that provides a reference frame for user actions, thus making human-computer interaction more natural. Figure 1.1 illustrates the position of AR-based interfaces inside the virtuality continuum by showing a real, an augmented, and a virtual representation of a LEGO Mindstorms® robot later appearing in application scenarios.

In Virtual Reality (VR) environments synthetic worlds inhabited by virtual objects act as an interface between applications and users. During the application development process the most effort is usually put into creating a faithful model of physical objects so that users have the "illusion" of seeing the "real thing". This model not only includes an accurate representation

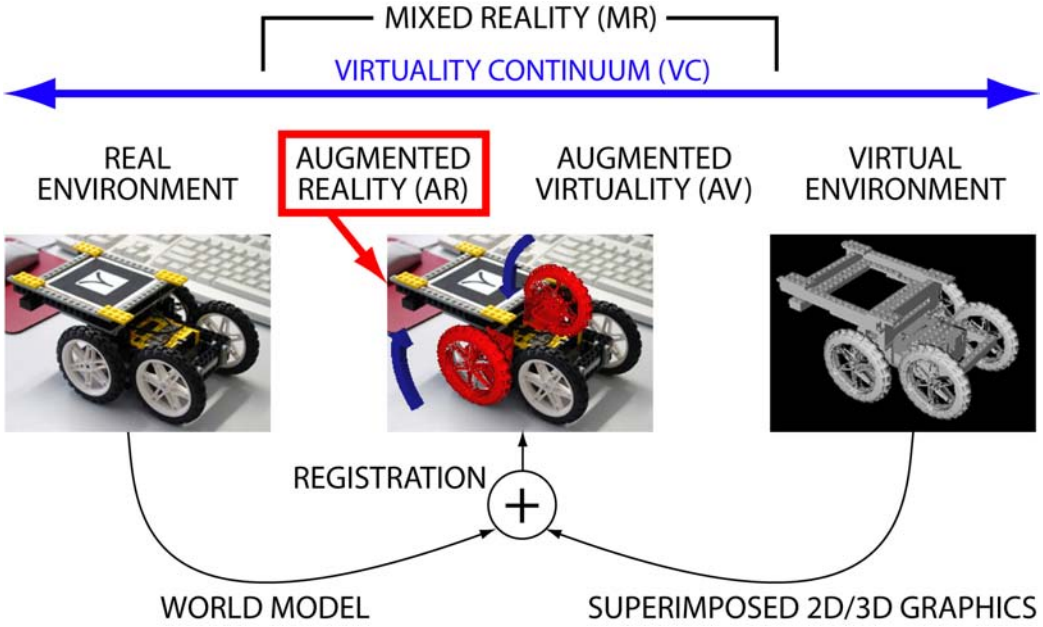


Figure 1.1: Milgram's virtuality continuum illustrated by a real, an augmented, and a virtual representation of a LEGO Mindstorms® robot

of the objects' visual appearance but a careful depiction of the respective objects' behavior as well.

A significant advantage of AR systems over their VR counterparts is the exploitation of the physical world. Application developers do not need to consider and model every single detail since these details are already physically present with infinite resolution and accuracy. The sensitive task of modeling appearance and behavior can be reduced to superimposing only meaningful, application-specific virtual information over real world objects. This reduction enables a better focus on efficient information visualization. Moreover, AR environments allow users' facial and body gestures and physical objects in the surrounding environment to remain visible, therefore users feel more comfortable while working with virtual interface objects.

Despite the obvious advantages AR environments offer, only few applications take full advantage of real world features. Besides traditional desktop-based virtual input data from a mouse, keyboard or speech recognition module, classic AR applications track and exploit real world attributes as well such as position, orientation, sound, light, or temperature as input modalities. These attributes are measured and processed by various sensors. Based on the sensor data an internal *world model* is built up serving as a virtual representation of the real environment, which is then rendered as computer

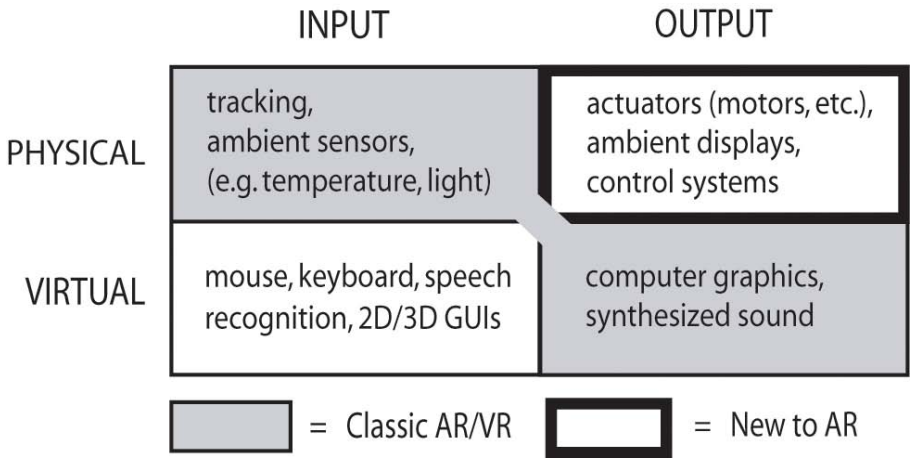


Figure 1.2: Input and output modalities in user interfaces

graphics images and synthesized sound on top of its real world counterpart. Typical examples are tangible AR applications [76], where the user manipulates virtual objects by physically manipulating real, tangible props.

As shown in Figure 1.2, AR systems have been relying on various virtual output modalities typical in VR applications. However, they have been so far lacking output modalities within the real world, although actuators and control systems have been used for a long time as output communication channels in various engineering fields such as robotics or industry automation processes. For instance the QRIO robot from Sony [90] transforms user voice commands and gestures into anthropomorphic movement by manipulating a complex network of motors. Other examples are ambient displays [108], which are physical devices that transmit information on the periphery of human perception using light, sound or movement. However, neither QRIO nor ambient displays combine physical and virtual output as AR systems do.

This thesis presents several AR applications taking full advantage of the physical world by using it both as an input and output modality simultaneously, which is a hitherto unexploited concept in AR.

1.1.2 Ubiquitous Computing

Ubiquitous Computing (Ubicomp) systems [104] pursue goals similar to those of AR environments: decreasing the seam between real and virtual, human and computer. Ubicomp systems aim to achieve these goals by distributing processing power previously associated with computer workstations into the real environment. By embedding digital devices into mundane objects,

heretofore passive everyday items can be turned into "smart" entities that the American writer Adam Greenfield [33] calls "Everyware". Computers and sensors designed to act as *Everyware* disappear in the background environment, making digital resources as accessible and ubiquitous as power sockets in the wall.

Beside their invisible and ubiquitous nature, Ubicomp systems are also prepared to opportunistically exploit dynamic resources offered by various mobile and stationary computing devices and a heterogeneous network of sensors. In contrast, most of today's AR systems operate as passive information browsers relying on a predefined, static set of hardware and software components. The only dynamic element of such systems is usually the world model (see Section 1.1.1), updated manually to store information about the physical environment and thus offer an interface between the real and virtual world. These world models are finite and deterministic, requiring application developers to exhaustively enumerate of its possible elements and states.

In contrast, Ubicomp systems maintain a flexible and indeterministic world model by enabling the seamless and automatic integration of diverse hardware and software components, thus making these disparate resources easily accessible as standard services. One important goal of this thesis is therefore to complement AR systems with Ubicomp techniques to exploit distributed and dynamic resources such as software services, sensors, and output devices in an effective and invisible way.

1.1.3 Software Agents

Help desk statistics and recent user studies [94] indicate that a significant number of people encounter serious problems during the installation and operation of technical devices in their home such as central heating controls, network routers or hi-fi systems. The bewildering complexity of user interfaces often intimidate customers from buying or using technical appliances. The boom experienced in the number of digital devices and available functions is likely to be soon reflected in the complexity of AR systems as well. Direct manipulation interfaces will become so saturated with controllable parameters that users will have no other choice than delegating interface manipulation tasks to autonomous software components, elevating users to a supervisory role.

These autonomous software components or *autonomous agents* are proactive software entities facilitating informed decision-making. Agents are able to proactively act on the user's behalf and carry out delegated tasks being uninteresting or time consuming to the user while making decisions without constant guidance. These decision-making capabilities are not necessarily

based on a deep understanding of the problem semantics, yet allow the agent to deliver a useful function in a complex, heterogeneous environment such as AR and Ubicomp applications. It is important to note that UbiAgents are not claimed to be *intelligent* in the classic artificial intelligence sense; this work is more influenced by what is generally described as Ambient Intelligence [67].

According to Reeves and Nass [77], users assign human characteristics to computers suggesting that the human brain has still not assimilated the huge array of 20^{th} and 21^{st} century technologies. By bolstering this illusion, AR user interfaces of this thesis rely on interface agents [46] possessing visual, often anthropomorphic representations that operate as assistants for direct manipulation interfaces. The framework presented in this work combines the advantages of interface agents and autonomous agents into an *autonomous interface agent* that acts parallel with the user without constant attention and explicit commands, and carries out tasks while monitoring the user's environment and actions.

In AR environments autonomous interface agents may be embodied by virtual and physical objects. Virtual objects are typically animated characters but are not necessarily anthropomorphic, as in some AR applications a fully fledged virtual human can be more distracting than a simple animated arrow communicating the same amount of information. Therefore, various character forms are examined. A novel and exciting new aspect of autonomous interface agents (or AR agents in short) is that physical objects such as a printer, a digital piano or an interactive robot can be turned into intelligent, responsive entities that collaborate with virtual characters. Several application scenarios will demonstrate such a collaboration.

Although interface agents and "smart" autonomous software components generate much controversy in the HCI community [89], AR and Ubicomp systems can benefit from software agent technology. Agents are designed to be independent from applications they are embedded into, enabling their employment in diverse application environments without reprogramming their core functionalities. Moreover, the incorporation of flexible high-level context elements such as application goal and user interest may more efficiently cope with the indeterministic nature of augmented physical environments than explicit direct manipulation techniques. Agents constantly monitor their environment and reevaluate the effectiveness of their currently executed actions, thus autonomous interface agents add flexibility and adaptivity to AR user interfaces.

Kotz and Gray [43] use the term *mobile agent* for autonomous software components that have the ability to transfer and reproduce themselves on various networked computing devices. By equipping interface agents with

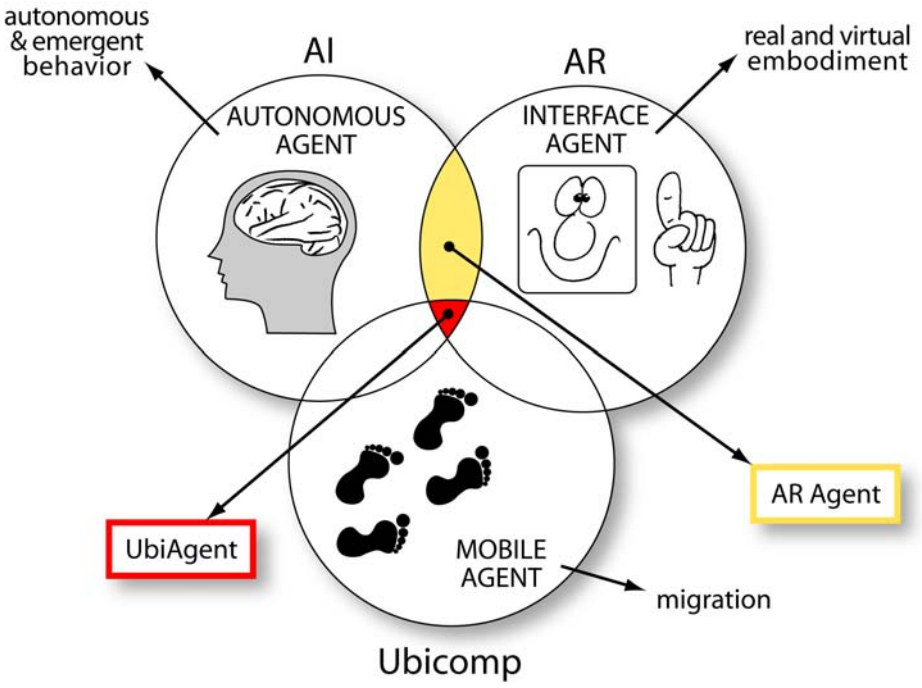


Figure 1.3: Research domains Augmented Reality Agents (AR Agents) and Ubiquitous Animated Agents (UbiAgents) are built on

mobile characteristics, they are no longer bound to a single, statically configured application and output device but may opportunistically migrate to and take advantage of other platforms more favorable for the agent's current needs. The extension of the aforementioned AR agents with mobile features yields *ubiquitous AR agents* (or UbiAgents for short) that enable exploiting concepts heretofore unexplored in AR such as adaptive user interfaces and massive user interface customization.

1.2 Contribution

The contribution of this dissertation is the design and implementation of a software framework that makes steps towards solving user interface problems described in the problem statement of Section 1.1. As Figure 1.3 illustrates, this thesis combines the aforementioned research domains of AR, Ubicomp and software agents yielding a novel, coherent human-computer interface paradigm called *UbiAgent*. UbiAgents offer the following fresh perspectives and novel features for AR environments:

- UbiAgent is the first general framework for autonomous animated agents that has been developed specifically for AR applications. The framework has been built on a powerful AR framework called *Studierstube* [85], which allows experimentation with a wide range of applications, tracking technology, platforms and displays.

- The framework examines agent-specific aspects of AR. New modalities enabled by the physical environment are exploited in animated agent behavior, and autonomous and emergent behavior is added to AR applications at low cost. By marking relevant input and output application attributes agents can easily monitor and thus react to user interaction and changes in the application state.

- The framework contains useful wrapper classes that can turn physical objects to intelligent, responsive entities and use them as input and output devices in AR environments. A unified command interface allows physical and virtual objects to be scripted in the same way.

- UbiAgents are dynamically configurable; their attributes and command fields can be intuitively controlled through various stationary and mobile devices.

- UbiAgents are able to monitor their environment through a network of diverse physical and virtual sensors and adapt to the current context derived from sensor measurements. Thus AR interfaces can dynamically adapt to user preferences and application history by accumulating a profile stored in a database. This database represents agent memory, adding persistency to agents.

- UbiAgents are migratable. They can opportunistically exploit dynamic resources such as multiple computing devices and displays, a diverse set of sensors, and various output devices by identifying the environment that is most optimal to achieve their current goals and leverage their capabilities. In case the ideal environment is different from the current one, UbiAgents are able to migrate to the new, desirable environment to more effectively exploit current system resources. Migration enables new features for AR systems such as load balancing and survival behavior as well.

This thesis is organized as follows. Chapter 2 provides an overview of related work on the combination of AR and software agent technologies as well as adaptive user interface technologies in AR. Chapter 3 and 4 discuss the evolutionary design of UbiAgents including implications of AR scenarios

on animated agents concerning their appearance, behavior, application areas and associated technologies. Framework design principles are illustrated by several example applications in Chapter 5. Chapter 6 presents frameworks forming the technological foundation for the agents' software implementation, followed by implementation details. Chapter 7 presents authoring concepts. Chapter 8 concludes the thesis with a summary, a discussion on the usefulness of UbiAgents in AR environments, and future work.

The work presented here contains material previously published and presented at several conferences:

- I. Barakonyi and D. Schmalstieg. Ubiquitous Animated Agents for Augmented Reality. To appear in *Proc. of the IEEE and ACM International Symposium on Mixed and Augmented Reality 2006 (ISMAR'06)*, Santa Barbara, CA, USA, 2006.

- I. Barakonyi and D. Schmalstieg. Augmented Reality in the Character Animation Pipeline. Sketch at *SIGGRAPH 2006*, Boston, MA, USA, 2006.

- J. Newman, G. Schall, I. Barakonyi, A. Schürzinger, and D. Schmalstieg. Sentient Environments for Augmented Reality. In *Advances in Pervasive Computing, Adjunct Proceedings of the International Conference on Pervasive Computing (Pervasive 2006)*, Dublin, Ireland, 2006.

- F. Ledermann, I. Barakonyi, and D. Schmalstieg. Abstraction and Implementation Strategies for Augmented Reality Authoring. Book chapter in *Emerging Technologies of Augmented Reality: Interfaces and Design* (M. Haller, B. Thomas, M. Billinghurst eds.), Idea Group Publishing, to be published in 2006.

- I. Barakonyi and D. Schmalstieg. Augmented Reality Agents in the Development Pipeline of Computer Entertainment. In *Proc. of the 4^{th} International Conference on Entertainment Computer (ICEC'05)*, Sanda, Japan, 2005.

- I. Barakonyi, M. Weilguny, T. Psik, and D. Schmalstieg. MonkeyBridge: Autonomous Agents in Augmented Reality Games. In *Proc. of the ACM SIGCHI International Conference on Advances in Computer Entertainment Technology (ACE'05)*, Valencia, Spain, 2005.

- I. Barakonyi and D. Schmalstieg. Exploiting the Physical World as User Interface in Augmented Reality Applications. In *Proc. of the IEEE Virtual Reality 2005 Workshop on New Directions in 3D User Interfaces*, Bonn, Germany, 2005.

- I. Barakonyi, T. Psik, and D. Schmalstieg. Agents That Talk and Hit Back: Animated Agents in Augmented Reality. In *Proc. of the IEEE and ACM International Symposium on Mixed and Augmented Reality 2004 (ISMAR'04)*, pp. 141-150, Arlington, VA, USA, 2004.

- I. Barakonyi and D. Schmalstieg. AR Puppet: Animated Agents in Augmented Reality. In *Proc. of First Central European International Multimedia and Virtual Reality Conference*, pp. 35-42, Veszprém, Hungary, 2004.

Chapter 2

Related Work

This thesis deals with an interdisciplinary topic combining multiple research domains. The result of this combination is a framework employing embodied autonomous agents to implement adaptive user interfaces for augmented reality applications. This chapter covers related work in research areas contributing to the design principles of the framework. These areas can be divided into two main categories:

- *Adaptive user interfaces:* This area provides an overview of techniques how AR systems in related research projects facilitate adaptive behavior by tailoring their user interface to dynamically changing context information. These techniques include information filtering based on current application context, adaptive user interface components, and user interface migration.

- *Software agents:* Software agents are "smart" software components based on principles of autonomic [37] and proactive computing [96]. The presented framework employs mobile and autonomous agents embodied by real and virtual objects as an interface and interaction metaphor, where agent bodies are able to opportunistically migrate between multiple AR applications and computing platforms to best match the needs of the current application context. The overview examines how related projects have exploited animated agents – in particular anthropomorphic animated characters – in AR applications and discusses how mobile agent technology increases the mobility of interface agents.

2.1 Adaptive User Interfaces

2.1.1 Information Filtering

The simplest form of automatic adaptation of AR content to current application context is information filtering based on a spatial or semantic world model. The primary objective of information filtering is to avoid cluttering displays by an unnecessarily large number of visual elements, and thus overwhelming and confusing users with unimportant information.

Classic computer graphics applications [3] apply spatial filtering based on simple context elements such as distance and visibility. These filters reduce computational and cognitive workload by culling away irrelevant visual objects or reducing their level of detail. A typical AR example for this approach is the work of Bane and Höllerer [9] (see Figure 2.1a), who developed interactive tools to enhance building visualization in a mobile AR setup.

While spatial filters are efficient tools for enhancing spatial tasks such as indoor/outdoor navigation, AR applications often need to consider a larger and more diverse set of context elements. The KARMA system [27] (see Figure 2.1b) employs a rule-based illustration generation system to exploit user viewpoint, object pose and communicative goals for efficient information visualization in an AR-based machine maintenance scenario. Although KARMA proved to be suitable for replacing manuals for a small-scale repair task, rule-based generation of visual augmentations for larger and more complex systems suffers from scalability problems. Julier et al. developed a hybrid approach [38] for their mobile AR system: a spatial model is used to prefilter visual elements to reduce input information for a rule-based filter component.

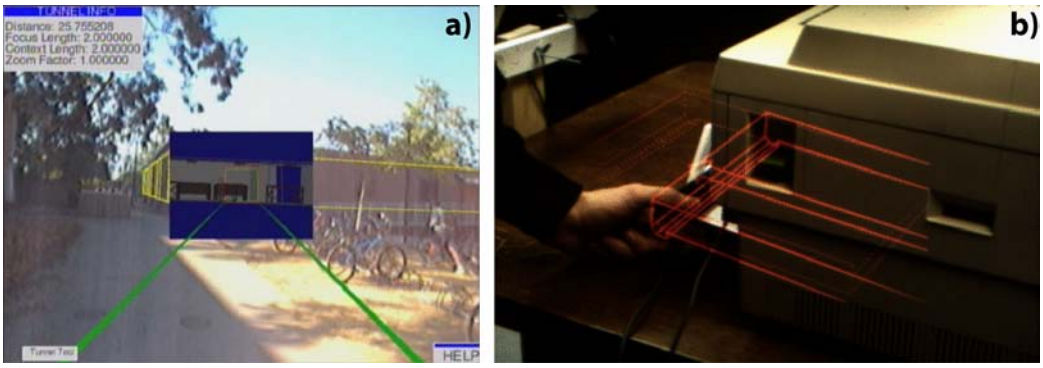


Figure 2.1: a) Virtual x-ray vision in an outdoor AR environment [9], b) Repairing a laser printer with the KARMA system [27]

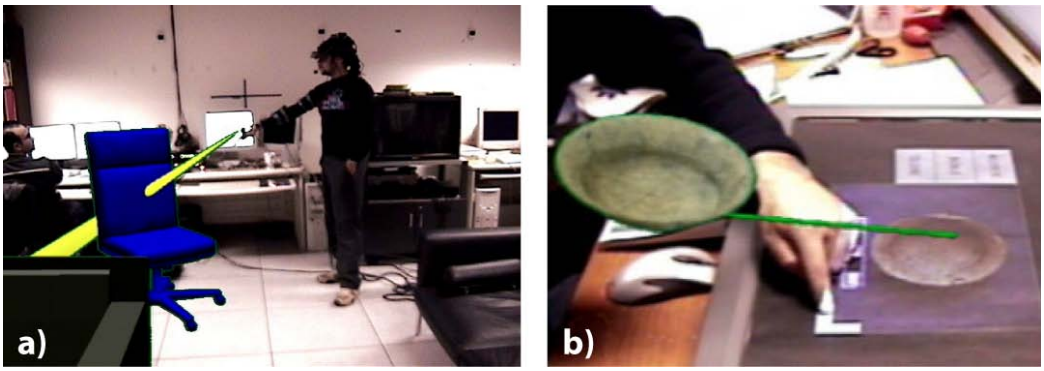


Figure 2.2: a) Disambiguating multimodal interaction in an immersive AR environment [39], b) Cross-dimensional interaction in an archaeology scenario [14]

2.1.2 Adaptive User Interface Components

Besides the rendering engine filtering out data, other system components may also actively adapt their behavior to dynamically changing context information. A typical example is the UbiTrack project [64], which eliminates dependencies on specific sensors by dynamically incorporating data arriving from a heterogeneous network of distributed sensors. Kaiser et al. [39] developed an immersive AR environment (see Figure 2.2a) that retrieves context information on demand to disambiguate multimodal interaction by estimating user intention from deictic speech utterances and 3D gestures.

The hybrid user interface developed by Benko et al. [14] (see Figure 2.2b) uses 2D and 3D gestures to switch between interaction contexts to determine the target display and privacy factors in a multi-display environment. The properties of the current target display impose dimensional constraints onto the available interaction methods in the user interface; for instance, a touch-screen panel permits only 2D gestures, while the immersive work environment of a head-mounted display demands 3D gestures.

2.1.3 User Interface Migration

A cross-dimensional interface is a notable example for a special type of adaptive component: the user interface migration controller. This component is responsible for the migration of user interface elements between computing platforms with different characteristics. Full or partial user interface migration between devices and displays allows the selection of the most suitable environment for presenting application information [59]. For instance, a PDA or mobile phone offers only limited rendering and interaction capabilities, but

Figure 2.3: Application migration in the Studierstube framework [86]

enables users to roam a large area without interruption in their workflow. Monitors and projection screens are stationary but support a shared view and richer presentation tools. The possibility to move application elements or entire applications between multiple devices eliminates the need of making a compromise as applications can dynamically select the hardware and software environment that best matches their current needs. The capability of making smooth transitions between platforms mitigates the seam between platform boundaries and thus increases productivity.

Further migration examples in AR include the playful SHEEP application scenario [55] that employs 3D gestures and tangible interaction to initiate the transfer of a virtual sheep model between multiple stationary and mobile displays. Schmalstieg et al. [86] created a shared collaborative workspace based on a distributed shared scene graph that enables the migration of applications between hosts (see Figure 2.3). Their work addresses ad-hoc collaboration and load balancing for AR environments. Rekimoto's pick-and-drop technique [80] extends the drag-and-drop direct manipulation technique already familiar from the classic WIMP world, and creates an interaction metaphor to move digital information physically between computing devices using a real stylus. Newman et al. [65] created the Speakeasy recombinant computing framework that provides an infrastructure through which device and service user interfaces can be delivered to users on multiple platforms dynamically and asynchronously.

The Augmented Surfaces project [81] (see Figure 2.4a) applies user interface migration techniques in a spatially continuous augmented physical workspace spanning multiple portable computers and fixed displays. Devices are identified by 2D fiducial markers, the relative pose of which triggers the migration of application objects. A new device can be added to the workspace if it implements an object serialization interface and is tagged by a unique marker.

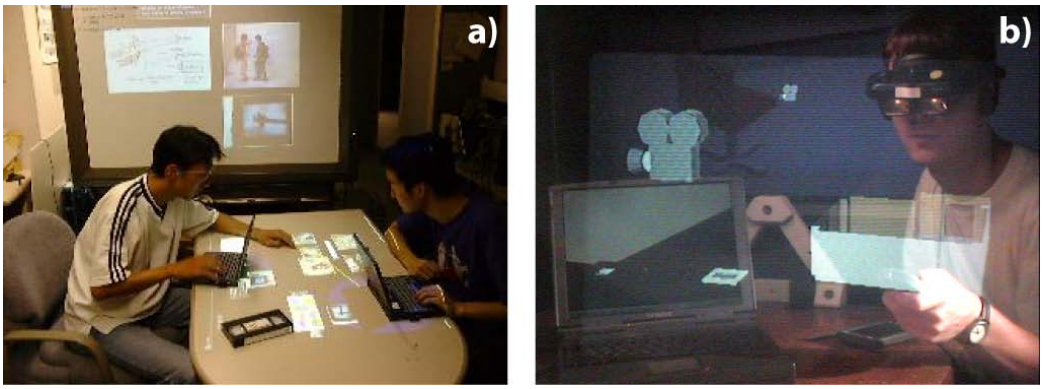


Figure 2.4: a) Interaction in the Augmented Surfaces project [81], b) Screenshot of the EMMIE framework's hybrid user interface [16]

The EMMIE framework [16] introduces a hybrid user interface for AR systems (see Figure 2.4b) enabling information management using a wide range of hardware devices. EMMIE's environment manager component addresses the needs of Ubicomp by providing techniques such as mixed reality interaction and privacy management to organize virtual information on several displays shared by multiple users. Augmented Surfaces and EMMIE implement ideas in a way that is conceptually closest to our work; however, neither EMMIE nor the Augmented Surfaces framework includes concepts such as proactive interface adaptation, persistent preference storage, and resource discovery.

2.2 Software Agents in AR

2.2.1 Animated Characters

Body and facial gestures as well as speech are familiar and widely accepted means of human communication. Animated characters, often with autonomous and affective behavior, have proved to be useful in man-machine communication since they are able to exploit and deliver information through multimodal channels and thus engage the user in a natural conversation. Autonomous agents have been actively researched in recent years as an interface to computerized systems bridging the communication gap between man and computer, and the real and virtual world. Augmented Reality (AR) applications share the same goal through the enhancement of the real environment with useful virtual information, where virtual objects appear to coexist with the real world.

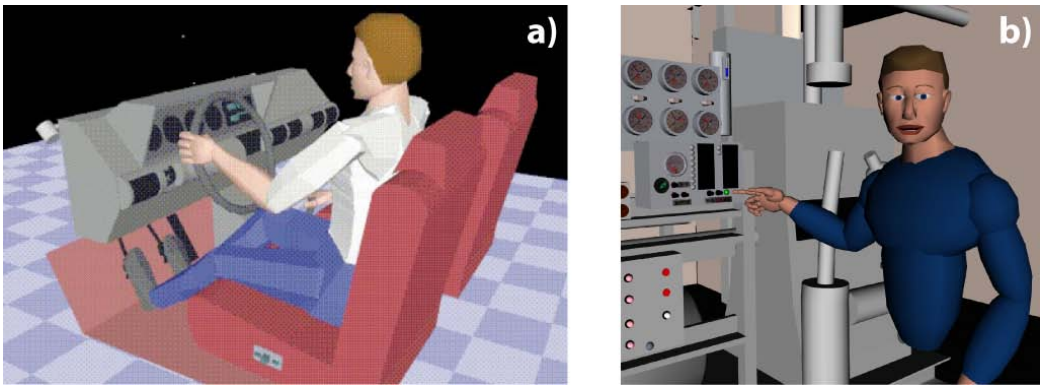


Figure 2.5: a) Jack animation system [69], b) STEVE animation system [84]

Virtual Reality (VR) is a more mature field than AR and has consequently already explored new interaction techniques involving animated agents, therefore it provides many useful ideas. One of the outstanding VR examples is the Jack animation system (see Figure 2.5a) from Noma et al. [69] that allows animated virtual human figures to be used in a wide range of situations from military trainings to virtual presentations. The Improv system [70] creates a novel interactive theater experience with real-time virtual actors on a virtual stage. The autonomous pedagogical agent of Rickel and Johnson [84] called Steve operates as a virtual trainer in an immersive VR environment (see Figure 2.5b) presenting complex interactive, educational machine maintenance scenarios.

Although users of these VR systems may have a strong sense of coexistence with virtual objects, they lack a connection to the real environment, as provided in AR systems. An early AR application providing character support is the ALIVE system [56], where a virtual animated character composited into the user's real environment responds to human body gestures on a large projection screen. This type of display separates the user's physical space from the AR environment, which demands carefully coordinated user behavior. As illustrated by Figure 2.6a, the Welbo project [4] features an immersive setup, where an animated virtual robot assists an interior designer wearing an HMD. Although the virtual robot character appears to be an integral part of the real user environment, it lacks a tangible physical representation and can only interact with virtual objects. The idea of the Steve agent recurs in an AR setting in the EU project, STAR [101], which aims to enhance service and training in real factory environments using virtual humans (see Figure 2.6b). Their robust machine maintenance scenario

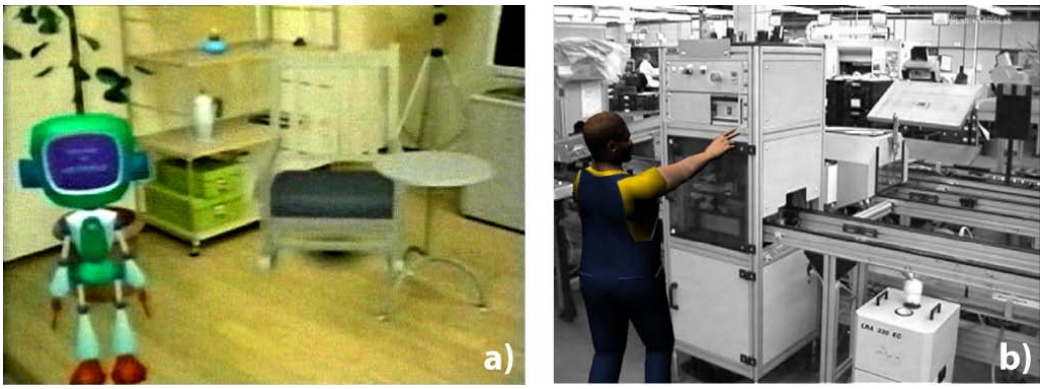


Figure 2.6: a) The Welbo interior design assistant [4], b) The STAR factory training assistant [101]

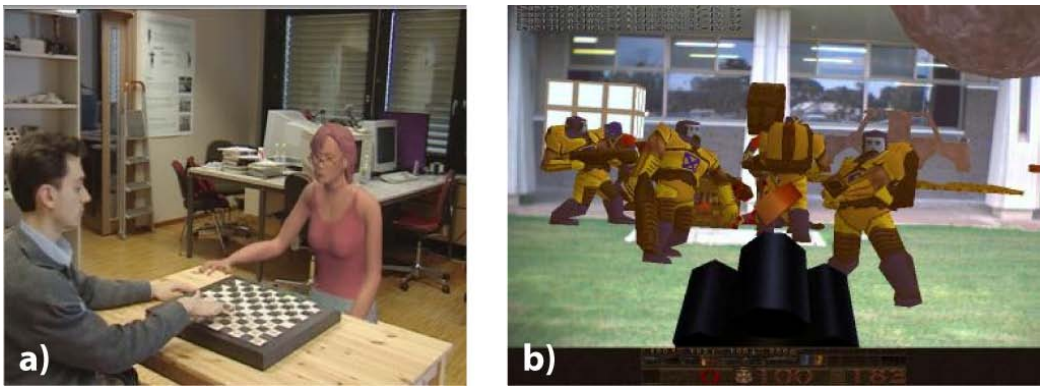


Figure 2.7: a) Virtual and real human playing checkers [8], b) The AR Quake game [71]

is a similar idea to our demo application, however, their system acts as an animated guided presentation, not a responsive interactive system.

AR has started to step beyond the usual instructional and presentational domain and is now being used to explore new application fields, for which animated agents open new perspectives. MacIntyre et al. [52] make the point that a new medium, such as AR, starts to gain wider public acceptance once it enters the game, art and entertainment domain. Their interactive theater experience places prerecorded video-based actors into an AR environment. The characters do not possess any autonomy, as their behavior is scripted, and interaction is limited to changing viewpoints and roles in the story. Cheok et al. [21] also experiment with Mixed Reality entertainment with live captured 3D characters, which enable real persons' telepresence in

a Virtual or Augmented Reality setting but without any control of the environment. Cavazza et al. [19] place a live video avatar of a real person into a Mixed Reality setting, and interact with a digital storytelling system with body gestures and language commands.

Balcisoy et al. [8] experiment with interaction techniques with virtual humans in Mixed Reality environments, which play the role of a collaborative game partner (see Figure 2.7a) and an assistant for prototyping machines. ARQuake [71] recreates the famous first-person shooting game in a real campus setting using a mobile AR setup, where the user has to shoot virtual monsters lying in ambush behind real buildings (see Figure 2.7b), and uses a tangible interface to fire virtual weapons. Both systems above exploit real world properties to control the virtual world, however, physical objects always serve as a passive background, rather than active performers.

With the exception of the DART system by MacIntyre et al. [54], which is an authoring framework for AR applications enhancing a commercial multimedia authoring tool, all of the related researches are bound to a single application and technology, and lack a general approach to create a reusable framework. Moreover, none of them consider physical entities as equal, active partners of virtual characters in dialogs, as they were predominantly used as passive objects like 3D pointers, tracking aids or interaction devices.

One of the main goals of this thesis is to create a set of software components that allows easy enhancement of AR applications with animated agents. Additionally, these components should allow physical objects like a robot, a printer or a digital piano to act as context-aware, interactive responsive agents that perform various tasks with digital actors, virtual presenters and other synthetic visual elements.

2.2.2 Mobile Agents

Unlike desktop agents that are limited to operate in the 2D world of a computer screen, agents in AR may move in the user's physical environment using all 6 degrees of freedom. With the simultaneous use of various stationary and mobile devices AR environments offer not only the freedom of a single three-dimensional physical space mapped to a display but several interconnected spaces. Moreover, the portability of a PDA or a mobile phone offers dynamic characteristics that enable agents to step out from their previously static environments and exploit mobile features such as current location and context. Thus autonomous interface agents increase their mobility and gain another output modality, as the current pose and choice of display may both carry an important message for users.

Kotz and Gray [43] use the term mobile agent for autonomous software

Figure 2.8: a) Mobile agents in the early C-MAP system [58], b) Mr. Virtuoso, an animated character as art history consultant in a handheld AR system [102]

components that have the ability to transfer and reproduce themselves on various networked computing devices. By equipping AR agents with mobile characteristics, they are no longer bound to a single, statically configured application and output device but may opportunistically migrate to and take advantage of other platforms more favorable for the agent's current needs.

Recent advances in hardware and software technology for portable devices such as PDAs and smartphones have eliminated the hitherto serious constraints on the visual representation of migratable agents. An early appearance of context-aware interface agents on mobile devices can be found in the C-MAP system of Mase et al. [58]. They employed simple 2D characters as tour guides (see Figure 2.8a) to deliver location-based information on portable PCs and PDAs. While C-MAP visualized its context-aware virtual museum guide as a sequence of static 2D images, Wagner et al. [102] presents a similar scenario in AR with a full-fledged virtual 3D character (see Figure 2.8b) exhibiting reactive behavior on a consumer PDA. Their Mr. Virtuoso character acts as a consultant in art history in a collaborative educational game.

The embodied mobile agents of the Virtual Raft project [99] appear to "jump" between tablet PCs carried by participants of a playful museum exhibition. Similar character behavior was implemented by Kruppa et al. [44], whose PEACH system experiments with visualization techniques using an animated cartoon character to preserve the continuity of an animated presentation spanning multiple displays. The system introduces a "multi-device" presentation agent that is able to move between devices (e.g. transfer from a PDA to a large display) to draw attention to a certain feature within the presentation.

Gutierrez et al. [34] use a PocketPC device to control the appearance and body posture of animated 3D characters of a large VR framework through standard MPEG-4 parameters. As described later, the UbiAgent framework enables the use of PDAs as multi-purpose interaction devices that serve simultaneously as platforms for agents to appear as well as an interface to dynamically control their behavior.

Besides our work, the Agent Chameleons framework [25] has been the only project to date allowing agents to seamlessly travel between real and virtual bodies while being controlled by a central control logic. Similarly to UbiAgents, Agent Chameleons are also based on a BDI agent architecture (see Section 4.1.4 for details), however, their system design lacks essential properties of ubiquitous AR systems such as application encapsulation, persistency, and dynamic agent platform management.

Chapter 3

Augmented Reality Agents

3.1 Design Requirements for Agents in AR

Our main goal is to create a set of software components that allow easy enhancement of AR applications with animated agents. Additionally, we want to turn physical objects like a robot, a printer or a digital piano into context-aware, interactive responsive agents that perform various tasks with digital actors, virtual presenters and other synthetic visual elements. Therefore an animation framework supporting animated agents in AR spaces needs to address the following major issues:

- A high-level command interface is needed where virtual and physical objects are treated as equal and active entities. This interface should also relieve authors and developers from the burden of working with low-level details such as absolute 3D coordinates or internal communication and synchronization mechanisms among agents, and thus rely on abstract entities and references to object attributes to let the framework resolve their current value internally.

- As AR applications typically consist of a complex network of software modules, it is not desirable to modify their internal structure to prepare them to work with animated agents. Therefore applications need to be encapsulated as black boxes that communicate with external software components – including AR agents – via relevant input and output attributes. Output attributes enable agents to monitor and deduce internal application status. Based on this status information, agents are able to make decisions and execute actions that provide feedback to applications through their input attributes that thus serve as a control interface.

- Agents should not be bound to a single hardware and software environment. They should be reusable in multiple AR environments and exploit dynamic resources offered by multiple stationary and mobile devices and AR applications.

This and the next chapter present design principles for an animation framework meeting the above requirements. For the sake of clarity, we present an evolutionary design that first tackles the issue of a uniform command interface for real and virtual objects and application encapsulation by *AR agents*, then an improved design to support the general requirement of Ubicomp systems about avoiding dependency on a single technology and application and exploiting multiple hardware and software environments.

3.1.1 Agent Representation

AR agents are embodied as three-dimensional virtual or physical objects. They share users' physical environment, in which they can freely move using all 6 degrees of freedom. Virtual agents in AR scenarios appear to have a solid, tangible body that can be observed from an arbitrary viewpoint, thus becoming integral parts of the physical environment. Virtual objects are typically animated characters but are not necessarily anthropomorphic. In some AR applications a fully fledged virtual human can be more distracting than a simple animated arrow that may communicate more information. Therefore, we experiment with various character forms.

A novel and exciting new aspect of AR agents is that physical objects such as a printer, a digital piano or an interactive robot can be turned into intelligent, responsive entities that collaborate with virtual characters. If we track and monitor relevant physical attributes and process this data, attribute changes can generate events that can be interpreted by other agents and application logics. Using network packets, infrared messages, MIDI code sequences or other means of low-level communication, physical objects can not only be queried for status information but can also be controlled by external commands that trigger actuators. Therefore physical objects act as *input and output devices* in AR spaces.

The combination of the real and virtual representation yields the augmented representation. This assumes the presence of a physical representation and only superimposes necessary virtual information on top of it. Virtual agent representations may have an associated tracked physical object, which serves as a tangible control interface, while this is a prerequisite for augmented agent representations. Screenshots in Figure 1.1 in Section 1.1.1 illustrate three different representations of a LEGO Mindstorms® robot.

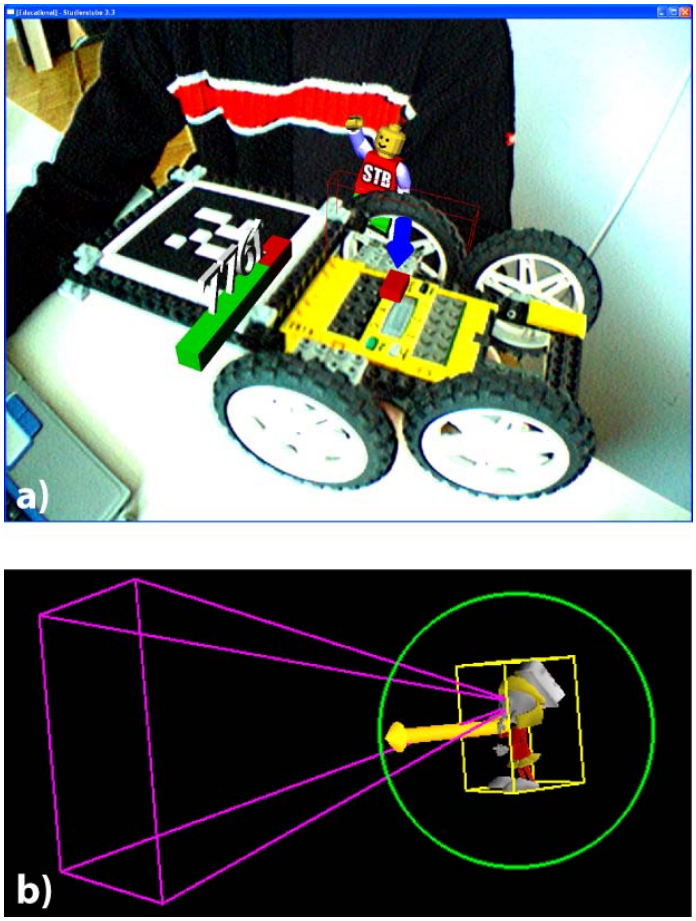


Figure 3.1: a) An augmented real LEGO robot occludes a virtual cartoon character, b) Virtual sensors for an animated character: yellow bounding box = touch sensor, purple frustum = virtual eye viewport, green sphere = virtual hearing volume, yellow arrow = velocity visualization

As AR uses physical objects as first-class entities, certain constraints imposed by the limitations of our human sensors are introduced that were formerly unknown in VR applications. We are unable to look through opaque physical objects or examine their internal structure, see in dark or foggy places, or hear distant sound sources. We also have difficulties in selecting relevant information in too much or too noisy data. The augmented representation may use several visualization techniques to overlay virtual information on top of physical objects to overcome these limitations. Without making an exhaustive enumeration, we describe a few examples from various potential application scenarios how the capabilities of our senses can be enhanced by AR techniques:

- *Labels and icons:* Virtual labels and icons placed next to parts of a complex factory machine can explain functionality or display information about the current internal machine state to aid a technician's work, e.g. labeling relevant buttons in a workflow stage, display of valves' pressure levels, or explanation of an engine's operation.

- *Wireframe overlay:* Instead of showing a fully detailed virtual model in a simulation, a simple wireframe model often yields better emphasis of certain features, such as marking suggested doors in a building leading to a selected navigation target, or enemy localization in a battle on a fighter pilot's head-up display.

- *Zooming:* Zooming onto some fine details draws the user's attention to information otherwise easily over-looked, such as problematic areas during an AR-aided surgery session on the patient.

- *X-ray vision:* This often quoted "superman"-like feature of AR allows the observation of opaque objects' internal structure by adjusting the transparency of the virtual objects rendered on top or in place of physical parts. Possible applications include displaying hidden components of a complex machine for explanation purposes or showing occluded rooms or doors in an indoor navigation system to aid identification of the navigation target.

- *3D fisheye:* A 3D version of the well-known 2D fisheye technique would prove useful when emphasizing a certain part in a highly complex spatial structure such as a car engine. The 3D fisheye tool can temporarily reorganize the virtual model by enlarging the parts around the center of focus while suppressing others further from the center to help a car mechanic identify potential erroneous areas in the real engine.

- *Anticipation with animation:* During the design phase of a critical work procedure for a factory it is useful to play animations to demonstrate what *would* happen before a button was really pressed or a lever was pulled, e.g. animating a network of engines revealing design or construction errors before damaging the actual device or injuring the user.

- *Gestures:* Virtual animated agents are able to perform human-like gestures over meaningful physical locations. These gestures can be used to visualize assembly steps for instance for DIY furniture, which would be the animated 3D version of the static explanation images often seen in assembly and maintenance manuals like that of IKEA.

The augmented representation also helps overcome the problem of correct visual occlusion. This means that we should ensure that physical objects placed between the user's viewpoint and virtual agents appear to cover parts of the virtual objects behind. This issue can be easily supported by tracking the occluding object's pose and associating it with an augmented representation that only renders an approximate virtual model into the depth buffer at the right location. Figure 3.1a provides illustration.

Autonomous AR agents may proactively choose the visualization technique most appropriate for the current context. They can recognize the application they are embedded into, critical stages within the workflow, uncertain user behavior calling for help, or occlusion conditions when visualizing dynamic information. Although most of the visualization techniques used in our demo have been previously presented individually by others [7][9][24][50][51], their use and combination with the augmentation of physical objects and autonomous agents is novel.

In a collaborative AR setting the attributes of visible virtual information depend on user location and orientation (e.g. only annotating objects currently visible to the user), and profile (e.g. novice users receive more basic explanation information than experienced ones). Therefore, virtual objects can be rendered in a user-specific way. The attributes of the information must be synchronized and shared among the users. The collaboration with physical objects is obvious, no technology is required.

In AR scenarios users are mobile, traveling between different physical locations and hardware setups, therefore they require cross-platform, mobile assistants. AR agents can "live" on several devices and displays such as HMDs, projection screens, PDAs or more recently mobile phones. Each has its own local coordinate system placed into the global coordinate system of the user's physical environment. As we describe in detail in Section 4.1.1, AR agents are able to smoothly travel between devices and coordinate systems.

3.1.2 Agent Behavior

An AR agent interacts in real-time with other agents in shared local or remote AR spaces, with users working with collaborative applications, and the applications they are embedded into. In addition to their capability of executing scripts, they possess certain autonomy, which means that they watch and automatically react to changes in the properties of AR spatial objects.

The scheme represented in Figure 3.2 depicts the interaction flow of autonomous agents within AR scenarios. The agent monitors the physical and virtual world by means of physical and virtual sensors. While processing

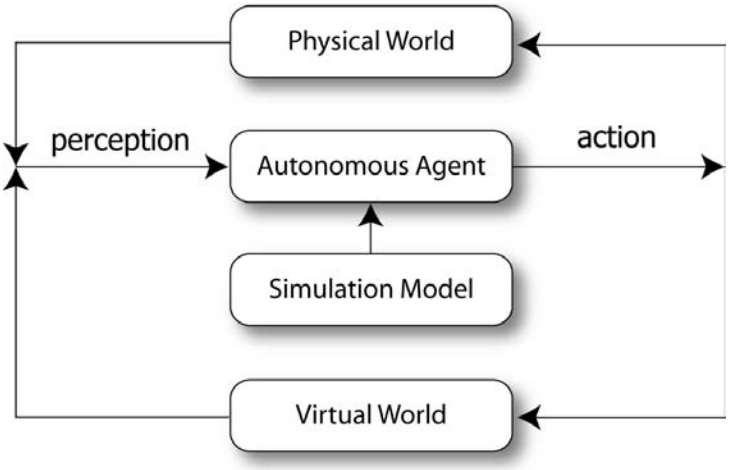


Figure 3.2: Autonomous agent behavior scheme

the information from light, push, angular, temperature, and other physical sensors is obvious, a rich research corpus [13][82][45] indicates that implementing a perception capacity for virtual entities is non-trivial. Examples of virtual sensors include the following:

- *Visual sensor*: Agents "see" users or other agents when their bounding box intersects with the viewing volume associated with the agent's virtual eyes, which may be a single frustum or several frustums, a box, or even partially unbounded space. Once in the viewport, agents start observing the position, orientation and hence velocity of users, interaction devices, agents and other physical and virtual objects. AR applications are also associated with a physical position and orientation since their working volume usually augments only a subspace of the real environment.

- *Audio sensor*: The agent can "hear" a sound object if the sound source's propagation volume intersects with the agent's hearing volume (typically represented by spheres).

- *Tactile sensor*: Touching is modeled by collision detection, therefore we need to properly calculate bounding volumes for both physical and virtual items. Some physical entities such as displays may not make use of a precise bounding box but instead a predefined "hotspot" area, which is not related to actual physical boundaries and triggers events once an object has entered it.

Figure 3.1b shows the visualization of a virtual touch sensor for a real object (see the bounding box of the LEGO robot), the virtual eyesight of a virtual cartoon character (a wireframe viewing frustum) and its virtual ears (a sphere around the character). Agents can be equipped with object and application-specific sensors as well that examine application attributes, GUI input, and internal state information of virtual objects (e.g. the emotional state of a virtual human) and physical objects (e.g. an error message of a printer).

Perception is followed by processing incoming information. With the assistance of an internal simulation model, the agent performs actions in response to input events. Traditional multimodal output channels can be opened between users and agents such as non-verbal communication (facial and body gestures), speech synthesis and recognition. However, AR offers novel, compelling modalities involving pose, velocity and status information of objects. The physical location agents inhabit, the direction they are looking into and the objects they control all convey important context information. These new modalities enable a wide range of new behavioral patterns such as the following:

- The user places a character into the physical working volume of an application. The character receives an event with the identity of the user and the application, and loads the user's application-specific profile and the state in which she last left the application. The character continues to work with this application.

- A virtual presenter is working with a user in an immersive AR setup and wears an HMD. She decides to work in another room with a projection screen suitable for a larger audience. She takes a pose-tracked PDA, moves it close to the character and "picks it up". The character continues to "live" on the PDA screen until it is carried over to the projection screen in the other room. It then becomes aware of the new environment and jumps to the projection screen, where the same application is running, maintaining the state of the user's work.

- A machine in a large PC cluster starts malfunctioning. A virtual repairman character identifies the computer in the cluster room, then leads the human operator to the computer's physical location. Once in the vicinity, the repairman points out the possible sources of error on the machine itself. An explanation is only begun once the operator looks at the repairman, in order to ensure appropriate attention and focus. The machine sends feedback to the repairman when it is back to its normal state.

Figure 3.3: Water puppets

3.2 The AR Puppet Framework

The author of this thesis had the chance to admire the famous *water puppet theater* ("mua roi nuoc") in Hanoi, Vietnam [28], where exceptionally skilled puppeteers animated a group of puppets using hidden, underwater controls while performing Vietnamese legends (see Figure 3.3). Although the puppeteers focused only on their own controlled puppets, they always stayed perfectly synchronized since they followed well-prepared instructions from a choreographer. The choreographer received instructions from the director, who actually breathed life into the legends told by storytellers.

In digital storytelling it is common to use a hierarchical structure similar to that used in a theater [91] since these terms, which often represent complex system components, are familiar even to non-technical people. Although the comparison is not novel, we found that tasks to control AR agents can be divided into discrete groups that closely match the layers of a puppet theater's multilevel structure. We therefore borrowed the stage metaphor for AR spaces, story metaphor for applications, puppet metaphor for AR agents, and the puppeteer, choreographer and director metaphors for various control logics. User interaction and other external events influence the director's behavior, while interaction is mediated by the storyteller assuring that the story proceeds in the desired direction. These components build up our hierarchical animation framework (see Figure 3.4), which we call *AR Puppet*. Each component's role in controlling our agents is now briefly explained.

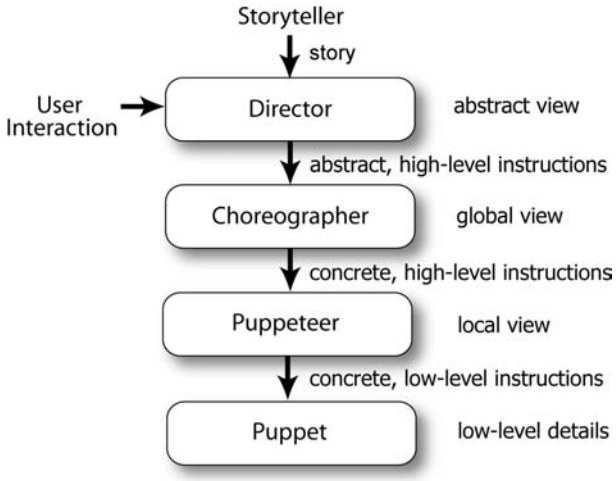


Figure 3.4: Overview of the AR Puppet framework components

3.2.1 Puppet

The bottommost component is the puppet level. A puppet stands for one representation of an AR agent, which can be physical, virtual or augmented, and may appear on various platforms ranging from an HMD to a PDA. Various puppets may require different code implementation, support for connection management and communication protocols with mobile and remote devices (e.g. open/close/recover connection, send/receive data with TCP/IP, IrDA or MIDI), and visualization methods (e.g. level of detail). Despite the diverse low-level implementation details, puppets of an AR agent belong to the same logical entity, namely the *puppeteer* visualizing a particular type of AR agent in various forms.

We created several embodiments for animated AR agents that are used in example applications of our framework:

- *Agents based on physical objects:*
  - an augmented LEGO Mindstorms® robot
  - an augmented MIDI keyboard [11]
  - an augmented UbiSense ultra-wideband tracking system [63]
- *Virtual agents:*
  - a skeleton-based animated virtual character built on the open source library Cal3D [18]

- a virtual character based on the Quake2 game's MD2 format
- a PDA-based virtual character built on the FPK library from Daniel Wagner's handheld AR software suite [35]
- an affective talking head

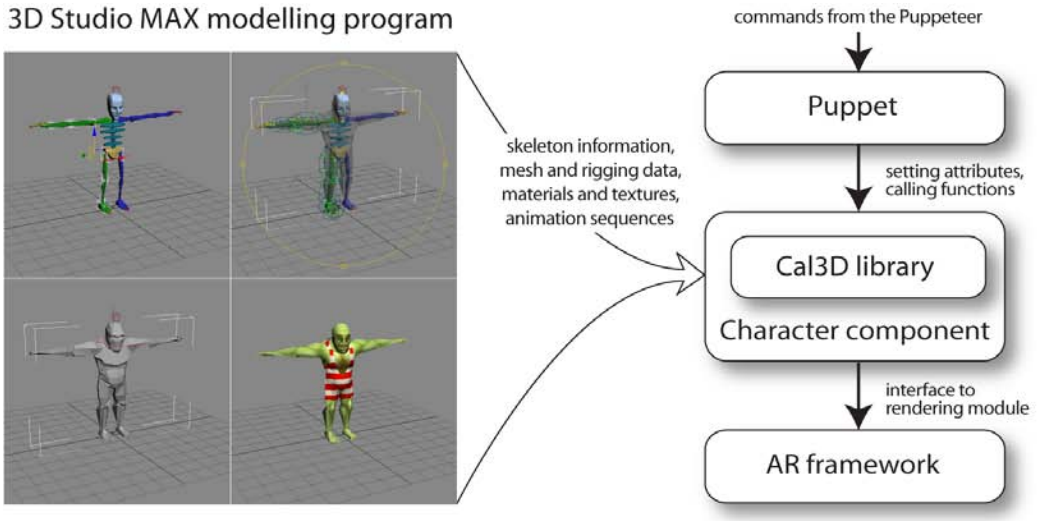
Most of our example applications rely on the Cal3D-based character that is capable of the import and display of high-quality animation exported from 3D Studio MAX's Character Studio [1], allowing for unlimited animation possibilities. It also allows direct access to bones, which permits inverse kinematics and the linking of objects to joints, for example to pick up and carry objects. Figure 3.5a illustrates how a Cal3D-based virtual character can be integrated into AR Puppet as a puppet component.

Using physical objects as puppets implies a bigger challenge as a bidirectional communication channel needs to be constantly maintained between the AR application's virtual control logic and the real object. This channel is used for activating the physical object's actuators to execute actions associated with the puppet's current state, and for querying the object's sensors to update internal object status in the AR application's world model. Figure 3.5b shows the integration of a MIDI keyboard into the AR Puppet framework. The associated puppet component contains a MIDI handler module that transforms puppeteer commands into MIDI messages and sends them to the keyboard. The same module is responsible for querying the keyboard again with MIDI messages to provide feedback about the internal status of the keyboard. This status will be reflected in the puppet's attributes. Besides low-level communication with the physical object, a virtual 3D representation is provided for the augmented keyboard representation, accompanied by registration information for a correct overlay.

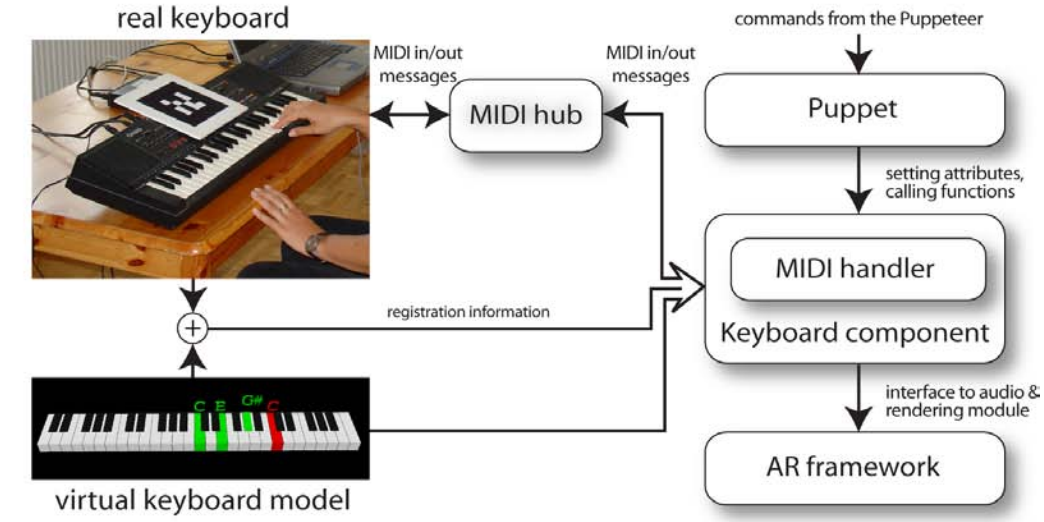
3.2.2 Puppeteer

On the next level the puppeteer is the component that groups puppets together and controls a selected set of agent representations at the same time. It knows exactly "which strings to pull", that is how to implement higher-level instructions for each puppet to obtain a desired effect. The puppeteer has the following responsibilities:

- Providing a unified command interface, which enables scripting of physical *and* virtual objects simultaneously. A default implementation is provided for a predefined set of commands including common locomotion and presentation functions. These default commands can be overloaded by derived objects.



a)



b)

Figure 3.5: a) Using a Cal3D library-based virtual character as a puppet in AR Puppet, b) Integrating the augmented representation of a real MIDI keyboard as puppet

- Customization of default parameters of virtual sensors (e.g. viewing frustum parameters, hearing sphere radius, bounding volume).
- Support for new tracking modalities such as pose and internal status
- Support for idle behavior
- Command adaptation

Command adaptation means that high-level puppeteer commands need to be tailored to the capabilities of its puppets. This is necessary in the following cases:

- *Switching between representations:* If one agent representation becomes unavailable, it needs to be turned off while some other representations have to be turned on. For instance, if the physical representation of a machine is broken or malfunctioning, we can switch to its virtual representation simulating the appearance and behavior of the real object.
- *Device adaptation:* A character has more freedom to move around when it appears on an HMD than on the screen of a mobile device, therefore certain motions cannot be performed. Instead of moving and pointing to a 3D location, a PDA-based agent could just give a visual or audio hint about its whereabouts.
- *Animation parameter adaptation:* A high-level motion command requires adaptation of animation parameters. For instance, if a character has to move to a distant location but a walking animation would appear unnatural within the allowed time interval, a running or flying animation sequence should be triggered.
- *Motion constraint adaptation:* The puppeteer facilitates the puppets' adaptation to motion constraints such as terrain and path following.

3.2.3 Choreographer

While puppeteers focus only on their respective puppets, the choreographer has a general overview of all puppeteers and their attributes. This level does not deal with character-specific details but uses high-level commands such as "go to my printer and point at the paper tray that has become empty" in a printer repair task. In a highly dynamic environment such as AR users move around and work with different applications, objects are displaced and devices may malfunction, low-level information (e.g. absolute

position coordinates or internal object status information) constantly change. Consequently these dynamic information elements need to be hidden from users and applications as these high-level entities deal solely with abstract names and spatial references.

The tasks of the choreographer component are the following:

- *Resolving spatial and object attribute references in commands:* The choreographer parses high-level director commands for spatial and object attribute references in an "`object_type(name).attribute`" format and substitutes them with the current attribute value to create commands that puppeteers having only a local view of the application understand. Revisiting the aforementioned printer maintenance example the director command sequence "`goNear trackedObject(printer) .position, pointAt trackedObject(printer).errorPosition`" for a virtual repairman character would be translated into locomotion and animation commands containing exact 3D coordinates as targets for the character's puppeteer.

- *Path planning:* The choreographer is aware of all objects between the source and target of a moving agent, therefore it is able to plan its locomotion to avoid obstacles. For instance, the virtual repairman character would be guided to avoid walking through a computer monitor located between the character's current position and the printer it wants to move to.

- *Feedback for synchronization:* If feedback is sent whenever a group of puppeteers finishes command execution, multiple agents in the same application can wait for one another, thus maintaining synchronization. For example the virtual repairman stops drawing attention to the printer's paper tray once it gets refilled.

3.2.4 Director

The director represents the application logic and interaction, and serves as the behavior engine of AR agents. The director's behavior can be decomposed into distinct elements associated with important "story" (that is application) parts: meaningful application states and attribute values. Switching between behavior elements drives the application forward based on events in the application environment, user interaction, and scripted behavior. Behavior states trigger animation commands passed on directly to the choreographer that lower-level components gradually transform into user interface actions executed by puppet actuators in the real and virtual world.

The director behavior can be well represented by a *finite state machine (FSM)*. FSM states correspond to individual behavior elements, while application events, user interaction, and script commands are mapped to transitions between states. Application events come from real and virtual sensors deployed in the real and virtual application environment. While choreographer and puppeteer functionalities have been designed mostly to be accessed through a scripting language, the current implementation of the director component is based on parameterized C++ code controlling the behavior engine's FSM. The director's code parameters can be dynamically adjusted by external components, offering some flexibility for the agent control logic.

The choreographer and puppeteer components encapsulate general functionalities of a 3D animation engine. However, the director component must be tailored to specific applications to prepare the agent for application-specific events and actions. This constraint and the somewhat limited flexibility of the parameterized FSM code decrease reconfiguration and reusability of agent framework components in diverse AR environments. Chapter 4 introduces an additional level of complexity by incorporating the hardware and software environment of AR spaces and presents solutions to overcome the aforementioned problems.

3.2.5 Storyteller

The highest-level component is the storyteller, which is a meta-component representing the author creating the application or story. It has only an abstract view of the components and the story flow, and contains instructions for the storyboard.

3.3 Integration with Applications

It has always been a challenge for interface agents to monitor the current state of the application in which they are embedded, and the behavior of the users they are interacting with, without modifying the application itself. One of the most powerful aspects of the AR Puppet framework is the easy way current applications and users can be monitored, which is grounded in the architecture of the *Studierstube* AR framework [85], which enables a wide range of distributed collaborative multi-user AR applications.

Both Studierstube and AR Puppet have been built on a scene graph database, where all entities are scene graph objects interacting with one another via input and output attributes called "fields". AR applications, users and components of AR Puppet are all parts of the same hierarchical

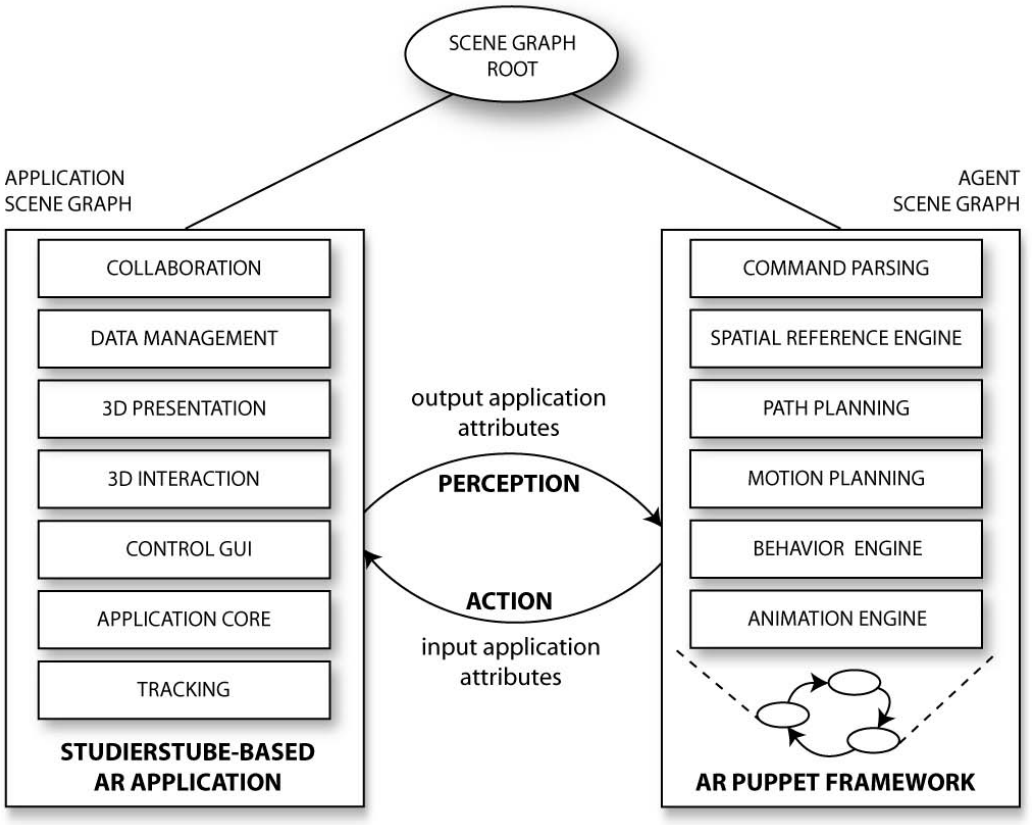


Figure 3.6: Communication schema between AR applications and AR agents

scene graph, therefore they can monitor one another's fields and immediately respond to changes. Numerous existing AR software frameworks [53][100][72] represent their components and applications by a distributed, hierarchical scene graph. AR Puppet is designed not only for use with Studierstube but can also be easily adapted to other scene graph-based frameworks that base their data representation and intercomponent communication on scene graph nodes and fields.

AR Puppet can also be tailored to AR frameworks that do not follow a scene graph-based approach. For instance, AR Puppet can be encapsulated as a component in the component-based DWARF framework [12] by adding an extra wrapper class transforming input and output attributes of AR agents and applications as *needs* and *capabilities* as specified in the framework model. A disadvantage of our scene graph-based AR platform is the lack of support for legacy applications. Interfaces to legacy applications must

be implemented on a case-by-case basis.

Figure 3.6 illustrates how AR applications can be encapsulated as black boxes by communicating only through input and output attributes with the agent framework. The internal decomposition of typical AR applications into modules has been described by Gerhard Reitmayr in his PhD thesis [78]. When an AR agent built on AR Puppet is to work with an AR application, sensors need to be attached to relevant output attributes of the AR application by making field connections in the scene graph. These sensors enable constant monitoring of the field values. Attribute value changes generate events inside the agent's behavior engine that may trigger a transition in the internal finite state machine. Whenever the state machine switches to a new state, behavioral actions such as animation sequences, playing sounds, etc. associated with the current agent state are executed. The agent is able to interact with the AR application by setting its input attributes.

Similarly to AR application, users also have fixed, standard attributes such as current tracked pose and display type to infer state and general profile information. User actions can be monitored through tracking user pose (e.g. by head tracking), biometric parameters (e.g. eye tracking, biosensors, etc.), and interaction devices and props. Animated agents in the AR Puppet framework are thus always aware of the application users' behavior.

Applications and agents should be prepared to interact with one another. If an application is to have the possibility to host agents, then it must supply them with functionality to exist:

- *Dynamic addition and mobility of agents:* A choreographer component has to be added to the scene graph, which can dynamically add/remove, enable/disable agent representations supporting mobility.

- *Monitoring the application's current state:* Agents need to be aware of the current state of applications. This internal state has to be mapped to a vector of attribute values, which can be easily observed by agents and the director component. Therefore we need to mark relevant attributes in the application's scene graph with a special tag, which is a quick and mechanical task in Open Inventor. When placed into an application, the agent can easily retrieve and query the marked attributes.

3.3.1 Example Application Scenario

Machine maintenance problems similar to the early work of Feiner et al. [27] are traditional AR scenarios. Let us imagine that a complex machine breaks down in a factory. Well-trained technicians are not always available due

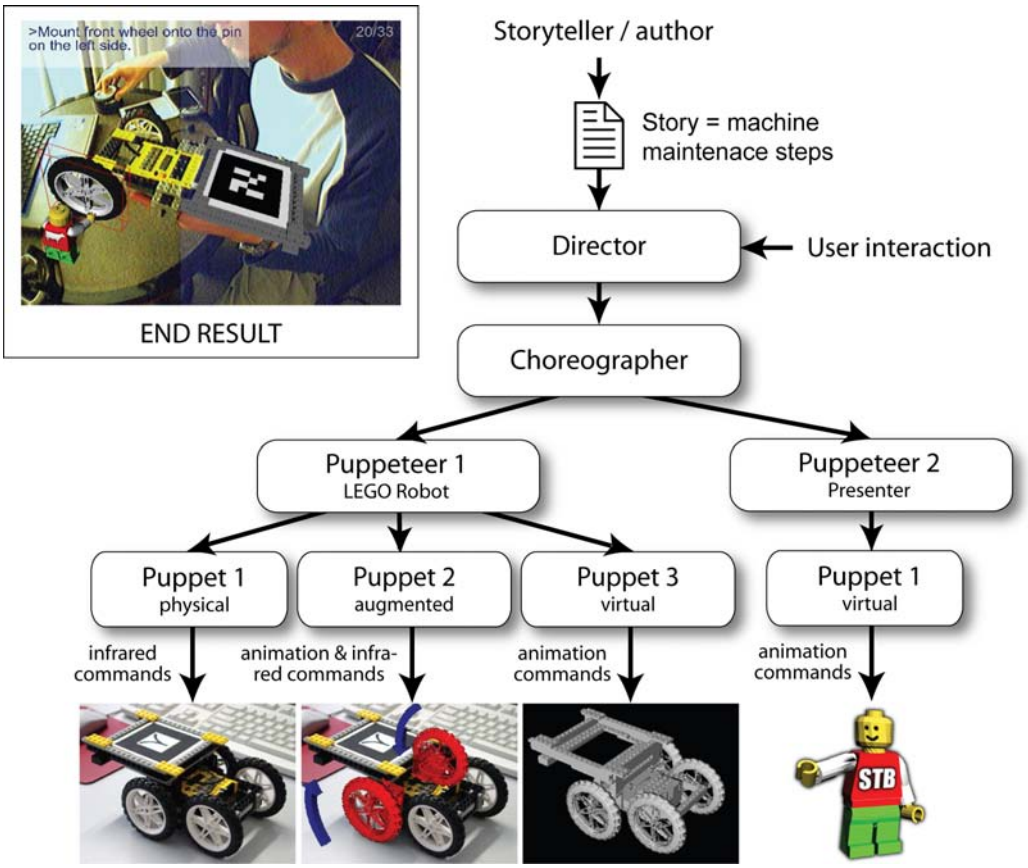


Figure 3.7: Decomposing the example scenario into AR Puppet components

to spatial, temporal or financial constraints, therefore lengthy manuals are supplied with the machine, full of illustrations concerning which button to press or which container to refill. Existing systems like the STAR project [101] provide an animated presentation by displaying a virtual manual superimposed on the real machine, however, they neither verify nor provide feedback whether the actual maintenance step has been executed correctly. A desirable feature involves continuously querying the physical system state and comparison with the demanded application state that should have been reached by the correct execution of instructions.

AR provides the overlay of virtual icons, images and animated models on top of the physical machine explaining what should be done and playing synthetic sound besides the original sounds of the machine. Simultaneously with the virtual information physical actions can be executed on the actual machine. With physical actions machine behavior simulation becomes un-

necessary since the real machine does exactly what it is expected to do in real life, and gives feedback through its own engines, instruments, control displays, LEDs, etc.

A further notable issue rarely tackled by AR-based maintenance applications is industrial safety. If an untrained operator could see what would happen if he pulled the wrong lever before the actual execution of the action, accidents could be avoided and industrial training would be more efficient by preparing trainees for the results of possible failures.

A possible implementation of the aforementioned scenario with the AR Puppet framework employs two agents to educate an untrained user to assemble, test and maintain machines composed of active (engines and sensors) and passive (cogwheels, gears, frames) parts. The two agents are a real robot and a virtual animated repairman. Figure 3.7 shows how this example scenario can be decomposed into AR Puppet components. This diagram serves only as illustration for AR Puppet as the actual application and implementation of this application scenario is described in detail in Section 5.1.

Both agent embodiments (agent representations in the machine maintenance application) are controlled by two puppeteers that are responsible for the robot and for the repairman character respectively. The robot puppeteer controls three robot representations or puppets: a virtual, a real, and an augmented representation. The three puppets stand for three different visualization modes, all controlled by the same control logic and behavioral model residing in the puppeteer component.

In case the real robot is available and a bidirectional communication channel can be established with it, then the real representation's puppet is preferred and activated. In case we are able to track the robot's pose (e.g. with a fiducial marker attached to the robot body), additional virtual information can be overlaid on top of the physical body that visualizes important status information such as remaining battery power and robot sensor state, and highlights internal hardware structure elements for explanation purposes.

The registration of virtual information with the real robot results in the augmented representation. The real and the augmented representation communicate with the robot via an infrared communication channel, through which they send commands to control the attributes of the active parts (e.g. engine voltage and direction, or type of the sensors) and query the current robot state (e.g. sensor configuration, current sensor values, communication channel failures, and battery level).

While the augmented representation does not render a full virtual replica of the real robot, the virtual representation contains a detailed and accurate model of the original machine's appearance. The puppeteer's internal behavior model is constantly updated to reflect the current status of the real robot.

In case the robot becomes unavailable due to breaks in the infrared channel or low battery power, or it is simply not physically present, the puppeteer detects the lack of communication with the real/augmented representation and automatically switches to the virtual robot puppet while deactivating the real and augmented puppet.

After the switch the virtual puppet's appearance reflects the last known status of the real robot and imitates real robot behavior by animation sequences triggered by the puppeteer. If the physical robot becomes available again, the puppeteer switches back to the real representation after issuing robot commands to synchronize the current status of the real and virtual model. The constant synchronization of representations ensures that the cognitive gap between the different representations is mitigated by maintaining a consistent and continuous visual appearance and behavior.

3.3.2 Communication Flow between Components

The director decomposes abstract story (i.e. application flow) elements into actual choreographer commands that are passed on to individual puppeteers: the robot puppeteer and the repairman puppeteer. The choreographer is aware of all public attributes of the puppeteers, therefore it is able to resolve abstract object attribute names and spatial references inserted into choreographer commands by the director.

The repairman's puppeteer has only a single virtual puppet to control. It provides a standard animation command interface for the choreographer component that is independent from the type of puppet used for visualizing the repairman character. As the puppeteer is the component that makes the translation into low-level, puppet-specific commands, it is transparent for the choreographer whether its high-level animation commands are executed by a skeleton-based anthropomorphic character, a 3D object relying on frame-based vertex animation, or just a simple animated arrow.

The robot puppeteer keeps pinging the real robot and automatically switches between real, augmented, and virtual representations. The current representation determines whether the puppeteer commands should be translated into low-level infrared robot commands controlling physical parts or animation commands manipulating a virtual 3D model. The augmented representation includes robot instructions as well as animation commands that are executed synchronously.

Figure 3.8 visualizes the communication flow between AR Puppet framework components in the example scenario of Section 3.3.1, which is described in detail in the following list:

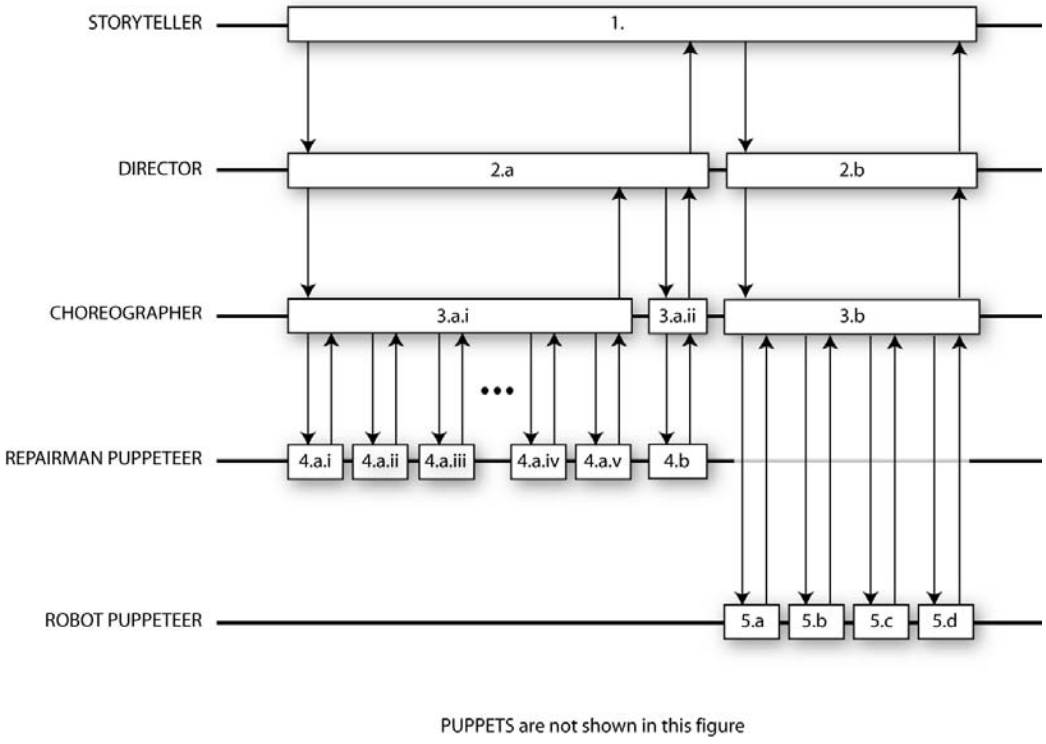


Figure 3.8: Communication flow in AR Puppet's machine maintenance example scenario

1. **Storyteller:**

   Tell the repairman to explain the user how to turn on the robot's engine

2. **Director:**

   Send command sequence to choreographer:

   (a) Tell the repairman to walk to the location of the robot button that turns the engine on and to play a button press animation over it

   (b) Tell the robot to turn its engine on

3. **Choreographer:**

   (a) Send command sequence to repairman puppeteer:

      i. Walk along a calculated $path(P_1,P_2, \ldots ,P_n)$ to avoid obstacles

ii. Perform button press gesture

(b) Send command sequence to robot puppeteer:
Reset and turn engine on

4. **Repairman puppeteer:**

Call low-level character functions in the virtual puppet:

(a) Walking:

i. Start walk animation
ii. Move between (P_1,P_2) with velocity v_1
iii. Move between (P_2,P_3) with velocity v_2
...
iv. Move between (P_{n-1},P_n) with velocity v_{n-1}
v. Stop walk animation

(b) Play button press animation

5. **Robot puppeteer:**

Call low-level robot functions and process messages from robot:

(a) Send an infrared command to ping the real robot puppet

(b) If the ping has been acknowledged, send a compound infrared command to set engine direction to "*forward*", set engine power to "*medium*", and finally turn the robot engine on

(c) If ping has not been acknowledged within a certain timeout, activate the virtual puppet by displaying the full virtual robot model and play an appropriate animation sequence visualizing the working engine

(d) If the real puppet is active and pose tracking is available, display animated arrows above the engine's cogwheels to visualize the direction they rotate in

6. **Virtual repairman puppet:**

Render mesh based on current animation keyframe sequences

7. **Real robot puppet:**

(a) Receive and execute infrared commands

(b) Send command feedback and status information via infrared messages

8. **Virtual robot puppet:**

   Render mesh based on current animation keyframe sequences

9. **Augmented robot puppet:**

   The actions of the real and virtual puppet together

3.4 Interaction between the Real and Virtual

Agent communication with the augmented environment includes four information channels to sense and affect the physical and virtual environment. The implementation of these channels is usually not a straightforward task. We provide some examples to demonstrate how the real and the virtual world can influence each other's behavior through various input and output communication channels.

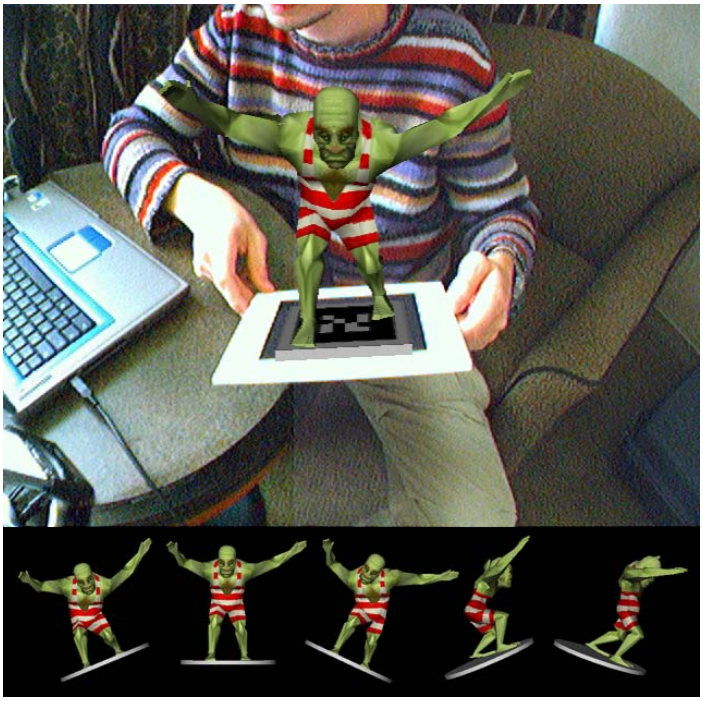


Figure 3.9: Animated character balancing on a tangible fiducial marker

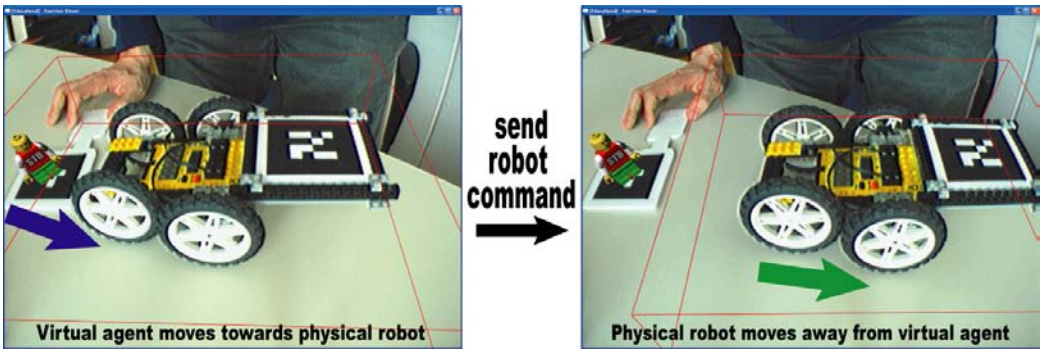


Figure 3.10: Defensive physical robot avoiding collision with a virtual character

3.4.1 Physical Input Affecting Virtual Output

Figure 3.9 illustrates a simple example how a virtual animated agent is able to respond to attribute changes of the real world. A tangible, physical fiducial marker acts as a platform for a virtual monster artiste to stand on. The user holds and tilts around the marker in front of a webcam, while the artiste agent appears to be struggling to maintain its balance on top of the real marker. If the angle of the marker becomes too steep, the monster falls down with a roar.

The application retrieves the current pose of the marker relative to the webcam using a fiducial marker recognition module [5]. The pose of the marker is directly mapped to the pose of the virtual platform of the artiste in the agent's world model. The webcam and the marker recognition library act as the agent's sensor to perceive changes in the physical marker's attributes. The agent's control logic then checks whether the platform orientation is still within bounds and decides whether to play the "fall down" or the balancing animation. The balancing animation is a blended motion interpolating between the neutral center and four extreme points in the animation space with factors calculated from the platform's pitch and roll rotation angles.

3.4.2 Virtual Input Affecting Physical Output

Sensing events generated by virtual objects is usually not a complicated task since virtual sensors such as vision, hearing or touching can be implemented in software. However, using physical objects as output communication modalities has several constraints. Our entire physical environment cannot be affected by virtual control logics, only by specially prepared objects that require communication channels and actuators to be set up.

The screenshots shown in Figure 3.10 depict a sample scenario implementing defensive behavior for the aforementioned physical robot that tries to avoid collision with a virtual character. The pose of the robot and the character is again tracked with a fiducial marker. If the character enters the virtual "safety" area around the robot, a command is sent to the robot from the PC via an infrared link, instructing it to move away.

Some researchers have made some efforts to create standard physical hardware components that are able to communicate with their virtual software counterparts such as the "phidgets" of Greenberg and Fitchett [32]. However, despite these efforts creating sophisticated responsive physical application objects still requires considerable electronics skills from developers.

3.4.3 Other Scenarios

In the case of physical input affecting the real world (e.g. two physical robots interacting with each other), the hard part of both previous scenarios is taken: sensors need to be installed in the real world again to monitor changes in addition to communication channels and actuators producing physical output. Generating virtual output based on virtual input is a trivial Virtual Reality (VR) problem that we do not discuss here in detail.

Chapter 4

Ubiquitous Augmented Reality Agents

4.1 Improving AR Puppet

With the AR Puppet framework we have shown how to construct animated agents in AR (*AR agents*), which have a physical as well as a virtual part in their input and output modalities. AR agents are able to monitor context information independently from the augmented environment they are embedded into and make decisions without constant user guidance. They can thus autonomously bridge the gap between the real and virtual part of mixed reality. However, the AR Puppet design described in Chapter 3 suffers from some limitations:

- The agent's brain is represented by a finite state machine that is hard-coded and not programmable on the fly. As a consequence, the agent operates with a static world model. While sensor information and user input provides live updates to the attributes of the world model, the structure of the world has to be defined in advance. The agent is thus programmed to work with a specific application and setting, and unable to adapt to new scenarios or react to unforeseen situations. In other words, the AR agent lacks typical capabilities of Ubicomp.

- The agent's behavior is mostly driven by the runtime system's flow of events and has only limited autonomy. The agents can respond to user input and other activities observed in the system but cannot derive any higher-order strategies that truly qualify as autonomous behavior. In combination with the static world model mentioned above, the set of different behaviors that agents can exhibit is limited and deterministic.

This chapter presents an improved design for a *ubiquitous augmented reality agent* (or UbiAgent for short), which overcomes most of these limitations. The UbiAgent works similarly to the AR agent, however, its behavior is not only event-driven but also proactive. Its decisions are based on a self-contained reasoning engine relying on a knowledge base that is externalized in a persistent XML database. Thus it becomes easy to influence the agent at runtime by updating its knowledge base.

The knowledge base is designed as a shared memory area, so that multiple UbiAgents and other distributed software components representing applications and real world objects can exchange messages. New software components can be dynamically added to this knowledge base, and the UbiAgent can learn to communicate with them through a standardized interface.

The persistency of the knowledge base allows the UbiAgent to preserve its state and preferences over time, so that it can opportunistically migrate from one networked AR environment to another, following a user around. Since the UbiAgent can also change its appearance in response to the current environmental conditions and these changes can be stored and retrieved on a per-user basis, it is capable of a behavior that we call *multi-user interface adaptation*.

The AR Puppet project integrated interface agents with autonomous agents to form AR agents, which are "smart" software components embodied by real and virtual objects operating in the user interface of AR systems. Section 3.3.1 demonstrates the capabilities of AR agents by a machine maintenance application, where a virtual animated repairman assists an untrained user to assemble and maintain a real robot. The behavior engine of AR agents relies on a finite state machine, where states represent sets of behavior elements (animations, sounds, etc.) and transitions are triggered by events coming from sensors deployed in the real and virtual world.

The virtual repairman agent operates independently from the robot maintenance application. It relies only on its own observations of application attributes and user input to generate an animated presentation of robot features without explicit user guidance. However, the state machine and the world model cannot be reconfigured at run-time. States and transitions in the agent-enabled application's control logic (the director component) are hard-coded, and the structure of the world model is manually configured before application start-up.

The manual, static configuration prevents the AR application from reconfiguring itself at run-time in case a more suitable environment appears for executing a maintenance step. The static approach hinders agent reusability and consequently increases software development efforts: the AR agent's rigid choreographer and director components need to be reprogrammed to

support another AR application the agent has not been previously "trained" to work for.

4.1.1 Increasing Mobility

To highlight key differences between UbiAgents and AR agents, we revisit the robot maintenance example in the following sections. UbiAgents enhance AR agents with mobile agent technology and migration capabilities by extending the previous finite state machine approach. Each state in the UbiAgent's state machine is now associated with certain requirements on the current software and hardware environment.

In the robot maintenance scenario the current construction step may require a minimum screen size and display resolution if it involves subtle visualization details of a complex engine. Other agent state preconditions may set a minimum CPU speed and maximum memory load for resource hungry animations. Portable computing devices may frequently demand agents to check the battery level and the wireless network status to avoid abruptly losing important application data.

If the requirements are met, the agent proceeds with the execution of the behavioral action sequence assigned to the current state in the state machine. However, if the capabilities of the current agent platform prove inadequate, the agent searches for another environment providing more favorable conditions to complete its job. If such an environment is found, the agent opportunistically migrates to it and executes actions in its new "home". Returning to the maintenance example, the virtual repairman may move to a bigger screen located near the user and continue to explain the robot's internal structure there, if the agent finds the current display too small for the current construction step.

Migration is signalized by a special behavioral sign such as an animation sequence, text warning or sound alert to make the user aware of the migration action or to instruct the user to prevent migration within a certain grace period. In case of the preventive scenario, the agent proactively suspends its current activities and advises the user to charge her PDA or set the screen resolution of the PC monitor to match minimum requirements. After the grace period expired without appropriate changes in the local workspace, the agent migrates to its new preferred environment and resumes its actions. Although migration causes an interruption in the application flow, this temporary break in the continuity of user interaction is invoked in favor of a more efficient work environment, shortening overall interaction times and increasing the quality of information visualization.

Migratable user interfaces demand that dominant interface properties are

preserved during and after migration to bridge the spatial and cognitive gap between disjoint workspaces. The user has to create a mental link between the old and new workspaces, thinking that the same virtual assistant continues to aid her work with the augmented robot, even if it migrated to a projection screen from a local display to increase its public exposure.

The agent should appear to continue its task exactly at the point where it left off before migration. Beside temporally continuous agent behavior, visual agent appearance also frequently needs to remain unchanged across multiple agent environments by migrating respective 3D models, textures, color schemes, spatial arrangement etc. together with the mental state. Nevertheless this is not a general requirement since in some cases the agent may deliberately choose a different visual representation. For example, a simple arrow may replace the pointing gesture of the full-body animated repairman of the robot maintenance application to avoid occluding subtle details of tiny mechanical robot parts. Similarly, the agent may choose to occupy a physical body taking advantage of sensors and actuators affecting the real world. For instance, the robot maintenance scenario can manipulate the real robot instead of a virtual robot model simulation, if the physical counterpart is available.

Agent migration and the preservation of agent attributes and mental state during migration necessitate the use of a central control logic that supervises agent embodiments. We call this component the agent brain being in charge of controlling multiple agent representations or agent bodies. The agent brain relies on a persistent information storage, where agent and workspace attributes can be saved and recalled.

4.1.2 Expect the Unexpected

Let us imagine that we buy a new microwave oven for our kitchen and want to employ an AR system to explain its operation. With current classic AR software design we would use a standalone application tailored to the explanation of our specific microwave oven model. We cannot get the already familiar animated repairman to introduce us our new household item instead of our robot, as this AR agent has never "seen" a microwave oven before and thus it does not know how to present it.

The UbiAgent framework teaches the old dog new tricks: we equip AR agents with capabilities to adapt to and work with hitherto unknown applications. Consequently, if we enhance the aforementioned household scenario with UbiAgent components, the microwave oven application becomes part of the dynamic world model of the new UbiAgent-based repairman character. If the application generates a request calling for an animated presentation

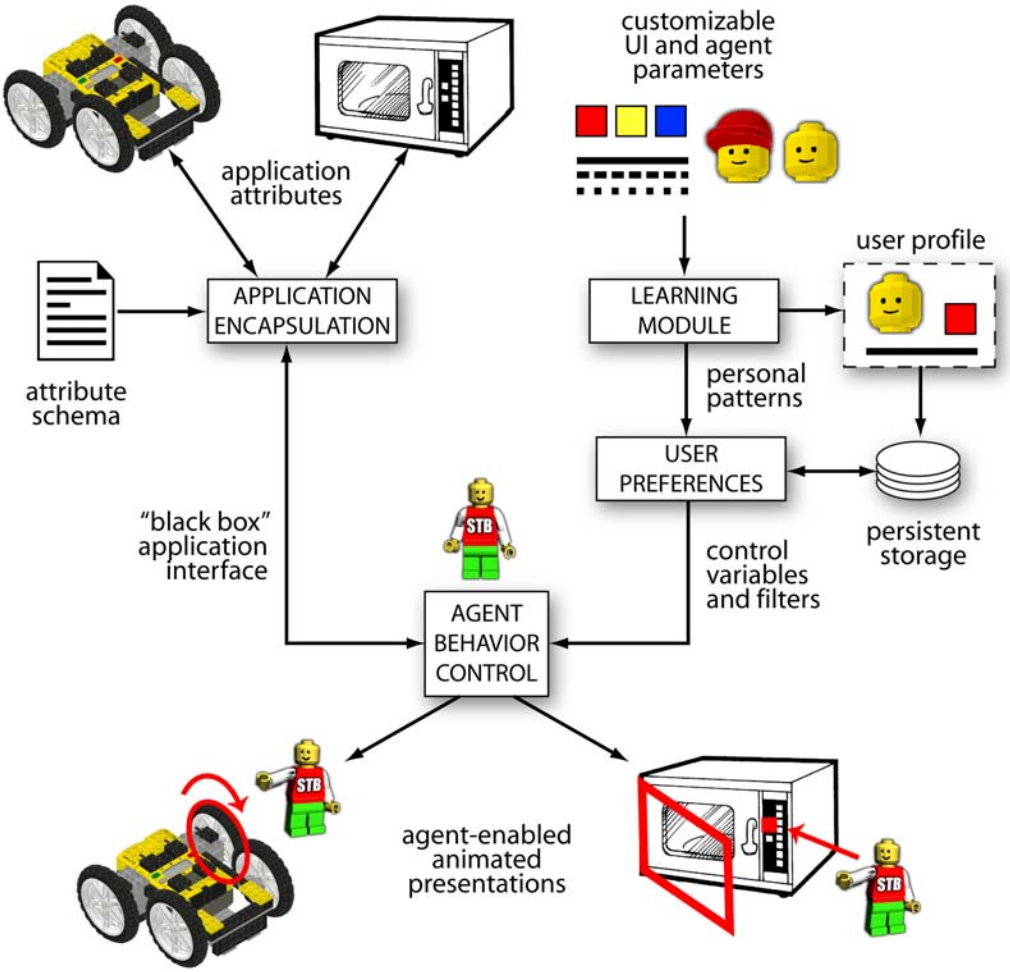


Figure 4.1: Application encapsulation with schema and adaptive user interface personalization

of its typical features, the repairman agent migrates to the new environment and starts the explanation.

UbiAgents encapsulate AR applications as black boxes hiding implementation details and communicating via relevant input and output attributes with the outside world. The black box interface maps private, internal application state to public attributes based on a well-defined schema. Any agent "understanding" this schema is able to automatically establish communication with the application, deduce application state information by monitoring these attributes, and influence application behavior by modifying attribute values. Figure 4.1 provides illustration for the concept about schemas and application encapsulation.

The diversity of AR application domains demands the creation of multiple schemas. Systems acting in the fashion of digital manuals need a schema enabling the presentation of a range of household devices, computing equipment and furniture, while indoor and outdoor navigation systems necessitate a schema for the encapsulation of parts of the physical environment such as floors, offices, streets, and buildings.

Multi-purpose agents need to understand several schemas to let the same agent work as a virtual tour guide or animated technician on demand. When a hitherto unknown application appears in the system and a UbiAgent wants to communicate with it, the agent first checks its schema. If the agent "speaks the language" of the schema, it includes the application in its control loop and reacts accordingly to attribute changes.

4.1.3 Multi-user Interface Adaptation

Users favor customizable interfaces over fixed ones. People have diverse preferences for the color, size, spatial arrangement, and numerous other style elements of user interface components, including accessibility features for the disabled.

Present AR systems offer offline tweaking of variables in parameterized user interfaces to dynamically change interface appearance, however, customization information is only considered in the current session without being stored in a persistent memory, ignoring adaptive and multi-user concepts. As illustrated by Figure 4.1, the UbiAgent framework includes a persistent database to store user preferences observed and accumulated by a learning module for future application sessions. The learning module captures typical patterns in the way users set interface customization parameters and stores them in a personal user interface profile in the database. This personalization profile follows users around while they are working with multiple distributed applications running on various computing devices.

A nearsighted user working with the previous section's machine presentation scenario would always enlarge objects to notice small details in the robot's mechanical structure. The learning module perceives this personal customization pattern and stores it in the database. The next time this particular user runs the robot maintenance application, all objects would be automatically enlarged based on the previously observed and stored preferences. When switching over to the microwave oven application, the user would find the virtual objects' default size already scaled up, saving hitherto efforts to tailor application interfaces to the user's taste and convenience.

The UbiAgent framework is based on a fast and robust database that enables storing and recalling preferences on demand for a large number of users,

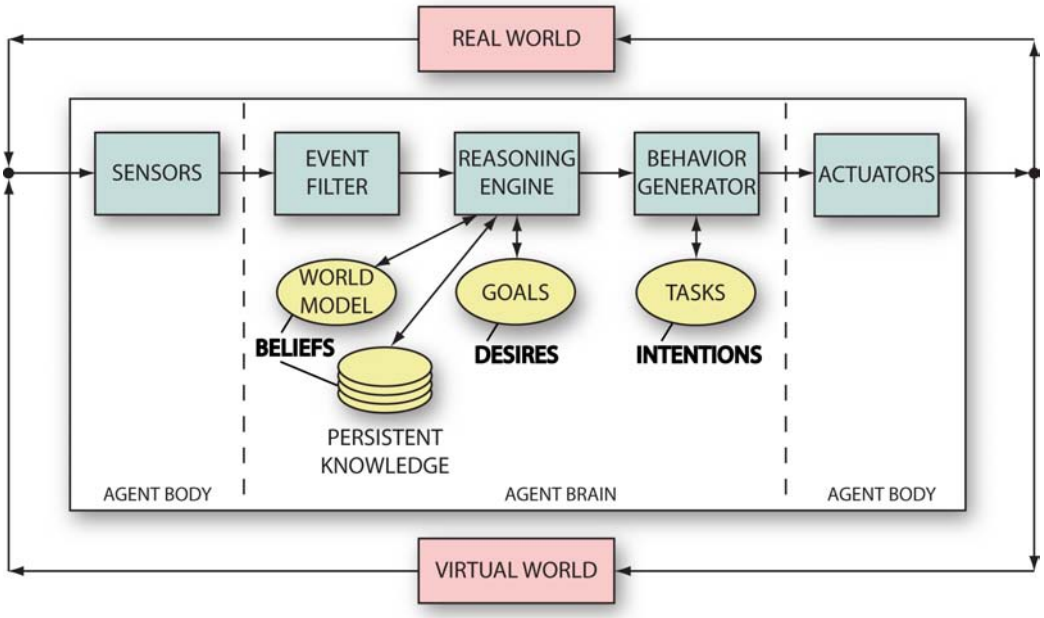


Figure 4.2: UbiAgent structure based on the BDI model

thus enhancing AR systems with multi-user interface adaptation capabilities. Identification of individual users relies on a unique user ID associated with personal devices such as PDAs or tablet PCs, or based on user accounts for shared public computers.

4.1.4 Beliefs, Desires, Intentions

Our framework follows the belief-desire-intention (BDI) model [15] for the implementation of the agent's reasoning mechanism. This model is not only one of the most well-known approaches for practical reasoning agents with a substantial research corpus, but is also highly suitable for dynamic and uncertain environments such as AR systems. Figure 4.2 depicts the BDI model-based structure of UbiAgents.

Beliefs represent the agent's current knowledge of the real world (such as the estimated pose and internal attributes of application objects) mapped to an internal world model. Since the world model represents only a potentially imperfect local view of the physical and virtual world, it needs to be regularly updated by measurements coming from sensors in the real and virtual environment. The database caches the current world model state between

measurements and stores persistent information such as user preferences, application attributes, and agent properties.

Desires stand for agent goals associated with a desired end system state. They represent high-level concepts in the UbiAgent's brain subordinating user interface components to adapt their behavior to achieve goals as quickly as possible. UbiAgents work towards their goals by carrying out tasks or *Intentions* using actuators in the real and virtual world. The currently executed tasks are constantly reevaluated to verify whether they are efficiently advancing the system towards the end state. The system may reconsider its decisions in case of inadequate progress, and kill suboptimal tasks while starting new, more promising ones.

According to Georgeff et al. [31] adaptive, goal-oriented systems offer a superior performance compared to task-oriented systems in dynamic environments requiring automatic recovery from erroneous situations. In task-oriented systems each task strives to achieve a local optimum without remembering the purpose of its execution. In ubiquitous AR environments, where failures and suboptimal working conditions are inevitable due to the simultaneous use of multiple interconnected hardware and software components, a flexible and adaptive software architecture is needed to effectively tackle issues such as computer crashes, load balancing, and resource discovery.

In UbiAgents, goals represent a combination of desired application and agent states, for instance "the repairman presents the operation of the robot's light sensor". This high-level goal is decomposed into subgoals or plans equivalent to application attribute changes and animated agent action sequences such as "proceeding the maintenance application to the step where the light sensor is activated on the real robot, and superimposing sensor measurement values over the real sensor while the agent explains the sensor's operation".

Plans are converted into concrete agent tasks that are executed by actuators available in the current agent environment. In AR actuators can be physical as well as virtual. Typical examples for virtual actuators include animation engines controlling 3D models and virtual characters, 2D text messages, sound players, and text-to-speech engines. Common physical actuators involve stationary and mobile computers with limited resources such as CPU speed and memory size, fixed and portable displays with a predefined size and resolution, audio speakers, and electric motors and control systems of mechanical engines.

Before task execution, the UbiAgent framework checks whether actuators required for the next task are available at the current agent platform. For instance, the light sensor scenario requires that the agent has access to an infrared communication port to exchange messages with the physical robot to let the repairman agent turn on the real light sensor during explanation.

This step also requires a PC with a fast CPU and a large display for 3D visualization purposes, and a 6DOF tracking system to facilitate the correct overlay of virtual information on top of the real world. If the desired sensors and actuators are missing or fail to meet the minimum requirements of the current agent context, the UbiAgent consults its beliefs (persistent knowledge) in the database about all available agent platforms, and looks for a more suitable environment to migrate to and complete its current job.

4.1.5 Autonomic and Proactive Behavior

As AR systems grow more complex, we need to reduce their perceived complexity to sustain usability by automating certain system tasks. Autonomic computers in IBM's system vision paper [37] mimic the human body's nervous system by taking over low-level maintenance jobs such as self-monitoring, self-healing and self-configuring without constant human guidance. The UbiAgent framework contains an autonomic "caretaker" component dedicated to low-level tasks such as resource management of agent platforms, garbage collection and integrity checks in the database, and repairing broken connections between UbiAgent components.

While autonomic systems are sufficient for typical office automation tasks, current trends in the embedded computer industry's growth rate and the accelerated expansion of the application domain of Ubicomp systems suggest that today's AR systems need to feature characteristics of proactive computing [96]. Autonomic systems deal with challenges of complexity but rely on a predefined finite set of system components enumerating all problematic situations and appropriate system reactions.

To effectively exploit novel resources offered by Ubicomp environments, AR systems need to dynamically reconfigure themselves to incorporate multiple stationary and portable devices, distributed software applications, and a heterogeneous network of sensors [41]. Due to the large number of possible system components, this reconfiguration task would be overwhelming to human users, which calls for a high-level supervisory role instead of a constantly alert manager.

UbiAgents constantly monitor relevant attributes of the real and virtual environment, and store their observations in the database. This persistent knowledge empowers UbiAgents to proactively make decisions about their own reconfiguration, leaving only a supervisory job to the user to intervene in case undesired agent actions are announced. If autonomic systems can be compared to the human nervous system, the organic counterpart of proactive systems would be symbiotic life-forms searching for a host they can exploit to live, while offering services for the host's benefit.

4.2 UbiAgent Components

In this section we present the components of the UbiAgent framework and explain their functions with references to the design principles described in the previous section. Figure 4.3a shows a diagram with all UbiAgent entities and their relationships.

Each UbiAgent consists of an *agent brain* and one or more *agent bodies*. The agent brain serves as a control logic and reasoning engine controlling global agent behavior. The agent body is a local representation of the agent brain and an embodiment of the agent. As UbiAgents operate in AR environments, they are allowed to possess real as well as virtual bodies, and thus appear to be integral parts of the user's physical surroundings. The agent body contains sensors observing the agent's real and virtual environment, and actuators affecting the physical and virtual world.

UbiAgents do not exist on their own. Referring to the aforementioned symbiotic comparison, they need a host environment to "live" in. In our framework we call this host environment a *habitat*, which is characterized by its *hospitability* attribute. The hospitability attribute symbolizes the amount of "nutrients" available for digital symbionts such as a UbiAgent, and thus combines diverse hardware parameters into a single value to describe the computational power of potential host machines. The single hospitability value hides irrelevant internal technical details of diverse computing platforms such as CPU load, available memory, or remaining battery power, yet allows agents to identify erroneous situations such as a crashing or overloaded device by their low hospitability value. Poor habitat conditions endangering agent operation trigger survival agent behavior that usually forces migration to another habitat. Graceful degradation of agent services is also possible as the availability of resources declines.

Computing devices serving as agent habitats in AR environments must provide one or more displays and a network of tracking systems called a locale to support the superimposition of virtual information over the real world. Some parameters such as display size or the degrees of freedom of tracking data rarely change, therefore they are stored in a hardware repository. Repository information is manually updated by a technician when a new display device or tracking system is installed.

Habitats continuously update their current attributes in a habitat information storage including pointers to applications they are hosting, references to agents currently embedded in the applications, and information about currently associated locales and displays to provide a quick overview of the present habitats' hardware and software infrastructure. Invariant display and locale parameters are loaded from the repository, while dynamic parameters

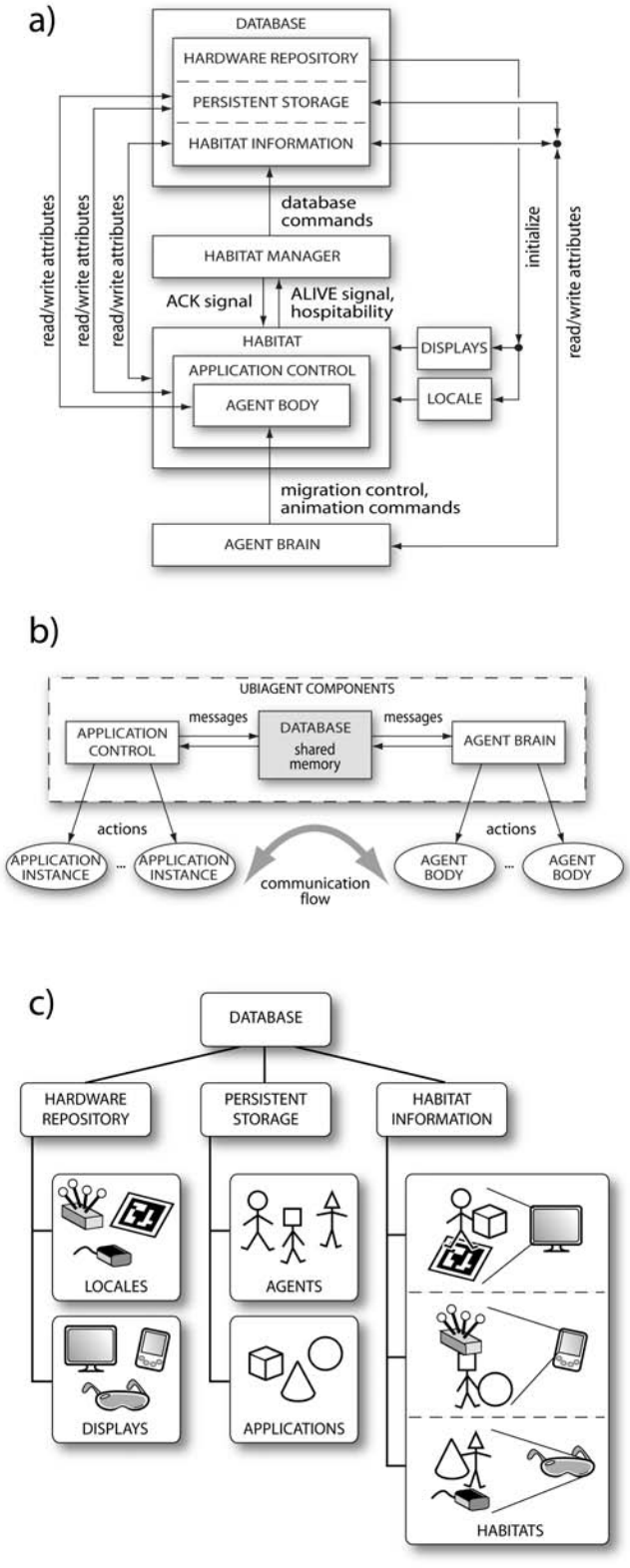


Figure 4.3: a) Entities and their relationships in the UbiAgent framework, b) Shared memory area between agents and applications, c) Structure of the UbiAgent database

such as screen resolution are updated by the respective embedded display or locale component.

As occasional interruptions in network connectivity and power failures of computing devices are inevitable, habitats need a specific component that detects their temporary inability to communicate with UbiAgents despite their presence in the agents' world model. This component is called the habitat manager, which implements the "autonomic caretaker" functionalities described in Section 4.1.5. To prevent UbiAgents from attempting to communicate with a non-responsive agent platform, the habitat manager removes habitats from the world model unless they periodically send an "alive" signal. The habitats keep pinging the habitat manager until it replies with an "acknowledge" message, which reassures faultless communication with UbiAgents until the next periodic "alive" signal is due.

4.2.1 Shared Agent and Application Memory

The dynamic nature of AR environments makes constant and reliable communication between framework objects crucial. Thus the database component plays an eminent role for UbiAgents. The database not only stores the hardware repository, current habitat information, and persistent UbiAgent component attributes, but also serves as a shared memory area between agents and applications (see Figure 4.3b). Components can register observers in the database, which are requests for notifications about changes in certain elements. When a particular application or agent writes a message into the shared memory (namely it adds or updates an element in the database), all applications and agents having registered an observer for that particular element receive a notification about the update. This mechanism makes the database an effective communication medium in our ubiquitous AR framework.

Distributed AR applications consist of several application instances controlled by a dedicated master or application control. The application control maintains the global application state and distributes updates to all instances. Similarly, UbiAgents possess a single brain that controls multiple bodies embedded into individual application instances. Application instances offer only an incomplete local view of the global application state for agent bodies, while agent bodies serve only as limited proxies of the agent brain for application instances. To overcome these limitations, communication between agents and applications happens at a higher level: the application control and the agent brain exchange messages through the database, controlling actions of application instances and agent bodies.

The application control maps internal application state to public database

elements. The mapping function implements the schema concept described in Section 4.1.2. The application control creates a transparent interface between multiple agents and applications to hide private implementation details. This interface enables already existing complex AR systems to exploit agent services without any modifications in structure and code. By employing multiple application controls, different schemas can be supported, allowing a versatile use of the application in various agent systems.

The brain implementation of our UbiAgents in the demo applications is also currently based on C++ code, however, dynamic scripting approaches for scene graphs such as Pivy [73] enable the dynamic uploading of procedural code, making the use of agents more flexible.

4.2.2 Agent Migration

We use the Muddleware real-time XML database [35] to implement the UbiAgents' knowledge base and shared application and agent memory. Muddleware provides fast and robust access to the UbiAgent database, which has a well-defined hierarchical structure (see Figure 4.3c). Please refer to Appendix A and Section 6 for a detailed description (internal structure, DTD and sample XML database content) of the UbiAgent database format.

The Muddleware technology uses XPath-based database queries and observers. The XPath language syntax well suits the hierarchical structure of the UbiAgent database and enables the use of complex queries and observers to receive information about UbiAgent components. With XPath expressions agents can quickly identify habitats matching a set of infrastructure requirements such as hospitability, display parameters, tracking data, and application and agent attributes.

The agent brain controls a finite state machine. Each state defines a set of desired actions for agent bodies. When entering an agent state, an observer is registered to represent minimum expectations about the ideal environment for the agent bodies' actions. If the observer reports the appearance of a more suitable habitat, the agent brain instructs the currently embedded agent body to migrate to the new location.

Migration can happen in two ways: either by serialization techniques of distributed shared scene graphs [86] to create a new agent body, or by activating already existing "sleeping" agent bodies while deactivating previous ones. We also created a GUI-based UbiAgent browser to issue custom queries for debugging purposes and to trigger forced agent migrations for simulations.

The last and current chapter have discussed design principles and requirements for an animation framework supporting the creation of ubiquitous augmented reality agents, their easy addition to AR applications, and their

migration between AR spaces. The next chapter shows how to realize design concepts by presenting implementation details for the AR Puppet and UbiAgent framework components, the Muddleware-based shared memory mechanism, the decision-making procedure for migration between habitats, and the scripting interface.

Chapter 5

Applications

This chapter introduces various AR agent-enabled applications with rich interactive multimedia content exploiting the AR Puppet and UbiAgent frameworks. All applications emphasize our most important design principle: there should be no seams in the visual representation and interaction between agents and agent-enabled applications while keeping the agent and application code implementation independent from each other. This means that we avoid modifying the internal application structure and code by relying on a well-defined communication interface between agents and applications. This interface allows agents to closely monitor the applications' internal state, which enables proper and timely agent reactions to application events and thus yields the impression of a close integration at code level and the agents' awareness of current application state. The following applications will be presented in detail:

- *AR Lego:* This application presents a machine assembly and maintenance scenario, where two AR Puppet-based agents assist an untrained user to assemble and maintain a LEGO Mindstorms® robot. The agents autonomously assess the physical and semantic status of the robot construction procedure and generate appropriate animated instructions.

- *Monkeybridge:* This application describes an AR game in which two users play against each other by building a bridge consisting of real and virtual elements for their monster-like avatars across a virtual ocean. The shape and spatial distribution of the bridge elements influence the decisions of the AR Puppet-based behavior engine that autonomously generates actions for character motion and affective behavior without explicit scripting or other user instructions.

- *Virtual Tour Guide:* This application demonstrates integration with the APRIL AR authoring framework [47] to simplify the authoring process. A virtual human agent is controlled by AR Puppet to serve as a tour guide for an indoor AR navigation system. The agent is turned into an APRIL component that communicates with the large and complex navigation application solely through dedicated input and output attributes. The AR agent-enabled application is automatically generated from an XML-based APRIL description.

- *Character Animation Studio:* This UbiAgent-based scenario creates a spatially and cognitively continuous workspace in a character animation studio by employing a pose-tracked PDA as a tangible data transfer device. The system automatically recognizes which animator workstation to activate based on the current application context derived from the PDA's current location and minimum hardware and software requirements set by the current character configuration and manipulation stage.

- *Ubiquitous Technician:* In this scenario the UbiAgent framework seamlessly combines several previously independent AR applications into a single ubiquitous application that assists a technician in locating and executing typical location-dependent installation and configuration tasks. The set of participating AR applications can be dynamically extended by providing an attribute schema mapping internal application state to standard attributes monitored by UbiAgent components.

5.1 AR Lego

The AR Lego application implements the AR Puppet framework's example scenario described in Section 3.3.1. This scenario demonstrates how to fully utilize real world features with the help of autonomous agents. AR Lego implements a machine assembly and maintenance scenario, in which two agents are employed to educate an untrained user to assemble, test and maintain machines composed of active (engines and sensors) and passive (cogwheels, gears, frames) parts.

The two agents are a real LEGO Mindstorms® robot and a virtual animated repairman. The PC-based application communicates with the robot via an infrared channel, through which it sends commands to control the attributes of the active parts (e.g. engine voltage and direction, sensor configuration) and queries the current robot state (e.g. sensor values, communication channel failures, or battery level).

Figure 5.1: Work environment in AR Lego

5.1.1 Application Scenario

Upon starting the application, the user has a set of real LEGO® building blocks (see Figure 5.1). The system provides step-by-step assembly instructions as to which block to mount next and how to verify whether the user is at the correct stage in the construction.

The verification of passive parts (i.e. inactive bricks) is only possible by visually comparing the appearance of the physical model to the virtual. We have built an accurate virtual 3D model of the full robot using a CAD editor [62] and superimpose it over the real robot. It is possible to switch between a full model representation allowing a careful comparison with the physical model or a reduced representation showing only the next brick to be mounted to prevent obstructing the view of the real parts during construction.

Although more complex, testing whether the active parts (engines and sensors) have been mounted correctly is a more straightforward task. After mounting an engine the application instructs the robot by an infrared command to turn the engine on. If mounted in the right position and correct direction, the engine and all moving parts connected to it should behave as demonstrated by overlaid animated virtual models. Similarly, if we mounted the sensors properly, the right type and range of data should arrive from the robot. The system checks and visually reports inconsistencies so that the user can go back one or more steps to double-check the construction.

5.1.2 Agent-Application Communication

Figure 5.2 provides an overview of relevant application output attributes that agents monitor to update their world model estimating the application state, and input attributes that agents control to interact with the application and provide feedback. The assembly application's interface object provides the agent with all information required to describe the next building block to be mounted. It outputs the current construction step so that the agent is aware of the user's progress, i.e. what has been and needs to be constructed. Then the appropriate LEGO block gets generated, which is then linked to the agent's hands.

If the size parameter of the block is below a certain threshold, the object is linked to and carried in one hand, otherwise it is held in both hands because the object has been perceived as "heavy". The position of the next block allows the movement from the agent's current location to the tile's target location to be planned by a path planning algorithm to avoid bumping into the already constructed model (see Figure 5.3). The block's suggested orientation instructs the agent to put the block into the correct pose before mounting.

Finally, the virtual block is added to the real robot using either a default gesture or special gesture sequences (e.g. turn around, push in, twist, etc.) if an animation hint is provided. There are steps that do not include any LEGO blocks, e.g. instructing the user to turn the construction around if the robot's relative orientation hides important details for the next step. Figure 5.4 shows screenshots of typical maintenance steps in the AR Lego application.

5.1.3 LEGO robot agent

Engines and sensors (push, light, rotation, temperature etc.) are important components of the robot since an erroneous connection or misconfiguration would lead to erroneous behavior. It is desirable that users can quickly verify whether they are connected and functioning correctly.

By turning our physical robot into an AR agent, these tests become intuitive. The robot agent is able to monitor the current status of the assembly application and generate actions that are synchronized with the repairman. This means that as soon as the user finishes mounting an engine, the robot attempts to turn the engine on. In case of incorrect behavior, the current or previous construction steps have to be repeated. Sensors are equally important entities. As soon as they are mounted, the user is invited to test them, for example by pushing the push sensor or holding a color plate in front of

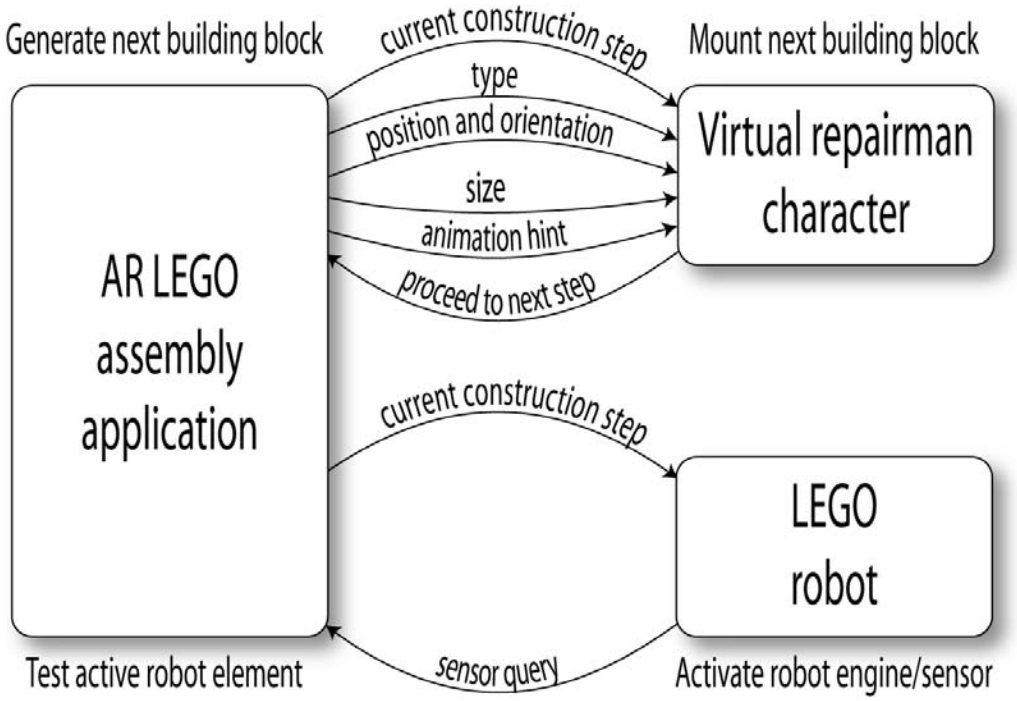


Figure 5.2: Communication between the agents and the AR Lego maintenance application

the light sensor while watching the results of the sensor query commands.

LEGO-like tangible interfaces for real-time interaction have already inspired researchers [87]. Using a LEGO Mindstorms robot appears to be an appropriate choice to build and test machine prototypes for the following reasons:

- It is easy to build and export a precise virtual counterpart of new robot models using a free LEGO CAD editor [62], therefore there is a smooth modeling pipeline.
- The LEGO Mindstorms kit enables application developers to build cost-effective and desktop-sized prototypes for real life, factory-sized machines. By extending our approach it becomes possible to create application scenarios where technicians of factory machinery receive instant help in the form of visual instructions superimposed on top of a real machine that maintaints a bidirectional communication channel with a control computer containing a user's manual.
- A "central brain" unit called RCX is able communicate with a PC run-

Figure 5.3: Path planning by the virtual repairman to avoid collision with real robot (the computed path is symbolized by connected orange line segments)

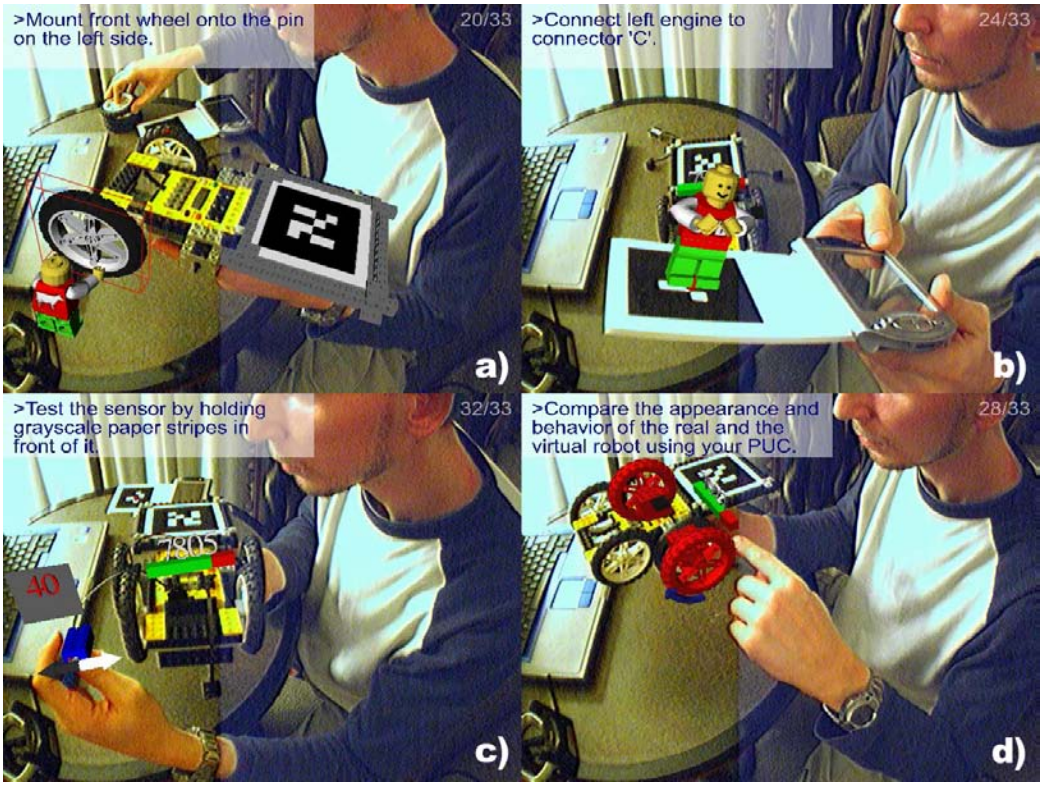


Figure 5.4: Screenshots of typical maintenance steps in AR Lego: a) The virtual repairman agent explains where and how to mount the next robot part, b) The agent instructs to connect a robot engine and a sensor, c), Visualization of live sensor readings from the physical robot's light sensors, d) Using the robot's augmented representation synchronized with the real representation to explain the robot's internal structure

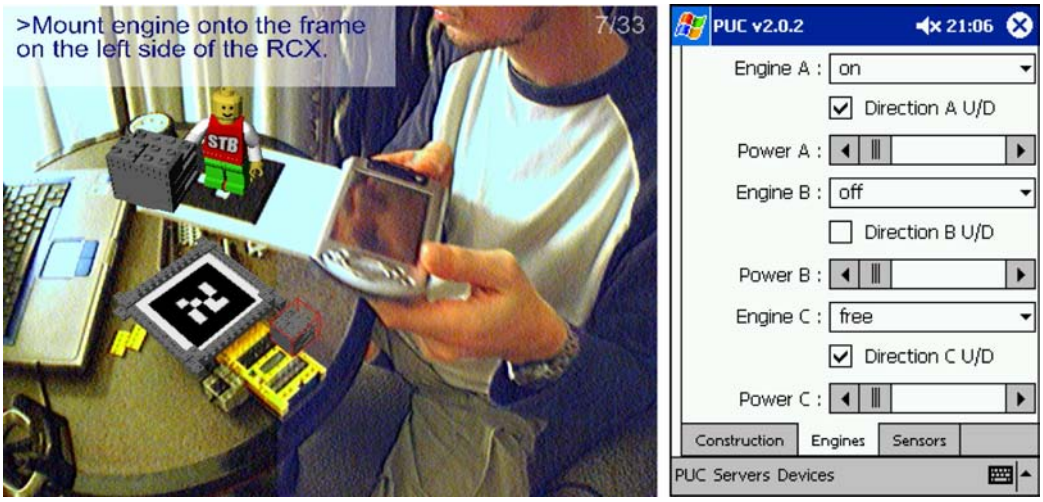


Figure 5.5: Tracked PocketPC as a multi-purpose interaction device: (left) Tangible interface to manipulate the virtual puppeteer, (right) Displaying a control GUI for the LEGO robot using the PUC technology

ning the AR Puppet framework with infrared commands using a communication tower, which makes controlling and monitoring straightforward. The RCX can set and query engine parameters such as on/off state, voltage and direction, configure and query various sensors, and send status information such as acknowledgement of a finished command, battery power and other useful data.

- LEGO is fun to work with and is familiar to many people.

To correctly overlay virtual information, we need to track the current construction model's pose in the physical environment. Mounting the tracking marker on the RCX is an obvious choice since this is the basic building block present in all interactive robots. It contains important control buttons and an LCD display, which should not be covered by tiles, so there is always free space for the mounting of markers, which are integrated into the LEGO bricks.

5.1.4 Interaction

The user interacts with the assembly application and the LEGO robot not only by carrying out assembly and maintenance steps but also by using a tracked PDA, which acts as a multi-purpose interaction device (see Figure 5.5). The PDA runs a client relying on the Personal Universal Controller

technology explained in Section 7.3.1. Thus the PDA not only serves as a tangible, physical interface to move the virtual repairman and virtual LEGO tiles around in the physical environment but it also renders a PUC-based graphical user interface (GUI) on the PDA screen, which can be used to move between the assembly steps and control the LEGO robot's engines and sensors at run-time. The PDA has a wireless network card for cable-free interaction and a tracking marker structure mounted on the back of the case.

We exploit the fact that the application is aware of the current pose of the PocketPC to create the following gestures:

- The virtual repairman holds the next block in the construction sequence while standing on the local user's PDA. The user can freely observe it from all angles by moving the device around.
- As soon as the user places the PDA near the real robot, the character moves from the PDA's local coordinate system to the physical LEGO robot's local coordinate system and starts explaining where and how to place the LEGO tile.
- After the explanation is finished, the user can pick up the character again with the PDA, which binds it to the PDA's coordinate system and commands it to display the next building block.

5.2 Monkeybridge

Monkeybridge is a multiplayer game, where users place real and virtual objects onto a physical surface, thus influencing the behavior of virtual animated characters (see concept image in Figure 5.6). During the development of the game we have been examining the use of embodied autonomous agents in AR gaming. We are interested in how "smart" software and hardware components influence multiplayer game experience.

5.2.1 Motivation of AR Gaming

A key element in games and stories is fantasy. While playing, a virtual, imaginary world is created within our mind, inhabited by characters obeying our imagination. In classic make-believe games this fantasy world and their characters connect to the real environment through physical game props to which various roles are assigned, thus making heretofore passive objects active players in the game story. AR applications are aiming at achieving the same effect by superimposing a virtual world on top of the real environment.

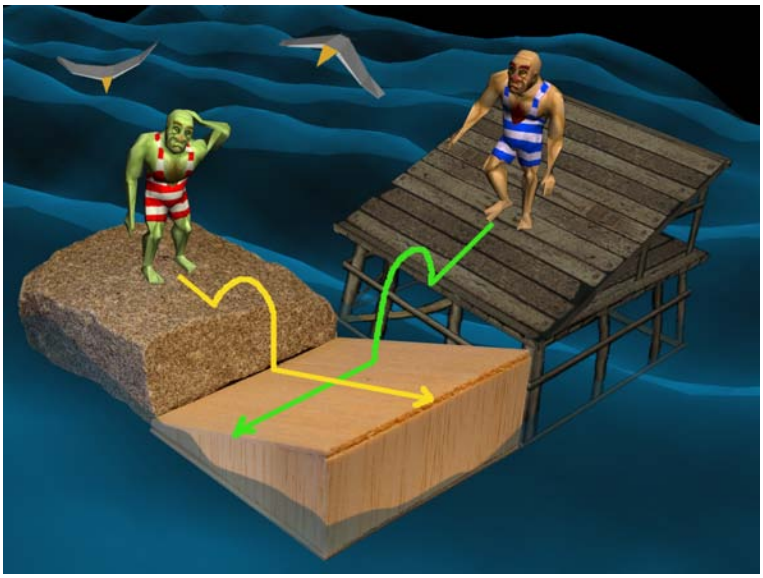


Figure 5.6: Concept image of the Monkeybridge game: the two monster-like characters autonomously follow the terrain defined by real and virtual building blocks

As pointed out by Stapleton and colleagues in their mixed-fantasy framework [92], the combination of the real and virtual help suspend disbelief and enrich the audience's fantasy experience. AR is able to visually change real world attributes, make passive objects appear animated and play sound effects besides sounds in the real environment to further enhance the atmosphere of the perceived mixed environment. In AR games physical game props appear to coexist with virtual game objects, enabling the creation of rich and atmospheric environments leveraging infinitely detailed and realistic physical game elements and interactive virtual objects.

In collaborative virtual game environments co-players are often represented by avatars. The behavior of the avatars is constrained and controlled by the game logic, which maps player actions to the set of capabilities and behavior elements of the avatars. In addition to players there is often a large variety of non-player characters serving as allies, bystanders or competitors. As games become increasingly complex, it is desirable that some workload is taken off from both game developers and players by adding autonomy to player and non-player characters.

An autonomous character does not need constant user guidance or thoroughly scripted behavior prepared for all possible situations. Instead, it proactively makes decisions based on events coming from sensors present in its environment. Thus only high-level goals are needed to be set, while

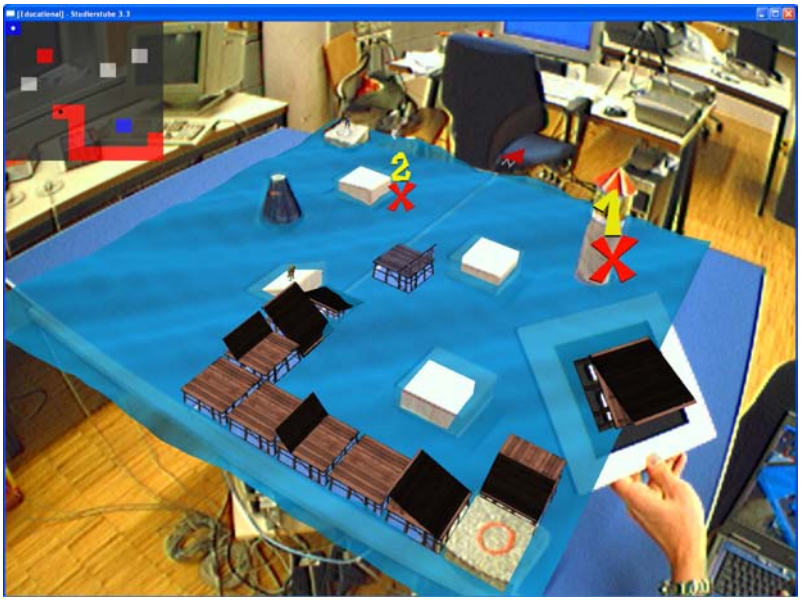


Figure 5.7: Building a bridge across a virtual ocean in Monkeybridge

the character's reasoning engine takes care of low-level details to achieve the goals as quickly as possible. Our Monkeybridge game employs such a character that appears to have a "brain" behind its movements, which is implemented by embodied autonomous agents.

5.2.2 Application Scenario

A "monkey bridge" is a fragile wooden construction over a river in South-East Asia [28]. People frequently risk their lives as they try to keep their balance crossing to the other side. In this game players dynamically build a monkey bridge for their own monster-like characters using virtual and physical pieces of landing stage, which vary in shape. The goal is to reach a dedicated target in a virtual ocean.

Figure 5.7 and 5.10 provide screenshots of typical game scenes from the users' viewpoint. In both images a bridge has already been built for a character, whereas the bridge consists of virtual blocks (models with the dark wooden texture) and physical tiles (bright balsa-wood and stone cubes showing through the virtual objects). A user is holding the next building block in his hand.

Players do not have direct influence on the game characters' behavior; instead they indirectly control character movement by providing the agents

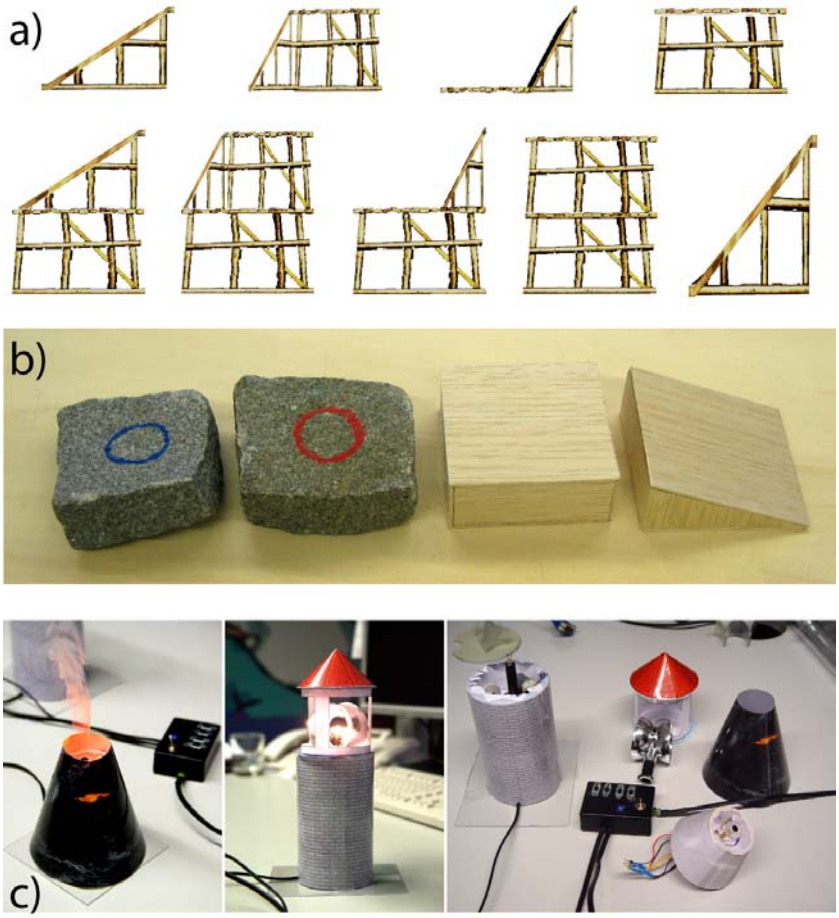


Figure 5.8: Virtual and physical building blocks in Monkeybridge: a) Virtual bridge elements, b) Physical bridge elements, c) Physical obstacles

with building blocks to walk on above the virtual ocean. In a typical setup the ocean is divided into 10x10 cells yielding a 1m x 1m rectangular physical game board, however, the grid and cell size can be customized. Each cell may host a building block functioning either as a bridge element or an obstacle. All possible building blocks can be seen in Figures 5.8a-c. Bridge elements are either physical or virtual and are composed of simple geometrical shapes that fit together smoothly, while obstacles are physical objects that serve as strategic hindrances to players as well as decoration elements.

The virtual bridge elements are auto-generated in a similar way to Tetris games and are dynamically laid onto the game board by the players. The position and orientation of the blocks are automatically snapped to the cells to make positioning easier. If left unmoved above an unoccupied cell for a

given time, the block becomes fixed, and the player occupies the given cell. A player is allowed to place a tile only into cells that are adjacent to other cells already occupied by the player.

The real building blocks are crafted from stone and wood. Unlike their virtual counterparts, the position and orientation of the physical blocks are fixed during the game, although they can be arbitrarily configured before start-up. The physical blocks represent the start and target platforms for the characters as well as strategic points to reach.

We have built two physical obstacles: a lighthouse with rotating spotlights and a volcano puffing real, illuminated smoke. The fun factor of seeing the volcano puffing smoke or the rotating lights of the lighthouse motivates players to lead the path of the monsters towards these objects, imposing influence on play strategy. The casing of the obstacles can be quickly assembled from paper templates which then host electric parts such as engines, LEDs and a smoke generator. Our test setup lets users manually control electric parts functions using a custom-made control box.

5.2.3 Autonomous Game Characters

The monster-like characters are embodied autonomous agents and therefore their behavior does not require careful and detailed scripting. Instead a dedicated control logic or virtual "brain" decides which animations and sound effects to play, which direction to turn or whether the target has been reached. The only factors that directly influence agent behavior are the spatial distribution, pose and shape of the virtual and physical building blocks placed on the game board. Figure 5.9a and b provide illustration.

The characters autonomously choose: the path they walk on; decide how to get from one platform to the other, e.g. jump up or down when there is a slight difference in height between platform edges; automatically choose the straightest path from several available tiles; and fall into the water if there is no suitable piece of landing stage to walk on. They happily cheer with their hands up when they win, and cry over a lost game to add affective content to the game.

5.2.4 Domains of Game Experience

Recent researcher papers on AR gaming [22] [57] [68] have established the convention of discussing how game elements contribute to user stimuli in individual domains of game experience, therefore we also provide our own summary.

- *Mental domain:* We attempt to create the illusion of a real marine landscape by using various visual and aural elements. Visual and audio effects help suspend disbelief, therefore enhance the mental image created by the players' own fantasy. In our game a flock of animated virtual seagulls fly around squawking while realistically animated 3D ocean waves boom below. The seagulls' movement is controlled by a boid algorithm based on the work of Reynolds [83]. The physical bridge blocks are made of stone and wood while their virtual counterparts use similar textures. Additional real world decoration elements such as our lighthouse and volcano offer unlimited opportunities to create a powerful game atmosphere. If registered correctly, physical objects appear to be washed by synthetic ocean waves, further blurring the boundary between the real and virtual.

- *Emotional domain:* The emotional experience is predominantly delivered by the animated monster agents. Animations and sound effects imply that the monsters possess a lazy and dull yet likeable personality. Since virtual characters situated in AR environments give the impression of possessing a real, tangible body that is part of the player's own environment, characters appear more lifelike. In addition, the almighty nature of the character control deciding about life and death attaches players more emotionally to the creatures they are responsible for.

- *Physical domain:* As previously discussed, physical activity greatly enhances the playful nature of games. Playing with Monkeybridge also involves a considerable amount of body movement because users constantly have to find the best cell to place the next building block while preventing the opponents from reaching their target first. A Head-Up Display (HUD) provides an overview about the currently occupied cells allowing strategic plans. Time pressure causes tension in the game that comes from the suicidal attitude of the characters. Similar to the virtual rodents of the famous Lemmings game [97], the monster agents keep walking forward, even if there is no bridge platform ahead to step on.

- *Social domain:* This domain is strongly connected to the physical domain. In AR opponents are natural parts of the environment preserving such important communicational cues as body and facial gestures. While the players compete against one another to occupy cells, they unwittingly block the other users' hand, camera etc., which may result in debates or jokes.

5.2.5 Game Setups

Similarly to all applications presented in this section, our game is grounded in the *Studierstube* AR platform, the modular structure of which enables experiments with several game setups using various tracking systems, displays and interaction devices. The game can be configured to be a distributed application or to run on a single computer. Before deciding for a setup, financial and technical factors such as cost, installation time and calibration efforts of the tracking system and display need to be considered.

We realized three demo setups to test Monkeybridge. The first and simplest one runs on a single computer without video background, and uses a simple keyboard-based tracking simulator. This is not an AR application, only VR, serving as a simulator for evaluating usability and development purposes. The second prototype relies on a multi-user setup with two computers sharing application data and tracking information provided by the ARToolKit optical marker recognition system. This setup requires two computers equipped with webcams and three optical markers: a large calibration marker to register the physical game board with the virtual game environment, and two small markers acting as user interaction devices to place the bridge blocks. The live video stream recorded by the cameras is augmented on the user's computer screen residing next to the physical game board. Although this setup is highly portable, requires simple calibration, lacks cables and can built at low cost, it has significant drawbacks such as inferior tracking quality severely affected by lighting conditions and occlusion, camera distortion and static viewpoint. The reason why the cameras have to remain fixed after an initial calibration phase is the fact that we place several props onto the game board at start-up which would cause the tracking algorithm to lose the game board calibration marker during recognition.

The third setup uses the Flock of Birds magnetic tracking system from Ascension to track two Sony Glasstron optical see-through HMDs and two Plexiglas pucs to place the virtual tiles. This setup requires a specially manufactured table lacking any metal parts (screws, bolts etc.) to avoid distortion of the magnetic field. As tracking quality is superior to the previous setup and the HMDs provide dynamical viewpoint change, users have a strong sense of presence. The tracking system and the HMDs need precise calibration every time the game is installed in a new location.

Despite the superior visual quality the magnetic setup offers, it is heavily tethered, which might come as a nuisance to some players. This problem could be overcome by a fourth setup variant which we haven't implemented yet: a natural feature tracking-based solution using a handheld tablet PC serving as a window onto the augmented world.

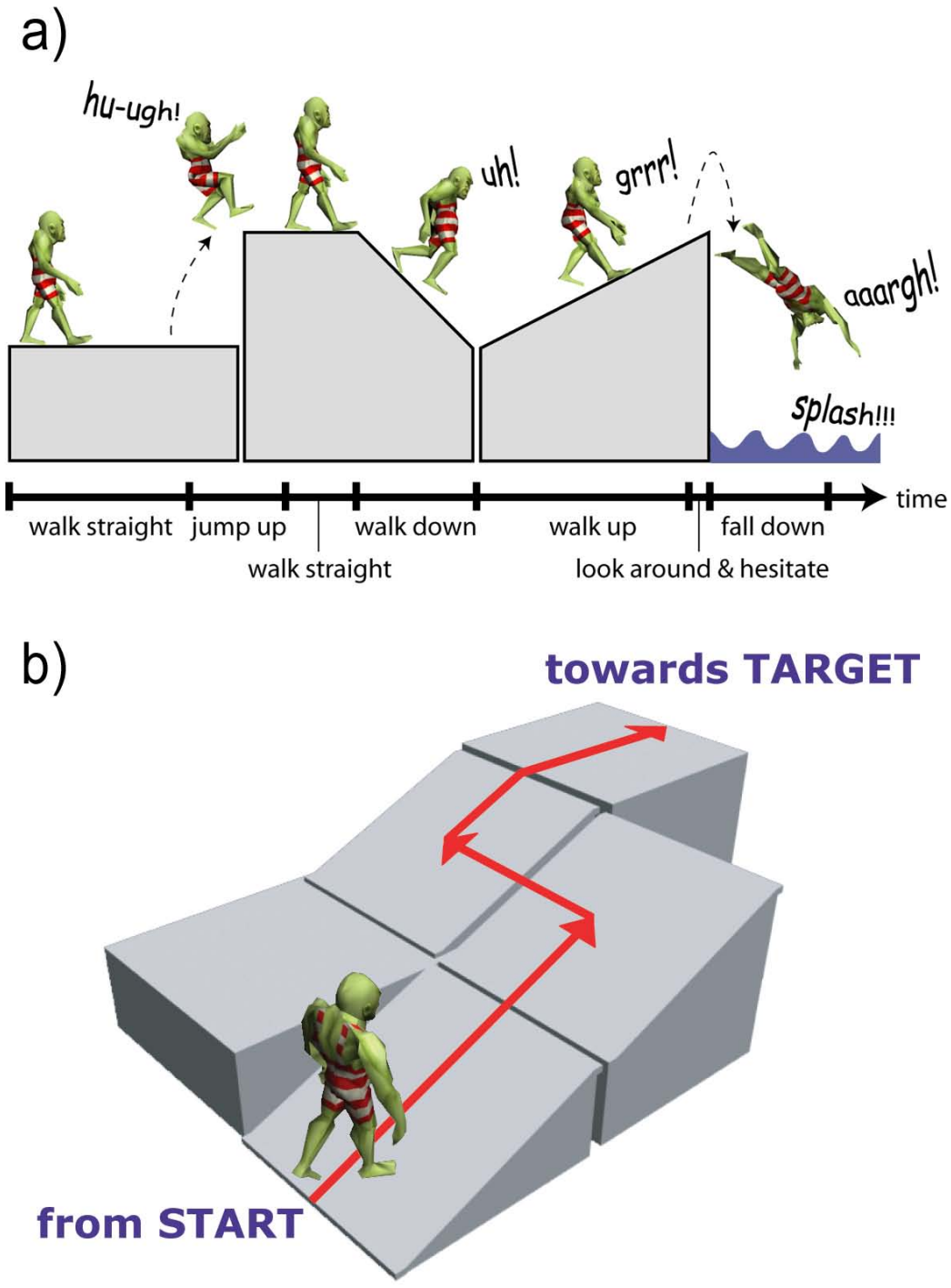


Figure 5.9: Autonomous agent behavior: a) Motion planning: choosing animation and sound based on platform type, b) Path planning depending on the spatial distribution of available blocks

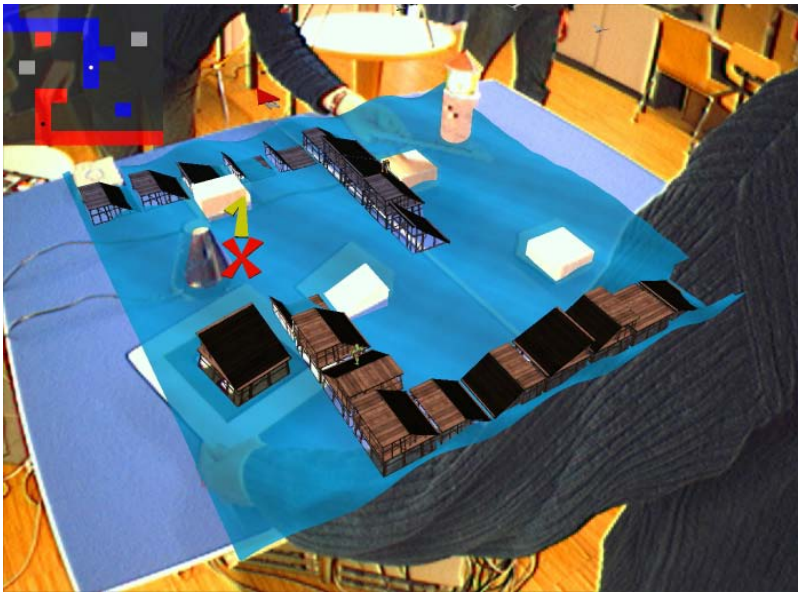


Figure 5.10: Optical marker tracking-based dual-user game setup

Figure 5.11: Magnetic tracker and HMD-based dual-user game setup

5.3 Virtual Tour Guide

This application demonstrates how AR Puppet-based applications can be authored by users without an in-depth knowledge of how to program our AR framework. We encapsulate framework components as building blocks within the APRIL framework (see Section 7.2 and [47]) that can be controlled by a state machine constructed from a UML-based storyboard in APRIL's authoring pipeline. Firstly we describe the application to suggest its complexity, then demonstrate how to hide internal implementation details and let authors focus on high-level application functionality.

5.3.1 Application description

The Virtual Tour Guide embeds a virtual human acting as a tour guide into a mobile indoor navigation application called Signpost [40]. The user wears a mobile AR setup (see Figure 5.12a,b) and perceives the augmented world through an HMD. A camera mounted above the HMD tracks fiducials placed onto walls of the building area covered by the application (see Figure 5.12c). The markers help locate the user within this area since the system knows their exact position in a precisely measured virtual model of the building that has been registered with its real counterpart.

The virtual tour guide character is placed into the reference frame of the real building. While walking around, the character provides assistance to find selected destinations and provides location-specific explanation about the content of various rooms and people working in them using body gestures (e.g. looking towards, pointing, asking the user to follow, etc.), 2D and 3D visual elements and sound. Since the tour guide is aware of the building geometry, it appears to walk up real stairs and go through real doors and walkways, thus further enhancing integrity with the user's physical environment.

5.3.2 Integration with the APRIL Framework

The tour guide is represented by a virtual animated character, which is controlled by the AR Puppet framework to execute various animation sequences depending on and parameterized by the user's current location within the physical environment. Both AR Puppet and the Signpost navigation application rely on the APRIL framework's component model by turning AR Puppet and Signpost into custom APRIL components, enabling the encapsulation of these frameworks' high-level functionality. These components can

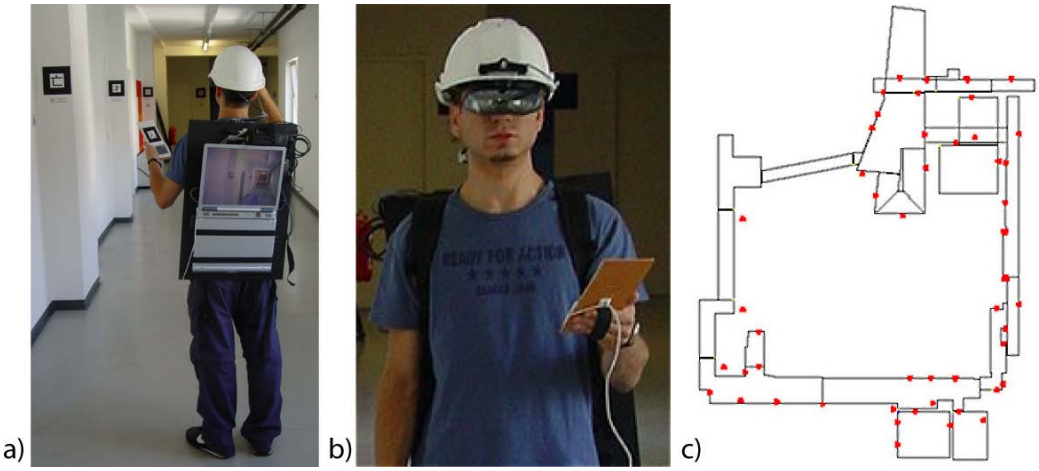


Figure 5.12: Signpost user wearing a backpack-based mobile AR system: a) Back view, b) Front view, c) Floor plan of the building where the user is guided around, fiducial markers are marked with red triangles (All images courtesy of Vienna University of Technology)

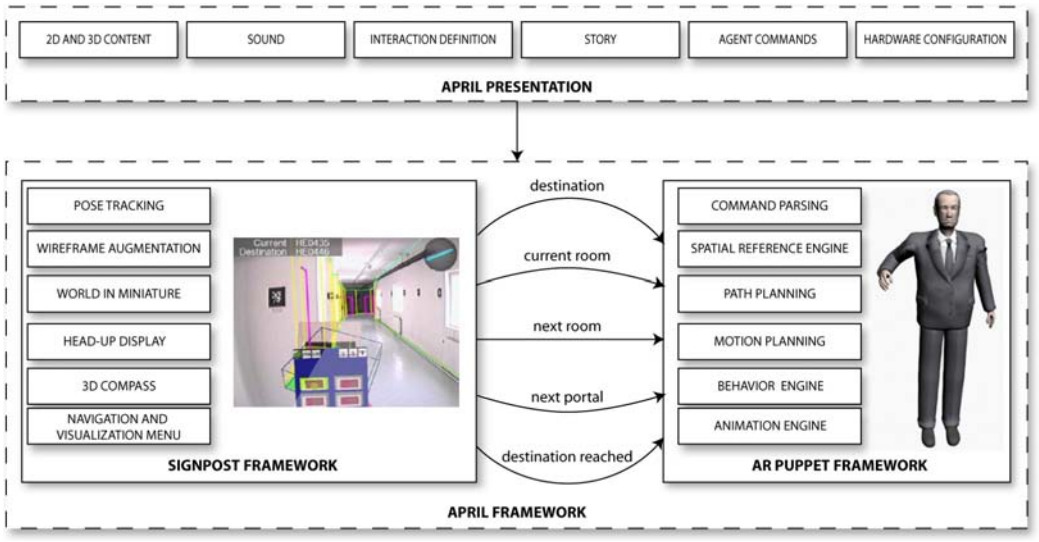


Figure 5.13: Communication between the AR Puppet and Signpost systems within the APRIL framework

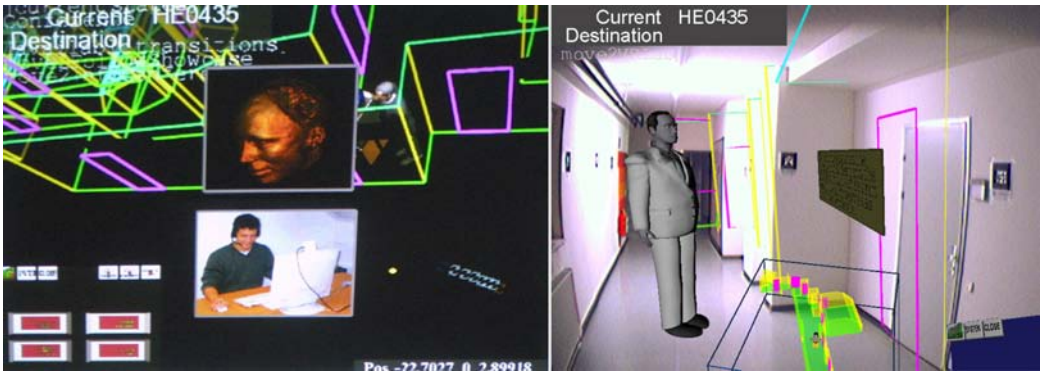


Figure 5.14: a) The Virtual Tour Guide application running on the desktop developer setup, b) An application view captured from the HMD of a user wearing the mobile AR backpack system. In both setups a world-in-miniature view of the building model is shown and location-dependent HUD overlay graphics is presented to the user as she roams the building

be used as black boxes by content authors that expose relevant input and output fields for communication with other, external components while hiding internal implementation details. Figure 5.13 illustrates the fields exposed by Signpost and monitored by AR Puppet to provide the tour guide character with relevant navigation information.

The indoor AR navigation system exposes location-based attributes to the virtual tour guide agent. The application outputs the selected destination of the user, the current and next room in the route suggested by the system, the next portal to go through and a flag indicating whether the user has reached its target location. By watching these attributes the virtual tour guide can deliver location-based descriptions about the current room and useful navigation information. Using hand and head gestures it is able to show the right direction, point out locations of interest in the building, is able to warn users when a door is approaching, and send a notification when the destination has been reached.

The APRIL authoring pipeline is described in detail in Section 7.2, we only briefly refer to individual stages now. The tour itself is modeled by the APRIL storyboard as a UML state engine (a small part of the complete state engine is shown in Figure 7.3 later. Individual stations of the guided tour are modeled as states, triggering linear presentations when the user arrives. The structure of the building and the different modes for the guided tour (linear or free mode) are modeled by transitions and superstates.

In each state (i.e. specific station of the guided tour) agent commands are issued by setting the AR Puppet component's *command* attribute to a

specific command string. These agent commands trigger appropriate animation sequences for the virtual human agent, therefore the agent appears to be aware of the user's location and surrounding environment. Transitions between states are triggered by changes in the Signpost component's output attributes such as the *position* attribute marking absolute coordinates of the user location within the building. If the state engine detects that the *position* coordinates penetrate a given bounding box marking a physical hotspot in the building, an event is generated that triggers a transition from the current presentation state to a predefined target state, again initiating associated agent actions.

5.3.3 Hardware Setups

The expensive and bulky mobile AR system required by the Signpost application in its original form makes content and application authoring, debugging and testing a difficult task, therefore we needed to develop a desktop simulator system that is able to run the same navigation application with simple keyboard input and screen-based output. The hardware abstraction feature of APRIL conveniently hides details such as the type of display or exact tracking setup from authors and components. Only symbolic names are used that allow exchanging the internal implementation of the hardware setup. When generating the executable Studierstube application, the desired hardware configuration can be chosen from multiple configuration files as long as they consistently use the same symbolic names when referring to trackers, displays, or input devices. With this modular approach the same AR application can be run on multiple hardware setups with minimal effort and no code modification. See Figure 5.14 for an illustration of the Virtual Tour Guide application running on the desktop simulator and the mobile AR system.

5.4 Character Animation Studio

The Character Animation Studio scenario illustrates how the UbiAgent framework can enhance distributed AR applications with migratable user interface elements that automatically recognize the current application context and proactively migrate to application instances best matching requirements dictated by the current context. In this scenario the distributed application is a character animation program and the migratable user interface element is an animated character following users around when switching between networked application instances located at diverse physical locations. Context

information is determined by current user pose and minimum hardware requirements for possible agent platforms such as CPU power of stationary machines and battery level for mobile devices.

5.4.1 Application Scenario

Character animation projects in big studios rely on the collaborative work of multiple people: modelers design the mesh, animators and programmers create expressive behavior, and producers supervise all stages. The production pipeline typically requires the simultaneous use of multiple computing environments. Artists prefer to work on their personal computers, while programmers need to test characters in the target production environment such as a game console, and producers report current progress to customers in the presentation room. UbiAgent-enabled characters decrease the seam between workspaces by proactively migrating to presentation environments demanded by the current animation pipeline stage (see Figure 5.15).

In the design stage, a PDA is used as a tangible transfer medium for characters. The PDA is pose-tracked by an ARToolKit-based fiducial marker and webcams mounted on the designer PCs. If the mesh designer wants to discuss potential modifications with the animator, he/she holds the PDA in front of the webcam on the PC monitor, which indicates an intention to "pick up" the character. The character senses the PDA's spatial vicinity and "jumps over" to the handheld agent platform, which is then carried to the animator's machine. There the character migrates again to the monitor if the PDA enters a predefined "hot" area around the webcam. Changes made to the character on the animator machine are persistently stored in the UbiAgent database, therefore the next time the character is transferred back to the modeler PC, its appearance is automatically updated to reflect changes.

In the presentation stage the character is taken to the presentation room's projection screen, where an Ascension Flock of Birds magnetic tracking system is installed. By mounting a magnetic receiver on the back of the PDA and penetrating a predefined presentation area around the projection screen, the character moves from a small, private handheld display to a large public screen to show its features to a larger audience.

The continuity of the visual interface creates a spatially continuous workspace for the collaborators and thus improves productivity. To avoid discontinuities in the interface and the interaction metaphor, the characters constantly check the availability of target environments and only attempt migration if the target platforms appear to be present and "hospitable". This includes for instance the periodic checking of the handheld device's battery

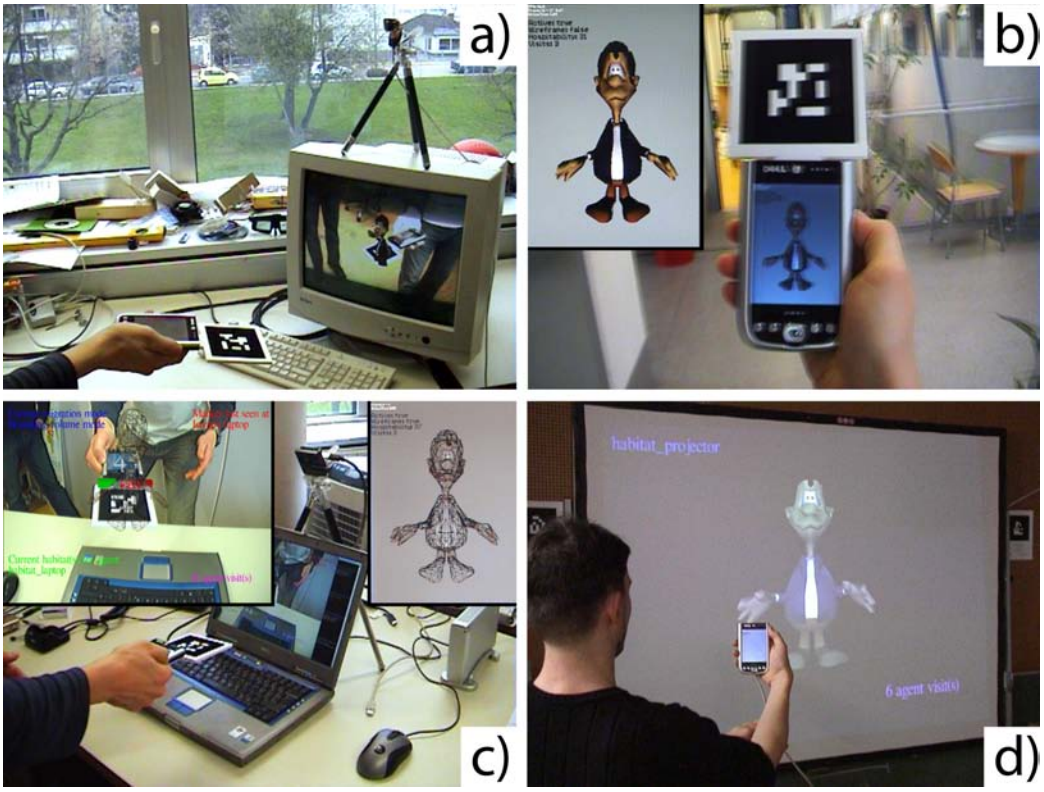


Figure 5.15: Enhancing the character animation pipeline with UbiAgents: a) Picking up character by PDA, b) Tangible character transfer, c) Persistent agent parameter: wireframe mode preserved on PC and PDA, d) Sending character to projection screen

level as a critical resource. If the battery level is too low, the character refuses to be picked up by the PDA, or escapes to the nearest available display.

5.4.2 Required UbiAgent Components

To support the aforementioned scenario, character animation software packages need to be extended with UbiAgent components without modifying often proprietary internal software structure. Although our test scenario currently relies on our custom AR application framework, popular commercial animation packages can also be enhanced by special plug-ins. Firstly, each character animation application instance is encapsulated by a UbiAgent application instance object. This object runs independently from the application but continuously monitors its internal state and maps the observed state information to external parameters such as the character's rendering

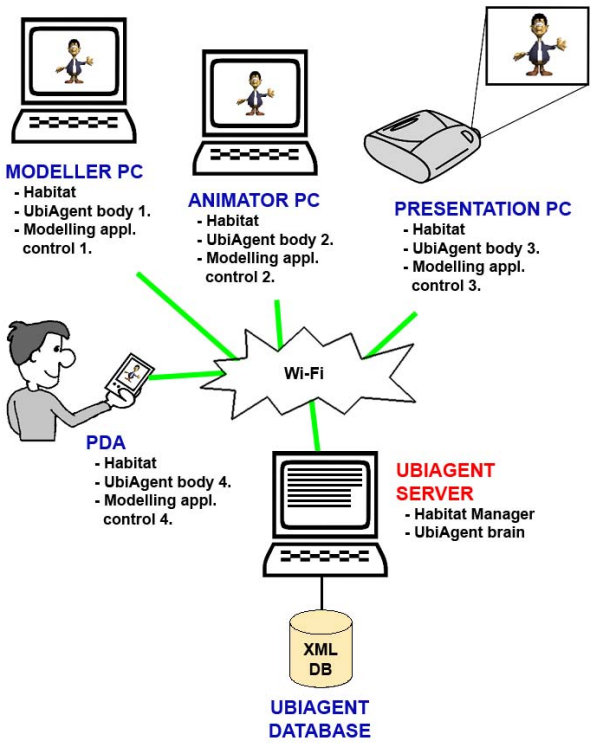


Figure 5.16: UbiAgent components in the Character Animation Studio

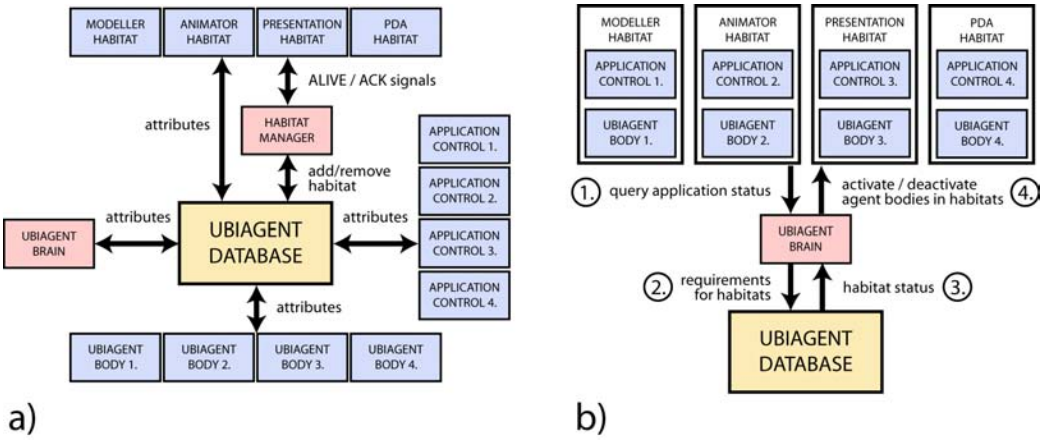


Figure 5.17: Communication scheme between UbiAgent components: a) Exchanging messages between components using the shared memory area, b) Communication flow among UbiAgent components during a migration process

mode, pose, scale, current animation sequence and level-of-detail.

The external parameters are defined by a character animation attribute schema understood by a dedicated application control logic generating commands for UbiAgent-enabled animation tools. All tools using this schema (even if they internally rely on previously unknown, exotic software packages) can dynamically join the virtual UbiAgent workspace without revealing their low-level details to the character control, and thus serve as shared rendering and manipulation resources that animators can exploit.

Besides the addition of the application control and instance objects, the characters themselves are represented by UbiAgent bodies controlled by the agent brain component. The agent bodies are rendered by the animation tools using the distributed character and rendering parameters retrieved from the shared database by the UbiAgent animation control. At the same time the brain component monitors the PDA's pose, and activates or deactivates agent bodies if a user penetrates a display's hotspot area and the "hospitability" parameter of the corresponding habitat is above a predefined threshold.

As Figure 5.16 shows, the modeler and animator PC, the PDA, and the presentation machine are all habitats expecting UbiAgents by running an application to render and tweak animated 3D characters. All habitats contain an instance of the distributed character animation application and a UbiAgent body represented by the character manipulated in the application instance. The habitats are added to and removed from the database by the Habitat Manager component running on a dedicated server machine. This component periodically pings all habitats to report error-free operation and communication, and removes unresponsive habitats.

Figure 5.17 illustrates the communication scheme between UbiAgent components using the shared agent and application memory area represented by a database. The left image shows the message types exchanged between the database and the components. The right image indicates the communication flow while the UbiAgent brain migrates an agent body from one habitat to another. Firstly, the current application status is queried by the UbiAgent brain. Based on this status information the UbiAgent brain sets specific hardware and software requirements, which are then transformed into habitat selection filters represented by database search criteria. By regularly querying the database with the given criteria, the UbiAgent brain gets an overview of the capabilities of all habitats where UbiAgent bodies may reside. Based on the current habitat requirements the UbiAgent brain activates one or more UbiAgent bodies within new habitats while deactivating bodies in previously active but no longer suitable habitats.

The agent brain, the XML database, and the habitat manager run on a dedicated control PC. All components communicate via WLAN using

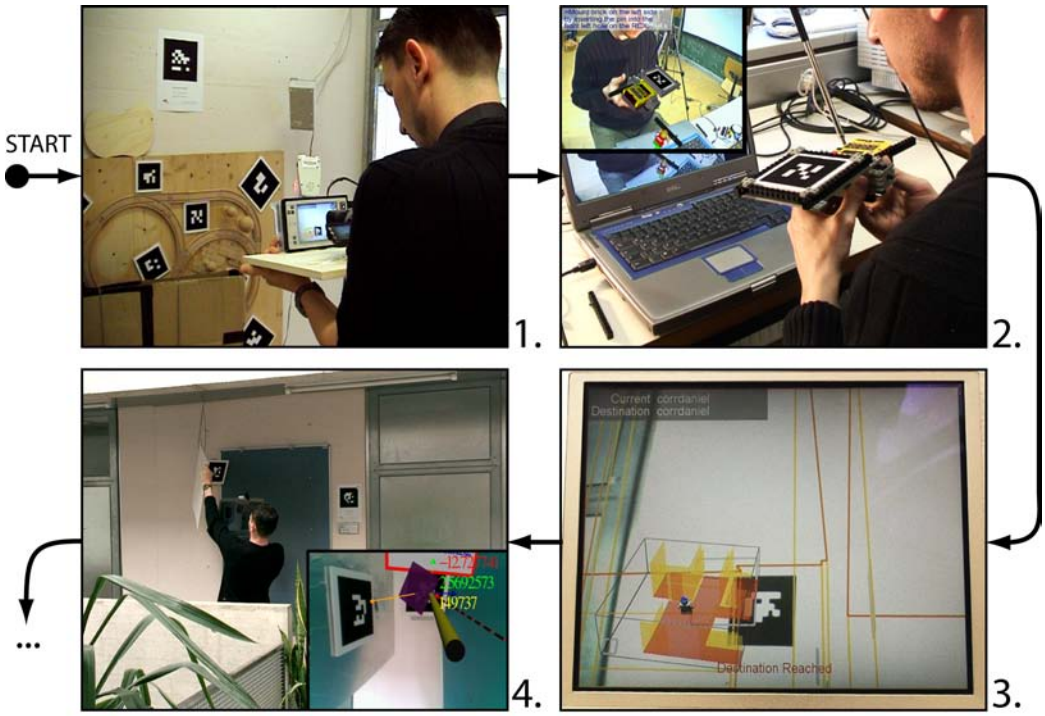


Figure 5.18: Screenshots from the Ubiquitous Technician application. This application seamlessly combines an indoor AR navigation system (a,c), an ultra-wideband calibration aid (b), and a machine maintenance application (d)

TCP/IP and UDP messages. The handheld agent platform is installed on a Dell Axim X51 PDA with hardware accelerated graphics, which runs Daniel Wagner's Klimt and FPK libraries [35] to render an animated 3D character. The PC-based characters are based on the Cal3D skeleton-based character animation library [18].

5.5 Ubiquitous Technician

The Ubiquitous Technician scenario demonstrates a technique how multiple AR applications can be seamlessly combined to form a spatially distributed "smart" environment using the UbiAgent framework. Any application complying to a predefined attribute mapping scheme can be dynamically incorporated into this environment as its internal status is represented in a way that can be "understood" by UbiAgents, enabling them to react to application status changes accordingly. Firstly the application scenario is described, which is followed by details of the attribute mapping.

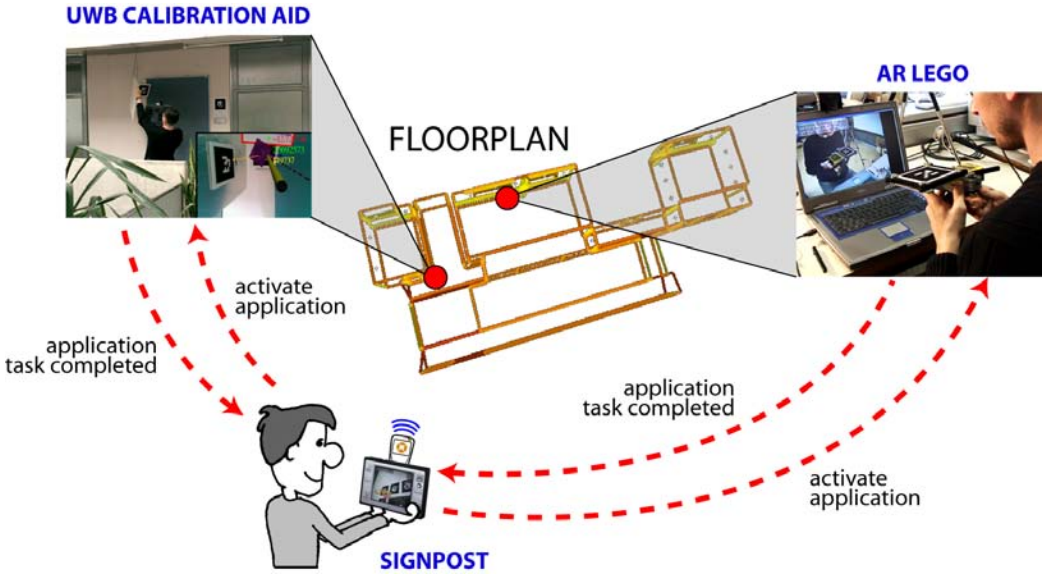


Figure 5.19: Inter-application communication flow

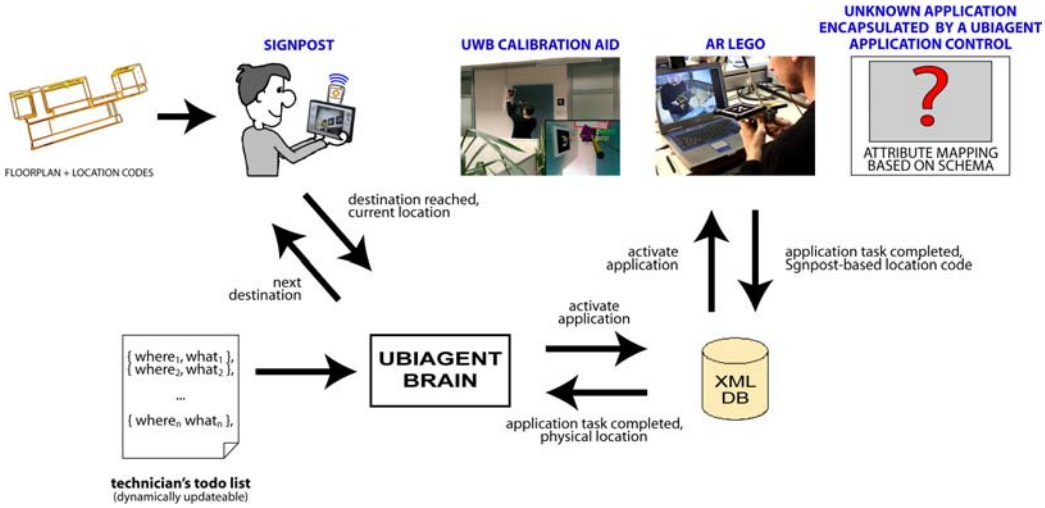


Figure 5.20: Communication flow between UbiAgent components

5.5.1 Application Scenario

Technicians are a scarce resource in every research lab and company. Their never-ending to-do list is constantly extended with requests to fix malfunctioning computers, calibrate tracking systems, and maintain copy machines, all located in different offices. The Ubiquitous Technician application provides assistance for our research lab's technician to complete various maintenance tasks on his to-do list. In this scenario we intentionally reuse elements from previous research demonstrations to test the encapsulation and communication of complex external AR applications, and to create richer content for our demo application.

The technician agent's brain creates a control loop to systematically go through the technician's to-do items (see Figure 5.18). As each task is located at a different place in our building, the loop first activates a modified version of the aforementioned Signpost AR indoor navigation system (see Section 5.3) to guide the technician to the next task's location. The navigation application runs on a Sony VAIO U70 portable computer equipped with a webcam tracking fiducial markers on office walls and corridors, a UbiSense ultra-wideband (UWB) position tracker, and an IntertiaCube3 inertial sensor to calculate the technician's current pose inside the building. The application displays a virtual compass suggesting the direction the technician should follow to reach the target.

This complex AR navigation system is encapsulated by an application controller object exposing only three attributes: current destination, current location, and a flag indicating whether the current destination has been reached. After arriving at the target location, the agent migrates to the AR application associated with the current maintenance task.

The first job is to calibrate a cell of a UWB tracking system. An AR application [63] visualizes angle-of-arrival sensor measurements by virtual rays emanating from the physical sensors, and helps overcome problematic situations such as multipath signals caused by reflections from metal ceilings and doors. When erroneous measurements are detected, the technician mounts a marker on a nearby surface suspected to cause the reflections. The marker calculates the surface's relative pose to the UWB sensors, enabling the agent to suggest a virtual baffle to block out unwanted signals. When the technician has finished the calibration procedure, he returns to the indoor navigation guide, which has already received a message with the next destination from the UbiAgent brain. Again, the technician is guided to the next task, which is the aforementioned LEGO maintenance scenario (described in Section 5.1).

5.5.2 Attribute Schema and Communication Flow

As Figure 5.19 and Figure 5.20 illustrate, the Ubiquitous Technician scenario includes three previously independent AR applications that communicate with one another using the XML database as a shared memory area. New applications can be dynamically added to the to-do list by encapsulating them with an appropriate application control. The schema of the application control must contain a Signpost-compatible description of the application's location in the building, a trigger to activate the application when the technician is nearby, and a flag indicating that the application task has been completed, which instructs the agent brain to proceed to the next to-do item.

The UbiAgent brain maintains a to-do list that can be extended dynamically at run-time. Each list item contains information about the location and name of a single AR application the technician can complete a task with. When the task finishes, the agent brain gets notified and proceeds to the next to-do list item. The item's associated location code is sent to the Signpost navigation system and becomes the current navigation destination. Then Signpost guides the user to the desired location. When the destination has been reached, Signpost notifies the UbiAgent brain, which activates the application associated with the current task with a trigger message for the application controller. The technician works with the active application until the task is finished. Then the agent brain gets notified again to select the next task, and the Ubiquitous Technician loop starts again.

Chapter 6

Implementation

In this chapter we describe the technological foundations and implementation details of the AR Puppet and UbiAgent frameworks and how they can enhance Studierstube-based AR applications with embodied autonomous agents.

6.1 Technological Foundations

AR applications based on the AR Puppet and UbiAgent framework assist the manipulation of rich multimedia content with autonomous and proactive software components and high-level user interaction. As described in detail by Reitmayr [78], the implementation of interactive AR applications includes numerous repetitive tasks such as data management, collaboration, 3D presentation, 3D interaction, control GUI, application core, and tracking. Instead of implementing our own modules from scratch, we build our animated agent framework on existing open source software toolkits that enrich our application development process with code reusability and the joint efforts and shared knowledge of online developer communities.

6.1.1 Requirements

We have set the following requirements for toolkits that will form the software base of our agent framework:

- The toolkits support the development of an object-oriented hierarchical animation framework to implement object roles described in Chapter 3 and 4. Framework objects enable the encapsulation of AR applications as black boxes to hide internal implementation details and to offer

a high-level interface for controlling and observing application status through input and output parameters.

- The toolkits provide a rich and extensible set of utility components and functions for high-level access of toolkit features while preserving a low-level interface for versatile customization.
- External toolkits can be easily integrated to extend framework functionality such as a character animation library enabling multiple character formats, a speech recognition and synthesis module for multimodal agent-human communication, or a database manager facilitating persistent agent memory.
- The toolkits support collaborative multi-user applications where status information is distributed over the network between application instances.
- Quick prototyping and rapid application development are supported by providing a scripting interface and a set of reusable configuration files.
- Cross-platform development is allowed so that applications running on multiple computer platforms can be enhanced with agent technologies without significant porting efforts.
- The performance of the generated executable applications allows real-time interaction with complex 3D applications.

This chapter presents the technical details of the toolkits which were used in the implementation of the AR Puppet and UbiAgent frameworks, and illustrates how these toolkits meet the above requirements by describing individual toolkit features exploited by our software frameworks to create agent-enabled AR applications. We selected the following toolkits to meet the aforementioned requirements:

- *Open Inventor:* an open-source high-level 3D graphics toolkit for developing cross-platform interactive 3D visualization and visual simulation applications
- *OpenTracker:* an extensible library solving different tasks involved in tracking input devices and processing tracking data for virtual environments
- *Studierstube:* a collaborative AR framework supporting the rapid development of mobile, collaborative and ubiquitous AR applications

- *Cal3D:* an open-source skeleton-based 3D character animation library written in a platform and graphic API-independent way
- *Muddleware:* a robust, high-performance XML database supporting a large number of clients and queries simultaneously

Figure 6.1 illustrates how agent-enabled AR applications build upon AR Puppet, UbiAgent, and the aforementioned toolkits. Agent-enabled applications built on our framework preserve the structure and implementation of the original core AR applications, while enabling the development of "smart" software components that observe internal application status and proactively generate an appropriate response by relying on AR Puppet and UbiAgent framework services. In our implementation all AR applications are based on the Studierstube collaborative AR framework. Studierstube extends Open Inventor, an object-oriented high-level graphics API, with an extensible set of utility classes supporting the rapid development of multi-user AR applications.

To facilitate collaborative AR applications where application state information is distributed over the network, Studierstube relies on services of the Distributed Open Inventor (DIV) library, which is an extension of Open Inventor supporting scene graphs distributed over the network. The functionality of standard Studierstube objects is further extended with the integration of external toolkits such as the Cal3D character animation library [18] for advanced character manipulation and rendering support, the Muddleware real-time database [35], and other utility libraries such as the Microsoft Speech API framework [60] for speech synthesis and recognition or the FMOD cross-platform audio library [29]. All external libraries are integrated into Studierstube by creating a special Inventor-based wrapper node encapsulating framework functionalities in a standard way that can be understood by all other objects within the scene graph.

An essential part of Studierstube is the handling of real-time 3D events coming from tracking systems and other input devices. A big emphasis has been put on device abstraction, which means that AR applications built on top of Studierstube do not have to be aware of the exact hardware setup including tracking hardware, interaction devices or networking capabilities. Instead, they can refer to these resources by abstract names, allowing application developers and content authors to focus on high-level concepts and functionality. This also increases portability by allowing the same application to run with different hardware configurations without modifying the application itself. Studierstube relies on the OpenTracker framework to gain device abstraction and to receive tracking events from tracking hardware devices in

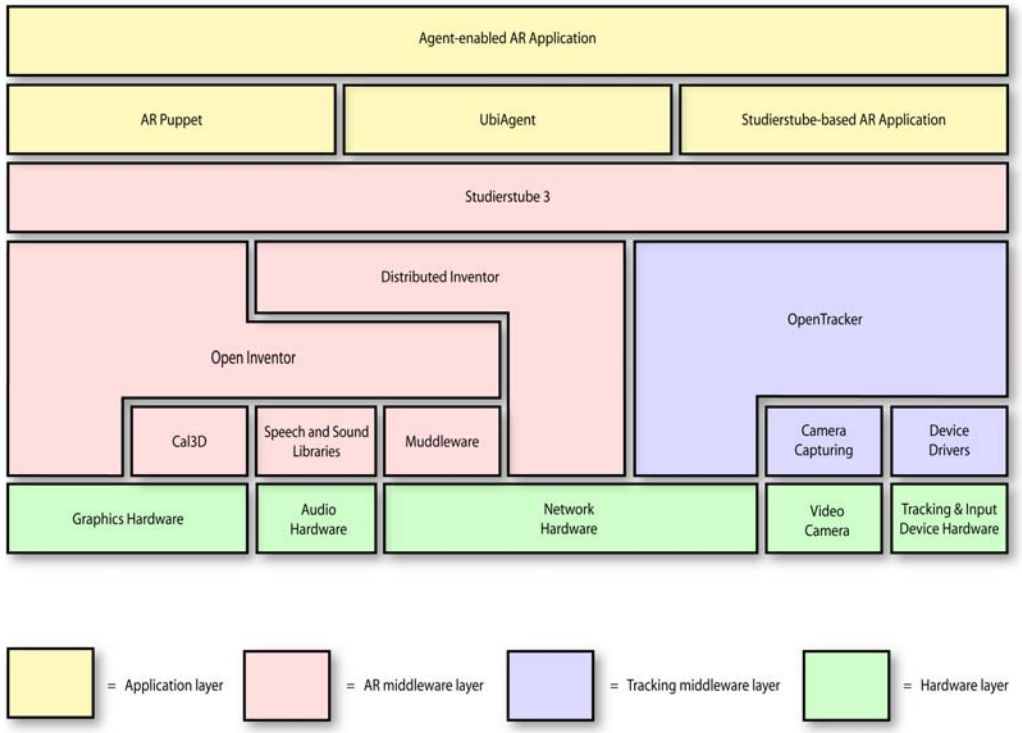


Figure 6.1: Software stack supporting the AR Puppet and UbiAgent frameworks

a standard format understood by dedicated Studierstube components. OpenTracker can be easily extended by new modules handling previously unknown hardware devices by wrapping up associated drivers as OpenTracker components with a standard communication interface.

In the following sections we describe how each software toolkit is exploited by the AR Puppet and UbiAgent frameworks. Instead of providing a detailed summary and an in-depth description of each toolkit, we reference respective literature and focus only on features relevant to AR Puppet and UbiAgent.

6.1.2 Open Inventor

Open Inventor (OIV) is a cross-platform "retained mode" 3D graphics API originally developed by Silicon Graphics Inc. We use Systems in Motion's implementation called Coin3D, which is portable over a wide range of platforms such as UNIX / Linux / *BSD platforms, all Microsoft Windows operating

systems, and Mac OS X. In-depth details about the Coin3D API and Open Inventor concepts are described in the books The Inventor Mentor [106] and The Inventor Toolmaker [107], and Systems in Motion's Coin3D website [23].

Open Inventor follows an object-oriented and structured approach to describing scenes containing 3D objects, common graphical needs of 3D applications and interactions with the 3D content. As described by Strauss and Carey [93], Open Inventor effectively tackles the "duplicate database problem", whereby interactive graphical applications use the same data structure to compute application state and to render the graphical output. This approach saves considerable storage and administrative overhead resulting in increased performance, and greatly simplifies the design of interactive graphical applications. This speed-up for developers, content authors, and users is particularly precious for complex and resource intensive applications such as AR environments.

One of the most important features of Open Inventor is extensibility. The library provides a rich default set of utility classes to create, manipulate and interact with 2D and 3D objects. Users can either add their own application-specific classes and methods or change the appearance and behavior of existing objects by overloading operations such as rendering or geometric computations. Objects are abstract representations of information that can render themselves when requested, influencing the final visual output.

The foundation of Open Inventor's abstract object representation is the scene database. It stores dynamic representations of 3D scenes as directed acyclic graphs (called *scene graphs*) of objects called *nodes*. Various classes of nodes can be used to represent different geometries, rendering properties, and database traversal behaviors. The database provides a set of actions that can be applied to scene graphs or scene graph parts such as actions for rendering, picking, computing a bounding box, event handling, or writing the scene to a file.

The rendering process is a traversal of the scene graph where each node affects the current state of the rendering process, therefore the order in which nodes are traversed has a significant impact on the final visual output. The rendering mechanism inside Open Inventor objects employs OpenGL and thus is highly optimized to take advantage of OpenGL accelerators. During rendering each scene graph object automatically makes proper, efficient calls to OpenGL.

Nodes can store instance-specific information in sub-objects called *fields*. Each node class can define one or more fields, with which specific value types can be associated. Fields can be used for unidirectional or bidirectional communication between nodes. If a field value represents an object status

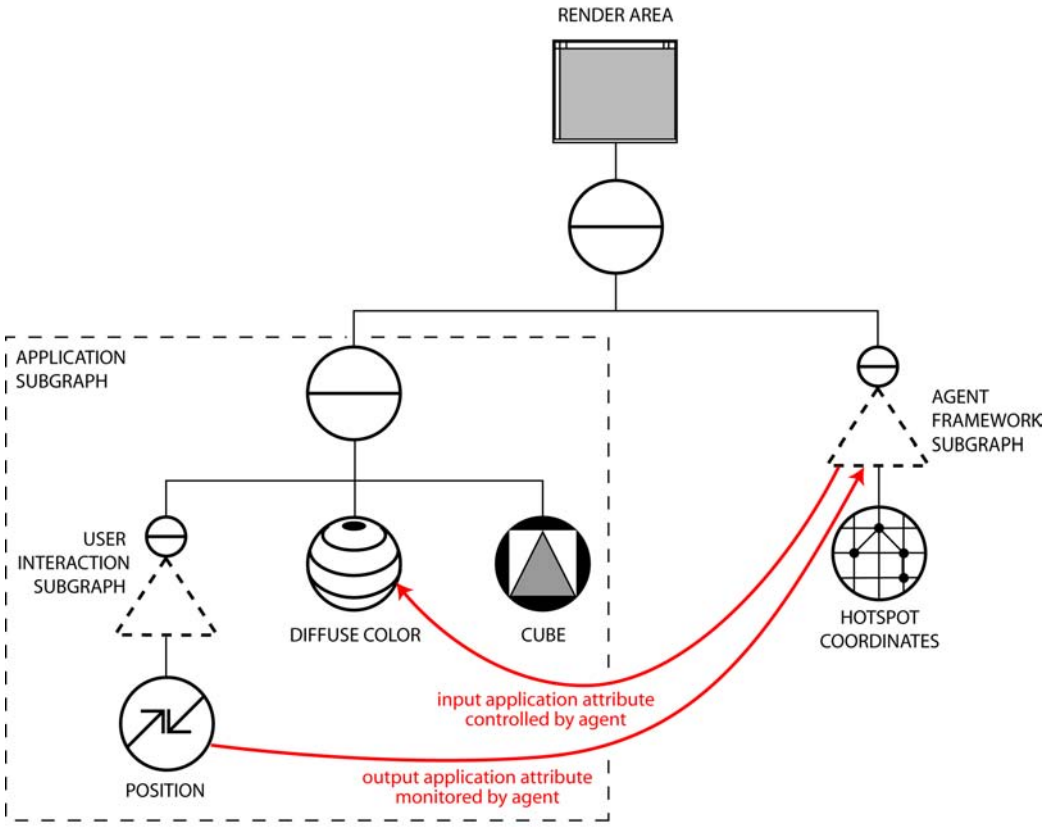


Figure 6.2: Example scenario for non-invasive Open Inventor-based agent-application communication

element and the field is regularly updated to reflect latest status changes, then the field represents an *output* or *status attribute*. If external changes in a field value are monitored by the object and influence internal object state, then the field is an *input* or *control attribute*. If a field is either an input or an output attribute but cannot be both at the same time (e.g. external changes in the field value of an output-only attribute are ignored and overwritten by the object), the field enables only unidirectional communication. In case the field retains both functionalities simultaneously, it supports bidirectional communication between nodes. This can be achieved by one component giving response to another component's stimulus.

Fields of nodes can be connected to receive updates from other fields, and thus forming a data flow network. A special class of objects called *engines*

can be embedded in the data flow graph to process field value changes and compute new updates to other fields. These objects are not part of the scene graph but only of an overlaid field network graph. Components in the AR Puppet and UbiAgent framework are all custom-made Open Inventor scene graph nodes, nodekits (special nodes internally containing a structured subgraph of other nodes), and engines. A few components overload some of the default action handler functions such as rendering or traversal order to implement custom behavior.

We exploit the fact that Studierstube-based AR applications are also built on Open Inventor. Hence AR applications can be easily enhanced by agent services by simply attaching a sub-scene graph containing AR Puppet or UbiAgent components to the application scene graph. This non-invasive approach does not modify the application's internal structure yet allows agents to monitor internal application state by establishing field connections from the AR application's output attributes to dedicated agent sensor fields respectively, and influence the application state by creating field connections from dedicated agent control fields to input application attributes.

In the case of attribute monitoring, unidirectional communication channels are created from the application to the agent without interfering with the original AR application's operation as new outgoing field connections do not break a field's existing outgoing connections. On the other hand, the agent's application control capabilities demand careful attention from the application developer when adding agent control services as incoming field connections break existing incoming field connections and thus alter internal application data flow mechanisms. Since agents do not influence the application state directly, only through dedicated fields and field connections, it does not matter whether the agent sub-scene graph is inserted before or after the application scene graph.

Figure 6.2 shows an example scenario for non-invasive Open Inventor-based agent-application communication. The application is represented by a simplistic sub-scene graph rendering a 3D cube. The color and 3D position of the cube can be changed through configurable parameters. The 3D position is controlled by user interaction nodes within the application, e.g. handling 3D events of a tracked input device. Field sensors in the agent sub-scene graph monitor changes in the cube's position output attribute. If the cube's center penetrates a configurable 3D hotspot area, sensors in the agent framework generate an event that instructs an agent control logic to change the color of the cube through the application's diffuse color input attribute.

As the following sample Inventor script illustrates, the aforementioned simple scenario can be quickly authored using Open Inventor's powerful scripting mechanism to assemble scene graphs and make field connections.

For the user interaction and agent framework subgraph we use pseudocomponents to simplify graph representation and to save space. In the code below we rely on the *SoRoute* Inventor-based utility object included in the Studierstube library (see Section 6.1.4) to make field connections outside nodes. The hashmark character marks comments.

```
...
# application subgraph
Separator {
    # transform node
    DEF TRANSFORMATION Transform {
        # user interaction pseudocomponent
        translation = MyInteractionComponent {
            ...
        }.position
    }
    # object material
    DEF COLORMATERIAL Material {}
    # 3D shape object
    DEF 3DSHAPE Cube {}
}
...
# agent framework subgraph
Separator {
    # agent framework pseudocomponent
    DEF AGENTCOMPONENT MyAgentComponent {
        hotspot Coordinate3 {
            # 3D hotspot = 1 m3 area around origin
            point [ -0.5 -0.5 -0.5, 0.5 0.5 0.5 ]
        }
        # field connection with standard Inventor mechanism
        objectPosition = USE TRANSFORMATION.translation
    }
}
# field connection with the "SoRoute" Studierstube object
SoRoute {
    from "AGENTCOMPONENT.colorControl"
    to "COLORMATERIAL.diffuseColor"
}
...
```

6.1.3 OpenTracker

To appropriately overlay virtual objects on top of the real world, AR applications need to create and maintain an accurate model of the physical environment. This world model is obtained and updated by sensors delivering estimates of physical properties such as pose, velocity, temperature or light. To allow agents to understand and react to various real world events coming from diverse tracking systems, interaction props, measuring instruments, and other input devices, a standard event format needs to be used. Moreover, the exact hardware configuration of event sources has to be abstracted for agents to allow them to work with portable applications that can run on multiple hardware configurations.

The OpenTracker framework [79] is an extensible C++ class library implementing a standard way of processing tracking data for virtual environments. It provides a multi-platform, open software architecture based on a highly modular design and a configuration syntax based on XML. OpenTracker provides device abstraction for applications using tracking services and feed them with real-time tracking events. It can be flexibly extended with new device handling modules by providing a well-defined software interface for previously unknown devices. OpenTracker manages a generic data flow network and describes complex manipulations of data as a series of simple transformations in an object-oriented graph structure.

Figure 6.3 shows the 3D event data flow between OpenTracker and agent-enabled AR applications. Firstly, tracking system components and other input devices deliver device-specific messages to OpenTracker where they are processed by specific handler modules integrating respective device drivers. OpenTracker translates these proprietary messages into a standard 3D event format containing position, orientation, button state, confidence, and time stamp information, and creates a data flow for each individual tracking source structure.

Studierstube employs abstract interface objects called *stations* to get selected OpenTracker data flows into the AR application scene graphs. Stations are identified with non-negative integer numbers that also serve as abstract names for tracking hardware configurations generating data for OpenTracker. Studierstube nodes may directly obtain tracking events consisting of position, orientation and button state information from dedicated stations through a general event handler object or use one of the numerous utility classes providing convenience functions for event handling.

Since the AR Puppet and UbiAgent frameworks are built on top of Studierstube, their components can also register for receiving tracking events from selected stations. Agent control logics then add semantics to standard

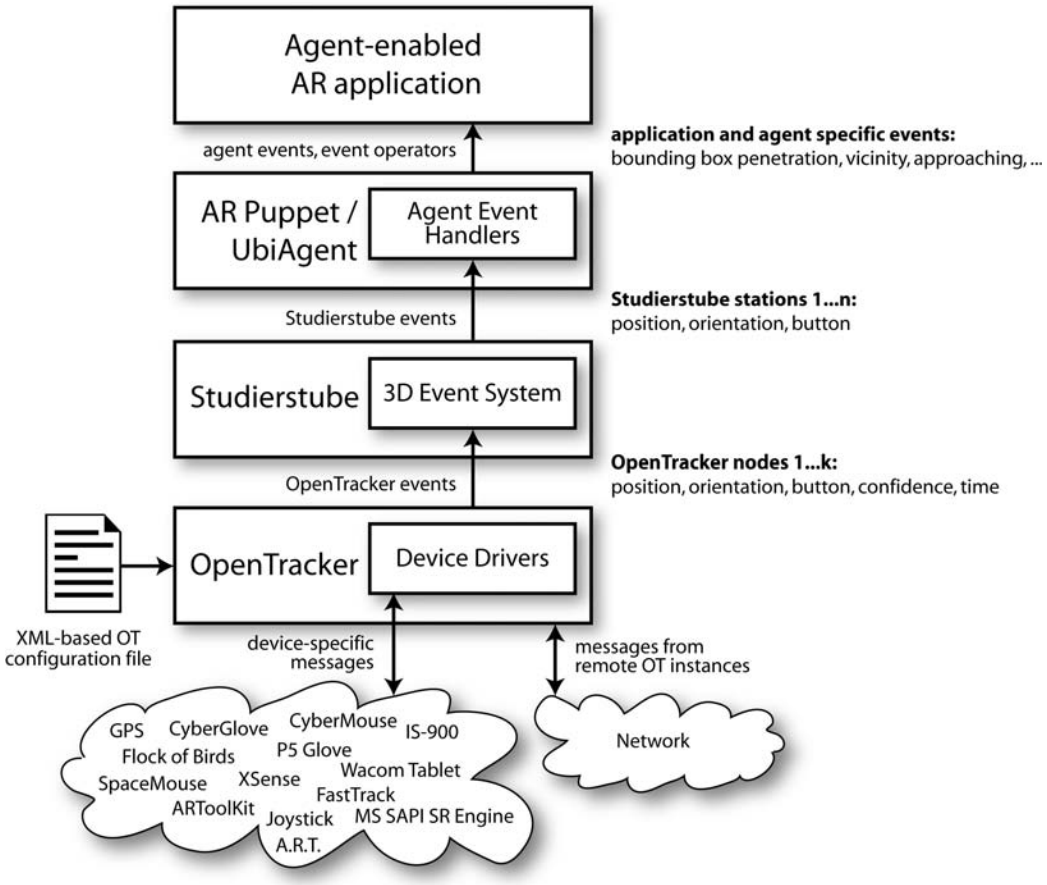


Figure 6.3: Event flow between OpenTracker and agent-enabled applications

Studierstube events by combining events from several sources with unary, binary or tertiary operators, such as an object penetrating a hotspot area around another tracked object or a fixed point in space, objects moving towards each other, objects facing each other, etc.

6.1.4 Studierstube

Studierstube is a framework that extends the Open Inventor library and offers a rich set of reusable building blocks for collaborative AR applications. The powerful scripting mechanism of the underlying Inventor API makes Studierstube highly suitable for prototyping and rapid application development as the building blocks can be quickly assembled and configured to create distributed multi-user applications with advanced interaction capabilities.

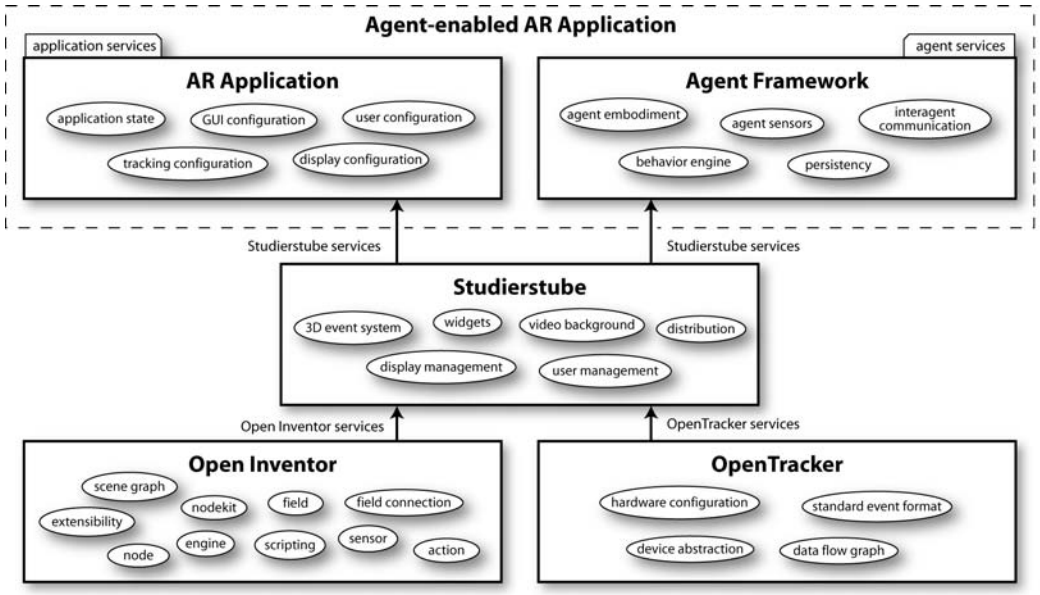


Figure 6.4: The hierarchical structure of software toolkits and services for Studierstube and agent-enabled AR applications

Figure 6.4 shows the toolkits and services Studierstube and Studierstube-based AR applications are based on. Studierstube integrates services from Open Inventor such as the scene graph philosophy, abstract object representation as scene graph nodes, configuration and communication through fields and field connections, and the high degree of toolkit extensibility. Studierstube also closely relies on OpenTracker services such as tracking device abstraction, an interface for obtaining tracking data flows in a standard event format, and easy configuration of underlying hardware structure via flexible and reusable configuration files.

The AR Puppet and UbiAgent frameworks are built on top of Studierstube as an extra software layer offering agent services for Studierstube-based AR applications. To create an agent-enabled AR application, the agent framework components are embedded directly into the AR application, which enables a close integration such as sharing the same rendering area for seamless visual output and receiving the same interaction events for maintaining an accurate world model.

Reitmayr [78] provides an extensive overview of the design principles, building blocks and software services of Studierstube, therefore we only provide a brief description of modules that are relevant to our agent frameworks:

- *3D event handling:* As described in Section 6.1.3, Studierstube receives 3D events from OpenTracker through *stations*, dedicated interface objects, in a standard format containing position, orientation, and button state. Each station corresponds to a data flow from a specific tracking hardware structure configured in an OpenTracker configuration script. Studierstube provides numerous utility objects and functions to obtain 3D events from stations and use them within scene graph objects. Our agent framework components rely on the *Stb3DEventGenerator* interface object, the *SoTrakEngine* engine, and the *SoTrackedArtifactKit* nodekit to receive events from selected stations.

- *Distributed application management:* Studierstube enables the development of collaborative distributed AR applications by using a distributed shared scene graph that follows the semantics of a distributed shared memory. The concept and implementation of distributed scene graphs is built on Open Inventor yielding the Distributed Open Inventor (DIV) library [36]. DIV relies on Inventor's notification mechanism and a reliable multicast protocol to distribute changes. Studierstube's built-in DIV support (e.g. with the dedicated *SoDIVGroup* node) allows the automatic management and synchronization of multiple application instances and master-slave rights.

- *Multi-user management:* Distributed applications with multiple application instances assume the presence of multiple users. User entities and their typical parameters are encapsulated by Studierstube's *SoUserKit* class, the instances of which are managed by the *SoUserManagerKit* object. Users view the augmented world through associated displays, which are represented by the *SoDisplayKit* class. Display objects provide configuration parameters and rendering support for multiple output devices common in VR and AR applications such as computer monitors, stereo optical and video see-through head-mounted displays, projection screens, or virtual workbenches.

- *High-level interaction support:* 3D applications demand 3D graphical user interfaces for interaction. Studierstube uses the Personal Interaction Panel metaphor [95] that employs a two-handed hardware setup consisting of a lightweight tracked handheld panel and a tracked pen. Virtual 2D and 3D interface elements are superimposed over the handheld physical panel, turning it into a tangible 3D menu (handled by *SoPipKit* and *SoPipSheetKit*) that is manipulated by a 3D cursor (handled by *SoPenKit*) controlled by the physical pen. Studierstube provides an extensive 3D widget set for quick GUI development.

- *Utility classes and integrated toolkits:* Studierstube contains a large set of miscellaneous utility classes such as geometric transformations, altering the current rendering state, network communication, or scene graph manipulation functions. Moreover, several external toolkits have been integrated into Studierstube to extend its functionality such as the Cal3D character animation library (see Section 6.1.5), the Muddleware real-time XML database (see Section 6.1.6), the FMOD audio library [29], the Microsoft Speech API [60] for speech recognition and synthesis, and the Personal Universal Controller technology (see Section 7.3.1).

6.1.5 Cal3D

Cal3D [18] is an open source 3D character animation library written in a platform and graphic-API independent way. It contains a set of utility classes to store, manipulate and render animated skeleton-based 3D characters in an animation package-neutral format. It does not handle the actual rendering and the texture-management itself to avoid hard-coded dependencies on libraries and APIs such as OpenGL or DirectX. The library is coded in C++ and only depends on the STL. Cal3D comes with exporter plug-ins for major 3D modeling and animation software packages such as 3D Studio MAX, Maya, and Blender, making Cal3D an attractive toolkit to insert high quality, artistic content into AR applications.

Integration into Studierstube is made by an Open Inventor-based wrapper class that encapsulates Cal3D's functionalities as an Inventor nodekit and offers an input and output field-based interface to monitor and control character appearance and behavior. The nodekit is called *SoCal3DPuppet*, following the naming convention of the AR Puppet framework it was initially created for. Although our agent frameworks support multiple character libraries such as a 3D talking head or MD2-based characters (see Section 6.2.1) for agent embodiments, Cal3D is the most advanced and flexible character animation framework for our purposes.

In the *SoCal3DPuppet* nodekit all major character manipulation functions can be accessed through fields: load and create a character from a given configuration file, start/stop animation sequences, mix multiple animation sequences with blend factors, set the level-of-detail, show or hide the character's skin and skeleton, and manually configure bone and joint parameters and constraints for inverse kinematics. *SoCal3DPuppet* manages an internal list shared by all nodekit instances to store already loaded characters. This character cache greatly improves performance when using a large number of characters to create an army or crowd as already loaded characters do not have to be initialized again, they just need to be instantiated saving

time and memory.

Besides fields, *SoCal3DPuppet* also contains several public nodekit parts (e.g. *bbCoord, bbMaterial, bbDrawStyle etc.* to configure the appearance of bounding boxes appearing around individual skeleton bones. These parts come handy when the character is used in a modeling program such as that presented in Section 7.3.2, where an animator adjusts the pose of each bone to create keyframes for animation sequences and highlighting the currently active bone provides important visual feedback. Figure 6.5 shows examples of field- and nodekit part-based character control in a scene graph.

As later described in Section 6.2.2, high-level animation commands for *SoCal3DPuppet*-based characters will be handled by the *SoCal3DPuppeteer* class in the AR Puppet framework, while UbiAgent uses the nodekit directly.

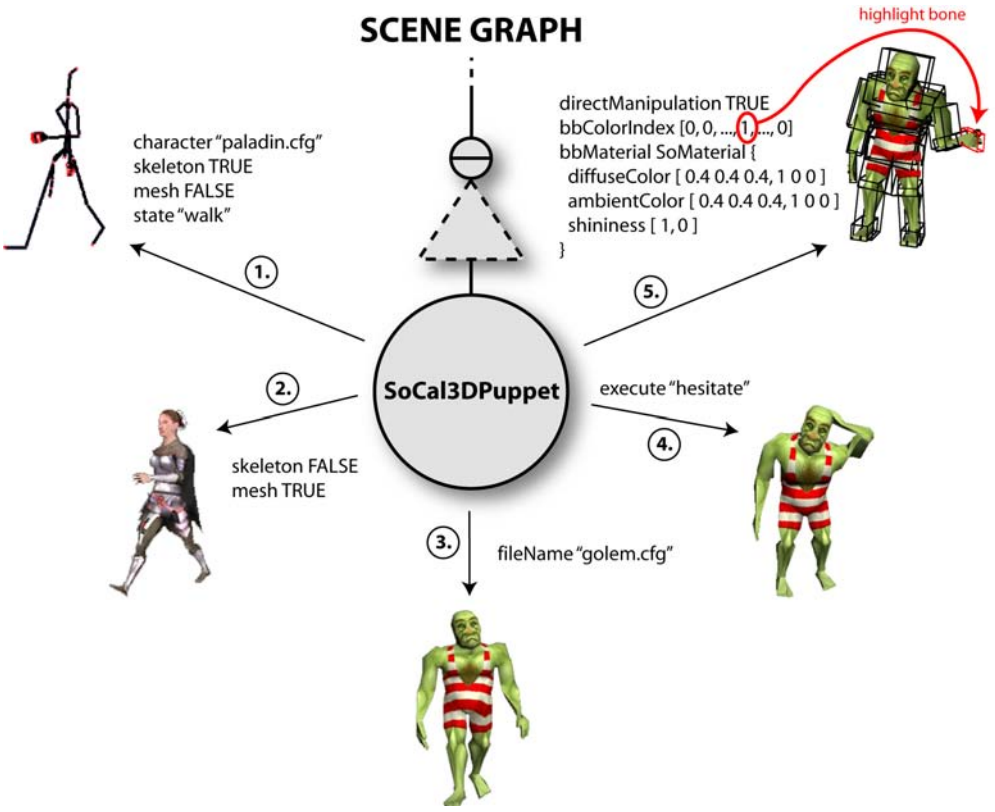


Figure 6.5: Effects of field value changes in the SoCal3DPuppet nodekit. Changes are executed in order: 1. load character file, turn on skeleton, turn off skin mesh, set looping animation, 2. turn off skeleton, turn on skin mesh, 3. load new character, 4. execute animation sequence, 5. turn direct manipulation mode on, highlight currently active bone with color index array and bone material nodekit part

6.1.6 Muddleware

Muddleware is a real-time database inspired by the concept of Tuplespaces [30] and implemented by Daniel Wagner. On the server side it is implemented as a memory-mapped XML database, where all data elements are stored as XML nodes by using an extended version of the TinyXML [98] library's DOM implementation. The XML data representation makes Muddleware highly suitable for storing hierarchical data structures, where clients can quickly retrieve and manipulate elements using queries based on the XPath 1.0 technology [112].

Besides the execution of ordinary database operations, Muddleware also allows clients to register *watchdogs* or database element update events. Watchdogs provide a simple yet powerful publish/subscribe mechanism and allow an easy setup of communication channels between clients. Muddleware is a high-performance database capable of handling thousands of queries per second and hundreds of clients simultaneously. Muddleware is highly portable as it is based on multi-platform libraries such as ACE [2] and TinyXML.

Muddleware clients can be integrated into Studierstube and UbiAgent by the Open Inventor class *XMLLogger*. Inventor nodekits derived from XMLLogger are capable of reading/writing field values from/into a Muddleware-based XML database. Fields can be the Inventor class' own fields or fields of named field containers such as nodes, nodekits and engines. XMLLogger-based Inventor objects enable encapsulation and persistency, whereby the database stores meaningful attributes from which object state can be restored. Muddleware can also serve as a possible implementation of a distributed scene graph where the database operates as a shared memory area between scene graph elements.

Besides setting parameters of the network connection to the database, low-level attributes of logging operations can be configured for each field the value of which is written into or read from the database by XMLLogger. The container name and field name together mark a field of a named field container in the scene graph for logging. The logging mode defines whether the field value should be read from or written into the database or should perform both operations in a given order. The logging mode can also define the logging condition i.e. a given database operation should be repeatedly performed at a configurable frequency or every time the field value is updated. XPath templates create parameterizable pointers to a part of the XML database the field values will be read from or written into. These templates contain parameters that are expanded at runtime such as the ID of the XMLLogger object, the name of the logged field, name of the respective field container, and an arbitrary number of custom parameters stored in an

ordered list that users can configure in a separate field.

Use cases in Figure 6.6 illustrate the versatile applicability of the XML-Logger component for application distribution, object persistency, and communication between objects through a shared memory area. The Inventor script below provides an example for the object communication use case. Figure 6.7 shows a graphical illustration of the scene graph constructed by the script.

```
...
Separator {
    Separator {
        DEF TRANS Transform {
            translation = DEF TRACKED_OBJECT SoTrakEngine {
                station 0
            }.translation
        }
        DEF TEXT SoText3 { string "Hello world!" }
    }
    Separator {
        Transform { translation -0.1 0 0 }
        DEF TEXT2 SoText3 { string "Muddleware Test" }
    }
}
...
DEF XMLLOGGER XMLLogger {
    id "MuddlewareTest" # object ID

    # XMLDB connection parameters
    addressXMLDB "192.168.0.2"
    portXMLDB 20000

    # field logging parameters
    logContainerName [ "TRANS", "TEXT", "TEXT2" ]
    logFieldName [ "translation", "string", "string" ]
    logXPathTemplateWrite "/test/writearea/$container$/@$field$"
    logXPathTemplateRead "/test/readarea/$container$/@$field$"
    logFieldMode [ LM_WRITE, LM_READ, LM_READ_WRITE ]
    logWriteMode [ WM_FIELDCHANGE, WM_TIMER, WM_TIMER ]
    logUpdateTime [ -1, 1, 4 ]
}
...
```

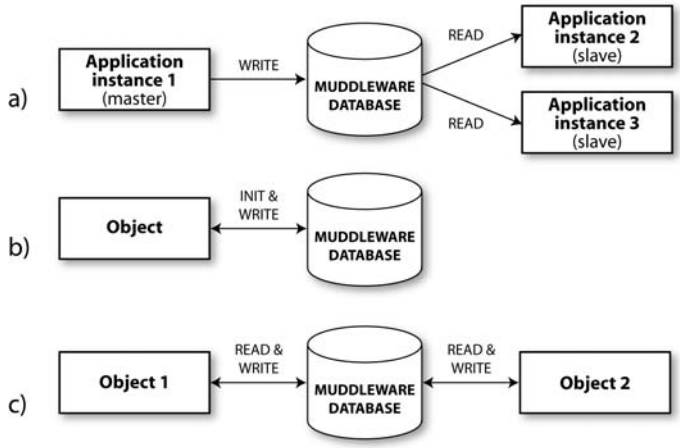


Figure 6.6: Application scenarios of the XMLLogger component: a) Distributed applications, b) Object persistency, c) Shared memory area between objects

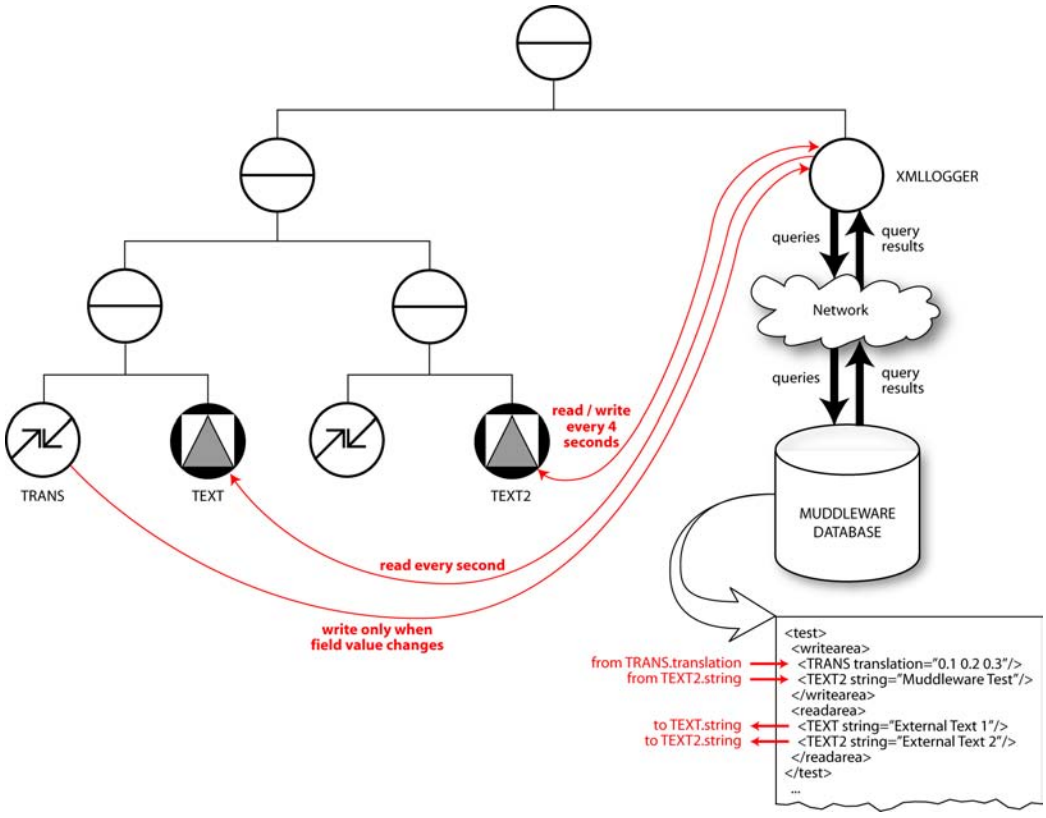


Figure 6.7: Graphical representation of a scene graph containing the XMLLogger object to communicate with the Muddleware XML database

The XMLLogger component and the Muddleware database are core building blocks of the UbiAgent framework, where they are used for maintaining a persistent storage for agent and application attributes and a shared memory area between agents and applications. Details can be found in Section 6.3.

6.2 AR Puppet Implementation

Similarly to Studierstube, the AR Puppet framework is a collection of Open Inventor classes extending Inventor's default functionality. They implement the abstract object hierarchy depicted in Figure 3.4. In our description we go gradually upwards in the object hierarchy by first explaining the implementation details of puppets followed by puppeteers, the choreographer, and finally the director component. Figure 6.8 provides a general overview of the inheritance structure of Inventor-based C++ classes making up the AR Puppet framework.

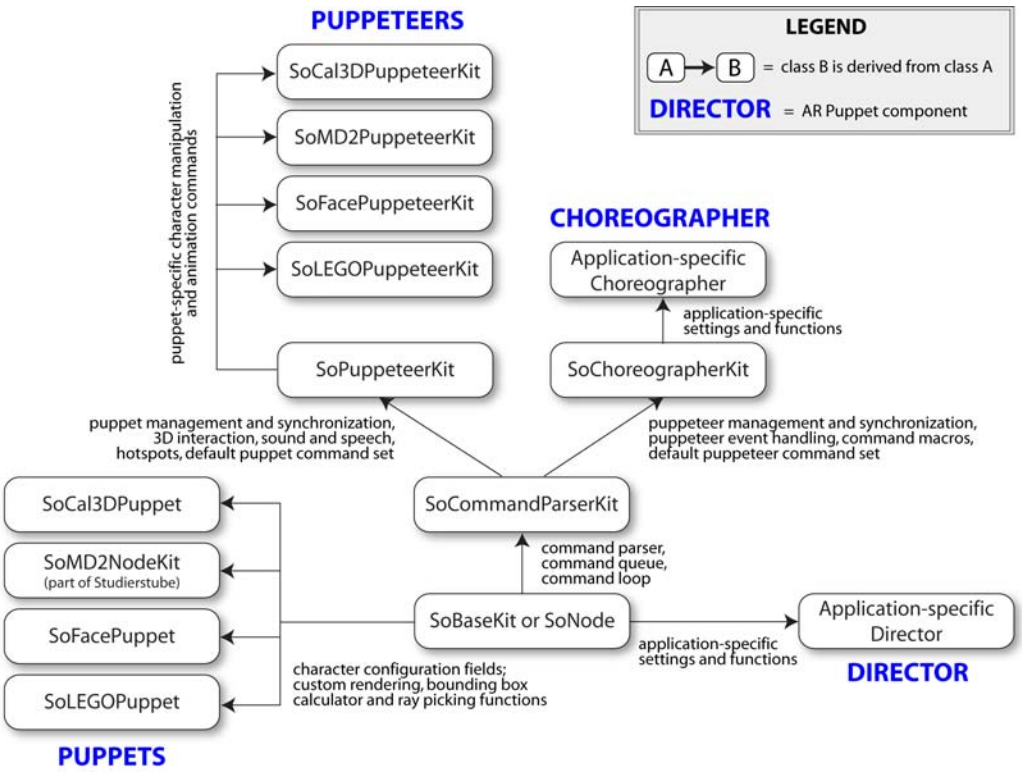


Figure 6.8: Inheritance tree with the AR Puppet framework's Inventor classes

6.2.1 Puppets

Puppets represent agent embodiments and serve as wrapper classes for scene graphs forming agent bodies and character animation libraries such as the Cal3D toolkit as shown in Section 6.1.5. Puppets consist of one or more nodes or nodekits derived from Inventor base classes to which character-specific attribute fields are added. For instance, the *SoCal3DPuppet* node adds fields to load and manipulate the appearance and behavior of skeleton-based 3D characters built on the Cal3D library. Besides new fields, puppets usually overload the parent node's or nodekit's callback functions to implement custom behavior for rendering, bounding box calculation, and raypicking.

Theoretically any Inventor scene graph can act as a puppet, however, for our application scenarios we created three reusable puppets:

- *SoCal3DPuppet:* This puppet integrates the Cal3D character animation library into Open Inventor, and thus Studierstube and AR Puppet. The puppet contains fields and functions to load, configure, and render skeleton-based animated 3D characters with rich artistic content exported from popular 3D modeling and animation software packages. See Section 6.1.5 for more details.

- *SoFacePuppet:* This puppet is a wrapper for an affective 3D talking head component with rich facial expressions. The talking head is described in detail in [10] and [20]. The talking head is capable of synthesizing numerous facial expressions based on a synthetic facial muscle model, performing lip synchronization to text-to-speech engine output, rendering vascular expressions such as blushing or pallor, and exhibiting lifelike behavior elements such as automatic insertion of eye blinks and gulping.

- *SoLegoPuppet:* This puppet encapsulates active physical objects assembled from the LEGO Mindstorms® robot kit. This kit facilitates the manual assembly and dynamic alteration of real robots and provides people lacking home electronic skills with an easy and flexible way to assemble responsive physical agent embodiments with a virtual control interface. The control interface is implemented by the *SoLegoPuppet* node, which communicates with the robot's "brain" or control unit via low-level infrared commands. It provides controls to turn engines on and off, set the engine direction and power, set the sensor mode, query sensor measurements, maintain an infrared message queue and periodically test whether the physical robot is "alive", that is whether it sends proper reactions to commands.

Agent embodiments may also use a fourth puppet called *SoMD2NodeKit*, which was already included in Studierstube and can render animated characters stored in the popular MD2 format of the Quake 2 computer game. The MD2 format is rather simple as it stores character animation sequences as vertex coordinate arrays, translations and scaling operations for each keyframe, and interpolates between keyframes for smooth animation.

6.2.2 Puppeteers

Command interface

While each puppet provides its own custom interface to control appearance and behavior, the puppeteer component implemented by the *SoPuppeteerKit* Inventor class is the lowest-level object in the AR Puppet hierarchy to offer a standard unified command interface for higher-level framework objects. *SoPuppeteerKit* is derived from the *SoCommandParserKit* class to gain an animation command interface facilitating a multi-level command queue, a configurable command parser and a customizable timed command loop.

Commands supported by a certain puppeteer instance are defined in the "*commandName argumentList*" format where the argument list contains the type, order and default value of command arguments. Currently the following basic types are supported: integer, float, and string symbolized by *%d*, *%f*, and *%s* respectively. Default argument values can be specified in brackets after the argument type. For example, the command "`setPosition %f %f %f %d(0)`" is a function setting the puppeteer's 3D position to a specific value within a given number of milliseconds. The function expects three float values for the position coordinates and an integer value for the time with a default value of 0 milliseconds. Missing arguments without default values and type mismatches are reported as errors during parsing.

The command parser also supports complex data types used by Open Inventor such as multi-dimensional vectors, quaternions, bounding boxes, etc. These complex types are composed of elements of one of the three basic argument types, e.g. a 3D vector consists of three float values. Arguments of complex data types are expanded into basic type elements again during the command parsing process, therefore the command syntax and parsing can be kept simple by restricting the format to basic types only.

Multiple puppeteer commands can be issued by using command groups to determine whether certain commands should be executed sequentially, in parallel or in combination. Besides command groups compound commands can also be created with complex internal dependencies that are expanded into a hierarchical structure of simple commands by the parser. For instance

the compound command *"go to target point within given time"* for a virtual human agent is expanded into the command sequence *"turn towards target point"*, *"start looping walk animation"*, *"set position to target point within given time"*, and finally *"start idle animation"* if no other animation command follows.

The puppeteer's command interface makes it transparent to other components which puppets (i.e. physical, virtual or augmented agent representations) execute puppeteer actions, which yields a unified interface for controlling virtual and real objects simultaneously. Besides command handling functions, the *SoPuppeteerKit* class extends *SoCommandParserKit* with several additional features that are discussed in the following sections.

Puppet management and synchronization

Multiple puppets can be attached to a single puppeteer to represent the physical, virtual, and augmented agent embodiments described in Section 3.1.1 or a distributed agent body where networked instances appear on multiple displays simultaneously. Puppets can be selectively turned on/off to enable/disable individual representations.

Puppeteer commands are sent to all active puppets, therefore command execution needs to be synchronized so that the puppets appear to act as a group. This means that a command executed by multiple puppets simultaneously is finished only after all puppets have reported completion. The puppeteer component includes synchronization mechanisms to implement this behavior.

3D interaction

Embodied animated agents in AR spaces appear to be able to freely move around in the user's physical environment. The position, orientation and size of an agent body communicate relevant visual information and thus act as symbols for guiding user attention to physical objects and locations. On the other hand, agents can be also distracting if they clutter the display or obstruct user view by covering important details of background objects. Agents in AR spaces therefore need to react to various ways of basic 3D user interaction to position or rescale agent bodies to match visual application appearance. The puppeteer component supports various interaction modes for each transformation element of its attached puppets. These transformation elements include translation, rotation and scale. Each element can be set independently from the others to one of the following four modes: *script*, *station*, *drag*, and *external*.

The *script* mode lets script commands (such as *setPosition*, *setOrientation*, etc.) control a transformation element through the standard puppeteer command interface. The *station* mode connects the element to the respective transformation field of a dedicated Studierstube tracking station. The *drag* mode allows control by one or more 3D cursor(s) of various Studierstube applications (see description about the Personal Interaction Panel in Section 6.1.4) in a drag-and-drop fashion. Finally, the *external* mode uses dedicated puppeteer fields for external control by a node in the application scene graph through field connections. The interaction modes can be changed dynamically based on the agent's current needs and application context.

Transformation elements consist of two components: a base and an offset transformation. Both components' interaction modes can be configured individually, which allows for complex interaction scenarios such as controlling an animated character's relative position and orientation by a script (*script mode*) on a moving tracked surface (*station mode*) while the size of the character is controlled through a field connection from an external control logic residing in the same scene graph as the character (*external mode*).

Sound and speech support

By using the Studierstube wrapper class *StbSpeech* integrating the Microsoft Speech API text-to-speech library [60] and the *SoFMODSound* class built on the FMOD audio library [29], embodied animated agents become capable of speaking and producing sound effects during animation sequences. The *SoPuppeteer* object enables the synchronization of sound and speech to animation sequences and provides toolkit-neutral control fields for speech synthesis and audio stream control.

Hotspots

Physical and virtual objects often contain special parts that play a dedicated and significant role within agent commands such as "push the power button on the fax machine" in a machine maintenance application or "pick up object with the character's right hand" in a computer game. We call abstract references to these object parts *hotspots*. Application developers and content authors may need to refer to hotspots frequently, however, low-level details should be hidden since command semantics is more relevant for them than quickly changing absolute parameter values. As hotspots are only valid within the context of their respective parent object, their attributes are always local and relative. This necessitates that absolute values be calculated before being used outside the parent puppeteer in a global reference frame.

Puppeteers based on *SoPuppeteerKit* can create an arbitrary number of hotspots by defining a string for a reference name, a 3D vector for local position, and a quaternion for local orientation. Typical presentation (e.g. "point at") and locomotion (e.g. "go to") agent tasks mostly exploit the position attribute, while a few other tasks require the orientation hotspot attribute as well, for instance when attaching an object to a character's moving hand.

To illustrate the usefulness of hotspots, let us consider a printer maintenance application assisted by a virtual animated repairman agent. An application content developer can issue the high-level command "show me the location of the most recent error on the printer" without actually knowing the nature and physical coordinates of the error. While processing the command, the puppeteer grouping and managing various printer representations first talks to the physical printer puppet wrapping up the printer driver. This puppet identifies the error source and its local coordinates in the printer's object coordinate system. The local coordinates are then translated and rotated by using the printer's current tracking transformation matrix to yield absolute coordinates in the world coordinate system. These world coordinates are then passed as arguments to a walking and pointing gesture command for the virtual repairman agent's puppeteer by the choreographer component.

Default animation command set

The *SoPuppeteerKit* class contains a set of convenient basic animation commands that cover typical presentation and locomotion agent tasks for application authors such as *"moving to a target"*, *"turning towards a point of interest"*, or *"pointing to a spot"*. Custom puppeteer objects derived from *SoPuppeteerKit* can use these commands and their default implementation as framework services, and implement their own custom behavior by overloading implementation functions and adding new commands. As described in Section 3.2.2, puppeteers automatically adapt animation commands for each puppet to match the current display and device profile they appear on, apply motion constraints such as path planning to avoid obstacles or terrain following, and implement puppeteer commands in a puppet-specific way.

Providing a default animation command set increases the reusability of animated agent components and saves implementation work for developers by facilitating the use of interchangeable animated agents. Let us consider an agent-assisted navigation application. In certain contexts a full body virtual human with rich face and body gestures can be a rewarding agent embodiment that serves as an exciting and motivating multimodal user interface

element to guide users around. On the other hand, there can be cases when the virtual character may be a nuisance to users during navigation as it may obstruct the view by covering important background elements or cluttering the display, or it may just simply be an overkill when a simple 2D arrow would do just as good as the complex body gestures of a virtual tour guide.

Readapting an application to work with an entirely different animated agent type (such as replacing the virtual human by a 2D arrow) usually requires a lot of work in classic agent-enabled applications including replacing animation commands, adopting a new command synchronization mechanism, and creating new agent configuration and control interfaces. The puppeteer component saves readaptation work by encapsulating all these features in a single component and providing access to them through a standard interface. This interface enables developers to simply replace a puppeteer responsible for a certain agent type by another without changing high-level component structure or code. Thus at application level it remains transparent which agent executes the commands issued by content authors.

Puppeteers can be either configured manually or automatically. Manual configuration is made by the application developer, while a control logic such as the choreographer component can manage puppeteers without user guidance based on preprogrammed context conditions, for instance to adapt the user interface to the current arrangement of visual objects on a certain display to avoid cluttering.

Default puppeteer set

The AR Puppet framework provides a set of agent types represented by custom puppeteer objects derived from the *SoPuppeteerKit* class:

- *SoCal3DPuppeteer:* Control puppeteer for the *SoCal3DPuppet* node with command interpretation for skeleton-based 3D characters, extra commands for locomotion, pointing and idle animation, and controls for animation blending, looping and non-looping animation sequences
- *SoFacePuppeteer:* Control puppeteer for the *SoFacePuppet* node with command interpretation for talking heads with built-in idle animation support and extra commands for pointing
- *SoMD2Puppeteer:* Control puppeteer for the *SoMD2NodeKit* node with command interpretation for 3D characters, extra commands for locomotion, pointing and idle animation, and animation controls for looping and non-looping animation sequences

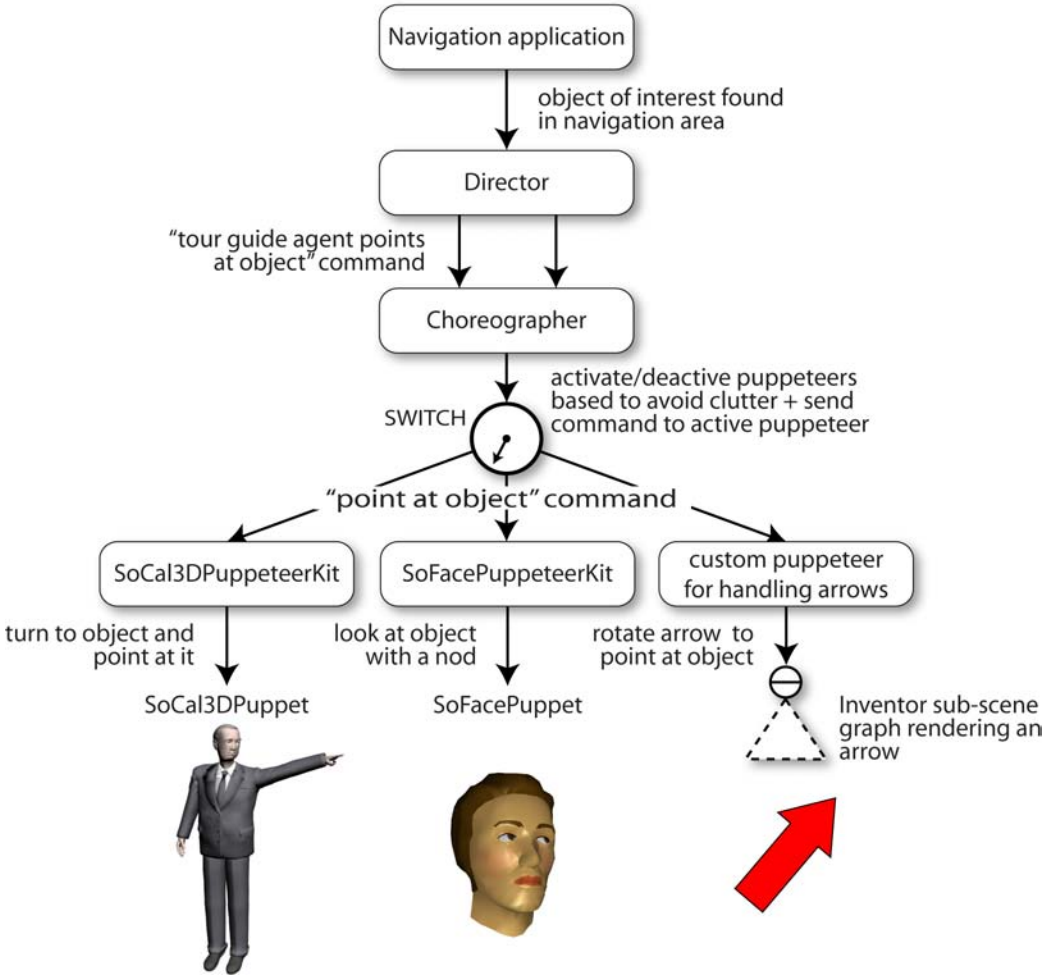


Figure 6.9: Replaceable puppeteers in a navigation application avoid to display clutter

- *SoLegoPuppeteer:* Control puppeteer for the *SoLegoPuppet* node with a custom command set for controlling and testing active elements such as engines and sensors of responsive physical objects and their virtual and augmented graphical representations

Figure 6.9 provides illustration for how elements of the default puppeteer animation command set are exploited and adapted by the choreographer component in AR Puppet.

6.2.3 Choreographer

The choreographer component is represented by the *SoChoreographerKit* Inventor class, which is also derived from the *SoCommandParserKit* to inherit facilities for scripting and command queue management. While the *SoPuppeteerKit* envelopes diverse virtual and physical agent embodiments by a unified command interface, the *SoChoreographerKit* builds upon well-defined services of *SoPuppeteerKit*. The most important choreographer functionalities are described in the following sections.

Resolving abstract attribute references

All puppeteer objects are attached to a dedicated choreographer nodekit part within a sub-scene graph so that puppeteers and their attributes can be directly accessed and manipulated. Due to this structure the choreographer has always an up-to-date overview of what is happening on the "stage", i.e. it is aware of the detailed status information of all puppeteers and user interaction props. As puppeteers are aware of only their own puppets and thus cannot "see" other puppeteers and the application environment, high-level agent commands coming from content authors through the director component must be first interpreted and decomposed here and replaced by commands understood by puppeteers. Choreographer commands target one or more puppeteers with a parameter list containing a mixture of concrete values (e.g. absolute 3D coordinates as parameter for a locomotion command) and abstract references (e.g. a puppeteer hotspot as parameter for a pointing presentation command).

Choreographer commands are first parsed for macros containing abstract attribute references written in the *entityType(instanceName).attributeName* format. The macros' *instanceName* element is a string identifying an instance of a named entity of type *entityType* within the application scene graph embedding the choreographer component. The following list enumerates all possible macros with their respective entity and ID types:

- *puppeteer(name):* a named puppeteer derived from the *SoPuppeteerKit* nodekit
- *puppeteer(name).hotspot(name):* a named hotspot of a named puppeteer in the puppeteer's local object coordinate system
- *puppeteer(name).hotspotoffset(name):* transformation offset of a named puppeteer's named hotspot in the puppeteer's local object coordinate system

- *puppeteer(name).hotspotabsolute(name):* a named hotspot of a named puppeteer in the AR application's global coordinate system (after applying the hotspot's local transformation offset and the global puppeteer pose transformations)
- *station(number):* a Studierstube tracker object interfacing OpenTracker and retrieving tracking data from a given station number
- *user(number):* a Studierstube user object with given user ID
- *pen(number):* a Studierstube 3D cursor object belonging to a user with given user ID
- *pip(number):* a Studierstube Personal Interaction Panel object belonging to a user with given user ID
- *node(name):* a named Inventor node in the application scene graph

Having found and parsed a macro, the choreographer retrieves a pointer to the referenced scene graph object and queries the value of its *attributeName* attribute by relying on Open Inventor's internal field container value and type retrieval functions and type casting mechanisms. The parser expands and replaces the macro by its absolute value and proceeds to the next macro. Having processed all macros, the expanded command containing concrete and absolute attribute values is passed on to the puppeteer, which tailors it to its puppets individually. Figure 6.10a illustrates how the Virtual Tour Guide application (see Section 5.3) exploits attribute reference resolving.

Path planning

The choreographer component not only decomposes abstract director instructions into explicit puppeteer commands with exact parameters but (true to its name) it also "choreographs", modifies high-level motion-related commands to make agents appear to be aware of their surrounding environment and thus behave in a more natural way.

A prominent example is path planning. AR applications are highly dynamic environments, where application objects, users and their interaction props frequently change their pose, size and other parameters influencing the locomotion of embodied agents. While executing the simple high-level agent command *"move to a given target location"*, agent embodiments need to calculate a complex path for their movement to avoid collision with other agents and real and virtual application objects to avoid appearing unnatural. Because of the dynamic nature of AR applications it would be overwhelming

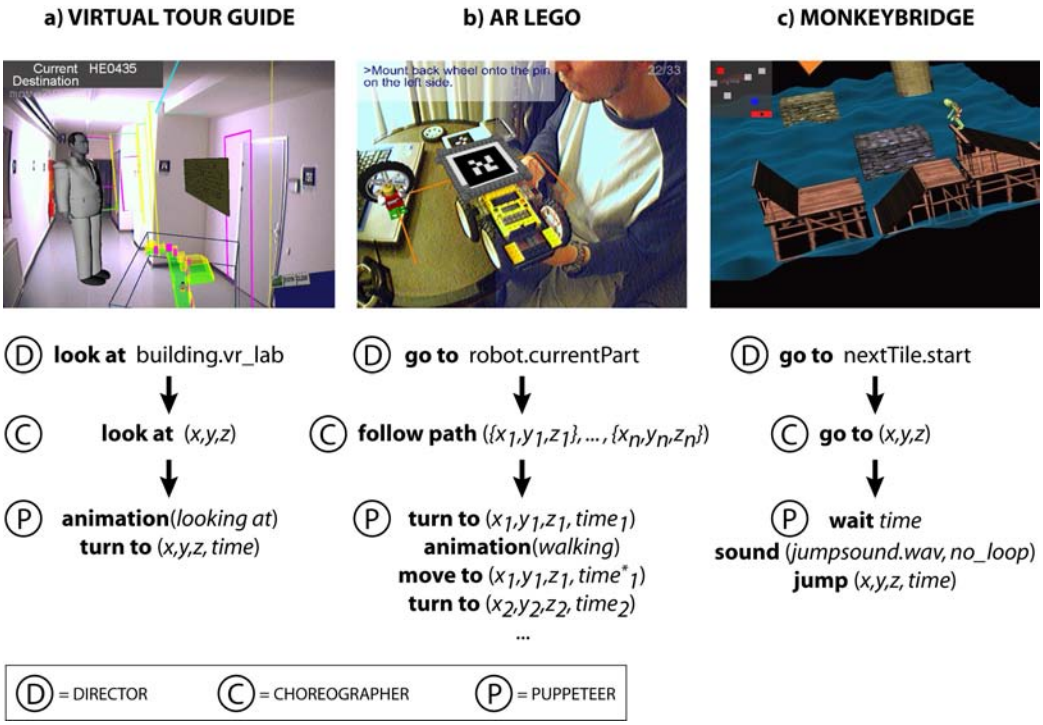


Figure 6.10: Choreographer functionalities illustrated by AR Puppet applications: a) Resolving an abstract object reference in the Virtual Tour Guide application, b) Path planning while carrying the next robot building block to its place in AR Lego, c) Motion planning in Monkeybridge while moving across bridge elements of diverse shapes

for application authors to take care of path planning by themselves, therefore the choreographer takes care of this service, making this task transparent to authors.

At the developer level path planning is rather application-specific (e.g. some applications may require additional, agent embodiment-specific motion planning as well), therefore there is no default implementation for path planning in *SoChoreographerKit*. Agent application developers need to derive their own choreographer component from *SoChoreographerKit* and add their custom implementation. Figure 6.10b and c provide illustration for path planning implementation in the AR Lego application (without motion planning) and the Monkeybridge game (with motion planning).

Puppeteer management

Section 6.2.2 describes the puppeteer component's replaceability feature, which ensures that agent representations best matching the current application context can be selected automatically. This selection is made by the choreographer component. Puppeteer sub-scene graphs are attached to a choreographer nodekit part having the Studierstube class type *SoMultiSwitch*. This *SoSwitch*-based class enables addition and removal, activation and deactivation for an arbitrary group of its children (i.e. the puppeteers) simultaneously at run-time. Puppeteer manipulation actions are triggered by a control logic, which (similarly to the path planning implementation) can be programmed in C++ by deriving a custom choreographer object from *SoChoreographerKit*.

Events

Higher-level components such as the director can register events with the choreographer to get meaningful status information about puppeteers without violating the object-oriented principle of information hiding. For instance, an event can be fired by the choreographer if a certain puppeteer penetrates a given bounding box around a specific 3D object. The director does not "see" where puppeteers currently reside in the application environment, it only gets notified by a "puppeteer entered bounding box" event. Using the choreographer's aforementioned abstract attribute reference mechanism, complex high-level events can be constructed such as testing whether puppeteers have collided, reached a certain distance from each other, or are approaching each other with a certain minimum velocity.

A special type of event is generated by the choreographer's command synchronization module. In case a complex agent command requires a puppeteer to wait for another to finish its actions, a synchronization mechanism is needed. Therefore puppeteers always report when they are done with command execution, generating a *"command finished"* event. In case of a temporal dependency, the choreographer does not forward its command queue to the next command until it receives such an event.

6.2.4 Director

While the puppeteer and choreographer components rely on predefined building blocks and services, the director has to be rewritten for each application. It is implemented as a state engine within a custom Inventor class derived from a node or nodekit, which is then attached to the AR application's scene

graph together with the choreographer, puppeteer and puppet components. The director's engine states contain choreographer commands. Whenever the director enters a state, commands contained by the state are sent to through a field connection referencing the choreographer's *command* field. The commands are then decomposed and "choreographed" by the choreographer to match the syntax of the target puppeteers. Transitions between director states are triggered by the aforementioned choreographer events and user input from interaction devices and the application GUI.

In all of the example scenarios presented in Chapter 5 the director's control logic is programmed in C++ and thus cannot be reprogrammed or updated dynamically. However, with Inventor-based scripting tools such as Pivy [73] it becomes possible to create programmable director components where the state engine's code can be dynamically uploaded and modified using a Python script.

6.3 UbiAgent Implementation

As described in Section 4.2.1, all UbiAgent components communicate with one another by means of a dedicated memory area stored in the Muddleware XML database. All components must therefore share a set of common functionalities:

- Establishing and maintaining a communication channel to the database using the TCP/IP protocol and featuring automatic connection recovery
- Writing/reading a dynamically configurable list of field values for selected Open Inventor nodes and nodekits into/from the database
- Automatically checking missing, erroneous or duplicate data elements and maintaining database integrity
- Logging connection and database operation information into a user-specified text file for debugging and analysis

All the above functionalities are implemented by the *XMLLogger* Inventor nodekit, which is described in Section 6.1.6 in detail. All UbiAgent components except the agent brain and agent body objects are derived from the *XMLLogger* class. They modify and extend default class behavior by overloading its functions and adding new, component-specific routines and fields. In the following sections we describe implementation details of each UbiAgent component.

6.3.1 Habitat

The habitat component is implemented by the *Habitat* Inventor class. It represents a host environment consisting of a software and hardware platform agent bodies can migrate to. The software platform stands for an application instance agents can reside in. The hardware platform includes all computing resources necessary to render animated agent services such as CPU power, available memory, battery power for mobile devices, a display where agent bodies can appear, and a tracking system or "locale" to let agents identify users, interactions props and other relevant tracked objects. Besides the aforementioned standard parts habitats may contain non-standard hardware elements such as a sound card and speakers for audio output or an infrared port to communicate with external devices using the IrDa protocol.

The agent brain continuously monitors potential habitats whether they are suitable for hosting agent bodies. As habitats may be comprised of diverse hardware components, it is difficult to provide a standard, absolute measure to characterize "how good" a hardware platform is to host a particular agent type. Therefore the *Habitat* object adds the *"hospitability"* attribute to the default fields inherited from the *XMLLogger* class. The *hospitability* field contains an integer number between 0 and 100, which lets the agent brain know the amount of computational resources available for rendering a particular agent body (0 = no resources, 100 = sufficient resources).

It is important to note that we do not aim at providing a complete solution to offer a complex and detailed hardware platform ontology and benchmark as usually required in software frameworks developed in the research areas of distributed computing, pervasive computing and UbiComp. Instead we rather provide a simple, initial implementation sufficient for testing resource-aware software components in AR environments, therefore we use an absolute measure for representing computation power of available agent platforms by combining available CPU power, memory, and remaining battery power to form a single number: *(CPU load in % / 100) * (available physical memory size / total physical memory size) * remaining battery power in %*. The remaining battery power is set to 100% in case of stationary devices.

For test purposes we implemented a *Hospitability Configurator* component on the Win9x/2k/XP platform, which can generate a *hospitability* value for *Habitat* objects in two modes. The first mode measures available CPU power, memory and battery power of the device it is running on while demanding only a negligible amount of system resources not to interfere with performance measurements. The second mode allows users to set the aforementioned system performance parameters manually to support the simulation of migration cases otherwise difficult or time consuming to test such

as low battery power of mobile devices. The *hospitability* parameter is then calculated from the real or simulated performance components using the aforementioned formula. The *Hospitability Configurator* is a GUI-based standalone C++ application (see Figure 6.11a for a screenshot) built on the Microsoft Foundation Class Library (MFC). It periodically sends the current *hospitability* value to one or more selected *Habitat* objects through a TCP/IP socket, therefore this component can run on a remote machine as well.

The *Habitat* nodekit contains one or more *Display* objects and a *Locale* object as nodekit parts. The *Display* and *Locale* inventor nodekits first initialize their fields from the database's *repository* section based on a unique ID. The repository contains permanent hardware information about the set of possible displays and tracking systems that may be used in AR applications. This set needs to be maintained manually by a technician. Whenever a new display or tracking hardware gets installed that can be used by AR applications, default hardware information must be entered into the repository database, e.g. typical resolution, size and type of a display, or the accuracy, update rate and degrees of freedom of pose data delivered by a given tracking system. The reason why human maintenance is favored is the fact that basic hardware characteristics rarely change, therefore occasional manual parameter update costs less time and coding efforts than implementing and maintaining a versatile set of device-specific monitoring software tools.

After the application instance containing the *Habitat* object has booted up and the display and locale hardware parameters have been initialized from the database repository, the *Display* and *Locale* objects add child elements containing their hardware parameters to their parent *Habitat*'s database element. This hierarchical database structure indicates hardware devices currently used by the habitat and enables the use of complex queries (see Section 6.3.4) for agents searching for platforms with suitable hardware parameters.

Besides the Muddleware database the *Habitat* also communicates with the *Habitat Manager* component using the UDP network protocol. This component runs on a dedicated server machine (ideally on the same machine as the Muddleware database) and ensures that only fully operational habitats are stored in the database to prevent agents from attempting to migrate to malfunctioning platforms. *Habitat* objects indicate their "alive" status by pinging the *Habitat Manager* periodically. In case a habitat machine fails to ping the *Habitat Manager* within a specific timeout due to a breakdown (e.g. a PDA runs out of battery) or a broken network connection, the *Habitat Manager* notices the error and removes the erroneous habitat from the database storing potential candidates for agent migration. Once recovered, the habitat announces its services again and restores its database entry after the acknowledgement of the *Habitat Manager*.

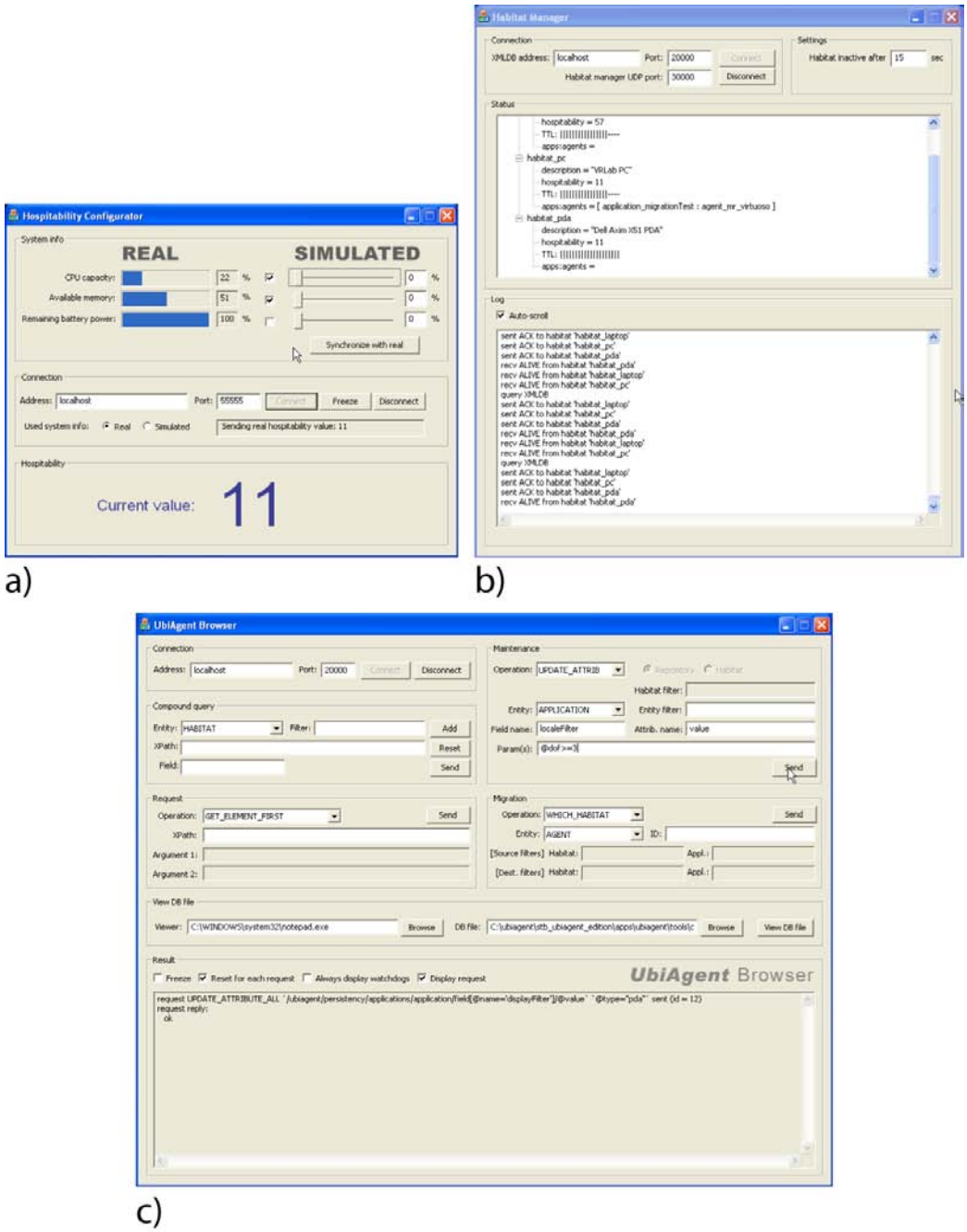


Figure 6.11: Screenshots of configuration and monitoring tools for the UbiAgent framework: a) UbiAgent browser, b) Habitat Manager, c) Hospitability Configurator

Similarly to the *Hospitability Configurator*, the *Habitat Manager* is not Inventor-based but implemented as a standalone application using MFC-based GUI elements. Figure 6.11b provides a screenshot. The GUI provides feedback about the currently active habitats, their hospitability value, time elapsed since the arrival of their last "alive" message, the remaining time until their potential removal (time-to-live parameter), and the agents and applications currently residing in them.

6.3.2 Application control

The *ApplicationLogger* object symbolizes an instance of a networked distributed AR application. In the UbiAgent framework all applications are persistent, which means that they are able to reconstruct any desired application state from appropriately selected state information elements stored in a persistent database. In our framework the state of a distributed application can be fully described by two components at an arbitrary point of time: the global, shared application state and the local, habitat-specific application state.

The two components are stored in two separate sections in the database. The habitat-specific application parameters (e.g. reflecting local customization settings or user interaction history) are stored in a child element within the parent element of the habitat the application is running in. The global, shared application state is stored in a special database area reserved for application persistency. The *ApplicationLogger* object provides dedicated fields to configure local and global application state information storage in the database.

6.3.3 Agent brain and bodies

Embodied animated agents in the UbiAgent framework consist of three components: the *AgentLogger* object, the *AgentBrain* object, and various agent-specific scene graphs representing agent embodiments. The *AgentLogger* Inventor nodekit is responsible for agent state persistency and communication with the Muddleware database. Similarly to distributed applications, agent state information also consists of two components: a global, shared agent state and local, habitat-specific information. Global agent state information includes agent-specific attributes about appearance and behavior and is stored in a dedicated agent persistency section in the database. The habitat-specific state information is stored as a child element within the habitat parent element to facilitate database queries of external UbiAgent components

wishing to know the application and habitat a particular agent is working with.

The *AgentBrain* Inventor node is a base class for control logics managing agent bodies. The *AgentBrain* decides which application context demands which agent bodies to activate or deactivate. It is typically implemented as a state engine, where states are associated with hardware requirements for habitats such as display resolution or a certain type of tracking data, and transitions are triggered by changes in application state attributes.

The *AgentBrain* is not derived from the *XMLLogger* object but directly from the *SoNode* class, facilitating complex database queries for habitats and processing database replies. The control logic's state engine must be implemented on a case-by-case basis for each AR application as hardware requirements associated with application state information vary, however, the result of habitat queries is presented in a standard format, namely a list of strings in the *AgentBrain* object's *habitats* field. Scene graphs representing agent bodies can simply include a Studierstube-based engine checking whether a given habitat ID is within the habitat list obtained by the *AgentBrain* from the database.

If the database reports that a habitat is suitable for the agent brain's hardware platform preferences dictated by the application context, an agent body needs to be created within the habitat using Open Inventor's serialization mechanism or if a body belonging to the agent has been already created, it needs to be activated. In case a habitat does not match agent preferences, the agent body needs to be deleted or simply deactivated. The agent body's creation/deletion and activation/deactivation mechanisms depend on the application developer's choice and needs to be implemented for each application.

6.3.4 Database structure and queries

Agent brain implementations use the *XPath 1.0* syntax [112] to store hardware platform preferences for the selection of habitats. The preferences later become parameters in queries sent to the Muddleware XML database, which also uses XPath in its query syntax. The hierarchical structure of XML and XPath fits the tree structure of the UbiAgent database, a graphical version of which is shown in Figure 6.12.

The following lists presents a few examples for XPath-based habitat filters to illustrate how agents can search for their preferred habitats using Muddleware queries. All XPath queries are used as a parameter in the following Muddleware command: `GETATTRIBUTE (from_all_elements, xpath)`

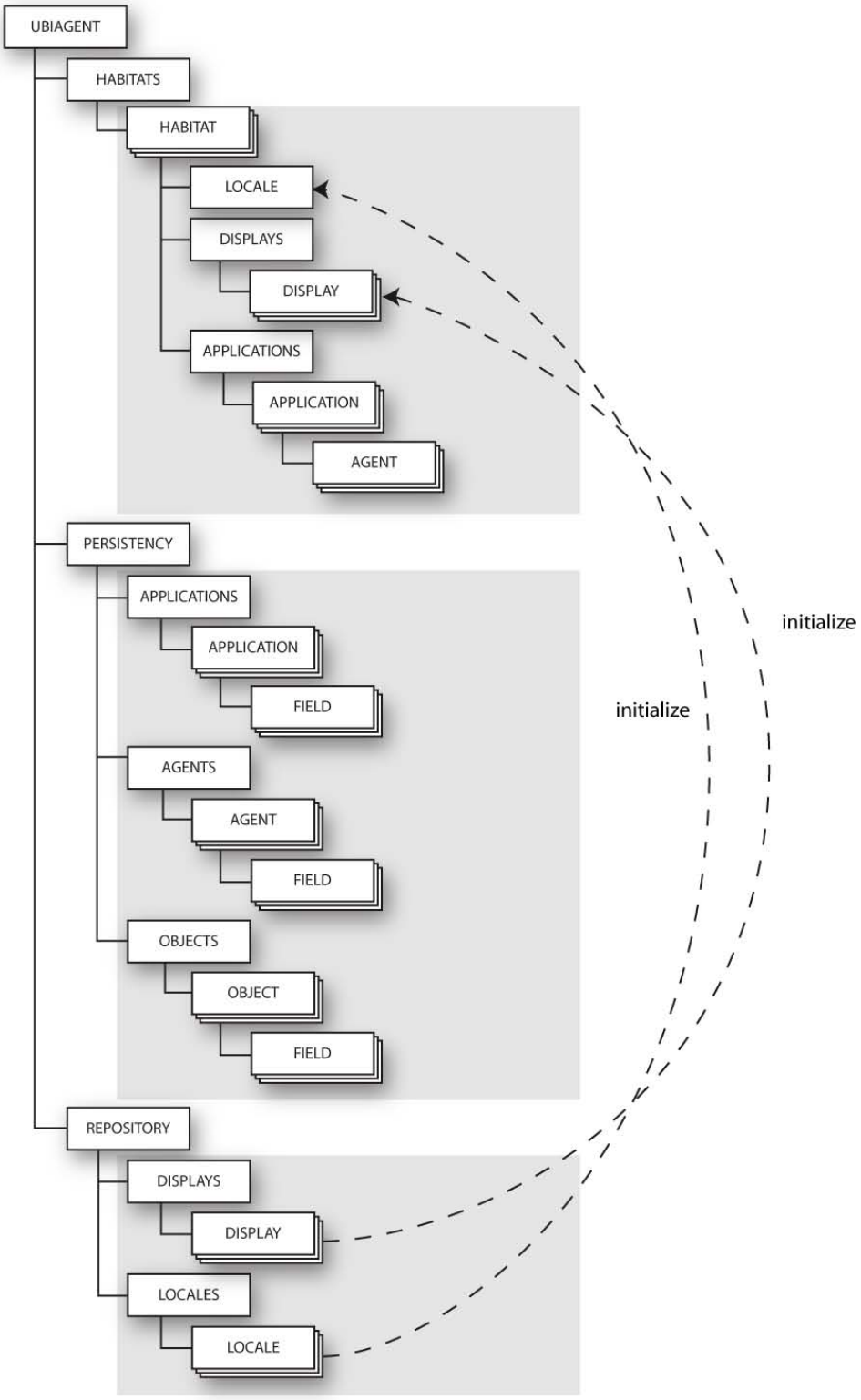


Figure 6.12: The hierarchical structure of the UbiAgent XML database

- look for habitats where agent called "legomaxl" currently resides:
  `/ubiagent/habitats/habitat/agents/agent[@id="legomaxl"]/ancestor::habitat/@id)`
- look for habitats where the application "arlego" resides:
  `/ubiagent/habitats/habitat/applications/application[@id="arlego"]/ancestor::habitat/@id)`
- look for habitats where the agent is needed ("agentNeeded" bit is high) in the application "ubitrack":
  `/ubiagent/habitats/habitat/applications/application[@id="agentNeeded"]/habSpecData/field[@name="agentNeeded" and value="1"]/ancestor::habitat/@id)`
- look for habitats where the hospitability value is greater than 50:
  `/ubiagent/habitats/habitat/[@hospitability>50]/@id/ancestor::habitat/@id)`
- look for habitats where there is a projector with a resolution of min. 1024x768:
  `/ubiagent/habitats/habitat/displays/display[(@res_x>=1024 and @res_y>=768) and @type="projector"]/ancestor::habitat/@id)`
- look for habitats where there is a 6DOF tracking system:
  `/ubiagent/habitats/habitat/locale[@dof="6"]/ancestor::habitat/@id )`
- look for habitats where agent "mr_virtuoso" resides and there is an hmd and a magnetic 6DOF tracking system:
  `/ubiagent/habitats/descendant::agent[@id="mr_virtuoso"] /ancestor::habitat/displays/display[@type="hmd"]/ancestor::habitat/locale[@type="magnetic" and @dof="6"]/ancestor::habitat/@id )`

For low- and high-level debugging and simulation purposes (e.g. forced migration) we developed the UbiAgent browser tool, which is an MFC GUI-based standalone program containing parameterizable shortcuts for UbiAgent-related Muddleware queries and commands. Figure 6.11c shows a screenshot.

6.3.5 Integration of AR Puppet into UbiAgent

The AR Puppet and UbiAgent frameworks have been developed to solve two different sets of problems for AR agents. Firstly, AR Puppet facilitates a

standard command interface for controlling virtual and physical agent representations and the resolution of abstract attribute references for high-level agent commands. On the other hand, UbiAgent supports agent migration between various hardware and software platforms by enabling agents to monitor current application state and respective platform requirements, and thus to choose the most suitable platform to render current agent actions on.

Figure 6.13 illustrates how application developers can exploit the services of both AR Puppet and UbiAgents simultaneously. As application status is monitored by the *Director* object in AR Puppet, it can be extended with *ApplicationLogger*'s capability of communicating with the Muddleware database to support persistent application state. As each application requires its own specific *Director* implementation, a combined Inventor class merging both component's capabilites can be implemented with little effort by deriving the *Director* from *ApplicationLogger* instead of a plain *SoNode*.

Another point of integration is at the agent representation management level as the *Puppeteer* and *AgentBrain* objects share similar functionalities. The *Puppeteer* manages software-based agent representations that usually run on the same machine such as a robot's virtual, physical or augmented counterpart. In contrast, the *AgentBrain* serves rather as a manager for hardware-based agent representations residing on different computing devices and displays. If an application needs to handle complex combinations of local and remote as well as software- and hardware-based agent embodiments at the same time, then a custom *Puppeteer* nodekit needs to be created that incorporates the *AgentBrain* capability of locating remote agent bodies and platforms in the Muddleware database based on preferences specified in the *XPath* format. Thus the AR Puppet framework's *Puppet* terminology fully covers the *agent body* entity within the UbiAgent framework.

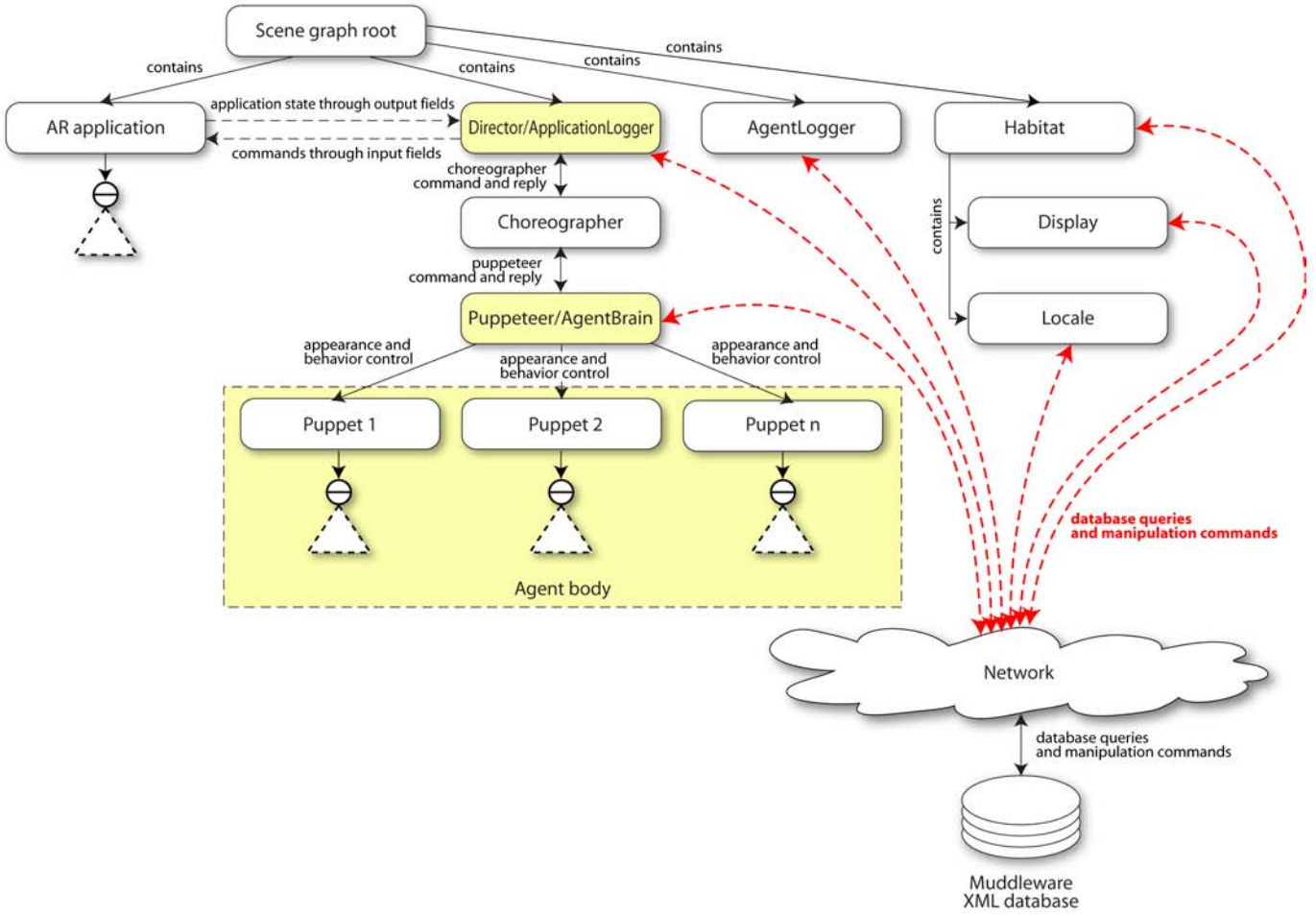


Figure 6.13: Integrating UbiAgent components into an AR Puppet-assisted Inventor application's scene graph

Chapter 7

Authoring

This chapter presents how to author agent-enabled AR applications in a quick and efficient way by integrating agent components and services into an application while leaving its structure intact. Firstly, the powerful Open Inventor scripting mechanism is presented followed by the description of the high-level authoring language called APRIL. Finally, scenarios for immersive, dynamic agent configuration and content creation are presented.

7.1 Scripting with Inventor

The AR Puppet and UbiAgent frameworks are both implemented in Inventor, therefore we can rely on its powerful scripting mechanism to author agent-enabled AR applications. In both frameworks agent components exchange data with applications only through field connections. This is a non-invasive way of communication (see Section 6.1.2 for details), which enables application authors to simply add the agent components' script representation to the application scene graph and create field connections either by the built-in Inventor mechanism or the Studierstube-based *SoRoute* object.

Figure 7.2 illustrates how AR Puppet components are added to the AR Lego example AR application scenario (see Section 5.1) in a simple, non-invasive way. The AR Puppet sub-scene graph's root is the *Director*, which is attached to the scene graph root outside the AR application's scene graph. The *Choreographer* object (derived from *SoChoreographerKit*), the puppeteers *SoLegoPuppeteer* and *SoCal3DPuppeteer*, and all associated puppets are attached to the agent sub-scene graph in a hierarchical structure shown in the figure. The director component communicates with the AR Lego application through field connections querying application output fields and controlling application input fields.

Figure 7.1: Integrating UbiAgent components into the Character Animation Studio application's Inventor scene graph

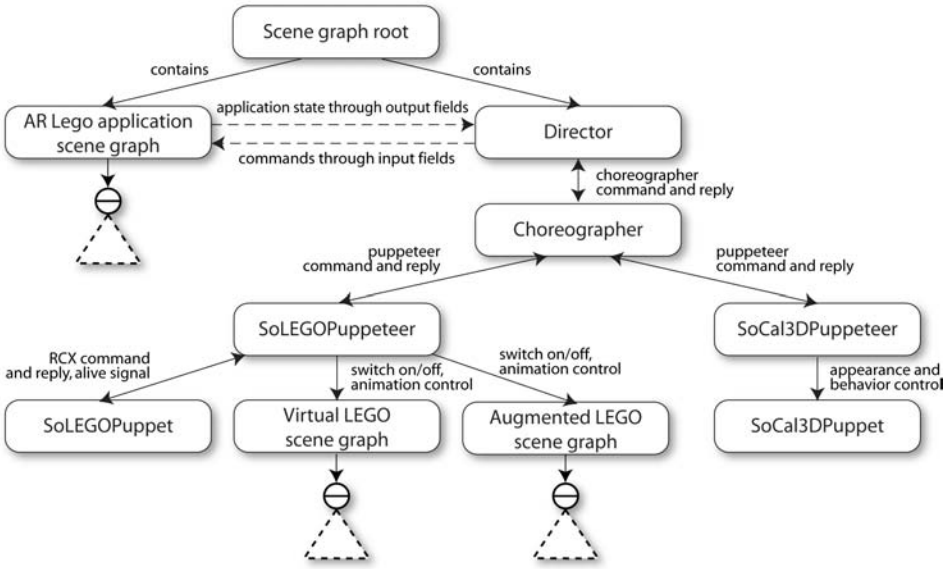


Figure 7.2: Integrating AR Puppet components into the AR Lego application's Open Inventor-based scene graph

Figure 7.1 shows how UbiAgent components are integrated into the Character Animation Studio application (see Section 5.4). In the UbiAgent framework the *ApplicationLogger* object monitors dedicated fields of the application scene graph through field connections in a non-invasive way. In case of a distributed application each application instance needs to be connected to a separate *ApplicationLogger* object that writes output application attribute values into the Muddleware database and reads input application attribute values from the database for that particular instance. Master/slave rights for reading and writing attributes for multiple networked *ApplicationLogger* objects are either preconfigured in the scene graph or assigned dynamically based on the respective application instance's activity i.e. where the latest interaction events have come from.

Besides the *ApplicationLogger* object delivering application events, agents rely on the *AgentLogger* object to send and receive attribute updates to/from the database that influence the agent body's appearance and behavior. In the example scenario of Figure 7.1 the agent body's scene graph contains an *SoCal3DPuppet* to serve as the actual agent embodiment. All distributed agent bodies embedded into application instances are controlled by the *AgentBrain* component through the network. The *AgentBrain* sits on a dedicated machine and contains some control program to manage agent embodiments.

The application instance's hardware environment is represented by the

Habitat object, which is also attached to the application scene graph. The *Habitat* object contains the *Display* and *Locale* objects that deliver information to the database about the display and tracking system the habitat's corresponding AR application instance relies on. The habitat communicates with the *Habitat Manager* object that - similarly to the *AgentBrain* object - resides on a dedicated machine and regularly checks habitats for erroneous behavior or broken communication. Communication between UbiAgent components residing on different machines is made through the Muddleware XML database using a communication network such as LAN or WLAN.

The Inventor-based authoring approach is naturally suitable for rapid prototyping and adding agent services to Studierstube-based AR applications. Currently there is no support for legacy applications and other AR software frameworks, therefore interfaces and wrapper classes for non-Studierstube-based applications must be implemented on a case-by-case basis.

7.2 Scripting with APRIL

APRIL [47] is a high-level descriptive language for authoring presentations in augmented reality. For agent-enabled AR applications AR Puppet objects need to be turned into APRIL components by wrapping up relevant input and output fields of respective Open Inventor objects such as *SoPuppeteerKit* and *SoChoreographerKit*-based nodes and nodekits. Appendix B presents an example how to turn the *SoChoreographerKit* class into an APRIL component, and then how to create a simple application with several APRIL-based AR Puppet components with a simple APRIL script.

In the APRIL workflow AR applications are modeled by the APRIL storyboard as a UML state engine in the user's favorite UML authoring tool. A small part of an example state engine from the Virtual Tour Guide application (see Section 5.3) is shown in Figure 7.3. Individual stations of the guided tour are modeled as states, triggering linear presentations when the user arrives. The structure of the building and the different modes for the guided tour (linear or free mode) are modeled by transitions and superstates. The graphical representation of the state engine was created with the Poseidon UML modeling tool [74].

The UML-based storyboard is exported into an XMI file [110], the official XML-based standard for serializing UML diagrams. This XMI description is then simplified and included into an APRIL script file to represent the presentation's story. This draft presentation script is then fine-tuned using standard graphical or text-based XML editors such as XMLSpy [111], where script commands controlling the APRIL-based AR Puppet component are

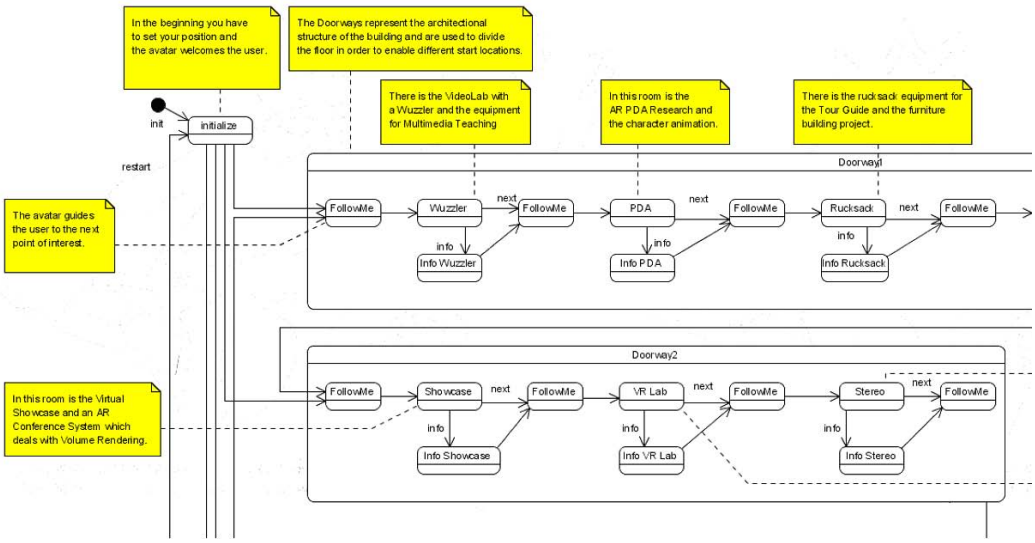


Figure 7.3: A part of the APRIL framework-based Virtual Tour Guide storyboard's UML state engine

inserted into the presentation states' action list. In each state agent commands are issued by setting the puppeteer or choreographer component's *command* attribute to a specific command string. These agent commands trigger appropriate animation sequences, therefore the agent appears to be aware of the application's current state.

7.3 Immersive Content Authoring

While tools to author content for AR applications and animated agents have been based on strong roots in classic computer graphics and VR, we argue that augmented reality should exploit medium-specific techniques to smoothen its production workflow. This means that AR has to penetrate its own authoring pipeline to let content developers fully experience and understand the novel environment of AR applications and tailor the content to this new medium.

In this section we do not describe a complete solution for immersive authoring of standalone applications as already presented by Lee et al. [48]. Instead we describe three case studies that explore novel ways of immersive configuration and authoring of content elements for AR applications and embodied animated agents. The software implementation of all case studies is built on our AR Puppet framework.

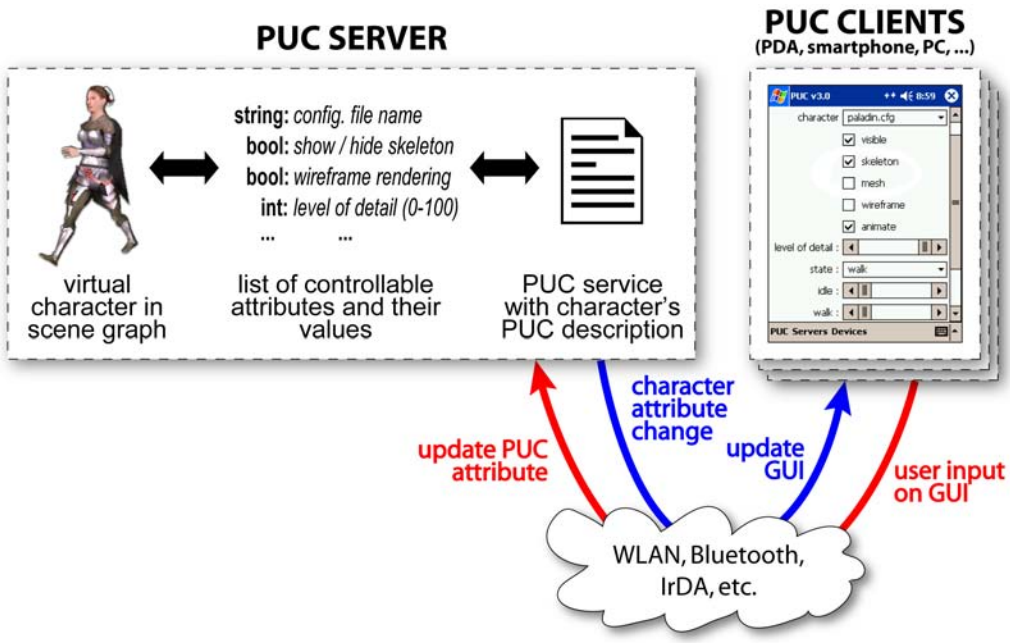


Figure 7.4: Overview of the Personal Universal Controller-based agent configuration pipeline

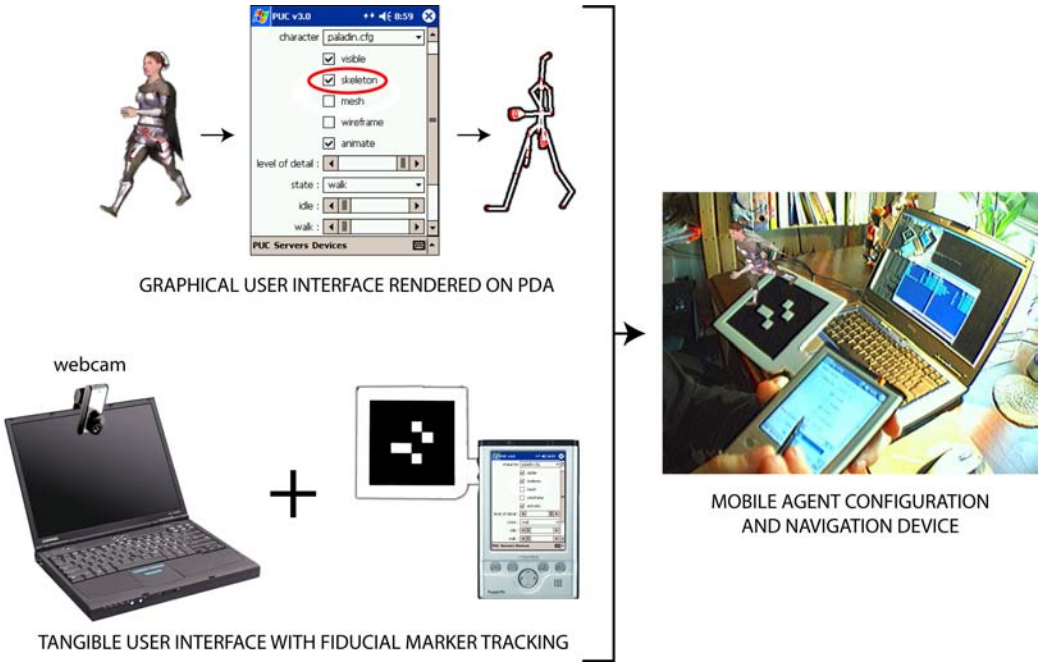


Figure 7.5: PDA-based PUC client as a Graphical and Tangible User Interface

7.3.1 Personal Universal Controller

An innovative way to configure AR agents is using the Personal Universal Controller (PUC) technology [66]. Figure 7.4 illustrates how the technology works. To support PUC, an agent needs to provide an XML-based description of its relevant, configurable attributes and its supported commands together with the command syntax. The agent runs a PUC service by attaching an *SoPucServer* Studierstube node to the agent scene graph and making field connections to relevant fields. The PUC service is listening to incoming connections from PUC clients. The client software is implemented on various devices and platforms including PCs, PDAs and smartphones. When connected to a PUC service, the PUC client queries the services attribute description, and then renders a graphical user interface (GUI) to control the listed attributes. A PUC service can accept multiple clients, which enables collaborative, multi-user configuration.

Figure 7.5 shows a control GUI rendered on a PocketPC. By checking the "skeleton" control checkbox (marked with an ellipsoid) the user changes the rendering mode from mesh to skeleton mode to reveal the underlying bone structure. Mobile devices implementing the PUC technology provide an intuitive way to configure AR agents as the user can simply walk up to an agent in her physical environment, connect to it to query its PUC description, and then tweak attributes dynamically using the GUI just generated on the fly. If the agent exposes the commands it understands together with their syntax in the XML description, a template can be generated with which the user can directly test the effect of script commands.

7.3.2 Keyframe Creation for Animated Characters

Modelers and animators often rely on real-life references to build and animate 3D characters for games or film production. Videotaping a real subject and the manipulation of mock-ups support the creation of precise and expressive character animation in virtual content creation environments such as 3D modeling and animation packages. Figure 7.6a shows a real person posing for an artist who is creating the balancing animation for the virtual monster character shown in Figure 7.6b and Figure 3.9.

Professional artists use motion capture techniques or other expensive means of acquiring motion data such as the Monkey kinematic tracker device [26] and the Dinosaur Input Device [42] to create an essential initial data set for the final, refined animation. Similarly, within an AR environment the animated virtual model and the real-world reference are merged to form a single interactive modeling instrument.

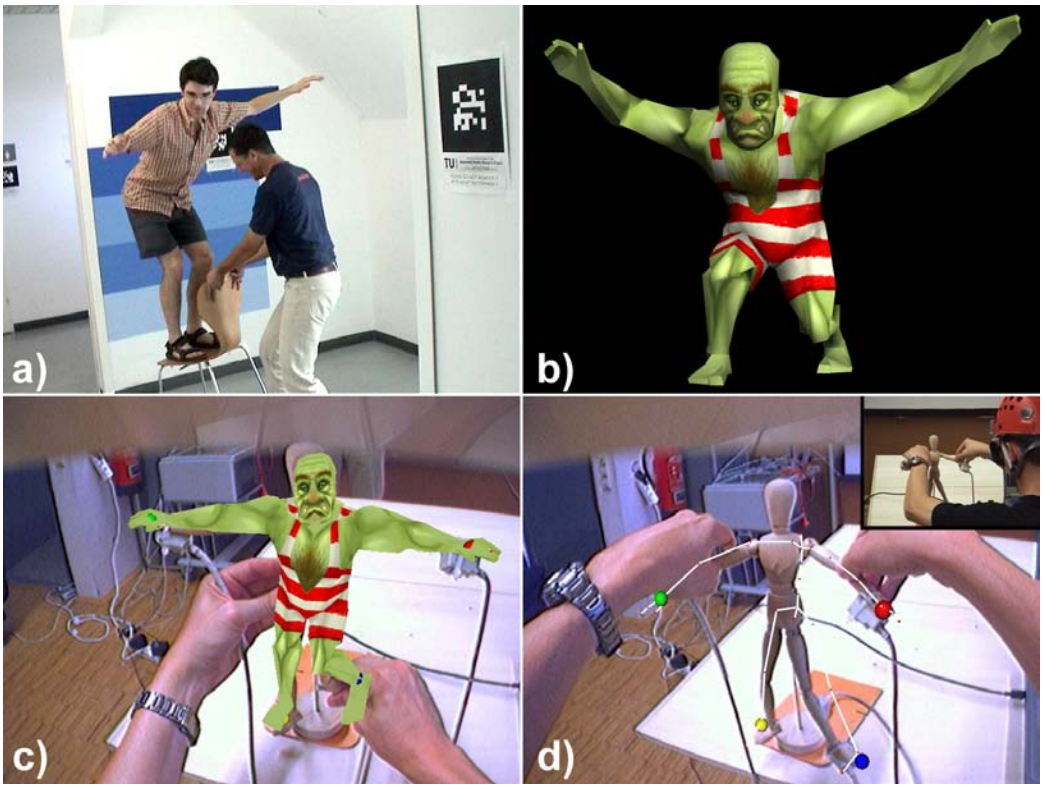


Figure 7.6: Immersive keyframe creation for animated characters: a) Real actor posing to provide reference for a virtual animated character, b) Resulting balancing animation, c),d) Screenshots of our immersive keyframe creation tool in mesh and skeleton mode

We use a wooden mannequin as an input device to animate skeleton-based anthropomorphic 3D characters. The head and limbs of the mannequin are pose-tracked. The system maps real-time pose data to rotation information for the joints of the character skeleton using the Cyclic Coordinate Descent (CCD) inverse kinematics technique [105] extended with rotational constraints for joints (see Figure 7.6c and d for illustration). We extended the Cal3D character animation library-based *SoCal3DPuppet* Inventor nodekit with the CCD inverse kinematics module to let users directly manipulate joint rotation. Keyframes are stored in the standard Cal3D format, therefore animations created with our tool can be replayed either within our modeling application to receive immediate feedback about the animation sequence's correctness, or they can be imported into 3D Studio MAX to allow animators to refine the animation in a professional modeling and animation software package.

Our AR-based immersive modeling tool not only enables close interaction with virtual models by using tangible objects but also the creation of complex motions such as walking up stairs or lifting a ball. Animators can use an actual physical model of stairs or a ball and their respective virtual counterpart in concert with the character to create realistic motions.

7.3.3 Immersive Music Composition

Music has only recently emerged as an important medium for AR applications [17] [75]. The Augmented Piano Tutor application serves as a piano teacher that educates users about basic chords and scales in an AR environment. It uses a desktop-based AR setup using a real keyboard that communicates with the control PC with MIDI in and out messages, a webcam, and a monitor to combine the real and the virtual scene. Section 3.2.1 and Figure 3.5b explain how the hardware setup and the MIDI keyboard's physical and virtual representation are implemented as puppets within the AR Puppet framework.

As shown in Figure 7.7, the real keyboard is augmented with virtual information that instructs users to hit certain keys in a defined order while giving feedback as to which keys have been pressed correctly and which ones were pressed by mistake. Sound is generated on the physical keyboard by MIDI commands in accordance with the visual feedback blended into the user's view, therefore the boundary between the user situated in the real environment with the instrument and the "teacher" giving instructions from the virtual world appears to be blurred.

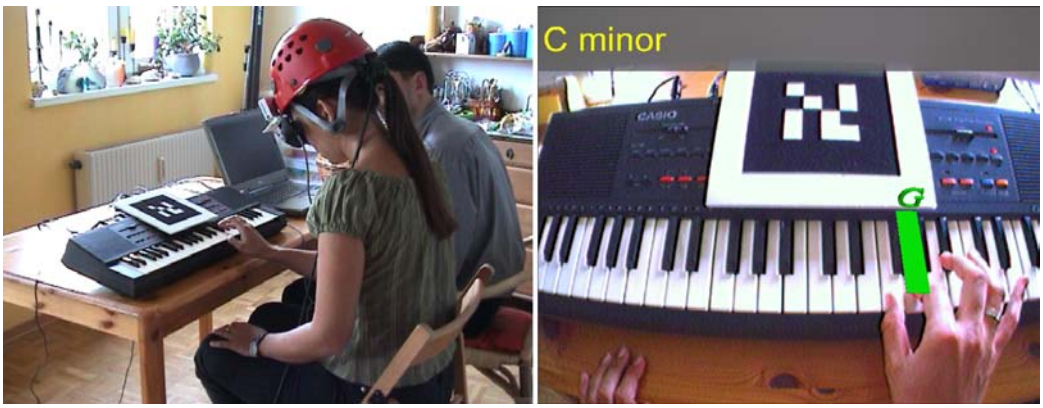


Figure 7.7: Screenshots from the AR Piano Tutor application

The synthetic information is always synchronized with the audio. When the user is instructed to play a certain chord on the keyboard, the piano keys yielding the chord are visually highlighted indicating which should be pressed, while the very same chord is played on the keyboard to create a mental connection between the two. When the user tries to imitate the chord, the correctly pressed and missed keys are marked with different colors on top of the real keys, giving a hint how the error should be rectified.

This system can be used as an advanced music composition tool offering the complex functions and rich visual feedback of a sequencer program running on a PC, while preserving the simplicity and freedom of a keyboard. While the sequencer module analyzes the tunes currently being played, it could also suggest harmonizing background chords and appropriate solo melodies to be played immediately on the keyboard.

Chapter 8

Conclusions

This thesis has presented steps toward creating "smart" and adaptive software components for AR applications and discussed techniques previously unexplored in AR such as using physical objects as output modalities in human-agent communication, emergent behavior of virtual animated 3D objects embedded into the physical world, multi-user interface adaptation, proactive interface migration and opportunistic exploitation of dynamic resources typical for UbiComp systems. We showed through numerous example application scenarios that AR interfaces enhanced by embodied autonomous agents possessing the aforementioned characteristics can enrich human-computer interaction.

The thesis introduced the AR Puppet and the UbiAgent software frameworks to support the embedding of autonomous animated agents into AR applications at low cost. These frameworks have been built on the powerful *Studierstube* collaborative AR framework, which allows experimentation with a wide range of AR applications, tracking technology, hardware platforms and various stationary and mobile displays. We presented in-depth implementation details how to program and author agent-assisted AR applications.

We believe that animated agents bring new challenges and fresh perspectives for AR. Although we tried to carefully obtain feedback about the usability of the demonstrated agent-enabled AR applications, as most important future work AR agents need to be formally evaluated in the following aspects:

- *Behavior:* the effectiveness of various gestures and the necessary amount of anthropomorphic features in agent behavior
- *Appearance:* the optimal size and placement of virtual agent embodiments on various displays to let agents be informative yet not obtrusive

- *Autonomy:* the amount of agent autonomy / user guidance required and tolerated by users
- *Interaction:* required set of real and virtual world events to be measured for efficient agent-human interaction
- *Migration:* required agent feedback in visual and aural form to mitigating the visual and cognitive gap while migrating between disjunct workspaces

Weiser [103] questions the usefulness of embodied interface agents by juxtaposing them with Ubicomp systems. He argues that assistant-like interfaces increase the seam between humans and computers, which conflicts with the fundamental goals of Ubicomp. We believe that empowering our interface agents with proactive behavior minimizes the required explicit user input to ensure correct agent operation, which makes agent presence less apparent in the interface. We also agree with Weiser's argument that some users do want personal assistants as butlers acting at their command (see Figure 8.1), therefore the potential use of embodied animated agents in Ubicomp environments is justified.

User preference for agent representations spans a wide spectrum between lifelike and non-anthropomorphic embodiments. The robot maintenance application used a human-like animated character, while the calibration aid employed simple geometrical shapes to visualize application state. While nonverbal communication may increase the information bandwidth of agents, in some AR systems a simple arrow may prove more useful than a full-body animated character. A possible solution to match preferences and purposes of a wide range of users and applications may be the employment of multiple agent bodies with varying level of realism and detail. The appropriate agent body would either be explicitly selected by the user, or automatically chosen by the application matching the amount of information currently shown on the display to avoid clutters.

Another important variable in agent systems is the amount of proactivity ranging between submission and aggression. Humans are normally suspicious about systems that exclude users from the decision making loop. On the other hand, the complexity of computing systems including AR systems will soon reach a level where direct manipulation interfaces become so saturated with controllable parameters that users will have no other choice than delegating interface manipulation tasks. We also share Lieberman's point [49] that agents are rather suited for making uncritical decisions, therefore we should let agents make a suggestion instead of immediate actions. A typical

UbiAgent example for this approach is the grace period allowing appropriate user response before agent migration.

We created our own ontology for adaptive AR systems without the intention of completeness. However, an exhaustive ontology would be highly desirable, in particular if based on standards such as WSDL [109]. This would allow information sharing between UbiAgents and other AR and agent systems, supporting resource sharing between diverse computing systems. Another important issue for future work is to eliminate vulnerability caused by the current framework implementation based on a single central database, which makes our system prone to network and computer failures.

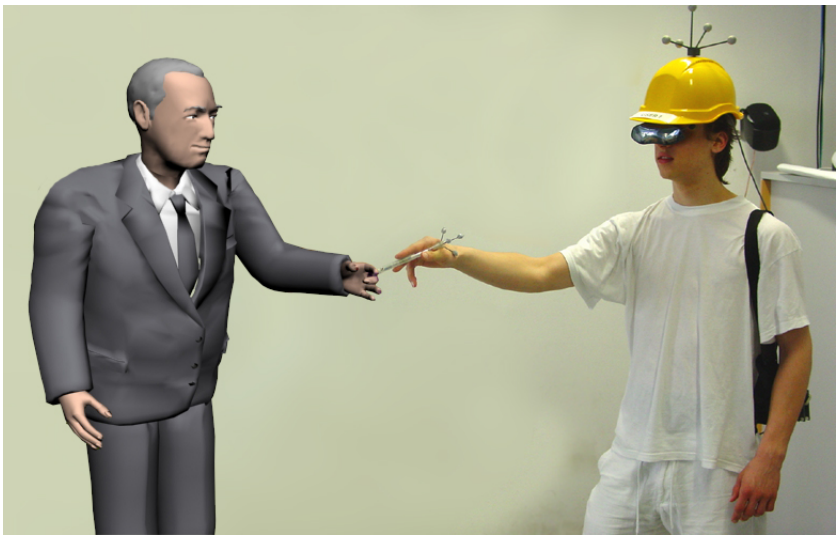


Figure 8.1: Some users do prefer to employ digital butlers (image created with Photoshop)

Appendix A

UbiAgent XML Database Format

The integrity of the UbiAgent XMLD database can be verified any time by using the following Document Type Definition (DTD):

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT agent (attribute?)>
<!ATTLIST agent
    ID CDATA #REQUIRED
>
<!ELEMENT agents (agent+)>
<!ELEMENT application (attribute?)>
<!ATTLIST application
    ID CDATA #REQUIRED
>
<!ELEMENT applications (application+)>
<!ELEMENT artifact (attribute)>
<!ATTLIST artifact
    ID CDATA #REQUIRED
>
<!ELEMENT artifacts (artifact)>
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
    name CDATA #REQUIRED
    value CDATA #REQUIRED
>
<!ELEMENT current_agent EMPTY>
<!ATTLIST current_agent
    ID CDATA #REQUIRED
>
```

```
<!ELEMENT current_agents (current_agent+)>
<!ELEMENT current_application EMPTY>
<!ATTLIST current_application
    ID CDATA #REQUIRED
>
<!ELEMENT current_applications (current_application)>
<!ELEMENT display EMPTY>
<!ATTLIST display
    description CDATA #REQUIRED
    resolution_x CDATA #REQUIRED
    resolution_y CDATA #REQUIRED
    colordepth CDATA #REQUIRED
    stereo CDATA #REQUIRED
    tracked CDATA #REQUIRED
    type CDATA #REQUIRED
>
<!ELEMENT habitat (location, display, current_agents,
  current_applications)>
<!ATTLIST habitat
    ID CDATA #REQUIRED
    hospitability CDATA #REQUIRED
    description CDATA #REQUIRED
    status CDATA #REQUIRED
>
<!ELEMENT habitats (habitat+)>
<!ELEMENT location EMPTY>
<!ELEMENT ubiagent (habitats, agents, applications, artifacts)>
```

A typical snapshot of the UbiAgent database during the Ubiquitous Technician demo:

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE ubiagent SYSTEM "ubiagent_XMLDB.dtd">
<ubiagent>
 <habitats>
   <habitat id="habitat_laptop" description="Istvan&apos;s
      laptop" hospitability="53">
      <displays>
        <display id="display_monitor_laptop" description="
           Laptop display" type="monitor" res_x="1680"
           res_y="1050" stereo="FALSE" tracked="FALSE" />
```

```
    </displays>
    <applications>
      <application id="application_migrationTest">
        <agents>
          <agent id="agent_mr_virtuoso" />
        </agents>
        <habSpecData>
          <field name="agentCounter" value="1" />
          <field name="positionPDA" value="0 0 0" />
          <field name="stationNumberPDA" value="0" />
        </habSpecData>
      </application>
    </applications>
    <locale id="locale_desktop_laptop" description="1
       local, 1 remote" type="artoolkit" dof="6" />
  </habitat>
  <habitat id="habitat_pc" description="VRLab PC"
     hospitability="37">
    <displays>
      <display id="display_monitor_pc" description="VRLab
         demo machine" type="monitor" res_x="1280" res_y
         ="1024" stereo="FALSE" tracked="FALSE" />
    </displays>
    <applications>
      <application id="application_migrationTest">
        <agents />
        <habSpecData>
          <field name="agentCounter" value="0" />
          <field name="positionPDA" value="0 0 0" />
          <field name="stationNumberPDA" value="1" />
        </habSpecData>
      </application>
    </applications>
    <locale id="locale_desktop_pc" description="1 local
       distributed" type="artoolkit" dof="6" />
  </habitat>
  <habitat id="habitat_pda" description="Dell Axim X51
     PDA" hospitability="30">
    <displays>
      <display id="display_pda" description="Dell Axim
         X51 screen" type="pda" res_x="640" res_y="480"
         stereo="FALSE" tracked="TRUE" />
    </displays>
```

```
    <applications>
      <application id="application_migrationTest">
        <agents />
        <habSpecData>
          <field name="agentCounter" value="0" />
        </habSpecData>
      </application>
    </applications>
    <locale id="locale_pda" description="rotation␣with␣
        stylus" type="mouse" dof="2" />
  </habitat>
</habitats>
<persistency>
  <applications>
    <application id="application_migrationTest">
      <field name="wantAgentInHabitat" value="habitat_pda
          " />
      <field name="lastActiveStationIndex" value="0" />
      <field name="habitatFilter" value="@hospitability&
          gt;40" />
      <field name="displayFilter" value="(@res_x&gt;=1024
          ␣and␣@res_y&gt;=768)␣and␣@type=&apos;projector&
          apos;" />
      <field name="localeFilter" value="@type=&apos;
          magnetic&apos;␣and␣@dof=6" />
      <field name="distanceThreshold" value="0.40000001"
          />
    </application>
    <application id="application_arlego">
      <field name="constructionStep" value="21" />
    </application>
    <application id="application_signpost">
      <field name="destination" value="vrlab" />
    </application>
    <application id="application_ubisense_calib" />
  </applications>
  <agents>
    <agent id="agent_mr_virtuoso">
      <field name="state" value="mr_virtuoso_idle1" />
      <field name="wireframe" value="FALSE" />
    </agent>
    <agent id="agent_technician" />
  </agents>
```

```
    <objects />
  </persistency>
  <repository>
    <displays>
      <display id="display_monitor_laptop" description="
          Laptop display" type="monitor" res_x="1680" res_y=
          "1050" stereo="FALSE" tracked="FALSE" />
      <display id="display_monitor_pc" description="VRLab
          demo machine" type="monitor" res_x="1280" res_y="
          1024" stereo="FALSE" tracked="FALSE" />
      <display id="display_projector" description="VRLab
          projector" type="projector" res_x="1024" res_y="
          768" stereo="FALSE" tracked="FALSE" />
      <display id="display_pda" description="Dell Axim X51
          screen" type="pda" res_x="640" res_y="480" stereo=
          "FALSE" tracked="TRUE" />
      <display id="display_monitor_tabletpc" description="
          Sony VAIO U70 screen" type="monitor" res_x="800"
          res_y="600" stereo="FALSE" tracked="TRUE" />
    </displays>
    <locales>
      <locale id="locale_keyboard" description="0-3
          keyboard stations" type="keyboard" dof="6" />
      <locale id="locale_desktop_laptop" description="1
          local, 1 remote" type="artoolkit" dof="6" />
      <locale id="locale_desktop_pc" description="1 local
          distributed" type="artoolkit" dof="6" />
      <locale id="locale_magnetic" description="5 local
          distributed" type="magnetic" dof="6" />
      <locale id="locale_pda" description="rotation with
          stylus" type="mouse" dof="2" />
      <locale id="locale_signpost" description="building
          tracking for indoor navigation" type="artoolkit"
          dof="6" />
      <locale id="locale_ubisense" description="UbiSense
          ultra-wideband tracking" type="uwb" dof="3" />
    </locales>
  </repository>
</ubiagent>
```

Appendix B

AR Puppet-based APRIL components

The following code illustrates how to turn Open Inventor-based AR Puppet objects into ARPIL components. The code shows the choreographer component's APRIL implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<component id="choreographer" xmlns="http://www.
    studierstube.org/april" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://www
    .studierstube.org/april ../../../../tools/APRIL/april-
    component.xsd">
  <interface>
    <input id="id" type="SFString" default=""/>
    <input id="command" type="MFString"/>
    <input id="commandType" type="MFString"/>
    <input id="puppeteerName" type="MFString"/>
    <input id="targetList" type="MFString"/>
    <input id="parameterList" type="MFString"/>
    <input id="dataNeedsValidation" type="SFBool" default="
        FALSE"/>
    <input id="dataValid" type="SFTrigger"/>
    <input id="purgeQueue" type="SFTrigger"/>
    <input id="responseIndex" type="SFInt32" default="0"/>
    <output id="commandList" type="MFString"/>
    <output id="commandFormatList" type="MFString"/>
    <output id="response" type="MFString"/>
    <output id="responseIndex" type="MFInt32"/>
    <output id="finishedCommand" type="SFTrigger"/>
```

```
  <output id="finishedCommandData" type="MFString"/>
  <output id="finishedAll" type="SFTrigger"/>
  <output id="responseString" type="SFString"/>
  <part id="puppeteers"/>
</interface>
<implementation type="OpenInventor" requires="../apps/
    ARPuppet/arpuppet#SoChoreographerKit">

  DEF <id/> Separator {
    DEF _Choreographer SoChoreographerKit {
      puppeteers Separator {
          <sub id="puppeteers"/>
      }
      id <in id="id"/>
      command <in id="command"/>
      puppeteerName <in id="puppeteerName"/>
      commandType <in id="commandType"/>
      targetList <in id="targetList"/>
      parameterList <in id="parameterList"/>
      dataNeedsValidation <in id="dataNeedsValidation"/>
      dataValid <in id="dataValid"/>
      purgeQueue <in id="purgeQueue"/>
    }

    DEF _Info SoInfo {
      string = SoSelectOne {
          type SoMFString
          input = USE _Choreographer.response
          index = USE _Choreographer.responseIndex
      }.output
    }
  }

  <out id="commandList"><id/>_Choreographer.commandList</
      out>
  <out id="commandFormatList"><id/>_Choreographer.
      commandFormatList</out>
  <out id="response"><id/>_Choreographer.response</out>
  <out id="responseIndex"><id/>_Choreographer.
      responseIndex</out>
  <out id="finishedCommand"><id/>_Choreographer.
      finishedCommand</out>
```

```
    <out id="finishedCommandData"><id/>_Choreographer.
        finishedCommandData</out>
    <out id="finishedAll"><id/>_Choreographer.finishedAll</
        out>
    <out id="responseString">_Info.string</out>

  </implementation>
</component>
```

Having created an APRIL-based component for all AR Puppet objects, we can create a simple, keyboard tracking-based AR application. The application has a choreographer component controlling two puppeteers: an SoCal3DPuppeteer-based embodied autonomous agent and a "dummy" puppeteer that we only use here to store hotspots for demonstration purposes. The application has two states. The first state contains a choreographer command that instructs the Cal3D-based character called "cally" to go to the location marked by the absolute 3D coordinates of the "dummy" puppeteer's "spot0" hotspot in 3 seconds. When the command is finished, the application waits for 2 seconds, and then an automatic transition is made to the second state, where the choreographer instructs the Cal3D character to go to the application coordinate system's origin in 2 seconds. When the user presses a button that is automatically displayed on a head-up display, a transition is made back to the first state, and the loop starts from the beginning.

The APRIL script of the aforementioned application is shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE april SYSTEM "../../studierstube/tools/april/
    april.dtd">
<april xmlns="http://www.studierstube.org/april" xmlns:ot="
    http://www.studierstube.org/opentracker" xmlns:xsi="
    http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.studierstube.org/april␣
    ../../studierstube/tools/april/april.xsd">
  <setup src="keyboard.aps"/>
  <presentation id="characterTest" name="APRIL␣+␣AR␣Puppet␣
      Test">
    <story>
      <scene name="one" initial="true"/>
      <scene name="two"/>
      <transition event="12" source="one" target="two"
          guard=""/>
```

```
  <transition event="21" source="four" target="one"
      guard="" />
</story>
<cast>
  <actor id="choreographer" src="../APRIL/components/
      choreographer.apc">
    <input id="id" value="choreographer" />
    <children id="puppeteers">
      <actor id="puppeteer1" src="./cal3dpuppeteer.apc"
          >
        <input id="id" value="cally" />
        <input id="whichPuppets" value="0" />
        <input id="tracked" value="1" />
        <input id="station" value="4" />
        <input id="boundingBoxOn" value="TRUE" />
        <input id="animationPrefix" value="cally_" />
        <children id="puppets">
          <actor id="cal3d" src="../APRIL/components/
              model.apc">
            <input id="src" value="content/cal3dPuppet.
                iv" />
            <input id="visible" value="1" />
          </actor>
        </children>
      </actor>
      <actor id="puppeteer2" src="../APRIL/components/
          puppeteer.apc">
        <input id="id" value="dummy" />
        <input id="hotspotName" value="'spot0','spot1
            ','spot2'" />
        <input id="hotspotCoord" value="-0.2 0 0, 0.2 0
            0, 0 0 0" />
      </actor>
    </children>
  </actor>
</cast>
<behaviors>
    <behavior scene="one">
      <entry>
        <set actor="choreographer" input="command" to="
            'cally: goTo puppeteer(dummy).
            hotspotabsolute(spot0).p 3000'" />
      </entry>
```

```
        </behavior>
        <behavior scene="two">
          <entry>
            <set actor="choreographer" input="command" to="
                'cally: goTo 0 0 0 2000'"/>
          </entry>
        </behavior>
    </behaviors>
    <interactions>
        <event id="12">
          <timeout time="PT2.0S"/>
        </event>
        <event id="21">
          <buttonaction virtual="true" caption="12" auto="
              false"/>
        </event>
    </interactions>
  </presentation>
</april>
```

Bibliography

[1] 3D Studio MAX product website. `http://www.autodesk.com/3dsmax/`.

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

[3] T. Akenine-Möller and E. Haines. In *Real-time Rendering, 2nd edition*. A K Peters Ltd., 2002.

[4] M. Anabuki, H. Kakuta, H. Yamamoto, and H. Tamura. Welbo: An embodied conversational agent living in mixed reality space, extended abstracts. In *Proc. of Conference on Human Factors in Computing Systems (CHI'00)*, pages 10–11, The Hague, The Netherlands, 2000.

[5] ARToolKitPlus website. `http://www.studierstube.org/handheld_ar/artoolkitplus.php`.

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

[7] R. T. Azuma and C. Furmanski. Evaluating label placement for augmented reality view management. In *Proc. IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR 2003)*, pages 66–75, Tokyo, Japan, 2003.

[8] S. Balcisoy, M. Kallmann, R. Torre, P. Fua, and D. Thalmann. Interaction techniques with virtual humans in mixed environments. In *Proc. of International Symposium on Mixed and Augmented Reality (ISMAR'01)*, Tokyo, Japan, 2001.

[9] R. Bane and T. Höllerer. Interactive tools for virtual x-ray vision in mobile augmented reality. In *Proc. of International Symposium on Mixed and Augmented Reality (ISMAR'04)*, pages 52–62, Arlington, VA, USA, 2004.

[10] I. Barakonyi and M. Ishizuka. A 3d agent with synthetic face and semiautonomous behavior for multimodal presentations. In *Proc. of the Multimedia Technology and Applications Conference (MTAC2001)*, pages 21–25, Irvine, CA, USA, 2001. IEEE Computer Society Press.

[11] I. Barakonyi and D. Schmalstieg. Augmented reality agents in the development pipeline of computer entertainment. In *Proc. of International Conference on Entertainment Computer (ICEC'05)*, Sanda, Japan, 2005.

[12] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riss, C. Sandor, and M. Wagner. Design of a component-based augmented reality framework. In *Proc. of International Symposium on Augmented Reality (ISAR'01)*, pages 45–54, New York, NY, USA, 2001.

[13] S. Benford and L. Fahlén. A spatial model of interaction in large virtual environments. In *Proc. of European Conference on Computer Supported Cooperative Work (ECSCW'93)*, pages 109–124, Milan, Italy, 1993.

[14] H. Benko, E. W. Ishak, , and S. Feiner. Cross-dimensional gestural interaction techniques for hybrid imrnersive environments. In *Proc. of Virtual Reality Conference (VR'05)*, pages 209–216, Bonn, Germany, 2005.

[15] M. E. Bratman. In *Intentions, Plans, and Practical Reason*, Cambridge, MA, USA, 1987. Harvard University Press.

[16] A. Butz, T. Höllerer, S. Feiner, B. MacIntyre, and C. Beshers. Enveloping users and computers in a collaborative 3d augmented reality. In *Proc. of International Workshop on Augmented Reality (IWAR'99)*, pages 35–44, San Francisco, CA, USA, 1999.

[17] O. Cakmakci and F. Berard. An augmented reality based learning assistant for electric bass guitar. In *Proc. of the 10th International Conference on Human-Computer Interaction*, Crete, Greece, 2003.

[18] Cal3D website. `http://cal3d.sourceforge.net/`.

[19] M. Cavazza, O. Martin, F. Charles, S. J. Mead, and X. Marichal. Interacting with virtual agents in mixed reality interactive storytelling. In *Proc. of Intelligent Virtual Agents*, Kloster Irsee, Germany, 2003.

[20] N. P. Chandrasiri, I. Barakonyi, T. Naemura, M. Ishizuka, and H. Harashima. Internet communication using real-time facial expression analysis and synthesis. *IEEE Multimedia*, 11(3):20–29, 2004.

[21] A. Cheok, W. Weihua, X. Yang, S. Prince, F. S. Wan, M. Billinghurst, and H. Kato. Interactive theatre experience in embodied and wearable mixed reality space. In *Proc. of International Symposium on Mixed and Augmented Reality (ISMAR'02)*, Darmstadt, Germany, 2002.

[22] A. D. Cheok, K. H. Goh, W. Liu, F. Farbiz, S. W. Fong, S. L. Teo, Y. Li, and X. Yang. Human pacman: a mobile, wide-area entertainment system based on physical, social, and ubiquitous computing. *Personal and Ubiquitous Computing*, 8(2):71–81, 2004.

[23] Coin website. `http://www.coin3d.org/`.

[24] D. Drascic, J. J. Grodski, P. Milgram, K. Ruffo, P. Wong, and S. Zhai. Argos: A display system for augmenting reality. In *Extended Abstract and Video, Proc. of INTERCHI '93: Human Factors in Computing Systems*, page 521, Amsterdam, Netherlands, 1993.

[25] B. R. Duffy, G. M. OHare, A. N. Martin, J. F. Bradley, and B. Schön. Agent chameleons: Agent minds and bodies. In *Proc. of International Conference on Computer Animation and Social Agents (CASA'03)*, New Brunswick, NJ, USA, 2003.

[26] C. Esposito, W. B. Paley, and J. Ong. Of mice and monkeys: A specialized input device for virtual body animation. In *Proc. of Symposium on Interactive 3D Graphics*, Monterey, CA, USA, 1995.

[27] S. Feiner, B. MacIntyre, , and D. Seligmann. Knowledge-based augmented reality. *Communication of the ACM*, 36(7):52–62, 1993.

[28] M. Florence and R. Storey. Vietnam. Lonely Planet Publications, 2001.

[29] FMOD website. `http://www.fmod.org/`.

[30] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[31] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The belief-desire-intention model of agency. In *Proc. Of International Workshop on Intelligent Agents*, pages 1–10, Heidelberg, Germany, 1999.

[32] S. Greenberg and C. Fitchett. Phidgets: Easy development of physical interfaces through physical widgets. In *Proc. of ACM Symposium on*

User Interface Software and Technology (UIST 2001), pages 209–218, Orlando, FL, USA, 2001.

[33] A. Greenfield. In *Everyware : The Dawning Age of Ubiquitous Computing*. New Riders Press, 2006.

[34] M. Gutierrez, F. Vexo, and D. Thalmann. Controlling virtual humans using pdas. In *Proc. of the 9th International Conference on Multimedia Modelling (MMM'03)*, Taiwan, 2003.

[35] Handheld AR libraries website. `http://www.studierstube.org/handheld_ar/`.

[36] G. Hesina, D. Schmalstieg, A. Fuhrmann, and W. Purgathofer. Distributed open inventor: A practical approach to distributed 3d graphics. In *Proc. of ACM Virtual Reality Software and Technology (VRST'99)*, pages 74–81, London, UK, 1999.

[37] P. Horn. Autonomic computing: Ibms perspective on the state of information technology. IBM Corporation, 2001.

[38] S. Julier, M. Lanzagorta, Y. Baillot, and D. Brown. Information filtering for mobile augmented reality. *Computer Graphics and Applications*, 22(5):12–15, 2002.

[39] E. Kaiser, A. Olwal, D. McGee, H. Benko, A. Corradini, X. Li, P. Cohen, and S. Feiner. Mutual disambiguation of 3d multimodal interaction in augmented and virtual reality. In *In Proc. of International Conference on Multimodal Interfaces (ICMI03)*, pages 12–19, Vancouver, Canada, 2003.

[40] 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 Workshop on ARToolKit (ART02)*, Darmstadt, Germany, 2002. IEEE Computer Society.

[41] G. Klinker, T. Reicher, and B. Brügge. Distributed user tracking concepts for augmented reality applications. In *Proc. of International Symposium on Augmented Reality (ISAR'00)*, pages 37–44, München, Germany, 2000.

[42] B. Knep, C. Hayes, R. Sayre, and T. Williams. Dinosaur input device. In *Proc. of Conference on Human Factors in Computing Systems (CHI'95)*, pages 304–309, Denver, CO, USA, 1995.

[43] D. Kotz and R. S. Gray. Mobile agents and the future of the internet. *SIGOPS Operating Systems Review*, 33(3):7–13, 1999.

[44] M. Kruppa and A. Krüger. Concepts for a combined use of personal digital assistants and large remote displays. In *Proc. of Simulation and Visualization (SIMVIS'03)*, pages 349–361, Magdeburg, Germany, 2003.

[45] J. J. Kuffner. *Autonomous Agents for Real-time Animation*. PhD thesis, Stanford University, 1999.

[46] B. Laurel. Interface agents: Metaphors with character. In B. Laurel, editor, *In The Art of Human-Computer Interface Design*, Reading, MA, USA, 1990. Addison-Wesley.

[47] F. Ledermann and D. Schmalstieg. APRIL: A high-level framework for creating augmented reality presentations. In *Proc. of the IEEE Virtual Reality 2005 Conference (VR 2005)*, pages 187–194, Bonn, Germany, 2005.

[48] G. A. Lee, C. Nelles, M. Billinghurst, and G. J. Kim. Immersive authoring of tangible augmented reality applications. In *Proc. of IEEE and ACM International Symposium on Mixed and Augmented Reality 2004 (ISMAR'04)*, pages 172–181, Arlington, VA, USA, 2004.

[49] H. Lieberman. Autonomous interface agents. In *Proc. of Conference on Human Factors in Computing Systems (CHI'97*, pages 67–74, Atlanta, GA, USA, 1997.

[50] J. Looser, M. Billinghurst, and A. Cockburn. Through the looking glass: The use of lenses as an interface tool for augmented reality interfaces. In *Proc. of International Conference on Computer Graphics and Interactive Techniques in Australasia and South-East Asia (Graphite 2004)*, pages 204–211, Singapore, 2004. ACM Press.

[51] D. J. C. M. Sheelagh T. Carpendale and F. D. Fracchia. Distortion viewing techniques for 3d data. In *Proc. of the IEEE Conf. on Information Visualization (INFO-VIS'96)*, pages 46–53, San Francisco, USA, 1996. IEEE Computer Society Press.

[52] B. MacIntyre, J. D. Bolter, J. Vaughan, B. Hannigan, E. Moreno, M. Haas, and M. Gandy. Three angry men: Dramatizing point-of-view using augmented reality. In *Proc. of SIGGRAPH 2002 Technical Sketches*, San Antonio, TX, USA, 2002.

[53] B. MacIntyre and S. Feiner. A distributed 3d graphics library. In *Proc. of SIGGRAPH'98*, pages 361–370, Orlando, FL, USA, 1998.

[54] B. MacIntyre and M. Gandy. Prototyping applications with dart, the designer's augmented reality toolkit. In *Proc. of Software Technology for Augmented Reality Systems Workshop (STARS 2003)*, Tokyo, Japan, 2003.

[55] A. MacWilliams, C. Sandor, M. Wagner, M. Bauer, G. Klinker, and B. Brügge. Herding sheep: Live system development for distributed augmented reality. In *Proc. of International Symposium on Mixed and Augmented Reality (ISMAR'03)*, pages 123–132, Tokyo, Japan, 2003.

[56] P. Maes, T. Darrell, B. Blumberg, and A. Pentland. The alive system: Wireless, full-body interaction with autonomous agents. *ACM Multimedia Systems*, 5(2):105–112, 1997.

[57] C. Magerkurth, T. Engelke, and M. Memisoglu. Augmenting the virtual domain with physical and social elements towards a paradigm shift in computer entertainment technology. In *Proc. of Advances of Computer Entertainment 2004 (ACE 2004*, pages 163–172, Singapore, 2004.

[58] K. Mase, Y. Sumi, and R. Kadobayashi. The weaved reality: What context-aware interface agents bring about. In *Proc. of Asian Conference on Computer Vision (ACCV'00)*, Taipei, Taiwan, 2000.

[59] J. P. M. Massó, J. Vanderdonckt, and P. G. López. Direct manipulation of user interfaces for migration. In *Proc. of International Conference on Intelligent User Interfaces (IUI06)*, pages 140–147, Sydney, Australia, 2006.

[60] Microsoft Speech API website. `http://www.microsoft.com/speech/download/sdk51/`.

[61] P. Milgram, H. Takemura, A. Utsumi, and F. Kishino. Augmented reality: A class of displays on the reality-virtuality continuum. In *Proc. of Telemanipulator and Telepresence Technologies, SPIE 2351*, pages 282–292, 1994.

[62] MLCAD website. `http://www.lm-software.com/mlcad/`.

[63] J. Newman, G. Schall, I. Barakonyi, A. Schürzinger, and D. Schmalstieg. Wide area tracking tools for augmented reality. *Advances in Pervasive Computing 2006*, 207, 2006.

[64] J. Newman, M. Wagner, M. Bauer, A. MacWilliams, T. Pintaric, D. Beyer, D. Pustka, F. Strasser, D. Schmalstieg, and G. Klinker. Ubiquitous tracking for augmented reality. In *Proc. of International Symposium on Mixed and Augmented Reality (ISMAR'04)*, pages 192–201, Arlington, VA, USA, 2004.

[65] M. W. Newman, S. Izadi, W. K. Edwards, J. Z. Sedivy, and T. F. Smith. User interfaces when and where they are needed: An infrastructure for recombinant computing. In *Proc. of the 15th ACM Symposium on User Interface Software and Technology (UIST 2002)*, pages 171–180, Paris, France, 2002.

[66] 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 *CHILetters: ACM Symposium on User Interface Software and Technology*, pages 161–170, Paris, France, 2002.

[67] A. Nijholt, T. Rist, and K. Tuijnenbreijer. Lost in ambient intelligence? In *Extended abstract in Proc. of Conference on Human Factors in Computing Systems (CHI'04) Workshop*, pages 1725–1726, Vienna, Austria, 2004.

[68] T. Nilsen, S. Linton, and J. Looser. Motivations for ar gaming. In *Proc. of Fuse 04, New Zealand Game Developers Conference*, pages 86–93, Dunedin, New Zealand, 2004.

[69] T. Noma, L. Zhao, and N. I. Badler. Design of a virtual human presenter. *IEEE Comp. Graphics and Applications*, 20(4), 2000.

[70] K. Perlin and A. Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *Proc. of SIGGRAPH '96*, pages 205–216, 1996.

[71] W. Piekarski and B. Thomas. ARQuake: The outdoor augmented reality gaming system. *Communications of the ACM*, 45(1):36–38, 2002.

[72] W. Piekarski and B. Thomas. An object-oriented software architecture for 3d mixed reality applications. In *Proc. of International Symposium on Mixed and Augmented Reality (ISMAR'03)*, pages 247–256, Tokyo, Japan, 2003.

[73] Pivy website. `http://pivy.tammura.at/`.

[74] Poseidon website. `http://gentleware.com/`.

[75] I. Poupyrev, R. Berry, J. Kurumisawa, K. Nakao, M. Billinghurst, C. Airola, H. Kato, T. Yonezawa, and L. Baldwin. Augmented groove: Collaborative jamming in augmented reality. *Proc. of SIGGRAPH 2000 Conference Abstract and Applications*, page 77, 2000.

[76] I. Poupyrev, D. S. Tan, M. Billinghurst, H. Kato, H. Regenbrecht, and N. Tetsutani. Developing a generic augmented-reality interface. *IEEE Computer*, 35(3):44–50, 2002.

[77] B. Reeves and C. Nass. In *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*, New York, NY, USA, 1996. Cambridge University Press.

[78] G. Reitmayr. *On Software Design for Augmented Reality.* PhD thesis, Vienna University of Technology, 2004.

[79] G. Reitmayr and D. Schmalstieg. Opentracker an open software architecture for reconfigurable tracking based on xml. In *Proc. of IEEE Virtual Reality 2001 (VR'01)*, pages 285–286, Yokohama, Japan, 2001.

[80] J. Rekimoto. Pick-and-drop: A direct manipulation technique for multiple computer environments. In *Proc. of User Interface Software and Technology (UIST'97)*, pages 31–39, Banff, AB, Canada, 1997.

[81] J. Rekimoto and M. Saitoh. Augmented surfaces: A spatially continuous work space for hybrid computing environments. In *Proc. of Conference on Human Factors in Computing Systems (CHI'99)*, pages 378–385, Pittsburgh, PA, USA, 1999.

[82] O. Renault, N. Magnenat-Thalmann, and D. Thalmann. A vision-based approach to behavioural animation. *Journal of Visualization and Computer Animation*, 1(1):18–21, 1990.

[83] C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics (Proc. of SIGGRAPH '87)*, 21(4):25–34, 1987.

[84] J. Rickel and W. L. Johnson. Steve: A pedagogical agent for virtual reality. In *Proc. of International Conference on Autonomous Agents*, pages 332–333, 1998.

[85] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavári, M. Encarnação, M. Gervautz, and W. Purgathofer. The Studierstube augmented reality project. *PRESENCE - Teleoperators and Virtual Environments*, 11(1):32–45, 2002.

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

[87] E. Sharlin, Y. Itoh, B. Watson, Y. Kitamura, S. Sutphen, and L. Liu. Cognitive cubes: A tangible user interface for cognitive assessment. In *Proc. of Conference on Human Factors in Computing Systems (CHI'02)*, pages 347–354, Minneapolis, MI, USA, 2002.

[88] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[89] B. Shneiderman and P. Maes. Direct manipulation vs. interface agents. Excerpts from debates at IUI'97 and CHI'97. *ACM Interactions*, 4(6):42–61, 1997.

[90] SONY QRIO website. `http://www.sony.net/SonyInfo/QRIO/`.

[91] U. Spierling, D. Grasbon, N. Braun, and I. Iurgel. Setting the scene: playing digital director in interactive storytelling and creation. *Computers and Graphics*, 26(1):31–44, 2002.

[92] C. Stapleton, C. Hughes, M. Moshell, P. Micikevicius, and M. Altman. Applying mixed reality to entertainment. *IEEE Computer*, 35(12):122–124, 2002.

[93] P. S. Strauss and R. Carey. An object-oriented 3d graphics toolkit. *Proc. of ACM SIGGRAPH'92*, pages 341–349, 1992.

[94] M. Stringer, G. Fitzpatrick, and E. Harris. Lessons for the future: Experiences with the installation and use of todays domestic sensors and technologies. In *Pervasive Computing, Proc. of Pervasive 2006*, Dublin, Ireland, 2006. Springer Press.

[95] Z. Szalavári and M. Gervautz. The personal interaction panel a two-handed interface for augmented reality. *Computer Graphics Forum*, 6(13):335–346, 1997.

[96] D. Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, 2000.

[97] The Lemmings Compendium website. `http://lemmings.deinonych.com/`.

[98] TinyXML website. `http://www.grinninglizard.com/tinyxml/`.

[99] B. Tomlinson, M. L. Yau, and E. Baumer. Embodied mobile agents. In *Proc. of International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'06)*, Hakodate, Japan, 2006.

[100] H. Tramberend. Avocado: A distributed virtual reality framework. In *Proc. of Virtual Reality Conference (VR'99)*, pages 14–21, Houston, TX, USA, 1999.

[101] L. Vacchetti, V. Lepetit, G. Papagiannakis, M. Ponder, and P. Fua. Stable real-time interaction between virtual humans and real scenes. In *Proc. of International Conference on 3D Digital Imaging and Modeling (3DIM'03)*, pages 449–457, Banff, AL, Canada, 2003.

[102] D. Wagner, M. Billinghurst, and D. Schmalstieg. How real should virtual characters be? In *Proc. of Conference on Advances in Computer Entertainment Technology (ACE'06)*, page to appear, Los Angeles, CA, USA, 2006.

[103] M. Weiser. Does ubiquitous computing need interface agents? In *MIT Media Lab Symposium on Interface Agents*, Cambridge, MA, USA, 1992.

[104] M. Weiser and J. S. Brown. The coming age of calm technology. In *Beyond Calculation: The Next Fifty Years of Computing*, pages 75–86. Springer, 1998.

[105] C. Welman. Inverse kinematics and geometric constraints for articulated figure manipulation, master thesis. 1993.

[106] J. Wernecke. In *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*. Addison-Wesley, 1993.

[107] J. Wernecke. In *The Inventor Toolmaker: Extending Open Inventor*. Addison-Wesley, 1994.

[108] C. Wisneski, H. Ishii, A. Dahley, M. Gorbet, S. Brave, B. Ullmer, and P. Yarin. Ambient displays: Turning architectual space into an interface between people and digital information. In *Proc. of International Workshop on Cooperative Buildings (CoBuild'98)*, pages 22–32, Darmstadt, Germany, 1998. Springer Press.

[109] WSDL website. `http://www.w3.org/TR/wsdl/`.

[110] XMI website. `http://www.omg.org/technology/documents/formal/xmi.htm`.

[111] XML Spy website. `http://www.altova.com/`.

[112] XPath 1.0 specification website. `http://www.w3.org/TR/xpath`.

Curriculum Vitae

István Barakonyi

Rossauer Lände 41/18
A-1090 Vienna, Austria
bara@ims.tuwien.ac.at

17th May 1977	Born in Zalaegerszeg, Hungary
1983-1991	Dr. Hamburger Jenő Primary School, Zalaegerszeg, Hungary
1991-1995	Zrínyi Miklós High School, special program in mathematics, winner of the Zrínyi Miklós Foundation Award, Zalaegerszeg, Hungary
1995-2000	Studies in computer science at the Budapest University of Technology, Faculty of Electrical Engineering and Informatics, Hungary, major in Multimedia and Communication Networks
Fall 1998	Exchange student scholarship at the University of North Texas, Denton, Texas, USA
June 2000	Graduation "Master of Science (M.Sc.) in Technical Engineering" from the Budapest University of Technology, Hungary, thesis: "Multimedia Demonstration of Object Networks"
October 2000 - March 2002	Visiting researcher at the University of Tokyo, Japan, Ishizuka Laboratory, supported by the Monbusho scholarship of the Japanase Ministry of Education, research topic: "Scriptable Affective 3D Talking Head"
June 2002-2006	PhD course at the Vienna University of Technology and Graz University of Technology, Austria
October-November 2005	Visiting researcher, National Institute of Informatics, Prendinger Laboratory, Tokyo, Japan, research topic: "Using an Unobtrusive Eye Tracker as Input Modality for Remote Collaboration in an Augmented Reality Conferencing System"
October 2006	PhD thesis: "Ubiquitous Animated Agents for Augmented Reality"