TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Diplomarbeit

An Authoring Framework for Augmented Reality Presentations

ausgeführt am

Institut für Softwaretechnik

der Technischen Universität Wien

unter der Anleitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

durch

Florian Ledermann

Matrikelnummer 9426416

Ungargasse 28/I/14, 1030 Wien

Wien, am 12. Mai 2004

Abstract

In this thesis, the design of APRIL, an XML-based language to create content-rich Augmented Reality (AR) applications and interactive presentations, is presented. The state of the art of hardware and software for AR systems is analyzed, to deduce the key concepts and features of APRIL. One central feature of APRIL is the separation of an application's content from the description of the hardware configuration the application should run on. This will allow users to run the same application with different hardware configurations, either reflecting different target platforms, or to replace the original target platform by a simulation environment in the development and testing phase.

While the question of content creation for AR applications has only recently moved into the focus of researchers, for other platforms like interactive CD-ROMs or web applications there is already a number of different approaches to create complex interactive presentations. These approaches are analyzed and compared, and the concept of using UML-statecharts as storyboards for interactive presentations is introduced. For APRIL, the storyboard, represented by a hierarchical concurrent state machine, will be the central document around which a presentation's content is arranged.

A prototype implementation of an APRIL player software, realized by transforming APRIL files into configuration files for the Studierstube AR system, is also presented. With this prototype implementation, several presentations for different AR setups have already been developed. APRIL allowed the authors of these presentations to realize them with much less effort than with a conventional approach, while at the same time providing increased flexibility and debugging possibilities.

Kurzfassung

In dieser Arbeit wird die Spezifikation von APRIL, einer XML-basierten Sprache zur Erstellung von multimedialen, interaktiven Augmented Reality (AR) Anwendungen, vorgestellt. Der aktuelle Stand der Technik von Hard- und Software von AR-Systemen wird analysiert, um daraus die zentralen Konzepte von APRIL abzuleiten. Ein wesentliches Konzept ist die Trennung des Inhalts einer Präsentation von der Beschreibung der Hardware-Konfiguration, auf der die Präsentation ausgeführt werden soll. Dies erlaubt die Ausführung der selben Anwendung auf verschiedenen Hardware-Systemen, die entweder unterschiedliche Zielplattformen oder auch Simulationsumgebungen zum Entwickeln und Testen darstellen.

Während die Frage der Erstellung komplexer Inhalte für AR-Anwendungen erst neulich ins Blickfeld der Forschung rückte, existieren für andere Systeme, wie etwa interaktive CD-ROMs und Web-Anwendungen, bereits mehrere Ansätze zur Erstellung interaktiver Inhalte. Diese Konzepte werden analysiert und verglichen, und die Verwendung von UML-Zustandsdiagrammen als Storyboards für interaktive Präsentationen wird vorgestellt. Für APRIL ist das Storyboard, dargestellt als hierarchisches Zustandsdiagramm, das zentrale Dokument zur Anordnung der Inhalte einer Präsentation.

Eine Prototyp-Implementierung einer Abspielsoftware für APRIL Präsentationen wird ebenso vorgestellt. Mit diesem Prototyp wurden bereits einige Anwendungen für verschiedene AR Systeme entwickelt. APRIL ermöglichte die Realisierung dieser Anwendungen mit deutlich geringerem Aufwand als mit herkömmlichen Ansätzen, während zugleich die Möglichkeiten der Autorinnen und die Unterstützung der Fehlersuche verbessert wurden.

Preface

After many interesting and joyful years as a student, this thesis marks the end point of a phase in my life that was primarily defined by this role. I hope that my meanderings through different disciplines during that time – architecture, computer science, sociolology and design – have contributed to a work and an aggregated competence that are solidly founded in the domain of computer science, but also open up perspectives to other disciplines, and allow to support and integrate specialists in these areas with their knowledge and skills. Throughout my studies, I tried to consequently strengthen the foundations of my own knowledge, while staying in close contact to experts and students of other disciplines. Only on the basis of such social networks of expertise we will be able to go beyond the rhetoric of interdisciplinarity and tackle many of the complex problems that our world is facing.

Being a student is not possible without support. People in generations before mine have fought for opening up universities, and to allow students from different social and economic backgrounds to study for free. Although the achievements of these social visionaries are being partially demolished at the moment, me and generations of future students are still benefiting from their vision and their contribution to a free society, and should not forget to be thankful for that.

I thank my father for supporting me throughout the first phase of my studies, allowing me to find the subject suited for me without the additional economic pressure of having to earn money for my living. Thanks to my mother and grandmother for inspiring discussions, challenging my thoughts yet always respecting my own opinion and style. A big “thank you” also goes to Niki, my girlfriend, for encouraging me in weak moments during the writing of this thesis and always providing a cozy place to hide from the rest of the world in busy times.

I thank my advisor, Dieter Schmalstieg, for giving me the opportunity to work as a project assistant in his group at Vienna University of Technology, despite being still an undergraduate student. Dieter taught me the fundamentals of scientific work, and he is an endless source of ideas and sug-

gestions. Gerhard Reitmayr contributed a lot of helpful hints and suggestions to my work, and I want to thank him for his brilliant analyses of software design and other problems. Thanks also to all the other members of the VRGroup team, namely Tamer Fahmy, Joseph Newman, Istvan Barakonyi, Thomas Psik, Hannes Kaufmann, Thomas Pintaric and Daniel Wagner for making our group an interesting and enjoyable workplace.

Contents

1 Introduction and problem statement **1**

2 Augmented Reality Systems **3**
2.1 Output devices . . . 3
2.1.1 Head-mounted displays . . . 4
2.1.2 Projection-based systems . . . 6
2.1.3 Virtual showcases . . . 6
2.1.4 Sound . . . 7
2.2 Integrating the real and the virtual . . . 8
2.2.1 Occlusion . . . 9
2.2.2 Lighting . . . 11
2.2.3 Surface properties . . . 11
2.3 Input devices . . . 12
2.3.1 Tracking devices . . . 12
2.3.2 Other input devices . . . 16
2.3.3 OpenTracker . . . 18
2.4 User interaction . . . 19
2.4.1 Pointing . . . 19
2.4.2 Gestures . . . 22
2.4.3 Widgets . . . 25
2.4.4 Widget containers . . . 27
2.5 Software systems . . . 29
2.5.1 Studierstube . . . 31

3 Authoring and Storytelling **33**
3.1 Modeling software . . . 34
3.2 Authoring solutions . . . 35
3.3 UML story modelling . . . 41

4 The APRIL language **42**
4.1 Requirements of an AR authoring language 42
4.2 XML technologies 45
4.2.1 Defining the language: DTDs and Schemas 46
4.2.2 Mixing dialects: XML Namespaces 47
4.3 Hardware description 47
4.4 Story modelling 49
4.5 APRIL components 49
4.5.1 Component definition 52
4.5.2 Using components 54
4.6 Animation and behaviours 57
4.7 Interaction 59
4.8 The APRIL workflow 63

5 Implementation **66**
5.1 Transformation and querying: XSLT and XPath 66
5.2 Studierstube configuration 67
5.3 Implementation details 70
5.3.1 XMI to APRIL translation 70
5.3.2 APRIL to Studierstube translation 71
5.3.3 The story engine 76

6 Results **79**
6.1 Scenario: The Heidentor in the virtual showcase 79
6.2 Scenario: A magic book 81
6.3 Scenario: Outdoor tourist guide 83

7 Conclusions and Future Work **86**

A The annotated graphical APRIL schema **90**

B The APRIL language specification **92**
B.1 Global Simple Types 92
B.2 Top Level Elements 92
B.3 Hardware Description Types 93
B.4 Hardware Description Elements 93
B.5 Storyboard Elements 97
B.6 Behaviour Types 99
B.7 Behaviour Elements 99
B.8 Interaction Types 102
B.9 Interaction Elements 102

Chapter 1

Introduction and problem statement

Over the last years, Augmented Reality (AR) has evolved from a pioneering niche in Virtual Reality research to a mature, versatile research discipline in its own right. With this development, interest rises in embedding AR technology in real-life scenarios and applications, instead of the mostly experimental fixtures used for developing the user interfaces or simple application prototypes of the early days. This raises new, interesting questions that were – up to now – not in the foreground of ongoing research: How do we model the detailed, dynamic, real environment of the user of such an AR system? How can we support multiple users concurrently using such a system? How can we create content-rich hypermedia applications, that make use of existing multi-media content and map it into space and time? How do users interact with such a rich media environment in contrast to classical, tool-like applications? How can we create standards that support global sharing and creation of content for these systems?

By asking these questions, we are moving into a new era of AR systems – in parallel to improvements in hard- and software, AR applications will be increasingly focused on *content*, originating from different domains than AR research itself, created to be consumed by a large, untrained audience like tourists, students or customers.

To draw an analogy, the internet was available as a networking platform long before the invention of HTML [15, 56] gave it's users a *language* – a language for the easy creation of information artifacts adequate for the media. Only when it's users could use their computers to create, share and consume ideas in a simple format, the internet would become the global, ubiquitous, multi-medial marketplace it is today. For Augmented Reality to be successful as a platform for content-rich applications, presentations

and ubiquitous systems, we need to give users a language to express their ideas, without having to be programmers for realizing even very simple ideas. AR systems are complex and heterogenous, and rarely we find two systems with identical specifications. Therefore, for special needs we will always need programmers and graphics experts, just as for creating a sophisticated website one wants a graphics designer and probably a programmer on the team. Simple things, however, and especially those the given media *calls* for (like, in HTML, embedding images in a document on a computer screen), should be supported on a core level.

To support users in creating content-rich applications for AR systems, we wanted to create a language especially designed for the needs of this task. To do so, we first have to have a look of the state of the art of AR systems that are in use today. An overview of the hardware and software technologies used in the field is given in chapter 2. Another important point to consider are the current standards for creating content for VR and multimedia presentations, and for structuring this content into interactive presentations that can be explored by the user. Especially the entertainment industry has contributed a lot to the improvement of these tools over the last years, and we will take a look at the present status in chapter 3.

After evaluating the state of the art in the relevant fields, the following chapters 4-6 of this thesis will be devoted to the design and implementation of an AR authoring language and the presentation of example results, created with this authoring framework. Chapter 7 will conclude the main part of this thesis by drawing conclusions and trying to peek into the future of AR authoring systems.

In the appendix, the reader can find the formal specification of the APRIL language, in graphical and textual form. As usual, the thesis is concluded by the bibliography of related work.

Chapter 2

Augmented Reality Systems

As mentioned already, Augmented Reality is rather a paradigm than a concrete hardware or software system. According to Ron Azuma's definition [13], AR systems have to meet three common criteria:

1. They have to combine "real" and "virtual" (computer generated) content.

2. They have to be real-time interactive.

3. They have to be registered in three-dimensional space. This means that the real space around the user defines the context for interaction and presentation.

Note that this definition does not limit AR systems to graphical applications – an AR application may also use only sound, as long as it is spatially positioned and one can interact with it (e.g. by walking around).

As this definition is very broad, there is a wide range of systems to be classified as AR systems. These systems can be primarily classified by the used output media, because this defines the user's impression and therefore to a wide extent the possible applications. Generally, output and input devices can be combined in a relatively independent manner.

2.1 Output devices

Most AR systems use graphics as their primary output media. As the definition cited above requires the registration of an applications content in three dimensions, a realistic, three-dimensional impression of the generated content is desirable in most applications. Recent developments in the PC industry have made powerful 3D-graphics card available to a mass market, but since

we want to blend the virtual content with the real world, pure photorealistic rendering is not sufficient – the computer generated content has to be mixed with the optical impression of the real world to reach our goal.

Humans are capable of stereoscopic vision and perceive the world around them through both eyes simultaneously. The brain merges the two slightly different images of both eyes to a single image of the world, allowing us to perceive the plasticity of 3-dimensional objects and judge distances and spatial relationships with a single gaze. For computer generated content to blend seamlessly with our view of the world, we have to provide two different images for our two eyes, matching the viewpoint of each of the viewer's eyes onto the scene. When these two images are merged in our brain, they will result in a truly 3-dimensional impression of the rendered object.

No matter how the image is displayed to the user, the system has to know the position of the user's eyes to render the perspectively correct images. Therefore, all AR systems require some means of *eye tracking* to be able to realistically blend the virtual with the real. Usually it is sufficient to track only a point on the head of the user and add a rigid, constant offset for each of the eyes. Although this method cannot provide exact measurements, for most applications and tracking devices it is sufficient. For a discussion of available tracking devices, see section 2.3.1.

To support the generation and display of stereoscopic content, and to combine this content with the real world, various systems have been developed and successfully used. Generally, these can be split into head-worn systems, providing a single user with special "glasses" to display the information, and projection based systems, that are located in the environment and can theoretically serve multiple users as a display device, with the drawback that these devices usually cannot be moved and have to remain at a static location in space.

2.1.1 Head-mounted displays

Head-mounted displays are worn by the user on her head, and provide two image generating devices, one for each eye (Fig. 2.1). Since the display surface is located very close to the eye, additional optics have to be provided to move the focal point further away from the user, allowing the eyes to focus on the environment and the overlay at the same time.

For image generation and merging with the real world, two approaches can be distinguished: Optical see-through systems, which allow the user to see through the display onto the real world, and video-based systems, that use video cameras to capture an image of the real world and provide the user with an augmented video image of her environment.

Optical see-through systems

Optical see-through systems use optical image combiners (such as half-silvered mirrors) in front of the user's eyes to blend together virtual and real content. Due to their working principle, not all of the light of the environment will reach the user's eye, resulting in a slightly shaded view of the world, comparable to wearing sunglasses. The computer generated images shown to the user always appear semi-transparent and cannot fully replace or occlude the real world (for a discussion of the occlusion problem, see section 2.2.1).

Figure 2.1: The Sony Glasstron, a widely used stereo capable optical see-through system, as a part of a mobile AR system.

Video see-through systems

Video-based systems, or *closed-view* systems, do not allow a direct look onto the real world. Instead, one or two video cameras at the front of the device are used to capture images of the real world, which are overlayed with the virtual content and then displayed to the user's eyes through two monitors inside the device. By overlaying the video images with the rendered content before displaying both to the user, virtual objects can, in contrast to optical see-through solutions, appear fully opaque and occlude the real objects behind them.

The drawback of video-based systems is that the viewpoint of the video camera(s) does not completely match the user's viewpoint. Although the eyes and brain can adapt to the new situation, for security reasons these systems cannot be used in applications where the user has to walk around or perform complex or dangerous tasks, since judgement of distances is distorted.

2.1.2 Projection-based systems

Projection-based systems use stationary display surfaces in the environment for image generation. These might be ordinary CRT or LCD monitors, or back- or front-projection systems to cover larger surfaces. To provide the user with stereoscopic images, in most cases a frame-interleaved approach is used: The images for both eyes are displayed sequentially, at high update rates (usually above 100 Hz). The user has to wear *shutter glasses*, which are synchronized with the display and block incoming light for each eye in the same alternating pattern as the display shows the image generated for the other eye. Therefore, each eye sees only every second frame, effectively cutting the frame rate of the display in half. Of course, the shutter glasses or the eyes of the user have to be tracked, to generate the perspectively correct images for each eye.

When using head-mounted display systems, the image plane is moving with the user, located at a fixed offset from her eyepoint position. For rendering the virtual content for these displays, usually a simple virtual camera model is used, positioned according to the user's current head location (Fig. 2.2a). In projection-based systems, the image plane remains at a constant position in space, while the eyepoint position of the user changes. To render the images for the user, projection based systems employ an *off-axis* camera model (see Fig. 2.2b), keeping the image plane at a constant location in space (reflecting the position and size of the real image plane of the projection device), while moving the eyepoint of the user.

Another use of projectors in AR applications is to project light onto real-world objects, in contrast to flat display surfaces. This technique can be used to dynamically illuminate real objects in the scene (see section 2.2.2), or to simulate alternate surface texture properties (discussed in section 2.2.3). In both cases, to produce correct results, the geometry of the object(s) to project on has to be known to be able to correctly render the image that is sent to the projector.

2.1.3 Virtual showcases

While the head-mounted AR display devices introduced in section 2.1.1 support the blending of real and virtual content sharing the same space, projector-based systems are generally mutually exclusive: a given region of space is either populated by real-world objects, or hosts a display surface for projecting virtual content on. A class of systems called *Virtual Showcases* [18] tries to overcome this limitation by introducing external, stationary image combiners (usually half-silvered planar mirrors) to merge light from a real

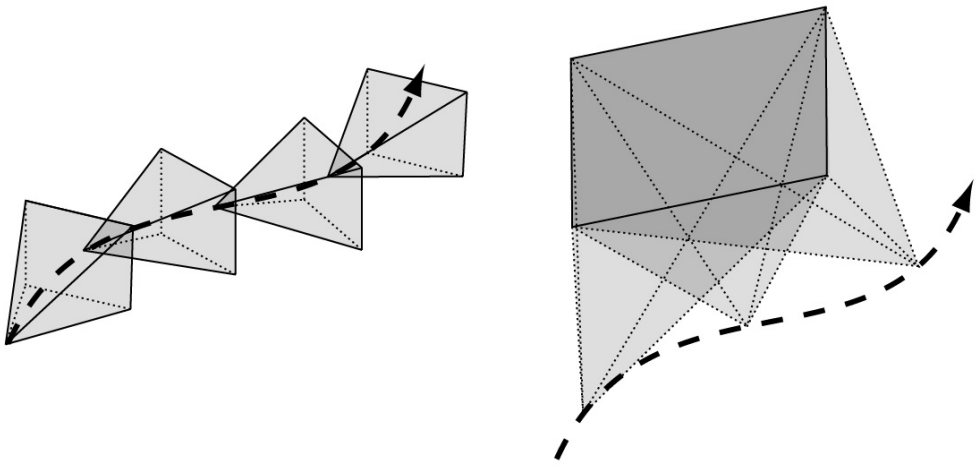

Figure 2.2: Conventional (left) and off-axis (right) camera models. While in the conventional model, a static viewing frustum follows the path of the user, in the off-axis model the image plane and viewpoint can move independently. In this example, the image plane remains static, while the viewpoint follows the user.

object behind the mirror and a computer driven display, which is reflected from the display (see Fig. 2.3).

Virtual Showcases can provide high-quality robust AR experiences within a limited region of space. The volume that can be used is defined by the geometry of the mirrors and the size of the display surface used for image generation – for small systems, CRT monitors can be used, for larger systems video projectors have to be used as image generating devices.

While the visual appearance of presentations using a virtual showcase is usually very good, these systems have an essential drawback: since the mirrors have to be located between the object and the viewer, it is usually impossible to reach into the showcase and interact directly with the real or virtual content. Instead, alternative interaction techniques have to be developed to allow the user to indirectly interact with the presentation.

2.1.4 Sound

While most AR systems and applications focus on visual aspects, sound can be an important component of such installations. Like geometry, sound samples can be placed in the space surrounding the user, and rendered according to her view– (or better: listening–)point. Few graphics packages, like the

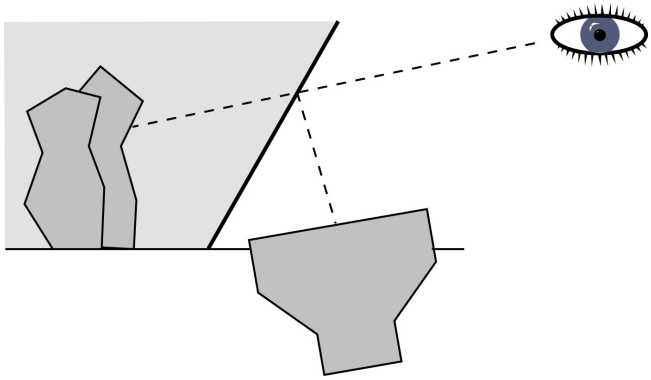

Figure 2.3: Working principle of a virtual showcase system (vertical section). The real object can be seen by the user, while at the same time the scene can be augmented with computer imagery reflected from the half-silvered mirror.

Virtual Reality Modeling Language (VRML) [2]), also support spatial audio with their API, but dedicated packages exist (like OpenAL [7] or FMOD [6]) that can be integrated with a graphics library to perform realtime audio rendering.

For accurate playback of spatialized audio located in the environment of the user, surround speaker systems are necessary. For applications with mobile users it is usually sufficient to use stereo headphones and let the software change the sound volume as the user move nearer or further away from the sound source.

2.2 Integrating the real and the virtual

Besides the mere blending of real and virtual images, special techniques can be applied to improve the realism and therefore the usability and possibilities of Augmented Reality systems. As already mentioned, the display systems alone do not provide any means of handling occlusions and intersections between real and virtual content; lighting of the real world does not automatically match the lighting of the virtual objects, and virtual objects will not cast shadows – a very important cue for the human perception of spatial relationships – onto real surfaces. Some software solutions for these problems have been developed, and will be described in the following sections.

2.2.1 Occlusion

Occlusion or intersections between real and virtual objects will occur in any AR application with reasonably complex content on both sides. Of course, this may not be the case if an application is run in an empty real room, or for very simple applications which only display annotations in front of the user. For other applications, depending on the display technology used, an overview of the visual result of different occlusion situations is given in table 2.1 (after [34], extended).

occlusion order → display system ↓	virtual object occluding real object	real object occluding virtual object
See-through HMD	semi-visible(*)	not supported(**)
Video-based HMD	inherent	not supported(**)
Projection based	impossible	inherent
Virtual Showcase	semi-visible(*)	not supported(**)

Table 2.1: The two possible occlusion orders and the result on various display devices. Results marked (*) can be improved by occlusion shadows, entries marked (**) can be improved by using occlusion phantoms.

The two most prominent approaches to improve the visual impression of occlusion situations are the use of occlusion phantoms or occlusion shadows. For both methods, the geometry of the real world objects has to be known to correctly render these effects.

Occlusion Phantoms

Rendering correct occlusion of virtual geometry hidden or intersected by real objects means simply *not* rendering the virtual content in those places which are hidden – since the display device will show the real world in places where no virtual geometry is rendered, the visual impression for the user will be correct.

If the geometry of the real object is known, we can simply render an invisible representation of the real object – an *occlusion phantom* – into the depth buffer of the graphics card, and render the virtual object after that, using the normal depth-testing algorithms provided by the graphics hardware. The result is the rendered virtual object, minus those parts which are hidden by the real object represented by its occlusion phantom.

Using this technique for static parts of the real world requires the correct acquisition of it's geometry – either by appropriate modeling, using exact measurements for large objects like walls or furniture, or by laser-scanning for smaller, more detailed objects. Occlusion phantoms have also been used to render occlusions between moveable objects or *users* and application geometry [34], requiring a more sophisticated approach which uses a kinematic model of a human body plus trackers mounted to various body parts of the user to dynamically generate a correct occlusion phantom.

Occlusion Shadows

Simulating occlusion of real world objects by virtual content is a more sophisticated problem – we cannot simply "cut away" parts of the real world, at least not with software only solutions [1]. In video-based systems, virtual content is always rendered "on top" of the video image of the real world, therefore we get occlusion support for free – any virtual content will automatically occlude all real content behind it, so we only have to take care about the correct occlusion of the virtual content. With the optical solutions – see-through HMDs and virtual showcases – the user will always see the real world behind the artificial objects as well, since the half-silvered mirrors used in these devices will always let through some of the light reflected from the real objects.

One solution to deal with this problem is to control the lighting of the real objects, so that the parts behind any virtual objects are simply not lit – and therefore will not reflect any light. While it is not possible to completely hide real objects with this technique, it is usually sufficient to give the user a correct impression, given the fact that the darkened part of the object is replaced by virtual content with much higher brightness and contrast, making the dark part "behind" it effectively invisible for human perception.

An algorithm to calculate the light pattern to be projected onto the real world is proposed by Bimber in [17]. The idea of the algorithm is to render those parts of the real objects dark, that are located behind virtual objects from the user's point of view. To be able to project this light pattern, emitted from a hypothetical "spotlight" mounted on the user's head, from a projector at a different location, the light pattern is applied to a virtual proxy of the real object (the same geometry that is used as an occlusion phantom for the occlusion of virtual content) using projective textures, and the scene is then rendered with a virtual camera resembling the projectors optical properties.

[1] And any hardware solutions one might think of will probably work only one time on a given object...

2.2.2 Lighting

Mutual occlusion is probably the most important visual cue for the shape of objects and their spatial relations, but not the only one. Other cues, like lighting and shadows, are used by human perception to gather information about a scene. If these cues are missing or inconsistent, our perception of the scene and therefore the usability of the application will not be optimal.

For the lighting of the virtual content of the application, the application creator has to take care of resembling the lighting conditions of the real world for rendering the virtual content. If lighting of the real world can change dynamically (like, for example, with a light switch or if daylight conditions change in an outdoor installation), ideally this dynamic behaviour is modelled in the application to result in consistent overall lighting.

Projector based illumination

For real objects, the idea presented in the previous section as "Occlusion Shadows" can be taken further and extended to using one or multiple projectors to illuminate the real world, but not only to darken the hidden parts of the scene, but also to create dynamic lighting effects on real objects. Bimber [19] introduces several ideas, like letting virtual objects cast shadows on the real world, or even Augmented Reality radiosity, where the light reflected from coloured virtual objects illuminates the real word in the corresponding colour.

2.2.3 Surface properties

In projector-based AR systems, the projection surface is not limited to flat screens. As we have seen in the projector based illumination approaches, once the real objects geometry is known, its surface can be covered with arbitrary light patterns, correctly aligned with the object, independent from the projectors location.

Raskar et. al. [58] propose an algorithm called *shader lamps* for using projectors to control the appearance of a (neutrally coloured) object – colour, texture and shininess of the object can be controlled by software. With this method, different alternative surface qualities can be simulated on a real object.

2.3 Input devices

Well-established standard computer input devices such as the keyboard or the mouse are practically useless in AR applications – often, users are moving around freely in space, or even roaming through buildings or outdoor areas. This leads to the requirement that input devices must either be ubiquitous, being able to follow the user's input without a fixed spatial location, or wearable, so that the user can carry the input devices with her.

Finding out where the user, her hands or some artifact she is handling is located in space is called *tracking* and is probably the most important type of input to be fed into an AR system. While tracking is often used to determine the context of a user's action (position of the user within the world, viewpoint, position of her hands, etc.), additional input is needed in most cases to trigger an action. Since in most cases a keyboard is not available to issue commands, actions are dependent on simpler (buttons) or computationally more complex (speech recognition, gesture recognition) methods.

2.3.1 Tracking devices

Giving a detailed overview of the state of the art in tracking technology lies beyond the scope of this work. Interested readers may consult Azuma's publications [13, 12] for a broad overview of available technologies and exhausting further references.

Typically, tracking devices used in AR applications deliver data about the *six degrees of freedom* (6DOF) of a tracked point in space: three position coordinates and three rotation components. Some devices also support one or more buttons, whose state information is delivered together with the positional data. Furthermore, there are several properties of tracking hardware that are important to consider for AR applications:

- The *range of operation*. Some devices work only in a given radius from a central unit, for others the targets must be within the field of view of a camera.

- The *update rate*, measured in Hertz (updates per second). For the primary interaction devices, this should ideally match the frame rate of the display, but at least about 15 Hz. Additional information, such as context or environment information, can be delivered with lower update rates, depending on the application.

- The *accuracy* of the measurement, measured in relative (percent) or absolute average or maximum deviations from the correct result.
- The *confidence* whether a tracking target has correctly been identified. This applies primarily to optical trackers.
- Whether the tracked target has to be *tethered* (connected with a cable), or supplied with electrical power (in this case it is called an *active beacon*).
- Finally, also the effort to set up and calibrate the device have to be taken into account when considering different tracking technologies. Some products come pre-calibrated, others have to be calibrated after installation, others even regularly.

Optical Tracking

Optical tracking systems use one or more cameras and advanced computer vision software to detect targets (often called *markers*) in the camera image and calculate their position and orientation information from that camera image(s). The properties shared by optical tracking solutions are untethered targets, reasonable update rates, an angle of operation limited by the field of view of the cameras and the problem of occlusion of the targets by real world objects (including other targets). All systems have to be calibrated, and their accuracy and confidence depends on the quality of calibration, lighting conditions and the cameras and software used.

The most popular tracking technology available today is without doubt ARToolkit [41, 42]. ARToolkit is an open-source software that uses a single ordinary "webcam" to track planar, square-shaped markers usually produced on an ordinary office laser printer. A user-definable pattern inside the marker allows the software to distinguish between different targets, allowing users to build complex applications with multiple tracked interaction devices and artifacts (see Fig. 2.4 for an example setup).

ARToolkit has enabled a whole "generation" of researchers, students and designers to experiment, often using their private hardware, with systems that previously cost hundreds of thousands of Euros to set up in a lab environment. Some researchers have contributed improvements and additions to the software, and up to now[2], two scientific workshops have been dedicated to the presentation of research projects using ARToolkit as a base technology.

Of course, a freely available software running on consumer hardware cannot deliver the results needed in professional lab setups. The accuracy of

[2]February 2004

Figure 2.4: In this setup, ARToolkit markers are used to track the hands and an interaction device of the user.

ARToolkit is impressive, yet low compared to commercial solutions, and the whole system is very sensible to lighting conditions and partial occlusion of markers. Commercial systems are available which deliver improved results for more demanding application requirements.

The DynaSight [9] system delivers robust, accurate position-only tracking information for a single optical target. It uses two cameras enclosed in a box that contains also the necessary processors and embedded software to perform the tracking calculation (Fig. 2.5). The box is connected to the computer with a serial cable, that delivers the tracking data ready to be used by the application. The DynaSight system is designed to be used as a head-tracking device for public installations, where no further targets have to be tracked.

A much more complex and powerful optical tracking system is the ART infrared tracking system [11]. ART uses static configurations of retro-reflective balls as markers and tracks them with multiple cameras (Fig. 2.6). The cameras are equipped with infrared emitters, and record the light reflected from the retro-reflective balls in the camera image. If at least two cameras see a marker (composed of several balls), it's position can be calculated.

By using multiple cameras, the ART system is much more robust against occlusion than the systems mentioned above. Also, the accuracy of the results is higher: carefully calibrated systems can be accurate to one or two millimeters. On the other hand, the calibration effort and the cost of an ART system are much higher than those of the other solutions.

Figure 2.5: The DynaSight optical tracking system. (Image source: Origin Instruments)

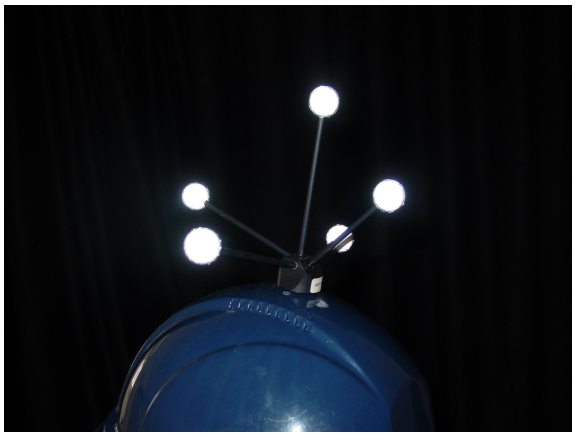

Figure 2.6: A target for the ART tracking system, composed of multiple retro-reflective balls. Note how the flashlight of the camera is reflected evenly from the surface of the balls.

Other technologies

One of the older methods used for tracking is electromagnetic tracking. An emitter generates an electro-magnetic field, which is detected by the electronics in the tracking targets and results in accurate 6DOF tracking information. The main drawbacks with magnetic tracking are that the targets are tethered, and the whole system is very sensible to metal in the environment. This has led to whole labs built out of wood, but of course also monitors or video projectors distort the tracking results.

Ultrasonic tracking devices use measurements of the time that a sonic signal takes to travel from an emitter to a receiver, to calculate the distance between emitter and receiver. For three-dimensional tracking information, three emitters in the environment and one microphone at the target are required. If 6DOF information is needed, multiple microphones on the target must be used to calculate the rotational information as well. Although ultrasonic devices do not need a cable connection between emitters and receivers, the targets have to be equipped with electronics and, in case of a wireless operation, batteries and wireless data transmission facilities. This makes the targets bulky and expensive, which, besides technical reasons, limits the number of targets that can be used in an ultrasonic tracking setup.

Inertial trackers are another class of devices, measuring the acceleration of a sensor in all six (orthogonal and angular) directions. This information is added up in every time step, resulting in the measurement of the current position and orientation of the device. While inertial trackers are of little use on their own (because they accumulate tracking errors rather fast, making the results unusable after a short period of time), they are an excellent complement to an optical tracking setup, to bridge the gaps in the information flow if no marker is visible for a small time span.

For outdoor applications, the global positioning system (GPS) can deliver rough positional data. For accurate positioning, GPS systems have to be accompanied by electronic compasses, inertial and/or optical tracking systems.

2.3.2 Other input devices

As mentioned already, tracking devices are often the most important input devices of an AR setup, but also often not sufficient to cover all requirements of user input. Some tracking devices offer buttons or other input devices such as little joysticks mounted to the tracking targets, but others, especially optical systems, don't even support a single button as additional user input.

This additional input is also a problem when it comes to connecting it to

a computer: nobody wants to write a USB driver, just to connect a single button to the machine. One approach that can support up to eight buttons is to connect the buttons to the data pins of the parallel port. The status of these pins is mapped to single bits in the computer's internal registers, which can be queried directly without requiring driver software to be written. Another approach would be to use the keyboard controller of a modified keyboard to connect the buttons to the keyboard connector.

Of course, in an untethered setup it is not desirable to use cables just for transmitting button information to the base station. Therefore, wireless transmission solutions have been developed, with a base station connected to the parallel port, and wireless buttons that can be carried around by users. Of course, these devices need power supply in the form of batteries, and, eventually, facilities for charging them. As one can see, although sophisticated solutions exist for tracking the position of objects, a simple button click can require quite an effort to be supported in an AR application.

For some applications and user interfaces it is desirable to get 2-dimensional input from the user, mainly used to point at objects or operate 2-dimensional user interfaces based on widgets or menus (see section 2.4.3). The mouse, used in conventional desktop interfaces as a pointing device, is not so well-suited for this task in an AR application, since it relies on a static planar surface to operate – something that will not be available for AR users who move around in the environment. Instead, "trackpads" like the ones used in laptop computers have been used successfully for 2-dimensional user input. They can be mounted in fixed places in the environment or even worn at the user's wrist for mobile applications (see figure 2.8b for an example).

The third basic input data needed in applications is text. Again, the device used in desktop applications – the keyboard – is not at all suitable for the needs of AR settings. However, it is hard to find alternative input solutions, especially ones that are suitable for entering large amounts of text. Speech recognition systems are not (yet) capable to replace keyboard input, at least not without intensive training for a distinctive speaker. Speech recognition systems have been used successfully for issuing short commands out of a limited set of instructions.

Other projects have used gesture-based text input methods, similar to the graffiti system known from Palm PDA devices. While this is a useable method to enter letters with a 2-dimensional input device, it is also not suitable for longer texts.

2.3.3 OpenTracker

With the wealth of different tracking systems and input devices available, it is impossible for application developers to deal with the details necessary to support each and every technology natively in their applications. Instead, it is desirable to add another level of abstraction, and try to encapsulate the details of the necessary software support for various tracking technologies in a tracking *middleware*, serving tracking (and other input) data to the application independent from the underlying hard- and software systems. Support for new devices can then be added to the middleware layer transparently, and all applications using a specific middleware can immediately benefit from improvements or additions made to the tracking subsystem.

Several middleware systems for tracking devices have been developed [72, 69, 68]. One of them is OpenTracker [61, 8], a freely available, research-based open-source system that supports the most common hardware and fulfills the basic requirements of a dynamically configurable tracking subsystem.

OpenTracker is configured by describing a *tracker tree* in an XML configuration file – the tree contains tracking sources (such as markers for an optical tracking system or the electronic sensors of a magnetic tracking system) as it's leaf nodes, and possibly multiple data sinks as top-level elements. Elements between leaf and sink nodes can transform or filter the data. The tree is represented in-memory as a DOM representation of the XML configuration file.

```
<OpenTracker>
  <ConsoleSink comment="pi">
    <Merge>
      <MergeDefault>
        <EventTransform translation="0 0 0.5">
          <ARToolKitSource tag-file="a.patt"/>
        </EventTransform>
      </MergeDefault>
      <MergeButton>
        <StbKeyboardSource number="0"/>
      </MergeButton>
    </Merge>
  </ConsoleSink>
</OpenTracker>
```

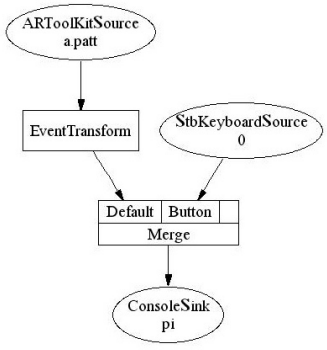

Figure 2.7: An simple OpenTracker configuration file and the corresponding in-memory data structure, the *tracker tree*.

2.4 User interaction

Input data from trackers and other interaction devices is delivered as a stream of events – most tracking devices deliver their data at a constant sampling rate, others have to be queried for their status or – like buttons – deliver events only when their state changes. In any case, these events have to be interpreted by software in terms of *actions* or *gestures* of the user, like, for example, pointing, selecting or dragging.

2.4.1 Pointing

Pointing at objects or locations in space is probably the most fundamental high-level interaction technique used in augmented reality. With a proper technique for pointing available, users can select objects and place or relocate them in the available space. When these objects are *tools*, these can then be used to modify other objects. Pointing is therefore the basic input technique to build interactive, constructive AR applications.

Generally speaking, pointing techniques map points in the space of the input device to points in the space of the application. As these two spaces do not have to match in dimensionality, extent or required/supported accuracy, application designers have to come up with different solutions depending on the available input devices and the requirements and dimension of the application. As the term *pointing* indicates locating something in space, we will require the input device to be at least two-dimensional, and leave out one-dimensional input devices like sliders or up/down buttons, which could theoretically be used to *select* something from a list or a menu.

$\mathbb{R}^3 \rightarrow \mathbb{R}^3$ Pointing

The obvious, trivial case of a pointing technique is a direct 1:1 mapping of input coordinates to application coordinates. If a tracking device with sufficient accuracy is available, and the user can move and point freely in the application's space, this is probably the most intuitive of all possibilities – one can just use the pointing device as one would use it in the "real" world, to touch, select or grab things. Tracking devices such as the Ascension Flock of Birds magnetic tracker or the ART system, but also ARToolkit can all be used in this manner for direct interaction with the application. For improved handling, the tracking target is often mounted onto a pen-, stylus- or "magic wand"-like device, possibly carrying additional buttons or other sensors to allow more complex gestures.

There are generally two circumstances that make it in some cases impossible to use this simple pointing technique: either an appropriate tracking device cannot be used, or the application's space cannot be fully reached by the user to directly point at the desired locations. Reasons for not using an appropriate tracking technology might be practical, such as a limited budget or a temporary setup at another location; It might be technically impossible to use an appropriate tracking device, such as in outdoor applications or in environments that make it impossible to use magnetical or optical trackers. The application itself may make the use of a 1:1 mapping impossible, if the application expects inputs in an area far away from the user or larger than the user's radius of action, or the area of interest may be partially or fully hidden or impossible to reach due to obstacles, that may well be parts of the application itself.

A practical improvement in these cases is an affine transformation of the input coordinate space to the application's coordinate space. Users of graphical operating systems know the idea in two-dimensional space from using a mouse: The input coordinates on the user's desk are translated and scaled to coordinates on the computer's screen. And although conventional mice are not even tracking devices in the sense that they do not really "measure" 2D coordinates on the user's desk, due to the visual feedback on the monitor the mouse pointer can be positioned with amazing accuracy.

If we want to achieve the same thing in three-dimensional space, one of the key requirements has just been mentioned: The user needs visual feedback about her current pointing position in application space, since there is no direct mapping anymore. This holds also true for almost all of the other pointing techniques that will be discussed.

Non-affine transformations have also be successfully used in some applications. The "Go-Go" pointing technique [55] uses non-linear scaling of the coordinates to allow the user to reach objects that are located far away from her. As the user reaches out with her hand, the depth coordinate of the target location is increasingly scaled up, resulting in a pointing technique that allows for high accuracy in positions close to the user, while at the same time allowing her to reach out and point to locations far away, with lower accuracy.

$\mathbb{R}^3 \rightarrow \mathbb{R}^2$ Pointing

Although current tracking technology can provide us with reasonably accurate measurements, it is often very hard for users to choose exact locations in relation to an existing (virtual) object. Therefore, for some applications like CAD programs, it is desirable to *constrain* the user's input [20] to re-

move some degrees of freedom and gain higher accuracy. 3D points received from the input device can be projected onto a plane in application space, so that the 3D input device becomes effectively (and possibly temporarily) a 2D pointing device, that can be used with much higher accuracy. Visual feedback of this projection is desirable, at least the virtual projection plane and the resulting point in application space should be shown to the user while doing her pointing task.

The target coordinate space does not necessarily have to be a plane or a line – target coordinates could also be located on an arbitrary surface, defined by a heightfield, a mesh or a mathematical formula. An important aspect when using such mappings is to give enough feedback to the user so that she can understand and intuitively use the mapping for her pointing task.

$\mathbb{R}^2 \rightarrow \mathbb{R}^3$ Pointing

To perform spatial input with a two-dimensional input device like a mouse, the input coordinate space has to be "blown up" to deliver three-dimensional output in application space. The trivial case is to set the missing coordinate to a constant value, to get a pointing device that operates in a plane parallel to one of the principal planes. If we have an additional, linear input device available to set this constant value, the user can add the missing information and perform "slice-wise" pointing in three-dimensional space.

Of course the target plane does not have to be parallel to one of the principal planes, but can be an arbitrary plane in application space. Again, the distance from that plane could be adjusted by using an additional input device, to allow the user to reach any point in application space by using these two input devices together.

A more sophisticated transformation is the mapping of 2D input data to points on an arbitrary surface. This technique is useful for selecting locations on the surface of an object, or choosing objects that are located on a non-planar surface (like a terrain model). Depending on the topology of the surface, different mappings have to be used to avoid ambiguities. In the case of a terrain model, implemented with a height map, points on the ground plane can simply be transformed by interpolating between neighbour values in the heightmap. For other surfaces, for each point in the ground plane there might be several points on the target surface, so this simple projection cannot be used. Other projections, like mapping the input coordinates to spherical or cylindrical coordinates before doing the projection, may be adequate.

Ray-pointing techniques

One class of pointing techniques proved to be very versatile and intuitive in AR applications: to let the user cast a ray (often visualized as a "virtual laser pointer") into the scene and select the first intersection point with any (or a defined subset) of the application's geometry. Ray-pointing techniques can be further distinguished by the method how the ray is generated from tracking input.

The obvious possibility is to use a 6DOF tracker as a virtual laser pointer. The ray can then be defined by just taking one of the local coordinate axis of the tracking target, transformed by the position and orientation of the target (only the positive or negative half of the axis would be taken to generate a ray that originates from the tracked location). For calculating the intersection point, most high-level rendering APIs provide the necessary functions to calculate the intersection point of a ray with a given scene.

Another way to generate a ray is to use two points in space, and calculate the ray originating in one of the two points, passing through the second one on its way to infinity. While this approach is also useful if no or inaccurate rotational information is delivered form the tracking device, it has one especially useful application: if the head of the user is tracked, then the originating point can be the eyepoint of the user, using a single tracked device to define the second point to be able to "aim" at the target. As for aiming with a gun, the user can use only one of her two eyes for aiming, to get accurate results.

The second point used for aiming does not have to be generated by a three-dimensional tracking device – any of the techniques discussed above can be used to generate an appropriate point in application space. A special case would be to use the *image plane of the user's camera* as the target plane, and use a 2D input device like a mouse for input – creating a 2D crosshair pointing device, well known from video games and desktop VR applications.

2.4.2 Gestures

Gestures are actions performed with a pointing device that are complete and can be interpreted unambiguously by the user *and* the application. While the pointing techniques discussed above generate points that can be used to perform gestures, pointing itself is meaningless unless it is interpreted as a gesture – does the user want to point out, select, delete, move or view an object?

We will look at the most common gestures that are used in AR applications and discuss how they are realized. Out of these simple building blocks,

more complex gestures can be composed by performing several gestures sequentially or simultaneous (e.g. with both hands).

Touching

Touching an object in real life means pointing to a location on its surface. The user will be provided sensory feedback from her fingertips, indicating not only that she has touched something but also reporting some of the object's properties like temperature, softness or surface structure. In addition, a small force will be applied on the object, even with the most delicate touch, and the objects reaction to that force will give further clues – for inanimate objects, the weight may be guessed from that feedback, and for living beings the reaction to the touch might lead to social interaction.

Touching an object in an AR application is much more primitive – in most cases, users will not be able to user their hands or fingers, but use a tracked input device like a pen instead. Virtual objects will give no tactile feedback at all when touched, and for real objects (or parts of them) the application has to know their position and exact shape to detect the touch. Since virtual objects have no physical boundaries, any *intersection* of the objects geometry with the current pointing location will be counted as a touch, and for real objects, inaccurate tracking data or uncalibrated tracking equipment might lead to the application not correctly detecting the touch gesture on the objects surface. Therefore, object boundaries are often extended beyond their visible geometry to provide a bounding geometry (often the object is simplified to a *bounding box*) for detecting intersections with touch gestures.

Since there will be no tactile feedback when touching virtual objects and probably insufficient feedback (and uncertainty about the correct detection of the touch) for real objects, applications have to provide other means of feedback for the user to indicate the possibility of a touch gesture and to acknowledge its beginning and end. "Touchable" areas and objects are often highlighted visually by rendering their bounding box in wireframe or using other additional geometry to give the user hints for which objects it makes sense to touch them. Once the user points to a location inside the object or region, it might be highlighted by rendering it in a brighter colour, increasing its size or rendering a transparent overlay geometry surrounding the object.

A touching gesture lasts only as long as the location pointed to by the user lies inside the geometry or region to be touched. It is therefore often used to display temporary information like additional information about the object (known from desktop applications as "tooltips"), or just highlight it to indicate the possibility of other gestures like selecting or dragging.

Selecting

In contrast to touching, *selecting* an object persists until it is explicitly undone – either by de-selecting the object or, in some applications, by selecting another object. Selecting an object means to set the focus of the application to a given object, to be able to perform additional operations on it. Usually, selecting an object is done by touching it (moving the pointer inside or close to it) and then performing an additional action like pressing a button on the input device.

Applications may allow multiple objects to be selected simultaneously, resulting in a *selection set*. If an application supports selection sets, user interaction becomes more complicated, since the application has to provide the user with the ability to add or remove objects from the set. In desktop applications, this is often provided by using modifier keys (like holding the "Control" key to select multiple objects). In AR applications, there is usually no keyboard available, so for more complex selection policies multiple buttons have to be fitted to the input device, which in turn makes the system more complicated and harder to learn.

Triggering

Triggering gestures perform some action associated with the touched object. On conventional desktop systems, triggering gestures are realized, depending on the object type, as single clicks (for buttons) or double clicks (for icons representing files) on the corresponding object. This mapping can also be used for AR applications, although the distinction between different object types (and especially something like "files") is not so well-established in these applications. Therefore, most often only a single click is used to perform an action on an object, which conflicts with the selection gesture discussed above.

To resolve this conflict, the application must use either only one of these gestures, or clearly distinguish objects that can be selected from objects that can trigger actions. One approach, also chosen in conventional desktop applications, is to use only special, visually distinguishable objects (called *widgets*) to trigger actions on other objects, that may be selected by the user.

Dragging

When we move around objects in the real world, natural forces like gravity or inertia give us important feedback during this process. In a virtual world, moving objects has to be made more explicit to avoid ambiguous situations

and interpret the user's actions correct. Moving, or *dragging* an object is often composed of a drag-start-gesture (selecting the object the user wants to move and indicating that it should be moved by a subsequent gesture), a drag-gesture (actually moving that object, pointing to its new target position), and a drag-end-gesture (releasing the object and placing it at the desired target location). Like pointing, dragging can be constrained to a subspace of the area surrounding the user (possibly a plane or a line), or to specific "valid" target positions – if the object is dragged elsewhere, it might "snap" to the nearest valid position or return to its origin.

Usually, dragging is realized by using a button on the input device. By pressing (and holding) the button, the user indicates that the object she is currently pointing at will be moved. The button is then held down during the whole dragging process, with the dragged object following the pointing position of the user. When the button is released, this indicates the end of the dragging procedure, and the system can check whether the object has reached a valid end position and adjust if necessary.

Other implementations use a "magnetic" pointing device that automatically picks up objects that are close to it, and a gesture to release the object, like holding the pointing device in a certain angle or performing a rapid "shaking" gesture. For this technique, no additional button is needed to enable the user to move objects around, at the cost of decreased accuracy.

Drag & Drop

If, depending on the release position of a dragged object, an action is performed when the dragged object is released, this process is called *drag & drop*. The dragged object can be released near a *drop target*, which in turn is notified of the drop action and can perform the necessary actions.

This gesture is also well-known from desktop operating systems, especially from the Macintosh platform, where it is widely used. The most prominent drag & drop operation is probably the deletion of files by dragging the icon representing the file to an icon representing a trash can. Similar techniques have been implemented in AR applications to manipulate content.

2.4.3 Widgets

So far we have discussed only actions that the user can perform on objects of the scene surrounding her. *Widgets* are special, standardized objects that allow more specific interaction with the application by interacting with them. Usually, the gestures performed on widgets are simple (clicking and dragging), but the widget establishes a context for these gestures, determining

the commands sent to the underlying application.

Buttons

A Button is a button is a button. As in reality, buttons can be pressed to trigger certain actions in the application. In addition, buttons may toggle their state with every press (checkbox behaviour) or the press on one button may automatically release other buttons of a group (radiobutton behaviour).

Menus

Menus are used in desktop operating systems to issue commands to the underlying program. Due to their hierarchical structure, a lot of different commands can be offered to the user without overloading the display area.

In VR and AR systems, menus have been less successful – being inherently 2-dimensional, users have difficulties navigating through layers of menu items that change dynamically. With heads-up displays (see section 2.4.4), conventional menus can be used. In true AR applications, the menus would float freely in the space around the user, and are only usable with constraints on the pointing device.

Some projects have used the fingers of the user [21] or multiple physical buttons on an input device to establish a hierarchy of menu items. While this is a feasible approach, it limits the number of menu items and requires additional tracking hardware to locate the hands or input device of the user.

Numeric Input

Buttons and menus allow the user to generate *events* which are sent to the underlying application to trigger a specific action or change its state. For the input of data, other widgets have to be provided.

An important class of input widgets in conventional desktop applications are text input fields – an area for entering text with the keyboard. Since most AR systems do not offer a keyboard to the user, the widgets supporting numerical and textual input have to be different.

For numerical input within a given value range, sliders are often used. Sliders display a handle, that can be manipulated through a constrained dragging gesture to adjust the value that the widget represents. Some sliders also feature two buttons, that allow the stepwise increment or decrement of the value.

Dials are another example of numerical input widgets. With a dial, the value is adjusted by turning the widget, like an adjustment knob on a (real) HiFi system. The turning gesture is often more difficult to perform, therefore

some systems translate a dragging gesture performed on the widget into a turning movement of the dial.

The advantage of dials is that they use less space, while not being quite as legible and easy to operate as sliders. They are therefore mostly used for adjustments that are not changed during normal operation of the system.

Text Input

As discussed already, text input is one of the most difficult input types to support in an AR application. Some widgets to support textual input have been proposed, but generally, programmers try to avoid the necessity of text input in their applications.

One naive approach is to render a virtual keyboard, composed of multiple individual button widgets. While this interface is well-known to most users, it is cumbersome to handle even with an accurately tracked input device, and impossible to operate if the tracking error of the input device is larger that the size of a single key.

Another widget that has been proposed is an area that can be used for "graffiti" gestures. The graffiti system is known from palm PDAs, and allows the user to enter letters by performing drawing gestures on a 2D surface. Again, this is only possible with a sufficiently accurate tracked input device.

Without powerful speech recognition systems, text input will remain a problem for all AR and ubiquitous computing applications in the near future.

2.4.4 Widget containers

The main problem that occurs when using widgets in an AR scenario, is that the user often is offered too few spatial cues to be able to operate these widgets accurately and precisely. Buttons and menus could theoretically float freely in the space surrounding the user, but she will have problems to correctly identify, locate and operate these widgets if no further spatial cues are given by the application designer. To solve this problem (or at least improve the situation), various approaches have been proposed, all leading to an improved spatial context for 3D-widgets.

Heads-up displays

Heads-up displays, or HUDs, use the image plane of the view of the user as a reference plane to display GUI elements. A HUD establishes a 2D user interface, floating in front of the user. Therefore, widgets used in a HUD

are not 3-dimensional, but very similar to conventional 2D widgets used in desktop operating systems.

HUDs have been used successfully in VR scenarios like computer games and design applications. For AR, the different focal plane of the HUD is a source of confusion for many users. While HUDs look convincing for desktop simulations and video-see-through AR applications, more research has to be conducted to make them fully usable for "true" optical see-through AR applications.

Personal Interaction Panel

The Personal Interaction Panel (PIP) [71], developed at Vienna University of Technology, provides a tactile reference surface to arrange widgets in AR applications. Basically, the PIP consists of a small tracked panel, that is held by the user with her secondary hand. The AR system can then overlay the PIP with the widgets that are offered to the user by an application (Fig. 2.8).

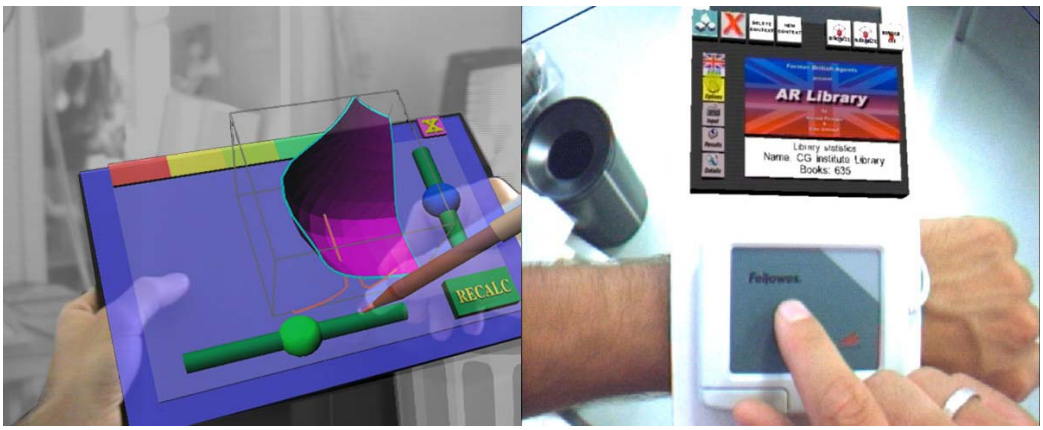

Figure 2.8: The personal interaction panel. The version on the right is using a wrist-mounted trackpad for 2-dimensional pointing input.

Since the space on the PIP is limited, multiple layers of widgets can be arranged on PIP sheets. A PIP sheet should carry widgets that are thematically related, and by switching sheets the user can control different widget groups for different tasks. One example would be a preferences sheet to control various settings of the application, and an application sheet to be used while actually working with the application.

2.5 Software systems

Most AR applications known so far were prototype implementations, rapidly implemented on top of a basic graphics framework to fulfill a relatively narrow-defined purpose or to prove a certain point in research projects. The few more complex software systems developed for augmented reality [66, 73] have focused on providing basic services for such applications, like user management, device configuration and distribution, amongst others. Like conventional desktop operating systems, AR systems offer to applications a set of services to use, and often a set of user interface elements and concepts for a consistent look and feel across applications. In conventional operating systems, although there are still notable differences between various brands, certain principles like the desktop metaphor or WIMP[3]-based graphical user interfaces are used successfully across brands and implementations. For AR-systems, no such standards have evolved yet, and every system uses its own metaphors and concepts for user interaction and application management.

One naive, but nevertheless partially successful approach is to try to simply take the WIMP set of interaction tools and the desktop metaphor into the third dimension. Some systems provide 3-dimensional "windows" [66] (rendering the original metaphor absurd, since 3-dimensional "windows" are actually cubic regions of space, instead of two-dimensional viewing regions), others make use of three-dimensional menu systems. Although one might think that the more realistic modelling of user interface elements and the more "natural" spatial representation would be more intuitive, users find it harder to control applications in three-dimensional space – with desktop operating systems, the screen and the strictly two-dimensional input device give a reference frame for the user's actions, which is completely missing in unconstrained 3D environments. A lot of user interface research has therefore focused on introducing *constraints* [51, 20] to give the user orientation and reference in using the system.

AR applications have been developed for a lot of different use cases, but one can identify a few scenarios that have traditionally been used as testbeds for AR systems: The first systems that qualify as AR systems have been used in maintenance and engineering scenarios, to support engineers in complex assembly tasks with additional, location and context-dependant information. Today, engineering scenarios have remained one of the main usecases for AR applications, with the focus shifted from pure information providing to more interactive design and prototyping systems [46, 29, 32].

[3]Windows, Icons, Menus, Pointer. The basic set of user interaction elements used in most 2D graphical user interfaces.

The early AR-based information systems also evolved into a second strand of applications, focussing on annotation of the user's environment. These systems often support multiple users, and some groups have built systems that support outdoor usage [60, 54]. Annotations of objects and regions in the real world can be text overlays or sketches, but current systems often support the full range of multimedia content types, including audio and video material.

Another topic for AR research is the support for office work – be it in the area of communication, where novel videoconferencing solutions have been proposed by AR researchers that allow users to see their partners as part of the environment [43, 30], or by integrating projector-based AR applications into an office workplace [57].

Medical applications have been another fruitful field for AR researchers. Especially in the areas of surgery there is big demand for the integration of context- and location dependent information systems to improve surgical processes. X-ray and ultrasonic scans of the patient can be overlaid in the view of the surgeon [70, 59], and external experts can be consulted during the operation without leaving the context of the current task.

Finally, the potential of AR technology to support education and teaching has long been recognized and actively researched [44, 63, 50]. Because of its ability to add abstract information to real-world objects, AR technology seems to be ideally suited as an accompanying technology in educational applications across domains. However, in contrast to most applications mentioned above, educational applications have different requirements on the AR system: While for most of the task-oriented scenarios, the AR application is like a *tool* that can be used in a given task in a relatively context-free manner, educational scenarios require a more persistent support and guidance by the application – one pupil is unlikely to repeat the same learning lesson more than a few times, but should be guided and supported throughout a possibly long-term learning process that may consist of many different tasks, lessons and even topics.

Few AR systems have yet recognized or tackled these issues. Traditionally, researchers in the area of *pervasive computing* are used to deal with more persistent, stateful applications that are "always on". However, to create successful learning and presentation applications, it will be important to incorporate ideas coming from these areas, to allow more complex presentations which possibly run over longer periods of time and guide users through a sequence of different scenes.

2.5.1 Studierstube

Studierstube [66, 67] is a software system developed to support AR systems and applications. Based on the Open Inventor realtime rendering framework and the OpenTracker input device abstraction middleware (see section 2.3.3), Studierstube allows developers to create complex, distributed AR experiences for multiple simultaneous users.

Studierstube can make use of various output devices through a generic, configurable off-axis camera model that can be extended with video background rendering. Therefore, all rendering devices discussed in section 2.1 can be used, and even mixed, in a single Studierstube setup. Configuration of the whole setup and the applications is accomplished through files conforming to the ASCII-based Open Inventor file format, and through XML-based OpenTracker configuration files. The whole Studierstube system can therefore be configured and adapted to a wide range of application scenarios without recompiling.

For user interaction, Studierstube provides a set of 3D widgets (buttons, sliders, lists, ...), that can be operated with a tracked pointing device. To give the user a reference for operating these widgets, and to add a physical constraint to the interaction, widgets are usually arranged on the Personal Interaction Panel (PIP) (see section 2.4.4).

Applications are implemented as custom nodes in the global Studierstube scene graph. Applications are composed of an `ApplicationKit`, which act as entry points for starting an application and contain an image to represent the application, a text describing the application and the actual class implementing the applications behaviour – the `ContextKit`. While the `ApplicationKit` is usually used as provided and can be compared to an applications *icon* in conventional desktop operating systems (an entry-point for identifying and launching the program), application programmers will usually subclass the `ContextKit` class to implement their own applications behaviour.

A `ContextKit` will define a `PipSheet` for the application, containing the widgets and interaction tools to be displayed on the PIP while the application is active. The `ContextKit` can also contain one or multiple "windows", which are cubic regions of space containing the applications geometry. If no windows are used, the applications content may fill the whole space surrounding the user.

Distribution of applications to multiple hosts and users is accomplished through the underlying *Distributed Open Inventor* (DIV) [39] implementation. DIV allows the distribution of parts of the scenegraph of a running Studierstube host to other hosts, transparently for the application devel-

oper. While some of the implications of using a shared scene graph cannot be completely hidden away from the programmer or even the user (like policies of dealing with concurrent or contradictory user requests), DIV allows the creation of complex, distributed applications without dealing with all of the complexities of realizing such a system.

While Studierstube offers a lot of support to the application programmer, it does not provide any scripting mechanism. The application's logic has to be encapsulated in a compiled subclass of `ContextKit`, and the programmer has to take care to keep track of all of an applications state and it's reaction to user input. Creating reasonably complex applications for the Studierstube system therefore requires advanced knowledge of the C++ programming language and the underlying support APIs like Open Inventor, OpenGL and ACE.

Chapter 3

Authoring and Storytelling

Authoring generally refers to a development process that is not focused on programming, but rather on arranging multimedia content to create a content-rich application. In recent years, we have seen authoring solutions for web applications, that allow the creation of complex websites or interactive movies even by non-programmers. Educational applications and interactive CD-ROMs are also domains where authoring solutions have successfully been used to create content-rich applications.

To create convincing and exciting presentations in Augmented Reality, we want to guide the user through a plot or a story that she can influence by her actions. Throughout history, there has been a dispute about what is the driving force behind a story – the *plot*, created by an author, or the *characters* who bring the story to life. Beckhaus et. al. [14] quote some historic positions on this dispute:

> Aristotle in his Poetics claims that '*the plot is the first consideration, and as it were, the soul of the tragedy. Character holds the second place, ...*' [10]. The opposite is claimed by Egri: '*The interference is unmistakeable: character creates plot and not vice versa*' [28].

While this dispute might continue in literature theory, for our purposes of creating an authoring system for interactive, yet deterministic presentations, the focus seems clear: We want to enable the author to create a reliable *plot* for the application, and not have her depend on the moods of her characters.

The term *storytelling* has been used in a lot of different contexts, and has recently become quite fashionable amongst computer scientists and VR researchers. For our purposes, an application that makes use of *AR storytelling* has to fulfill the following criteria:

1. It has to consist of several scenes. A scene is an atomic unit of the story with finite duration, in which one or more actors (media objects) expose some sort of behaviour.

2. The scenes should be modelled explicitly by the author (in contrast to character-driven approaches where the actors drive the story). As mentioned above, we want the author to be in control of the presentation's overall plot.

3. Depending on the user's actions, the development and outcome of the story will vary.

As mentioned in section 2.5, most AR applications and systems that exist today are designed as more or less stateless *tools* that do not provide the features we need to create compelling stories. This is partially because the AR software systems available do not offer the necessary support for stateful, interaction-driven presentations.

3.1 Modeling software

Within the last years, computer games and digitally enhanced movies have set new standards for computer graphics, and the visitors expect similar quality even in more constrained, real-time rendering environments. Together with the results, the quality of the tools available for 3D modelling has increased a lot, and as there are people specializing in 3D modelling as their profession, there is a big market for modelling software and add-ons. For any new authoring framework to be successful, providing a smooth workflow by integrating these tools is crucial.

Support for real-time rendering file formats like Open Inventor is still limited in commercial modelling tools. The only modelling tool that natively supports the Open Inventor file format is CosmoWorlds - a now discontinued product from SGI. Since SGI stopped the development of VRML and CosmoWorlds (which was mainly used for authoring VRML files) already several years ago, it does not include state of the art modelling or animation features that we need for AR presentations. Other software packages - like the widely used Maya or 3DStudio Max packages - offer state of the art modelling, but are not optimized for a real-time graphics framework like Open Inventor.

3.2 Authoring solutions

For authors and professional scriptwriters, there is software available to assist them in storywriting by structuring the authoring process and providing inspirational tools for evolving the story [5]. These tools focus on the literary side of storytelling and support the author in generating text. However, what we need for presentation authoring is not a support for writing text, but a tool to structure and modularize the story, possibly in a non-linear way that allows multiple story paths to be explored by the user in an interactive setting.

Traditionally, complex presentation applications have been developed by using two tools: timelines and scripting. While a timeline keeps track of the linear flow of events and the evolving of the story over time, scripting can be used to add user interaction, random events and conditional branches in the story (jumping from one timeline to an alternative one). Authoring tools for multimedia presentations have used these features to allow their users to create complex interactive presentations.

The Alice system [25] was designed with novel computer users and students in mind. It allows people with no prior knowledge of computer programming to create animated 3D scenes from existing building blocks. The idea behind using a 3D environment to teach programming is compelling: A virtual world is inherently stateful, and is therefore ideally suited to visualize the state of a program controlling this world. The creators of Alice use the "key feature" we are looking for as a debugging tool for simple programs. Alice offers very limited interaction possibilities, and no way to structure the story into multiple parts or scenes. Stories created in Alice are simple programs or scripts, that execute linearly and perform the actions chosen by the user.

Examples for software that supports non-linear stories are Authorware [3] and Director [4], both by Macromedia Software. While Authorware supports the idea of an abstract graphical representation of the flow of the story (see Fig. 3.1) to create educational applications, Director exposes a theater metaphor (as already indicated by its name), which enables the user to put members of the *cast* onto a *stage*, where they act according to a *score* created by the user.

While the theater metaphor of director is compelling, story modelling is based on linear scenes with a single timeline, and more complex behaviour can only be realized by scripting the application in its own scripting language called "Lingo". Authorware offers an interesting model of the presentation as a flowchart diagram, however the system is limited to only a single active scene at a time and cannot be extended by scripting at all. Being a tool for

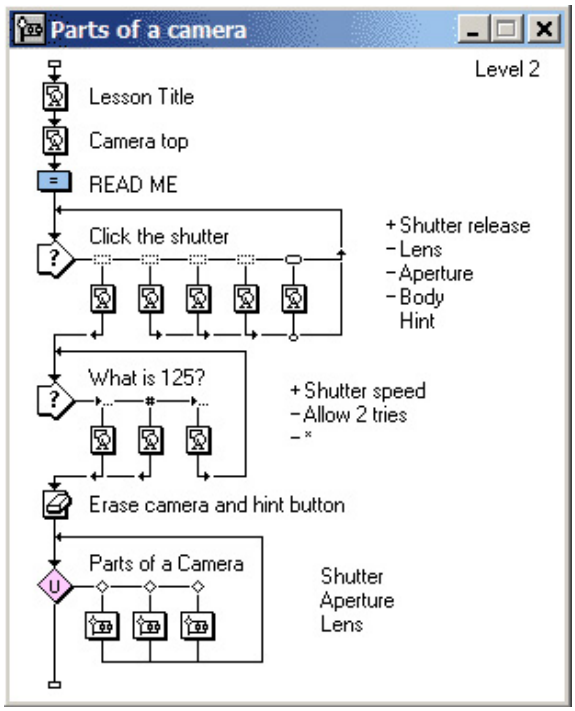

Figure 3.1: The flow of a presentation in Macromedia Authorware.

educational screen presentations, interaction is also limited to conventional 2D interaction techniques and text input.

VR and AR researchers also explored the possibilities to create more complex presentations in recent years. The Virtual Reality Modeling Language (VRML) [2] offers the possibility to create new, custom nodes (so called PROTOs) and to use JavaScript [27] to control their behaviour. For more complex tasks, developers can implement the desired functionality in Java and link their PROTOs to the compiled classes. However, this remains a low-level approach where all content has to be created from scratch by a programmer. While the PROTO mechanism allows the creation of re-usable components, no mechanism is provided to keep track of the overall state of a presentation or to control interactions and animations from a central point.

Generally speaking, scripting solutions are a powerful tool to support the rapid prototyping and development of complex applications. However, there are two main drawbacks that make scripting a less desirable solution: First of all, only people with sufficient programming skills can use scripting to realize complex processes. It might be feasible to teach non-programmers how to accomplish very simple tasks with scripting, but we are looking into complex,

non-linear and interactive applications that will be too complex for novice programmers to realize. The second drawback also applies to professional programmers: Scripted applications usually have very poor explicit structural information. The state of the application is hidden inside variables and data structures, and there is no explicit model of the application's states and the user's possibilities. Therefore, scripted applications are hard to debug, and round-trip communication between programmers and designers can be very tedious.

The Virtual Reality Slideshow System (VRSS) [35] adresses this issue by providing a set of Python macros for the creation of virtual reality slide shows. VRSS Slides can contain text, images and objects created in the VRML file format. The macro language allows to define the sequence of slides for the presentation, and the transition effects that should be used for each slide. VRSS only supports linear presentations, but it could be easily extended to support loops and branches in the execution. All the interactive content within a slide (or scene) has to be provided in the VRML file format, sharing the drawbacks of this relatively low-level format.

The concept of *presentation scenarios* is introduced by the authors of VRSS, allowing the system to support several setups including frontal presentations, multi-user collaborative scenarios or mixed (collaborative users with spectators) settings. This set of scenarios is fixed, and is not generalized into a generic setup description part.

Other groups have used conventional slideshow creation software like Microsoft Powerpoint to create virtual reality content. In the PowerSpace system [38], Powerpoint presentations are exported to a simple XML-format, defining the slides of the VR presentation. Similar to VRSS, PowerSpace allows only linear presentations, with the possibility to further divide each slide into a sequence of animations, which make additional elements of the slide visible. The 2D-animations, provided by Powerpoint, are not preserved in the VR presentation, but replaced by simply hiding or showing the corresponding element.

The XML file created with Powerpoint is then loaded into the PowerSpace viewer software. Elements of the XML file (such as slides, text or shape objects) are instantiated as DOM nodes and transformed into an Open Inventor scene graph for viewing. The PowerSpace system creates VR presentations, that can also be played back on AR systems, but doesn't take into account the special requirements and features of such systems.

The Designers Augmented Reality Toolkit (DART) [49] is one of the few systems specifically targeted towards Augmented Reality presentations. DART uses Macromedia Director as a base system, and provides the necessary extension classes and utilities to make it possible to create AR presen-

tations in Director. The advantage is that designers can use a tool that is widely used and professionally supported, and is traditionally used by web- and multimedia designers. Unfortunately, at this time an evaluation of the DART software is not possible, but from the published material one can get a good impression of the key properties of this authoring solution. With the use of Director as a base system, developers get timeline and animation support, and a powerful scripting solution to realize complex processes. However, it remains unclear whether DART can support the full range of AR input and output devices. As stated above, scripting is a feasible solution for programmers and specialist designers, but prohibits novice users from creating complex presentations. In addition, for complex projects, it is hard to keep the overview and organize communication with domain experts and other designers.

One key feature of DART is that designers can use *informal content* like sketches or textual annotations in the early phase of their project. This supports the incremental prototyping of a presentation, developing and improving ideas for the final content as the project develops, without having to produce labour-intensive content before testing the application. The informal content can also be used as a communication artifact for teams of designers, or to communicate with external people like content creators about the ideas of the presentation.

Within the Geist project [47], a detailed theory of Computer Supported Collaborative Interactive Stroytelling (CSCIS) has been developed out of literature theory. Based on a Prolog engine, several layers of abstraction control the evolving story and the user's interaction possibilities. Rendering this story to an AR system is only one of many output possibilities of the story engine. While the theoretical background and the complexity of the Geist engine is compelling, the system lacks support for the creation of simple stories by non-experts, and for support of AR specific features for authoring and playback of the created presentations.

Another tool that supports the creation of 3D hypermedia narratives is the MARS authoring tool developed at Columbia University [37]. MARS follows more closely the hypermedia paradigm of interlinked media artifacts, which can be distributed in a space to be explored by the user. MARS focusses on supporting non-programmers in creating narrative multimedia experiences.

MARS clearly distinguishes between an *authoring component* and a *presentation component*. The authoring component is used to create the presentation on a desktop system, simulating the space the user will later explore using the portable presentation component. Following the hypermedia paradigm, presentations are composed out of *snippets*, which encapsulate

media content and can be arranged to *clips* and *icons*, which is a visual representation of the media content that can be placed in the scene. Clips are interconnected by hyperlinks, allowing the author to guide the user through a non-linear story and reference related information.

MARS uses an extended version of the Contextual Media Integration Language (CMIL) [26], an XML-based markup language, to store presentations created with its authoring component. Similar to HTML, individual clips are represented by individual files, which are interconnected by hyperlinks. The creators of MARS added proprietary extensions, like transition effects and the specification of a snippets location in world space, to CMIL.

While MARS offers excellent high-level support for authoring hypermedia documentaries by non-programmers, it has a few drawbacks. First of all, only one specific AR setup is supported to be used as a presentation component for the finished presentation. The presentation is structured implicitly by interlinked files, which will make it hard to keep the overview with complex projects. On the content side, MARS is focused towards multi-medial content, and supports the playback of images, videos and sounds, but does not provide mechanisms for controlling animations or virtual actors that are added to the scene.

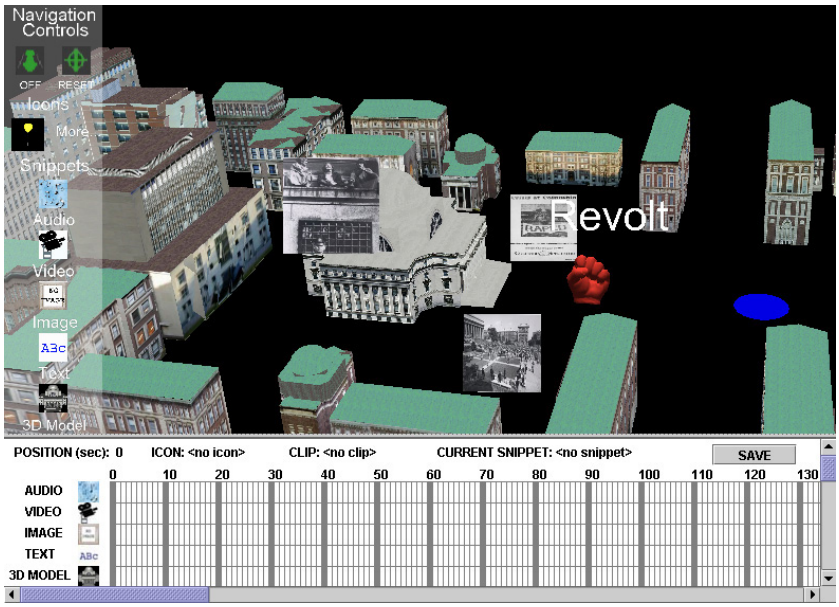

Figure 3.2: The MARS authoring toolkit. (Image © 2003 Sinem Guven and Steven Feiner, Columbia University)

Zauner et. al. [74] present an authoring wizard for AR assembly instructions. Although the usage scenario is limited to creating interactive, mixed

reality assembly instruction, the software uses some interesting concepts that may be helpful for general authoring scenarios. A state engine is used to represent the assembly process, allowing for non-linear navigation through the presentation, according to the user's needs. Presentations are created by putting the parts together in the right way, identifying parts and their correct relationships with a 6DOF pointing device. The presentation is therefore created in the same space as it will be played back.

The AlVRed project [14] aimed to create tools for storytelling in virtual environments. Beckhaus et. al. discovered the need for a structured approach to storytelling, and propose the use of Hierarchical Finite State Machines (HFSM) as a conceptual model for interactive presentations. This conceptual model can be used to discuss the intended story with domain experts, designers and programmers and to make adjustments before the content of the actual presentation has been created.

State machines can be modelled in a standardized way by using the Unified Modeling Language (UML) [52]. Other projects [64, 65] have used UML to model multimedia presentations, but none of these projects has applied these ideas to VR or AR presentations.

After reviewing these systems, we get an idea of the authoring process that we are looking for. First of all, we want a solution that does not rely on scripting to create complex presentations – we are aiming for novice users, designers and other domain experts as users of our system, and we want to enable them to create at least prototypes of the planned presentations without having to consult a programmer. The Alice system is geared towards novice users, and offers them building blocks that can be arranged to create simple, yet meaningful scenes with a visual interface. Director provides a powerful terminology, coming from a movie directors background. Yet both systems rely heavily on scripting, and do not support any of the input and output devices needed for AR presentations. Authorware allows the visual modelling of a non-linear presentation's plot – a very helpful feature. This idea is taken further by using a standardized conceptual model like an UML statechart and use it for story modelling.

Recent projects like DART or MARS have offered high-level support for authoring AR presentations. However, DART relies on scripting and therefore requires it's users to have sufficient programming skills, and MARS is following the hypermedia paradigm with its limitations for creating fully interactive virtual presentations. Both systems run only on a single hardware setup, and do not support alternative AR display technologies like virtual showcases or light projectors.

3.3 UML story modelling

As mentioned already, the capability of modelling non-linear paths through an interactive system can be found in state chart diagrams, a method used in computer science to model the behaviour of entities. State charts can be annotated in various ways, including attaching text, images or other media content to the elements of a state chart diagram, turning it into a tool for the specification of stories, allowing the author to add notes, content suggestions and ideas early in the design process.

The Unified Modeling Language (UML) [52] defines a standardized way of drawing state chart diagrams, and with the recent standardization of XMI [53], the extensible metadata interchange format, there is a standardized, platform independent way of serializing these diagrams to an XML-based format. Therefore, one can use any standard UML modelling suite, or take an open source project like ArgoUML [1] and extend it to support story-authoring specific features. The story can then be exported to an XML format, which offers a convenient and standardized way to import the created storyboard into any other software.

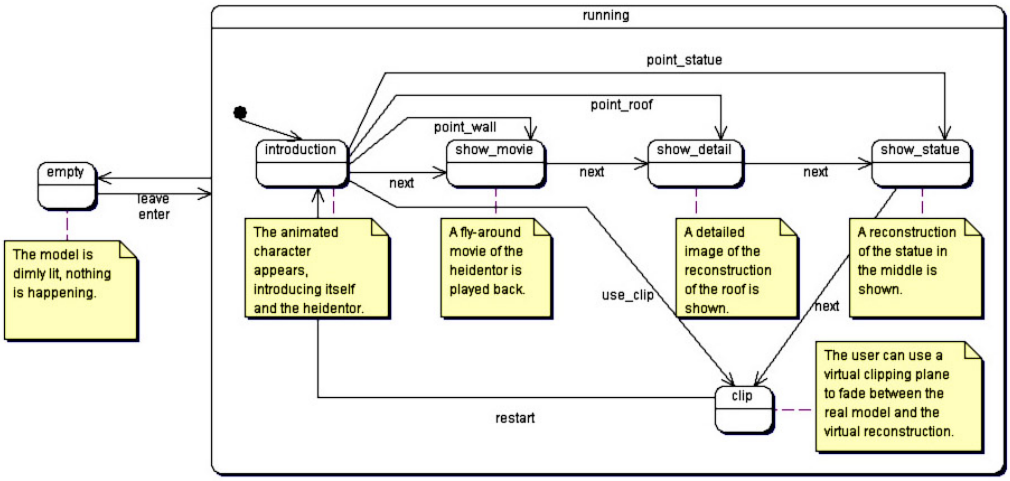

Figure 3.3: An example of an UML storyboard for a simple presentation.

The state chart is created by the story author very early in the process, acting as a sketch pad during the brainstorming phase and as a communication tool in meetings with domain experts and other people. Later on, when the story has stabilized, it is the main specification document to hand out to content creators and component implementers, to define the overall context of the project and to visualize the flow of the presentation. An example storyboard is shown in figure 3.3.

Chapter 4

The APRIL language

After reviewing the available AR systems and looking at solutions for authoring and content creation, I want to present the design for a language that supports the authoring of Augmented Reality presentations. The Augmented Reality Presentation and Interaction Language (APRIL) proposed here should provide high-level concepts, derived from the state of the art in each of the fields, for direct usage by presentation creators.

The first step for designing APRIL is to sum up the requirements for such a language, based on the findings in the previous chapters. Following that, a design for the central concepts of APRIL will be proposed and the roles in the workflow for the collaborative creation of such presentations will be defined. In subsequent chapters, I will present a prototypical implementation of an APRIL player software and some sample presentations that have been realized using APRIL.

4.1 Requirements of an AR authoring language

As stated already, an authoring language for a given media has to take into account the special features and constraints of that media. Some of the requirements that will be identified come out of the specifications of AR systems, others are derived from the anticipated applications that users should be allowed to create.

As shown in chapter 2, there are probably more AR "systems" and possible configurations than research groups in the field, used to conduct various experiments or verify user interface ideas. Those systems which do persist over several setups and applications have to support a wide variety of input and output devices, with sometimes completely different data and media

types to process – video and audio data, tracking information, geometry to be rendered are only the most common data to be processed by an AR system. In this heterogenous landscape of devices and software systems, a generic authoring language should support different combinations and configuration of hard- and software, to be usable across a wide range of installations.

Requirement 1 *Support at least a reasonable subset of input and output hardware and their manifold combination possibilities.*

As most AR systems are prototypes, they are usually also a sparse resource. It should therefore be possible to develop presentations in a (desktop-based) simulation environment, without having to occupy the target system for the whole time of the development process. Also, presentations developed for a specific setup should be portable to other setups with minimum effort.

Requirement 2 *Support the portability of presentations by separating the presentation's content from the system and hardware specific definitions. Also support desktop based developer setups for creating and debugging presentations.*

On the content side, there is a variety of tools in use for 3-dimensional modelling and animation, let alone audio and video production. In the 3D modelling world, formats like VRML have established themselves as standards for geometry interchange, acting as a hub between authoring software with their proprietary file formats and rendering and playback solutions. Unfortunately, in the area of animated geometry, most software packages use their own formats, and animations are usually not exported to formats like VRML. Only recently, with the gaming industry as a driving force, standards are beginning to emerge in this area to make a seamless toolchain across different vendors possible.

Requirement 3 *Support standards for geometry and media data, and support upcoming industry standards for animation of 3-dimensional content.*

Content creation for Virtual- and Augmented Reality presentations is a tedious process. To make the most of this effort, it should be possible to reuse content across different applications, creating a repository of components of such applications as new projects are realized. The complexity of these components can range from static, 3-dimensional models or media objects to fully interactive objects, interaction techniques or even whole applications.

Requirement 4 *Support the re-use of content by providing a modular structure of presentations. Allow the creation of content archives and the sharing of content between multiple users and setups.*

Not only the input devices for AR systems, but also the interaction techniques and policies realized with these devices vary greatly across installations. Unlike in 2D desktop environments, interaction in 3D space is not yet standardized, and applications usually have to be adapted to a specific platform before they can be used. For this reason, most VR and AR applications are either very limited in the interaction possibilities they offer to the user, or run only on a single, specialized hardware setup. For interactive presentations, highly sophisticated interaction techniques, as needed for example by CAD applications, are not required – however, the user should be able to perform basic interactions, similar to physical interactions (walk/look around, point at objects, grab objects, press buttons ...) to interact with the scene. These interaction techniques should be available on all hardware setups and use the available input devices in an optimized way to provide a natural mapping.

Requirement 5 *Provide a well-defined set of interaction techniques and support various input devices to implement them.*

In conventional media (like books, recordings or movies), a presentation can only progress linearly, guiding the audience through a story (usually consisting of introduction, main part and conclusion). Hypermedia systems like HTML have extended our possibilities with the introduction of *hyperlinks*, a way to let the audience interactively choose their way through the content by successively selecting one of several links to the next hypermedia document. For our interactive presentations, the system should also support non-linear stories, but not through an implicit, static hypermedia structure. Instead, our presentations should *evolve* with the user's actions and be able to react to various conditions and events.

Requirement 6 *Provide authors with tools to structure their presentations in a non-linear way for the user to explore them interactively.*

These requirements are the starting point for the design of the planned presentation framework. Since, as has been argued in chapter 3, we do not want to rely on scripting for defining an applications behaviour, our language will be a structural language in contrast to an interpreted one – a document written in this language defines a static structure that exposes the desired behaviour at runtime, and is not *executed* by an interpreter or scripting engine.

One important base technology that helps to encode structural information in human readable and machine readable code is XML [22, 36].

While it would be possible to choose a different base technology for the APRIL language, or design a proprietary syntax from scratch, using XML as a base standard to create the AR presentation language offers several advantages:

- It is a widely used *standard*. As stated in requirement 3 in the introduction, the language should be based on standards that are used in industry and research. Nowadays, a lot of tools for editing, storing and processing XML data are available, and these tools will be of benefit for authors using our framework.
- XSLT (see section 4.2) allows the translation of XML documents into other XML- or arbitrary text documents. This is the basis for machine processing of APRIL documents, allowing for automatic conversion, rendering to various output formats and updating of documents to new versions of the standard.
- As stated above, XML is optimized for machine processing, yet it is saved in a human-readable ASCII-format. This supports debugging and creation of APRIL files by hand, and allows new tools to be created by using available XML parser frameworks.

In the following section, we will look in more detail at the consequences of choosing XML as a base technology for APRIL.

4.2 XML technologies

While XML itself is a quite slim standard, defining only the basic syntactic rules for *well-formed* documents, it is accompanied by a set of second-level technologies that make it the powerful tool for storing structural information it is today.

The atomic unit of information in XML is an *element*, which can contain *attributes* and child elements. A special type of element are text elements, containing only (ASCII-)text and no attributes or children. Every XML document contains a single root element, which can contain an arbitrary number of children, which can in turn contain children, and so on. In its memory representation, the root element is the root of a *tree* data structure representing the XML document, with all elements represented as child nodes in the tree. The first step in defining a new language (or *dialect*) using XML is to define the elements and attributes that the language is composed of.

4.2.1 Defining the language: DTDs and Schemas

While the core XML standard defines rules for well-formed XML documents, it does not allow to check whether such a well-formed document is a *valid* document of a given XML dialect – i.e. if the correct elements are used in the correct places. To allow users to define the set of valid documents of a given XML dialect, additional standards have been proposed: Document Type Definitions (DTDs) and XML Schemas.

DTDs have been supported by the very first XML standard as a means to define XML sublanguages. DTDs follow their own syntax, and define element names, and their attributes as well as allowed children. A few shortcomings of DTDs have inspired developers to look for alternative, more advanced ways of describing XML dialects:

- DTDs are not XML documents, but follow their own, proprietary syntax. Therefore it is not possible to apply XML tools on DTDs (for example, it is not possible to define a DTD for DTDs, and therefore not possible to check the validity of a given DTD)
- DTDs are not typed. It is not possible to define classes of elements, or types or value ranges for attributes.
- The possibilities for defining the validity of element structures are very limited – it is therefore not possible to create sophisticated validity checks.

However, since DTDs are a relatively simple standard to use as well as to implement, they are still widely used in XML tools and applications.

An improved way to define an XML-based language is provided by XML schemas [31]; schemas provide solutions to the main shortcomings of DTDs: they are XML documents, provide a sophisticated type system for elements and attributes, and allow the expression of complex conditions for defining the validity of documents.

For APRIL, an XML schema will be provided as the primary language specification. As most XML parsers and tools available today do not fully support schemas, also a DTD, which can be generated automatically from the schema, will be created. A graphical overview of the APRIL schema is presented in appendix A, and the full schema with all elements and attributes is listed in appendix B.

4.2.2 Mixing dialects: XML Namespaces

In some cases it is necessary or desirable to mix elements from different XML dialects in a single document. Doing so without further preparations would lead to problems in identifying which dialect a given elements belongs to, especially if the element names are identical (which cannot *a priori* be ruled out).

To solve this problem, the concept of XML namespaces was introduced. All elements from a given dialect are marked with a (user defined) prefix, which is in turn mapped to a unique identifier for the dialect (for this purpose, often the URL of the organization which developed the language is used, because it is a world-wide unique identifier). In the XML parser, the short prefixes are expanded to the long, unique identifiers, resulting in a set of uniquely identified elements that can be unambiguously mapped to their respective XML languages.

In APRIL, namespaces will be used to include OpenTracker configuration information (expressed in the OpenTracker configuration XML dialect) directly inside an APRIL file. Therefore, OpenTracker can be used as a part of APRIL, and we do not have to re-invent the wheel for the specification of low-level tracking configuration. An example of a file using XML namespaces is shown in figure 4.1.

4.3 Hardware description

The first thing to consider when describing an AR application is the description of the hardware setup the application will run on. As required, an application can be run on different hardware setups, and a single hardware setup can host multiple applications, sequentially or simultaneously. It is therefore desirable to separate the hardware description part of APRIL from the description of the content and behaviour of the presentations that will run on the setup, since it can possibly be reused for other presentations.

The root element of an APRIL file, `april`, contains two child elements: a `setup` element describing the hardware setup, and a `presentation` element, which holds the definitions for the presentation's content and behaviour. Alternatively, the `setup` element can be empty, containing only a `src` attribute, specifying the URL of an (APRIL-)file from which to include the setup element. By using this approach, it is possible to share a single hardware setup description amongst multiple presentations.

An AR hardware setup is typically composed of one or more computers, displays connected to those computers, tracking devices used for head-

tracking, pointing and other purposes, and buttons for triggering certain actions in the presentation. Within the `setup` element, users can define such a setup. For tracker configuration, the elements of the OpenTracker configuration files are used. Since OpenTracker uses XML-based configuration files, these elements can simply be included by using XML namespaces.

The `host` element is a wrapper element for all devices that are connected to a single host. In setups with multiple hosts, the APRIL execution platform can establish network connections between hosts for distributing application [39] or tracking data [48]. Each `host` element can therefore carry additional information about its IP address and other networking properties as attributes.

Inside the `host` element, the video outputs of the computer are configured with corresponding `screen` elements, specifying the resolution of the video output(s). Logical displays do not necessary fill the whole area of a video output, therefore they are configured independently using the `display` element – containing all the attributes necessary for the definition of a display surface for AR applications. Displays can be mono or stereo, using video-see-through, optical-see-through or virtual reality (non-see-through) techniques or projector based approaches. Furthermore, the real-world size and position of the display surface and its position and orientation have to be defined. For applications with mobile displays or head tracked users, the `display` element can contain an additional `headtracking` and/or `displaytracking` element(s), containing the necessary OpenTracker configuration elements to configure the tracking.

All tracked elements in the hardware description part can either contain OpenTracker elements, included directly inside the element, or just specify a named OpenTracker node in an external document to define the tracking. Therefore, existing OpenTracker setups can simply be referenced inside an APRIL configuration file.

Some pointing techniques (see section 2.4.1) are only applicable in conjunction with a display surface or a tracked eyepoint. For these techniques, a `pointer` element is used inside the `display` element it refers to, clearly associating the pointing device with the display. Other pointing devices are specified outside the `display` element, as a direct child of the `host` they are connected to. All `pointer` elements contain an attribute specifying the technique used, and can contain OpenTracker elements to define the tracking source. If no tracking source is specified, the mouse is used as a default input device, if possible. Tracked objects that are not used for pointing can be configured using the `station` element. Again, the tracking is defined by inline or referenced OpenTracker elements.

Finally, `button` elements define push-buttons available in the installation

to control the application. The input for the button can be an OpenTracker source (although a button is not a tracking device, OpenTracker can provide button information for various devices and supports buttons connected to the parallel port of the computer), or a key on an ordinary keyboard. For public installations, a keyboard controller can be used to connect arcade-style buttons to it, to act as a cheap and reliable interface.

The complete arrangement of hosts, displays and tracked devices is the hardware setup to run APRIL presentations. Figure 4.1 shows a sample hardware configuration file, taken from a configuration for a virtual showcase setup.

4.4 Story modelling

The first part of the actual presentation's content is the storyboard of the presentation. The storyboard is modelled as an UML statechart (as presented in section 3.3). A state in the diagram represents a scene, a transition denotes user interaction. Additionally, statecharts can be annotated with text or other media elements to embed hints for the design of the presentation's content in the storyboard.

For visual editing of UML statecharts, developers can choose among several commercial or open-source tools that support the XMI format. Since XMI is a very complex standard with different syntactic alternatives for identical concepts, the generated XMI code is not directly included in the APRIL file, but a simplified syntax has been developed to define the state engine. For a smooth workflow, an automatic conversion tool, based on XSLT, has been developed, to translate XMI documents to the simplified APRIL syntax. This converter tool will be described in section 5.3.1.

The simplified syntax offers elements for states, transitions, and composite states that can contain multiple concurrent substates. Figure 4.2 shows an example story definition, using the APRIL syntax to define a simple storyboard.

4.5 APRIL components

As stated in the requirements section, the content of our presentations should be composed out of reusable components. Components should be defined outside the presentation, in individual files, to allow for re-use across presentations and setups.

```
<april xmlns="http://www.studierstube.org/april"
       xmlns:ot="http://www.studierstube.org/opentracker">
  <setup>
    <host name="showcase" ip="10.0.0.77">
      <screen resolution="1280 1024"/>
      <screen resolution="1024 768"/>
      <display screen="1" screenSize="fullscreen" stereo="true"
        worldSize="-0.4 0.3" worldPosition="0.098 0.162 0"
        worldOrientation="-0.1856 0.9649 0.1857 1.6057" mode="AR">
          <headtracking>
            <ot:EventVirtualTransform translation="0.00 0.20 0.01">
              <ot:NetworkSource number="1" multicast-address="10.0.0.7"
                port="12345"/>
            </ot:EventVirtualTransform>
          </headtracking>
        <pointer mode="2D-RAY"/>
      </display>
      <station id="tool">
        <ot:NetworkSource number="2" multicast-address="10.0.0.7"
            port="12345"/>
      </station>
    </host>
  </setup>
</april>
```

Figure 4.1: A sample hardware configuration file. In this case, a single host with two configured VGA outputs is used to drive a single, head-tracked display. Additionally, a pointer (using the mouse for ray-picking) and a tracked object are defined.

```
<story>
  <scene name="empty" initial="true"/>
  <scene name="play">
    <concurrentScene>
      <scene name="A1" initial="true"/>
      <scene name="A2"/>
      <transition event="go" source="A1" target="A2"/>
    </concurrentScene>
    <concurrentScene>
      <scene name="B1" initial="true"/>
      <scene name="B2"/>
      <transition event="move" source="B1" target="B2" guard="A2"/>
    </concurrentScene>
  </scene>
  <transition event="enter" source="empty" target="play"/>
  <transition event="leave" source="play" target="empty"/>
</story>
```

Figure 4.2: As simple story, defined using the elements provided by APRIL. Figure 4.3 shows the graphical representation of this storyboard. Note the use of a guard, that enables the transition from B1 to B2 only when state A2 is active.

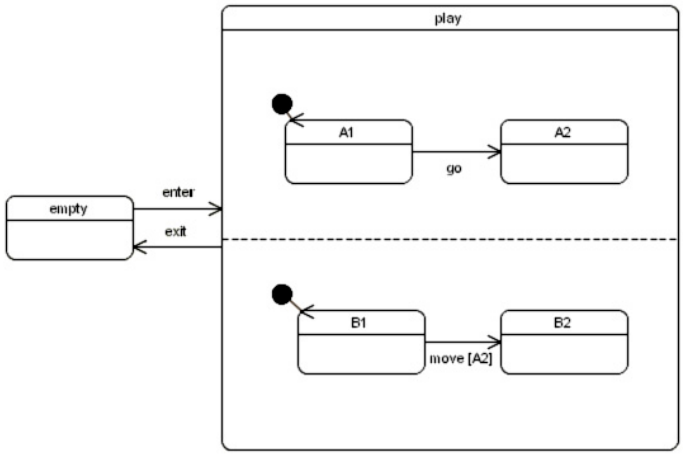

Figure 4.3: The graphical representation of the story from figure 4.2.

4.5.1 Component definition

As these components will constitute the content of our presentations, sophisticated means to express geometry and multimedia content, that make up an object in a presentation, will be needed. Instead of creating a new XML-based syntax for defining the content of such an object, another approach has been chosen: an APRIL-component is basically an ASCII-based template, using *any* existing, ASCII-based language to express the intended content, plus added XML-markup to define the *interface* of the component, a collection of inputs and outputs that will be accessible from the APRIL presentation.

While this component mechanism works with any ASCII-based language (including XML-dialects), APRIL components will generally use a content description language that can be read by the target runtime platform. In the case of our reference implementation (which will be discussed in chapter 5), we will use Open Inventor as the basis for defining the content of our components. While these components will not be portable across platforms (since they use a platform-specific content specification format), the APRIL component mechanism itself is platform independent and can make use of any host language.

Using a platform specific language for content definition reduces portability of components, but makes all features and optimizations of a given platform available to developers. The alternative would have been to create a platform-neutral content definition language, that could always only use a set of features supported by all platforms – an approach that does not allow the creation of sophisticated content for our presentations that uses state of the art real time rendering features. To support the portability of components across platforms, components can contain multiple alternative implementations for different platforms, letting the APRIL-translator choose at translation time the suitable implementation for a given runtime platform.

An APRIL component definition file contains two main parts: the components *interface definition*, and one or multiple *implementations*, expressed as XML-enhanced templates in the chosen language. A components interface is composed of the available input and output fields, and the specification of possible sub-components (parts) that can be added to the component.

A field in the interface of a component can contain one or multiple values of a primitive data type. A list of possible data types is given in table 4.1, where single value fields are prefixed by the string `SF` and multiple value fields are indicated by a `MF` prefix. In addition to the primitive data type, the `SFPose` data type is provided for processing pose information (position and orientation), since this is data frequently used in AR applications.

Type	SF	MF	Description
`Trigger`	×		A trigger field, not carrying any value. The trigger fires every time its value is changed.
`Bool`	×	×	A boolean value, either 'TRUE' or 'FALSE'.
`Int32`	×	×	An integer value.
`Float`	×	×	A floating point number.
`String`	×	×	A text string.
`Vec2f`	×	×	A 2-dimensional floating point vector.
`Vec3f`	×	×	A 3-dimensional floating point vector.
`Rotation`	×	×	A rotation value, composed out of 4 float values.
`Pose`	×		Pose information, consisting of position and orientation.
`Color`	×	×	A color value, composed out of 3 float values for the red, green and blue component.

Table 4.1: Field types supported by APRIL.

Fields can be used for input, output or both. This is indicated by using an `input`, `output` or `field` element for defining the field, respectively. Inside the element, the fields type and default value are specified with attributes, and the `const` attribute specifies whether the field can be changed during the presentation.

The second part of a components interface definition are parts, or children, of the component. Parts are specified using the `part` element, and define a hook for possible sub-components, that will be inserted into the given component. By using the parts mechanism, components can be created that influence other components – for example, the `group` component (shown in figure 4.4) transforms all its children by a translation, scale and rotation factor that can be configured using its `inputs`. As parts can be named, a component can have multiple parts, each carrying one or more children, for various purposes. If a component has only one part named "children", it is the default part of the component and can be used without specifying its name explicitly.

The implementation part of a component contains the specification of the components content in the chosen host language, which is indicated by the `type` attribute of the `implementation` element. Inside this element, the inputs and outputs used in the interface definition have to be marked by special XML marker elements, to indicate the (platform specific) entry points for setting and retrieving values from the components fields. These marker elements are the `in` element for indicating input fields, the `out` element for

```
<interface>
  <field type="SFVec3f" id="position" default="0.0 0.0 0.0"/>
  <field type="SFRotation" id="orientation" default="1.0 0.0 0.0 0.0"/>
  <field type="SFVec3f" id="scale" default="1.0 1.0 1.0"/>
  <field id="visible" type="SFBool" default="TRUE"/>
  <part id="children"/>
</interface>
```

Figure 4.4: Interface definition of the group component.

indicating output fields, and the `sub` element for indicating the position of a part. Additionally, the `id` element can be used and will be replaced by a unique id for each instance of the component. This can be used to generate unique IDs for various purposes.

As the usage of these marker elements depends on the runtime platform that the presentation will be executed on, no general rules can be given for using them. For Open Inventor components, which will be used in the prototype implementation, the `in` element is used as a simple placeholder for input values in the Open Inventor ASCII file format. Also the `sub` element is simply used as a placeholder, which will be replaced by the corresponding sub-components in the generated Open Inventor file. The usage of the `out` marker for retrieving a field value is not so straightforward: Since in Open Inventor the values of fields can only be referenced by using the unique name of its parent plus the field name, the full reference of a field cannot be inferred by using only a local marker, without parsing the Open Inventor code in the implementation. Instead, the `id` element is used to create a unique prefix for the name for the fields parent node, and the fully qualified field name (including the unique name of the parent) is then referenced inside the `out` marker, placed at the end of the implementation specification. This demonstrates the versatility of the template-based component approach, where complex references can be generated without parsing the host language by using relatively primitive XML-based markup. As an example, the implementation section of the group component is shown in figure 4.5.

4.5.2 Using components

After defining a component, it can be used in the APRIL presentation by instantiating it one or possibly multiple times. All components are instanti-

```
<implementation type="Open Inventor">
Switch {
  whichChild 1 = DEF <id/>_Bool BoolOperation {
    a <in id="visible"/>
    operation A
  }.output  # convert from Bool to Int32
  Group {} # Dummy Child for switching off
  Group {  # actual content
    DEF <id/>_Transform Transform {
      translation <in id="position"/>
      rotation <in id="orientation"/>
      scaleFactor <in id="scale"/>
    }
    <sub id="children"/>
  }
}
<out id="position"><id/>_Transform.translation</out>
<out id="orientation"><id/>_Transform.rotation</out>
<out id="scale"><id/>_Transform.scaleFactor</out>
<out id="visible"><id/>_Bool.a</out>
</implementation>
```

Figure 4.5: Open Inventor implementation section of the group component.

ated in the `cast` section of the presentation, by using `actor` elements. The source file of the component is specified with the `src` attribute, and a unique id has to be assigned to each actor by using the `id` attribute. This attribute will later be used in the presentation to reference the actor, for changing or referring its field values.

Within the `actor` element, field values can be set with `input` elements. These values will be used as default values for the presentation, and override the default values specified in the components interface definition. Parts of a component are set by using a `children` element as a wrapper for other component instances, specified by nested `actor` elements. The parent/child relationship defined by parts is therefore expressed by nesting `actor` elements in a hierarchical manner.

Figure 4.6 shows a snippet from a presentation's `cast` section, instantiating a group component, which contains two sub-components: A model and a hotspot.

```
<actor id="group" src="group.apc">
  <input id="position" value="0 0.14 0"/>
  <children>
    <actor id="ryla" src="model.apc">
      <input id="src" value="content/Ryla.iv"/>
      <input id="orientation" value="0 1 0 -1.57"/>
    </actor>
    <actor id="hotspot" src="hotspot.apc"/>
  </children>
</actor>
```

Figure 4.6: Instantiation of the group component, containing two children that will be transformed by it.

Representing the real world

As already mentioned, one key feature of AR systems is the integration of the real world as part of the application. In APRIL, real-world objects are represented by `stage` elements, containing the geometric representation of the real objects surface, which can be obtained by 3D-scanning or modelling. This geometry is then used by the APRIL player software to calculate occlusion between real objects and virtual content, or to render special effects

such as projector based illumination.

4.6 Animation and behaviours

The components defined and instantiated in the `cast` section of a presentation are available during the whole running time of the presentation. Dynamic behaviour is added to the presentation on a scene-by-scene basis by adding *behaviours*, which control the field values of the components of a presentation.

Behaviours are bound to scenes of the presentation – each scene (defined by a state in the `story` definition) can hold an arbitrary number of behaviours for all the fields of all actors of the presentation, which are arranged on a *local timeline* to allow behaviours to be scheduled or performed sequentially.

Each scene is further divided into three pseudostates – `entry`, `do` and `exit`. All behaviours in the `entry` state are performed when the story enters the given scene and are guaranteed to be executed. Therefore, behaviours with a nonzero duration, specified in the `entry` substate, *block* the presentation until they are finished.

The internal timeline holding the behaviours of the `do` substate will start when all behaviours in the `entry` state have completed, and will only run as long as the parent state is active. If the state is left before all behaviours have been effective, the remaining behaviours are ignored and the presentation continues with the next scene.

Finally, the `exit` substate holds behaviours that will be performed upon exit of the given scene. Again, all behaviours in the `exit` substate are guaranteed to execute, and will block the transition of the presentation to the next state (which has, at this time, already been triggered) as long as their completion takes.

As indicated already, these substates are represented in the presentation file by the corresponding elements. A behaviour is defined by a `behavior` element, specifying the scene it should be bound to, and its contained `entry`, `do` and `exit` elements. Inside these elements, APRIL provides four fundamental behaviours for controlling an actors inputs – it can be set to a given value, animated over time, connected to an output of another actor, or controlled by the user. Each of these possibilities is represented by a separate element in the specification.

Setting field values

The `set` element allows to assign a new value to any input field of an actor. Depending on the field type, the `to` attribute contains the textual representation of the target value. Multiple values (for setting multiple value fields) are separated by commas, the components of a multi-dimensional field value (like `Vec3f`) are separated by spaces. The target field of a `set` operation is specified by setting its `actor` and `input` attributes to reference the target field.

All `set` behaviours are, by default, executed immediately when the parent state becomes active. Alternatively, `set` behaviours can be scheduled on the states timeline by using the optional `time` attribute. As with all attributes that contain time values, the `xs:duration` data type, provided by XML Schema [31], is used to express the value. The `xs:duration` type allows to express time values ranging from years to milliseconds, allowing for presentations that run in real-time over long periods of time.

Animating fields

In addition to the information needed for setting a fields value (target field and starting time), animating a field needs input for the intended duration of the animation (again expressed as an `xs:duration` value, stored in the `duration` attribute) and the target value of the animation. The target value can be expressed in two different ways: either as an absolute value, by using the `to` attribute, or as a relative offset specified in the `by` attribute. If an absolute value is used, the input field is simply animated (linearly interpolated) to the new value, if the `by` attribute is used, its value is added to the current value of the field over the time of the animation.

Generally, animations use the current value of the field and start the interpolation from there. Alternatively, an explicit starting value can be specified, which is identical to `set`ting the field to the starting value at the beginning of the animation and running the animation from this starting value.

Connecting fields

The third possibility to control an input field is to connect it to another output field of the same or another actor. Connections persist as long as the state they are defined in remains active, routing any changes in the output (source) field to the input (target) field immediately. By using connections, complex behaviours can be realized by calculating the desired field value in a

custom component and routing its output to an input field of another actor, establishing a controller/slave relationship between the two actors.

For defining a connection, only the source and target actors and fields have to be specified by using the corresponding attributes `actor` (for the target actor), `input` (for the target input), `master` (for the source actor) and `output` (for the source output).

Letting the user control a field

Finally, control of a field can be given to the viewer of a presentation by using the `control` element. This is mostly used for adjusting parameters of the story, but can also be used for more complex interaction between the user and a presentation's content. As for field connections, the control of a field by the user lasts as long as the corresponding scene is active, therefore it only makes sense to use `control` elements inside the `entry` or `do` substates.

Besides the usual specification of the target field, authors can define minimum and maximum values (for numeric inputs) of the adjustment. Additionally, a `label` can be specified that can be used by the target platform to annotate the user interface that is generated for controlling the field value.

How the control of the field value is realized depends on the implementation of the APRIL runtime platform. Depending on the system, two- or three-dimensional user interface elements might be rendered with the corresponding label to enable the user to make adjustments. For numerical values, this interface will most likely be a "slider" type widget, while for boolean values it will be a toggle-button or a checkbox-like widget. Position and rotation information can be controlled by using a `station`, defined in the setup file.

Figure 4.7 shows an example of a behaviour binding element for a single scene of a presentation. All four element described above are used in this example.

4.7 Interaction

The last part missing to realize interactive presentations is the integration of user interaction into the APRIL authoring concept. Like states, that represent scenes of the story and can be bound to specific behaviours of actors, transitions represent changes of the story and can be bound to specific user interactions or conditions.

By separating the definition of a transition in the storyboard from the specification of the user interaction that should be used to trigger that tran-

```
<behavior scene="introduction">
  <entry>
    <set actor="ball" input="visible" to="TRUE"/>
    <animate actor="ball" input="transparency" from="1.0" to="0.0"
      duration="PT2.0S"/>
  </entry>
  <do>
    <control actor="ball" input="transparency" min="0.0" max="1.0"
      label="Transparency of ball"/>
    <connect actor="ball" input="position" master="plane"
      output="cargoPosition"/>
  </do>
  <exit>
    <set actor="ball" input="visible" to="FALSE"/>
  </exit>
</behavior>
```

Figure 4.7: Behaviour binding for a single scene. All four different behaviour elements are used in this example.

sition, the mapping of transitions to interactions can easily be changed and replaced by an alternative mapping. Therefore, it is easy to realize different versions of a story, be it for different target platforms or different audiences. A presentation may be run in a fully automatic mode for a presentation in front of a large audience, and switched to an interactive setting, where the user can actively influence the story, for presentations to single users.

APRIL offers the story author a catalog of interaction techniques to be used. This catalog contains the fundamental interaction tools and techniques mentioned in section 2.4, plus a few specific concepts for interacting with components.

The binding of an interaction to a given transition of the storyboard is done by using an `event` element, specifying the id of the transition in question. Inside the element, interaction elements are placed, which will be bound to the corresponding transition. If multiple interactions are specified, any of the interactions can be used to trigger the corresponding event.

Head tracking

A basic interaction feature of nearly all AR installations is tracking the user's head and rendering the graphics displayed dependent on the user's estimated eye point position. In the hardware description file, headtracking is defined by a `headtracking` element inside a `display` element, containing the necessary OpenTracker definitions for realizing the tracking. In the interaction layer, headtracking can be used to detect if a user has entered a given region or if the setup is currently unused.

Touching Objects

The `touch` element can be used to define an interaction that is triggered whenever the user touches an actor, referenced by the `actor` attribute, in the scene. All pointing devices defined in the setup part can be used to touch the object. A complimentary element, `untouch`, ic provided to detect the end of a touching gesture previously started.

Buttons

Buttons are the most basic hardware interaction tools that can be used in a AR installation, but due to their robustness and versatility are a very important interaction tool for presentations. Buttons can be used as simple presentation controls ("next scene", "previous scene"), but can also change behaviour depending on the context of the presentation. Buttons can also be embedded into the floor or realized as photo-sensors to detect the user's

movement. In addition, buttons can be easily simulated by rendering them onto the display surface as virtual buttons, if the presentation runs on a different hardware setup with fewer or no hardware buttons installed.

A button interaction is defined by placing a `buttonaction` element inside the `event` element for the transition, specifying whether a virtual button should be used and (optionally) the label of the virtual button to be rendered.

Hotspots

Hotspots are cubic regions in presentation space - they might be parts of the real or virtual content of the presentation, and mark significant parts of an artifact that might be selected by the user. The normal scenario would be that additional information about parts of an object is displayed when the user selects it.

Similar to buttons, hotspots trigger the transition when the user clicks on them or touches them – depending on the configuration. Depending of the definitions in the setup description, various direct or indirect pointing techniques may be used to trigger a hotspot.

To realize a hotspot interaction, a hotspot component has to be placed into the scene, which can then be used for interaction by using a condition for its `touched` or `triggered` outputs (see below).

Conditions

To overcome the simple, hard-coded interaction techniques offered by APRIL and to realize custom, story-specific interaction, general conditions can be used to trigger events to drive the story. A condition compares the value of an output field of any actor to a constant value or another output field. If the condition evaluates true, the transition that the condition is bound to is triggered.

Conditions are specified using the `evaluator` element. Besides specifying the output field to be compared and either a constant value or another output field to compare to, the comparison itself is selected from a list of available ones. Comparison types provided by APRIL, that can be specified in the `comparator` attribute, are: equal (default), lessThan, greaterThan, lessOrEqual, greaterOrEqual, notEqual, isInside (performs a bounding box check), notInside.

Timeouts and automatic transitions

Besides direct user interaction, also the absence of user actions can be used to drive presentations. APRIL provides a timeout mechanism, specified by

using the `timeout` element, to be able to trigger events if for a certain period of time no significant user interaction took place. Additionally, transitions can be configured to trigger always (as soon as they become available) or never (to disable parts of the story, for example for demo presentations) by using the `always` or `disabled` elements.

4.8 The APRIL workflow

Authoring interactive, dynamic presentations is a process of varying complexity that may include several professionals working with different tools, but may also be performed by a single individual with more limited resources. The authoring process should therefore be scalable, offering all the possibilities to model simple presentations or prototypes quickly, and offering a structured workflow to teams working on larger presentations, incorporating various tools for modelling and content creation.

APRIL supports such a workflow by separating presentations into different parts – story, components, media objects, hardware description, behaviours and user interaction are the main aspects mentioned in the previous sections. The APRIL authoring process helps a single author to structure her work, and teams of specialists working together to coordinate their efforts. For the authoring process, it is possible to define various *roles* of people contributing to the presentation. These roles may be embodied by distinct professionals (or teams of professionals), or by fewer or even a single person authoring a simple presentation on her own.

The following roles of people that contribute to a VS presentation have been analyzed:

- Domain Expert

  The domain expert is the individual or group with the necessary knowledge about the presentation's subject. For history presentations, this might be an archaeologist or historian, for scientific presentations an expert on the given subject.

- Story Author

  The story author is the person who comes up with the ideas of how the subject should be presented in an interactive way, and defines the storyboard for the presentation. In our model, the story author is also the communicator between the domain experts and the content creation people.

- Content Creator

  Content creators design and deliver multimedia content for the presentation, following the storyboard as a specification document. Content creators deliver images, graphics, video, sound and 3D-models to be used in the presentation.

- Component Implementer

  For sophisticated presentations, static media content has to be turned into components that can expose behaviour and react to user input. Additionally, custom components may be needed to realize complex user interaction or behaviours.

- Story Integrator

  Finally, the story integrator puts together the components and media elements according to the storyboard, and specifies interaction techniques offered to the user. This is a similar, integrative position as the story author, and might well be performed by the same person. But while the story author acts a priori to the content creation to specify the details of the presentation, the story integrator takes the results of the content creation phase and puts them together.

Using the definition of roles given above, we get a sequence of steps to create an APRIL presentation, using the various tools that have been integrated. A graphical representation of this workflow is given in Fig. 4.8.

The first step in the APRIL workflow is research of the subject of the presentation. Raw material (text, images, video, sound, models) is collected, and the idea for the presentation is developed, possibly in sessions with domain experts and museum staff. This brainstorming phase results in the story document, the UML model of the flow of the presentation.

The story document acts as a specification for content creation (using the raw material found in phase 1) and component authoring. Components can be re-used from a set of default components or earlier presentations, and for sophisticated interactive presentations new, customized components will be developed.

Integrating the components, interaction tools and content items is the goal of the final phase, story integration. The result is the complete presentation specified in the APRIL language. Independent from the story authoring, for each hardware setup there is a configuration file, describing the arrangement of displays, interaction hardware, speakers, and other aspects of the available hardware. During story development and testing, it is not

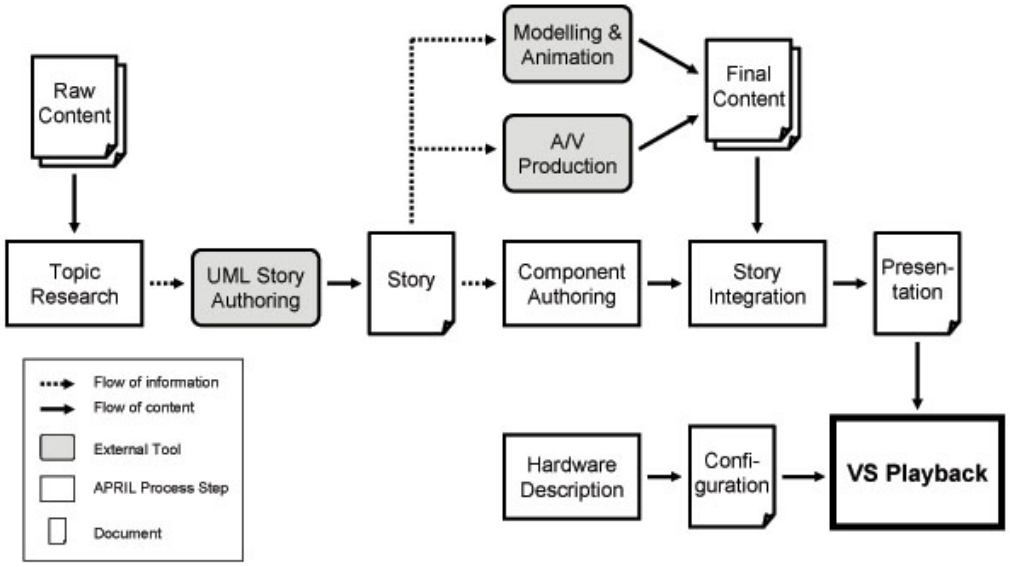

Figure 4.8: The APRIL workflow for creating a presentation.

necessary to run the presentation on the actual hardware setup; the same presentation can be run in an "emulation mode" on the developers PC.

Media content, components and hardware description files are re-usable parts of an APRIL presentation, and are only loosely coupled by the story to contribute to a specific presentation. The same media, components or hardware can be used to tell other stories about the same or completely different subjects.

Chapter 5

Implementation

In this chapter, a prototype implementation of an APRIL player software, based on the Studierstube AR system [66], will be presented. The approach taken is to transform the APRIL files that make up the presentation (presentation file, setup description and the components used in the presentation) into Studierstube configuration and content files in an offline process, and then run the native Studierstube application resulting from that process.

5.1 Transformation and querying: XSLT and XPath

Besides the "core" standards XML, DTD, schema and namespaces, a lot of useful tools have been developed for various purposes. One of them is XSLT, the Extensible Stylesheet Language Transformations [23]. XSLT allows the transformation of XML input files into one ore multiple output files in any ASCII-based file format (including XML).

XSLT follows a template-based approach. The result of an XSLT transformation is determined by two inputs: the XSLT file, containing a collection of templates, and the XML input file to be processed. The XSLT engine loads and parses the input XML file, and, starting at the root element, looks for templates that match the current element in the XSLT file. If such a template is found, it is *expanded*, which means that the contents of the template are written to the output of the XSLT processor. If no template is found, an internal default template of the XSLT processor is used, that usually outputs the character data contained inside the element and continues with processing all the children of the current element (again looking for a matching template in the XSLT file). With this approach, all elements of the input file are processed.

Templates can not only contain static ASCII text to be written to the output, but can themselves process information contained in the input XML file. This processing capabilities range from the access of the current element's attributes and children, over conditional processing of parts of the template, to complex querying of the XML input data. Queries of the XML input are formulated in a syntax called XPath [24], which is a powerful language to address and query elements and attributes in an XML document. XPath also provides mechanisms for loading and querying external XML files, and therefore allows the processing of multiple input documents (Which is needed for APRIL to load external hardware configuration and component files).

Within an XSLT template, other templates can be called by using the `xsl:apply-templates` element. The element of the input XML file that the template will be applied on can be selected by an XPath query, effectively allowing each element in the input file to access and process all other elements of the input file(s). With this mechanism, complex processing patterns, like 'jumps' or recursive processing of input, can be realized.

Multiple output files can be generated by using the XSLT 1.1 function `xsl:document`. All templates that are called inside an `xsl:document` element will write their output to the file specified in the `href` attribute of the `xsl:document` element. For this attribute, it is also possible to dynamically calculate a filename, by using information stored in elements or attributes of the input XML file.

Several implementations of XSLT processors exist today. For the prototype implementation, the Saxon XSLT processor, Version 6.5.2 [45], will be used. Saxon is implemented in Java, and offers a lot of extensions to the core XSLT features and an extension mechanism to add new functionality to the XSLT processor. While these features were not used for the implementation, some features that were first introduced in Saxon and later incorporated into the XSLT specification were needed for translating APRIL files.

Describing the full functionality of XSLT is beyond the scope of this work. Interested readers are referred to [23, 36] for a complete overview.

5.2 Studierstube configuration

To run an application in Studierstube, several files are necessary to define the configuration of the Studierstube systems and the content of the application. These are the files that will be generated with the XSLT-based transformation process.

For the configuration of Studierstube and its displays, usually one or multiple files are provided, containing the Open Inventor-based specification of

the setup's UserKits. In contrast to APRIL, where no strong representation of a presentation's *users* is provided (only displays and pointing devices are configured, which could be, conceptually, taken by any user), Studierstube assumes that all displays and pointing devices are assigned to specific users. The UserKit therefore contains a DisplayKit, providing all the necessary fields to configure a display, and a PenKit to define a pointing device for that user. Additionally, Studierstube makes heavy use of PIPs, and therefore allows to specify a PipKit in the UserKit.

Each DisplayKit carries information about the position of the display on the screen, and the configuration of the display window (such as whether to show the window's title bar and decoration, or whether to use a video image as the background for the rendered scene). Additionally, head- and/or displaytracking is configured by adding a specific CameraControl node to the DisplayKit. The virtual camera itself is defined by an OffAxisCamera node, which is usually loaded from an external file to re-use it across users and installations. Figure 5.1 shows an example Studierstube UserKit, with inline definitions of the display, camera and pointer settings.

For all tracking requirements, Studierstube uses OpenTracker as a middleware layer that deals with the details of supporting and configuring individual tracking devices. The connection between OpenTracker nodes as data sources and Studierstube nodes as data consumers is established through StbSink nodes in OpenTracker, each being assigned a unique *station number* to identify the source. Studierstube nodes that consume tracking data (like the DisplayKit for headtracking, or the PenKit for tracking the pointing device) usually have a station field to configure the OpenTracker StbSink node that should be used for delivering the tracking data.

The OpenTracker configuration file that is used for the Studierstube setup is another input file necessary for running a Studierstube application. If no such file is provided, an internal default file, using only the keyboard for tracking simulation, is used.

Finally, the applications that should be run in a setup are defined in another Open Inventor file. This file contains (usually) one or multiple ApplicationKits, which contain the ContextKit implementing the applications behaviour. This may be a subclass provided by an application programmer, to provide custom behaviour and complex processing capabilities, or an 'empty' ContextKit that contains Open Inventor nodes to define the applications content and behaviour. All applications can define PipSheets to provide widgets to control the applications, and geometry, possibly contained in one or multiple WindowKits, to define the content that is visible to the user.

```
#Inventor V2.1 ascii
SoUserKit {
  userID 1
  display SoDisplayKit {
    backgroundColor 0 0 0
    decoration FALSE windowBorder FALSE

    xoffset 0 yoffset 0        # position and
    width 800 height 600       # size on monitor

    display SoFieldSequentialDisplayMode {}  # stereo vision
    cameraControl SoTrackedViewpointControlMode {}
    station 1                  # station used for haedtracking

    stereoCameraKit SoStereoCameraKit {
      eyeOffsetLeft   -0.035 0 0
      eyeOffsetRight   0.035 0 0
      camLeft DEF cam SoOffAxisCamera {
        position 0 0.3 0.5     # only configure the image plane,
        orientation 1 0 0 0    # since the viewpoint position is
        size 0.4 0.3           # delivered by a tracker
      }
      camRight USE cam
    }
  }
  pen SoPenKit{
    station 2                  # station used to track the pen
    geometry Separator { Cone { bottomRadius 0.02 height 0.05 } }
  }
}
```

Figure 5.1: An example Studierstube `UserKit`.

5.3 Implementation details

Since all input files used by Studierstube are using ASCII-based file formats, it is possible to implement the complete transformation of the APRIL file(s) to Studierstube input files in XSLT. Studierstube uses Open Inventor as its content format, therefore components used in the presentation have to provide an Open Inventor implementation section. Additionally, as some of the concepts like state-engine driven presentations or light projectors are not supported by Studierstube, the necessary extension classes have been implemented and integrated into the Studierstube framework.

In the following sections, the details about the necessary transformations and the extensions added to the Studierstube framework will be presented in detail. In addition, the pre-processing step to simplify statecharts, saved as XMI, to include them in APRIL presentations, is explained.

5.3.1 XMI to APRIL translation

As mentioned in section 3.3, the official standard to save UML diagrams to XML files, XMI, is a complicated standard, allowing for many syntactic variations and alternative representations of identical models. We therefore want to introduce an automated simplification step, implemented in XSLT, to be able to include storyboards created with external UML tools into an APRIL presentation, using the simplified syntax provided by APRIL.

A single XMI document can contain, among other UML diagrams, multiple state diagrams. We want to extract the state diagram information, producing a separate output file for each diagram encountered in the input, ignoring the other kinds of UML diagrams that might be present in the input. The transformation script therefore defines a template for the root element that selects all the `UML:StateMachine` elements (the XMI wrapper element for a state machine) in an `xsl:apply-template` call, effectively applying the remaining templates only to these elements, ignoring all others.

The template matching `UML:StateMachine` creates a new output file with the name of the state machine, and calls the templates for its children, ignoring the first `UML:CompositeState` in the hierarchy, which represents the overall state machine and is always represented by the `story` element generated as the root element of the output file. From this point on, `UML:SimpleState` and `UML:CompositeState` elements in the input file are mapped to `scene` element in the APRIL file, and `UML:Transition` elements are mapped to `transition` elements.

Since references to other elements in the XMI file are using machine-generated, unique identifiers, while APRIL uses the human-readable `id` at-

tribute, usually containing the name of the state or the transition, a mapping of the unique id's used by XMI to state and transition names has to be created. For every state, referenced by a transition as its source or target, the state element referenced by the unique id has to be queried from the input XMI file, and its name is used as its identifier for the APRIL output.

Comments, which can be added to the UML diagram, are also preserved, allowing to put remarks for content-creation or presentation design directly into the UML diagram. These comments, stored in `UML:Comment` elements, will be exported to the APRIL file as `annotation` elements inside the states they are referencing.

5.3.2 APRIL to Studierstube translation

For our prototype implementation, an APRIL presentation, including it's hardware- and tracking-configuration and component definition files, will be transformed into a complete set of Studierstube configuration files, including a batch file to be able to start the presentation with a double click on it's icon.

The first step in this process is to load the contents of the referenced hardware description file, if an external file is used, into an internal variable for later reference without having to parse that file over and over again. After that, the filenames for content and configuration output files are figured out, based on the presentation's name, and the batch file for starting Studierstube with the necessary input files is produced. The `story` element is extracted to a separate file, story.xml, which is loaded at run-time by the story engine (see section 5.3.3).

The remaining files to be generated are the configuration files containing `UserKit` definitions with displays and pointing devices, the tracking configuration files and the main application file, containing the presentation's content.

Displays and pointers

Since Studierstube relies on the concept of users and allows only one pen and one display per user, the following approach was chosen: Each `display` element encountered will generate a `UserKit`, containing a `DisplayKit`, defining the displays size and position on screen and referencing the camera configuration file, which is also generated from the information contained in the `display` element. Of the pointers assigned to the display (possibly more than one), only the first one is defined in the `UserKit` node – all other pointers are written directly to the application file and therefore have no direct

connection to a user in the Studierstube model.

Depending on the type of pointer(s) used (which is defined in the `type` attribute of the `pointer` element), either a `PenKit` or a `RayPicker` node is generated. If a raypicker is used, an additional OpenTracker station has to be provided, routing the output of the raypicker (the picked point) to the application (see below).

For the `DisplayKit`, the screen coordinates and size is calculated from the information in the setup file, which allows the specification of percentage values or the keyword `fullscreen` for the display size. These values have to be converted to absolute pixel values on a single, virtual screen to be used by Studierstube. For this purpose, the information specified in the `screen` elements in the setup file is used to calculate the resulting pixel positions.

All OpenTracker elements used in the setup file are collected and copied to a single OpenTracker file, optionally including content from another OpenTracker file specified in the `otfile` attribute of the `setup` element. All OpenTracker sources are wrapped by `StbSink` elements and assigned a unique, automatically generated station number, which is used for subsequent references of the corresponding tracker element. In addition, as already mentioned, for all `RayPicker` nodes, `StbSource` elements are introduced into the OpenTracker file, providing the tracking output of the raypickers to the application.

For tracked elements in the setup file that do not contain or reference OpenTracker elements to define their tracking behaviour, default `KeyboardSource` elements are generated to allow users to control these inputs from their keyboard. Additionally, a warning message is printed to the console to notify users of this fact.

Story engine

As stated already, the `story` element with the definition of a presentation's storyboard is copied unmodified to a separate file. This file will be read at runtime by the `StateEngine` node, that has been implemented to support the necessary runtime behaviour and is described in section 5.3.3. The `StateEngine` node acts as a black box, taking event tokens, generated by interaction components, as input and exposes the names of the currently active states and available transitions.

This story-engine is the central part of the presentation, triggered by interactions and triggering behaviours. Therefore, most of the engines created to drive the presentation will be connected to the story engine with one of their inputs or outputs.

Components and actors

Since components are essentially code templates, they can easily be processed by XSLT to be included in the final presentation. For each instantiation of a component (any `actor` element referencing the component in question), the whole implementation body of the component is simply copied to the content file.

If a marker for an input field (marked by the `in` element) is encountered, the transformation script looks up the behaviours of all scenes, if any behaviour is defined on the given input. If no behaviour is defined, the `in` marker is simply replaced by the default value for the given field, specified in the `actor` element or the component definition file. If some behaviours for the field are found, this means the field will change during runtime, depending on the state of the story engine. In this case, engines are created to control the value of the field, dependent of the current state of the story engine. This case will be described in detail in the next section.

Locations for parts or sub-components are indicated by a `sub` marker element in the component file. If such a marker is encountered, the transformation script performs a lookup in the `actor` element if any sub-components for that part are defined. If this is the case, the processor will just continue to create that sub-component in a recursive process, and, after processing all children, continue with the current component. If no sub-components are found, the marker element is simply ignored.

Output fields are not directly used in the component instantiation process. Only if a `connect` behaviour or `comparator` interaction make use of the value of the output field, a reference to the field is created, using the information in the `out` marker element.

Behaviours

Since all behaviours are essentially controlling the value of a single field, they can be implemented as engines controlling the specified field of an actor. Since a single field can be controlled by multiple (sequential) behaviours, a fan-in for values generated by behaviours has to be created to route all value changes into the corresponding field. Open Inventor does not provide a mechanism for fanning in multiple engine outputs into a single field, therefore a `FanIn` node has been implemented, which takes up to 10 inputs of a given type and routes only the one that was most recently changed to its output (which is, in turn, connected to the input of the component).

Since we cannot rely on the fact that only 10 behaviours are defined for each field (in fact, this limit is quickly reached even in simple presentations),

additional measures have to be taken to allow an arbitrary number of inputs for each field. For each behaviour, the XSLT transformation script checks if it's position (starting at 0) in the list of all behaviours assigned to a specific field is a multiple of 9. If this is the case, the last input of the current `FanIn` node has been reached, and another `FanIn` node is created and connected to the last input of the previous `FanIn` node. The behaviour engine is then connected to the first input of the new node.

After the last behaviour has been written, the generated `FanIn` nodes have to be completed by closing brackets. This is realized by a helper template that generates as many closing brackets as `FanIn` nodes have been generated, by recursively calling itself, outputting a closing bracket and reducing the counter by 9 with every call.

The individual behaviours themselves are realized as engines, which are triggered by changes in the story engine. If the state, that a given behaviour should react on, has been reached, the behaviour is triggered (or, if a time has been specified, the behaviour is triggered by an additional `OneShot` engine, waiting for the specified period of time after the scene has been reached). In the case of a `set` behaviour, the behaviour is realized by a `Gate` engine, holding the new value and copying it to the output when triggered.

Animations are realized with interpolator engines, provided by Open Inventor. The interpolator engines linearly interpolate a value between a starting value and an end value. The interpolation itself is driven by a `OneShot` engine, interpolating it's output from 0 to 1 in a given time frame. This number is then used to control the interpolation of the field value.

Connections of fields are defined with the `connect` behaviour, and, as for the `set` behaviour, realized with the `Gate` engine provided by Open Inventor. In the case of field connections, the output of the source field is referenced in the `Gate` engine as an input, while the `enable` field of the gate, that controls if the value is copied to the output, is controlled by an `OnOff` engine reacting to the current state of the story.

Figure 5.2 shows an example of the code that is generated to control a single field by `set`, `animate` and `connect` behaviours in subsequent states. This also illustrates how complex it would be to implement the high-level features and concepts that APRIL provides directly in Open Inventor.

Interaction

Since all interactions generate input events for the story engine, all nodes that represent the various interaction techniques have to be connected to the `StateEngine` node in the presentation. Similar to the approach chosen for behaviours, all interactions are connected using multiple nested `FanIn` nodes

```
DEF ball_model_Material Material { # from the component definition
  transparency 0.0      # the default value from the actor element
    = DEF val_ball_transparency SoFanIn { type MFFloat
# <set actor="ball" input="transparency" to="1.0"/> in first:exit
      in0 = DEF set_d0e124 Gate { type MFFloat
        input 1.0
        trigger = DEF trigger_d0e124 SoConditionalTrigger {
          triggerString "first"
          stringIn "" = USE StoryEngine.exitStates
        }.trigger
      }.output
# <animate actor="ball" input="transparency" from="1.0" to="0.0"
#  begin="PT1.0S" duration="PT2.0S"/>
      in1 = DEF anim_d0e132 InterpolateFloat {
        input0 1.0
        input1 0.0
        alpha 0 = OneShot {
          duration 2.0
          disable TRUE = DEF start_d0e132 OneShot {
            duration 1.0
            trigger = DEF trigger_d0e132 SoConditionalTrigger {
              triggerString "second"
              stringIn "" = USE StoryEngine.currentState
            }.trigger
          }.isActive
          trigger = USE start_d0e132.isActive
        }.ramp
      }.output
# <connect actor="ball" input="transparency" master="ghost" output="transparency"/>
      in2 = DEF connect_d0e146 Gate { type MFFloat
        input = USE ghost_model_Material.transparency
        enable = OnOff {
          on = SoConditionalTrigger {
            triggerString "third"
            stringIn "" = USE StoryEngine.currentState
          }.trigger
          off = SoConditionalTrigger {
            triggerString "third"
            stringIn "" = USE StoryEngine.exitStates
          }.trigger
        }.isOn
      }.output
    }.out
}
```

Figure 5.2: Generated code to control a single field by `set`, `animate` and `connect` behaviours. Comments have been added by the author to annotate the code and show the original APRIL definitions.

to route all events to the `eventIn` field of the `StateEngine`. Interaction nodes are outputting a string token, containing the name of the event that has been generated.

For `buttonaction` elements, a button widget is created and placed on the HUD of all displays. The HUD is rendered by placing its content in a `Separator`, that contains an `OrthographicCamera` as its first node which renders the content in a static, orthographic way, not affected by head-tracking of the users.

Depending on the availability of the transition that the button is bound to, it is dynamically hidden or shown to the user. Therefore, only buttons that will actually trigger a transition in the story will be visible at any time in the presentation. Additionally, all buttons are hidden during the `enter` substate of any scene, since it has been defined that the enter section should be guaranteed to execute and not be interruptible by user interaction.

Clickable hotspots are implemented as APRIL components, that can be triggered by using any pointer defined in the setup configuration file. The hotspot component has two `SFBool` outputs for detecting interactions, `touched` and `triggered`. These outputs can be used for interaction by evaluating their value with an `evaluator`.

The `evaluator` interaction is a generic tool that can be used to trigger changes in the story whenever an output of an actor fulfills a given condition. Fields can be compared to a constant value or to other outputs of other actors. In Studierstube, the `evaluator` is implemented differently for different field types – simple types simply use the `ConditionalTrigger` engine implemented to support the story engine (see next section), while composed types such as `Vec3f` use a `Calculator` engine to compare their values.

The remaining interactions, `always` and `timeout`, are implemented by just triggering an event whenever the parent state is reached. This is accomplished by using a `ConditionalTrigger`, which will be explained in the next section.

5.3.3 The story engine

To be able to realize the desired state-engine based run-time behaviour of APRIL presentations, Open Inventor had to be extended by some nodes that implement the desired functionality. This is the `StateEngine` node, for providing a black-box implementation of a generic hierarchical state engine, and some helper engines to support the complex trigger networks, controlling behaviours and interactions, connected to the state engine.

The state engine has been realized in C++ and is a generic implementation of the necessary behaviour. The binding to Open Inventor is provided

through a separate adapter class, that exposes the necessary fields as an engine (namely, an `eventIn` field for routing events into the state engine, and the fields `currentState`, `availableTransitions` and `exitStates` to query the current state).

In addition, a field is provided to specify the name of a file, from which the specification of the state engine's structure (defined in the APRIL story format) should be read. This file is parsed by the generic state engine implementation with a SAX-compatible parser [33], which builds up an internal representation of the state engine defined in the APRIL file. For this internal representation, the classes `State`, `Transition` and `StateEngine` are used, together with the struct `Region` to represent one of possibly multiple concurrent regions inside a state.

The basic interface of the C++ implementation is the `event(string token)` method of the `StateEngine` class. When this method is called, with the name of an event as an argument, a recursive lookup is performed on all transitions currently available, in all substates. If a transition is found to match the event, the transitions target state becomes the new current substate of its parent state, following the transitions source state, which is now inactive. The `event()` method is replicated in the Open Inventor adapter by the eventIn field. Whenever this field changes, its value is simply routed to the `event()` call of the encapsulated `StateEngine`. After processing the event, the `StateEngine` is queried for the active states, which are written to the `currentState` field.

To be able to generate event-strings in the Open Inventor engine network, a versatile helper class, named `ConditionalTrigger` is provided. The `ConditionalTrigger` has inputs for every primitive type (`MFBool`, `MFInt32`, `MFFloat` and `MFString`), and inputs for *reference values* of these types. The semantics of the `ConditionalTrigger` is as follows: if *any* of the input values of *all* used input types are equal to their reference value, the `trigger` output of the engine fires. In addition, the string value set in the `token` field is copied to the `tokenOut` output.

Usually, the `ConditionalTrigger` is used with only one input type connected. Whenever the input reaches its reference value (other comparisons, like `LESS_THAN` or `GREATER_THAN` can also be configured), the `tokenOut` output is set to the string defined in the `token` field. If the `token` field contains an event name, and the `tokenOut` field is connected to the `eventIn` field of the state engine, this can be used to effectively generate an event in the state engine whenever a certain condition is met.

The `ConditionalTrigger` can also be used the other way round: if the `currentState` output of the state engine is connected to the `stringIn` field, and the `triggerString` field contains a state name, the engine will trig-

ger every time the given state becomes active. This can be used to trigger behaviours, as shown in figure 5.2.

Chapter 6

Results

In this chapter some early results accomplished with the APRIL authoring framework will be presented. Three applications will be presented, intended for three different target platforms. The first presentation deals with an archaeological subject, the "Heidentor" in Carnuntum/Austria, which will be presented in a virtual showcase system. The second scenario implements the well-known "Magic Book" scenario, which has been used for a long time to demonstrate the capabilities of ARToolkit, allowing children to interactively explore a story in a (real) book. In the third scenario, APRIL is used to control the content of an AR outdoor tourist guide application, explaining details about the Resselpark in Vienna.

6.1 Scenario: The Heidentor in the virtual showcase

The Heidentor (Heathen Gate) [40] is an ancient roman ruin, located in Petronell-Carnuntum/Austria, and probably the most well known roman ruin in Austria (Fig. 6.1). Originally, the Heidentor was not a gate, but it had 4 pylons forming a so called *tetrapylum*, a double-passage arc, located at the intersection of two major roads. The pylons were supporting a 2-floor building on top of them. Today, only 2 of the pylons are still intact, and form an impressive, gate-like ruin, which is visible from far away. The exact original purpose of the Heidentor remains unclear; it might have been a tomb or a triumphal arch. Due to its impressive size and location in the countryside, it has inspired artists and storytellers over the last centuries, and a lot of myths focus on this historical site.

For the virtual showcase, we have built a scale model of the current Heidentor ruin, to be augmented with additional explanations and reconstruc-

Figure 6.1: The Heidentor ruin, located in Carnuntum/Austria. A low-tech augmentation device is used on site to show a possible reconstruction of its original state.

tions of the original building. Users should be able to interactively explore the model and discover explanations about the historical facts as they go along. The interactive features should help to gain attention and raise interest in the subject by offering the possibility to explore it individually instead of just watching a linear presentation.

The scale model was built out of cardboard based on exact archaeological plans and measurements. It was then laser-scanned to obtain a virtual model, to be used as a proxy for the real content for correct lighting and occlusion effects (see section 2.2.1). In addition, the virtual model can be used for purely virtual presentations outside the showcase and during development of the presentation as a placeholder for the real contents of the showcase.

For the virtual part of the presentation, we use two different historical reconstructions of the original building, as well as photographs, sketches, and virtual models of a tombstone and a statue. After the introduction, the visitor can either use virtual buttons to advance the presentation in a linear fashion, or use the provided tools for interactive exploration to get additional explanations for certain parts of the building.

Since the model is enclosed inside the showcase, visitors cannot interact directly in the space of the model. Instead, a virtual laser pointer is used to enable users to point at parts of the model that they are interested in

(Fig. 6.2). A 6DOF tracking device is used to control the raypicker, so it does not have to be assigned to a specific display in the setup configuration file, and is available to all users of the presentation. When a part is selected by the laser pointer, it is highlighted and the appropriate information is shown. This might be an image or a video displayed on a plane inside the showcase, possibly accompanied by audio commentary, or a whole scene involving virtual actors and audiovisual information.

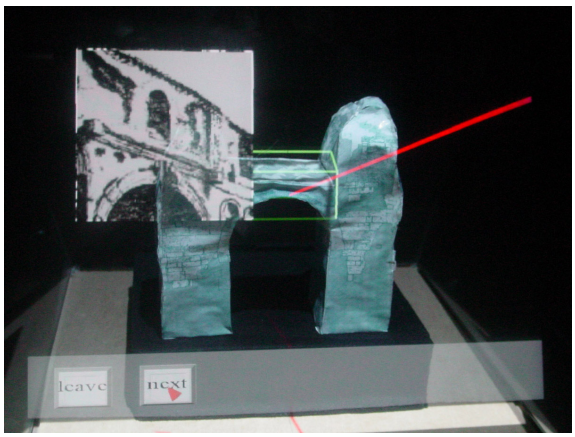

Figure 6.2: Using a virtual laser pointer to explore additional information. Here, a sketch of the reconstruction of a similar building is shown.

Another possibility for user interaction is using a virtual clipping plane, to "cut away" the virtual model and make the real model underneath it visible (Fig. 6.3). To realize the clipping plane effect, a custom component has been implemented that provides geometry clipping of its children. Position and orientation of the clipping plane are `connect`ed to a `station` that can be used to control its location. The clipping plane helps the visitors understand the relationships between the ruin that can be seen today, and the original state of the building. The same technique can also be used to blend between and compare different possible reconstructions of the same building.

6.2 Scenario: A magic book

The original magic book presentation, created by Billinghurst et. al. [16], used the ARToolkit optical tracking system to create an augmented reality reading experience for children: Markers were attached to the pages of a book, and through a set of video-see-through glasses, the user could explore virtual models "attached" to the pages of the book. Additionally, by pressing

Figure 6.3: Blending between the real model and its reconstruction with a virtual clipping plane.

a button, the user could "dive into" the page and explore a VR presentation of the story.

Our magic book scenario was realized on a desktop setup, using a moveable webcam to capture the book and display the image on the computer screen. With this approach, multiple users at the same time can view the presentation. In addition to the markers on the pages, two interaction markers were used: a "read" marker, which, when put onto the current page, would trigger an audio file playback, reading the story of the current page. The other marker, when put on top of the page, would display additional geometry, illustrating the story.

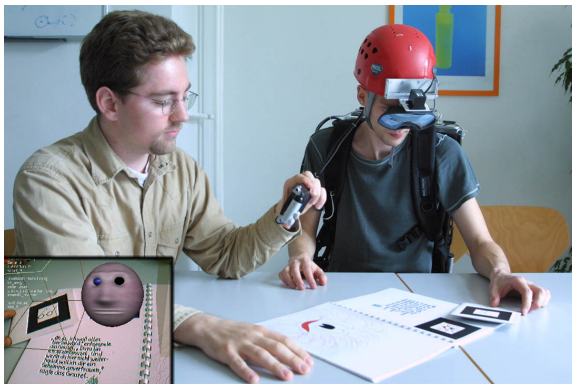

Figure 6.4: The magic book application.

In the storyboard, transitions from an "idle" state, where no page-markers are visible, to each of the pages, which were modelled as separate states, had to be taken into account. Additionally, transitions from each page to all the other pages, and to the idle state, were added to the storyboard. This would allow the user to browse through the book in a non-linear manner. The alternative would have been to only allow a linear path through the book, displaying an error message when the intended order had not been followed by the user.

Each of the scenes for the pages was further divided in two concurrent substates – whether the "read" marker was visible, and whether the "content" marker was visible. The resulting state engine (for simplicity containing only two pages) is shown in figure 6.5.

ARToolkit generates events only when markers are visible – no event is generated, when a marker is not visible any more in the camera image. Therefore, the detection of marker visibility had to be implemented on the OpenTracker level: if for a given time span (800 milliseconds) no updates of a marker's position are received, the marker is considered to be invisible, and it's position information is set to a fixed value, outside the visible area of our application. In the interactions section, we can then just use an `evaluator`, to check whether a certain marker is at the "invisible" position, and react accordingly.

For the behaviours, the subscenes representing the visibility of the interaction markers simply switch the `visibility` input field of the model actor, or the `play` input of the sound actor for that scene to `FALSE` or `TRUE`, reflecting the state of the corresponding marker.

6.3 Scenario: Outdoor tourist guide

For the third scenario presented here, the goal was to interface with an existing Studierstube application: An outdoor tourist application, tracking the user by GPS, developed by Reitmayr [62]. While this application provides already basic capabilities for overlaying graphics in the HUD of the user or in the environment, creation of sophisticated content or stateful presentations like guided tours was, up to now, tedious.

To allow the tourist application to integrate with the APRIL presentation, a component had to be developed that acts as an interface between the native Studierstube application and the presentation. This component interacts with the navigation application through global fields, a mechanism provided by Open Inventor to create fields that are accessible from any point in the global scenegraph. The navigation application creates these global fields, and

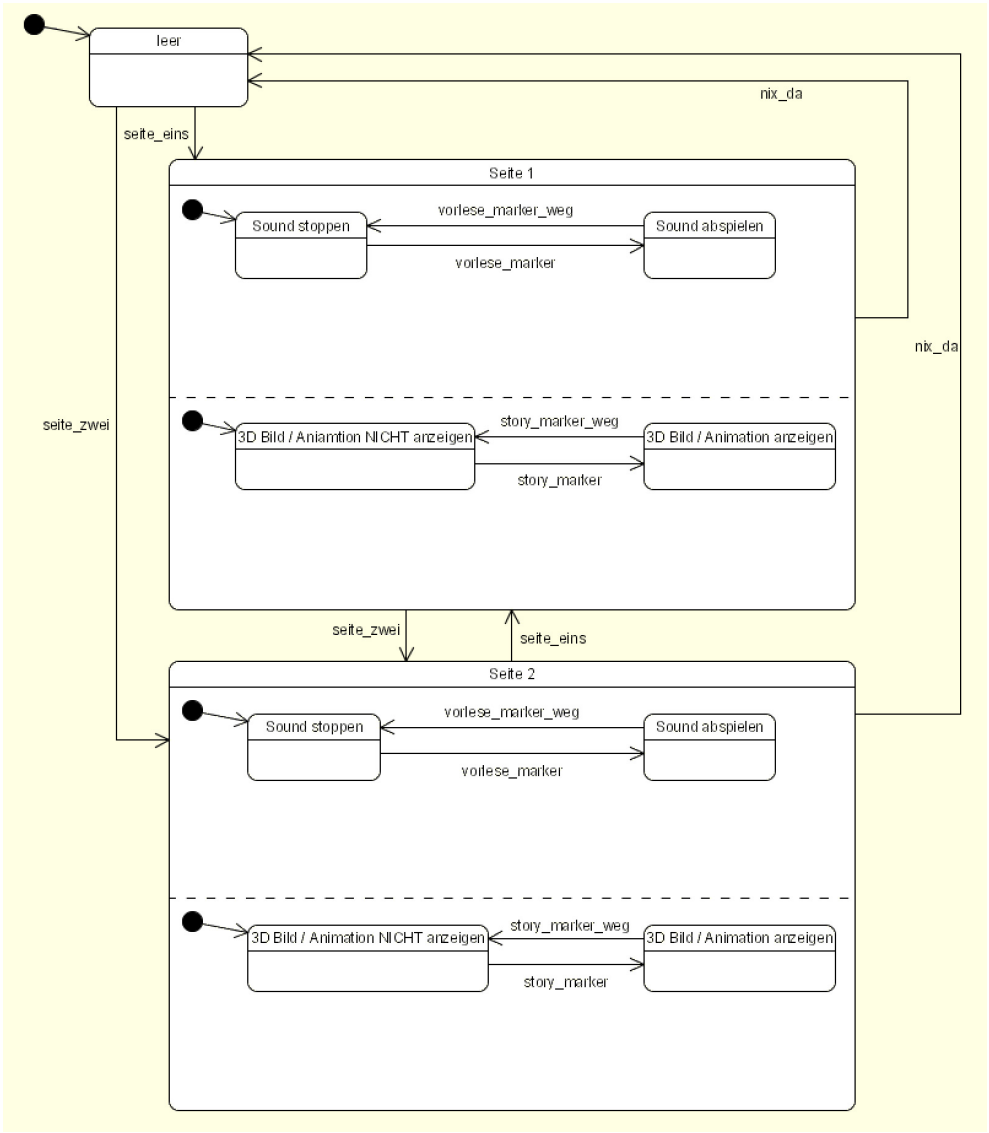

Figure 6.5: The storyboard of the magic book application. For simplicity, only the states and transitions for two pages are shown.

outputs the navigation data (such as the position of the user, or the name of the current waypoint) to these fields. The global fields are referenced in the APRIL navigation component, and can therefore be read from and written to from within the APRIL presentation.

The story is primarily driven by the user's position, and depending on the selected navigation mode: in the guided tour mode, the user is guided by Clara, out virtual tour guide, from one interesting waypoint to the other. The system always waits until the correct waypoint is reached, to continue with the presentation. In the free mode, the user can walk around freely in the environment, and will be notified every time she comes close to some interesting place or artifact.

Additional components have been developed for interactive content: to simulate spatial audio, the position information, provided by the navigation component, is fed into a `Calculator` engine to calculate the distance from the sound source. This information is then connected to the `volume` input of the sound component, increasing the volume as the user comes closer to the audio content.

Chapter 7

Conclusions and Future Work

In the previous chapters, a novel authoring approach for Augmented Reality and Virtual Reality presentations has been presented. Support of multiple AR systems and setups, explicit structuring of the presentation through a hierarchical, concurrent state engine, used as a storyboard as well as re-usable components and the flexible binding of interaction methods to transitions of the state engine are the key features of the proposed solution.

In practice, the explicit structuring of the presentation, accomplished by using standardized UML-statecharts as a storyboard, allows even non-programmers to invent and implement complex, interactive presentations. Additionally, through structuring the story explicitly (in contrast to the implicit definition of presentation behaviour that can be found in scripting or hypermedia solutions), the presentation behaves deterministic and the overall state of the story can, at any point in time, be deduced from the individual substates that are currently active. This has helped authors with debugging their presentations, and allows teamwork and an integrated workflow, including various professionals from different domains in the design process, by using the UML storyboard as a central artifact for communication.

Another design choice that has a great influence on the possible solutions that can be realized with APRIL is the decision to not provide a proprietary content format, but support a generic component model that can make use of whatever content format the target platform supports. While this means that components have to be re-implemented for different runtime platforms, the chosen approach allows component implementers to optimize their components for the given target platform and use special features that are only available on that specific system. The alternative solution, to create a generic (XML-based) content format that can be translated to all runtime platforms, would have meant that only the minimum subset of features, supported by all platforms, could have been supported. Additionally, such a standard would

have to be maintained and updated continuously, to reflect the current state of the art of realtime rendering platforms. By using the platform's native content format, improvements and new developments can be used immediately by the developer, without having to wait for the APRIL specification to evolve.

The rationale behind providing the component model was to provide a mechanism for extending the features of APRIL beyond the anticipated needs of presentation developers. If only a fixed set of features or content types would be provided (e.g. models, images, video, sound and transformation groups), these could be represented by elements provided by the APRIL specification. However, presentation developers would then be constrained to use exactly these features, and could not invent new content types. As we saw in the results section, this was already necessary to provide the clipping plane feature of the Heidentor presentation or the proxy component of the pathfinding application for the outdoor tourist guide.

In contrast to actors, behaviours and interactions are not extensible by using a component model, but limited to only those provided by the APRIL language. For behaviours, the four provided basic types – `set`, `animate`, `connect` and `control` – and the possibility to arrange them on a timeline seem to cover all needs of controlling an input field of an actor. For sophisticated calculations of or user interaction with an input field's value, the field can always be connected to an output of a custom component that performs the necessary calculations. For interactions, advanced presentation authors can also encapsulate user interface elements in custom components, and use an `evaluator` interaction to trigger state changes. However, since user interaction is usually more complex that simply calculating or comparing field values, and also non-programmers may want to use different and more sophisticated interactions for their presentations, the catalog of interaction concepts provided by APRIL should be continuously expanded to include new techniques.

Another key feature of the design of APRIL is the possibility to separate the definition of a presentation's content and the description of the hardware setup the presentation should run on. This allows users to run a single presentation on different setups, or a single setup description file to be used by several presentations. However, we are facing a common computer science problem here: if we want to keep a loose coupling between the two fields, it is not possible to reference elements from one file (e.g. a `station` in the hardware description) in the other – authors cannot be sure that the referenced element will exist in the setup description file that is used for running the presentation. One way to deal with this problem is the use of *naming conventions* – if all APRIL users agree to always name the `headtracking`

elements in their setup files `head1`, `head2`,..., then the headtracking data of the first user can always be referenced with the id `head1`. If no such element is found, an error message can be generated, and the presentation will not run.

A better way to deal with the issue of loose coupling is the introduction of *roles*: to establish a fixed, well-defined roles, like, for example, `HEAD` as the headtracking of the default user, and `HEAD1` as the headtracking of the primary user. Roles are assigned to elements in addition to their `id`, and any element can have multiple roles assigned to it. Additionally, default roles will be assigned if no roles are explicitly specified, for example both roles `HEAD` and `HEAD1` can, by default, be assigned to the first `headtracking` element encountered in the presentation. For creating a reference, a list of roles is specified that should be used to provide the necessary reference. This list specifies the roles that should be looked for, in order of decreasing priority. For example, the list `"HEAD1, HEAD"` would cause the APRIL runtime to first look for the headtracking element of the primary user, and if no such element is found, use the default headtracking element of the specific hardware setup (note that in most simple setups, this will probably be the same element). Only if no element is found that fulfills any of the criteria, an error would be produced. An additional attribute could be used to specify if in the case of an error the presentation cannot run, or if it should be ignored.

Currently, no mechanism for assigning and referencing roles is provided by APRIL. For realizing complex presentations that should run on multiple, inhomogeneous setups, the concept of roles would allow to keep the coupling between the presentation's content and the hardware description low, while at the same time allowing the detailed specification of complex presentation behaviour and user interaction.

In other cases, it is also not possible to separate hardware and content descriptions as clearly as desirable. Sometimes, even virtual content can be part of a hardware setup – for example, in desktop setups used for authoring and debugging presentations for other platforms, one might want to reproduce some of the properties of the target platform (for example the dimensions and geometry of the mirror optics of a showcase system) as virtual content, that is in this case not part of the presentation, but part of the specific hardware setup. In the future, we might want to introduce the possibility of placing content also in the hardware description file, to allow the creation of more sophisticated simulator setups.

Another are where APRIL is currently limited in its possibilities is the placing of content in the environment: actors can either be placed on the HUD of the user, or in the world, using absolute world coordinates to specify the location and size of the object. While group components, that allow

to transform a group of children simultaneously and are the building blocks of a tree-like scenegraph structure, provide some means to structure a presentation's content, they are not the versatile mechanism for specifying the intended location of a presentation's content that we are looking for. Instead, the concept of *stages* could be introduced, allowing authors to specify possible locations for content in the setup description file. Examples for stages would be the world space, the HUD, a small model of the world, tracked by a marker, or a 2-dimensional plane, displaying rendered content (like a TV-screen placed in the AR scene). To reference the target stage(s) for a given actor, the roles mechanism, explained above, could be used to avoid strong coupling between content and hardware description files.

Regarding the reference implementation, future work would primarily focus on supporting multiple operating systems and platforms. While Studierstube already runs on the Linux operating system, and it would therefore be trivial to make APRIL also work on this platform, the support of handheld devices for running AR presentations is more challenging. While a large and complex system like Studierstube will not run on the handheld devices that can be expected to be available in the next years, realtime graphics libraries like OpenGL and Open Inventor have already been ported to these platforms. It would therefore be a goal for the future to provide an "APRIL player light" version, that runs APRIL presentations on handheld devices, providing all features that can be supported on these devices (for example, it will not be necessary to provide multiple displays or headtracking for handheld devices).

Finally, the preferred way to create AR presentations is probably not that of editing XML files manually. While APRIL provides a format for the description of content, behaviours and interaction, it would be very much desirable to create a visual authoring tool, that makes us of the APRIL format and its features, but allows authors to interactively place actors in space and wire their interactions and behaviours by providing a visual representation for the abstract concepts of APRIL. Such an authoring platform would use APRIL as its file format to save and load presentations, and offer the author an intuitive way to explore the full power of the proposed solution.

Appendix A

The annotated graphical APRIL schema

Appendix B

The APRIL language specification

B.1 Global Simple Types

FloatList	List of double values, base type for restricted data types like vectors, rotations, etc.
IntegerList	List of integer values, useful for list of indices.
Vec2f	A simple type storing two float values, separated by spaces.
Vec3f	A simple type storing three float values separated by spaces.
Vec4f	A simple type storing four float values separated by spaces.
Vec2i	A simple type storing two integer values separated by spaces.
ScreenSizeType	Either two numbers, indicating absolute pixels, or two percentage values, or the keyword "fullscreen".

B.2 Top Level Elements

april

The root element of every APRIL file.

presentation

This is the top-level wrapper element of a APRIL presentation.

Attributes

attribute	type	description
id	xs:ID	Unique ID for this presentation. Depending on the implementation of the transformation of APRIL to a specific platform, this will be used for generating file- or folder names for the resulting files.
name	xs:string	The name of the presentation. This can be used by target platforms as a human-readable title for the presentation.

Allowed children
`story, cast, behaviors, interactions`

B.3 Hardware Description Types

TrackableType

Base type for trackable objects. Allows specification of an OT source either by reference or by inline OT code.

Attributes

attribute	type	description
id		ID to identify this element.
otsource		DEF name of an OpenTracker node in the included OpenTracker file (or an earlier configured inline OT node)

B.4 Hardware Description Elements

setup

Wrapper element for input and output device configuration. The config section of the APRIL file specified as src is imported (if present), and overlayed with the information of the local config section (this means that local elements with the same id as in the src file, and their child-elements, override the elements specified in the src file). The low-level tracking configuration is done in the OpenTracker file specified with the otsource attribute, or inline in the corresponding interaction elements (see below). If there is no setup element, APRIL looks for a file called "default.aps" in the presentation's directory.

Attributes

attribute	type	description
src	xs:anyURI	URI of external file containing the config section to use. If a src is specified, any inline specifications will be ignored and the configuration specified in the given file will be used.
otfile	xs:anyURI	URI of OpenTracker config file to use as a basis for the tracking configuration. Named OpenTracker nodes referenced in the APRIL file are referenced from this file.
multicast-address	xs:string	
multicast-baseport	xs:integer	

Allowed children
`host`

host

Wrapper element for all devices connected to a single computer.

Attributes

attribute	type	description
name	xs:string	
ip	xs:string	
platform		
master	xs:boolean	

Allowed children
`screen, display, pointer, station, button`

screen

Configures a VGA output port.

Attributes

attribute	type	description
resolution	Vec2i	Screen resolution in pixels.

display

Defines a display for rendering content on. The containing displaytracking and/or headtracking elements define optional tracking, this elements configures the position of the display window on an output port, its resolution and the default camera pose in the world.

Attributes

attribute	**type**	**description**
id	xs:ID	ID to uniquely identify this display.
screen	xs:int	The screen number this display window should be rendered on.
screenPosition	ScreenSizeType	position of the top left corner of the display window on the screen in pixels or percentage.
screenSize	ScreenSizeType	Size of the display on screen in pixels or percentage.
stereo	xs:boolean	Flag indicating if frame-interleved stereo should be used.
worldPosition	Vec3f	The position of the center of the image plane in world coordinates. Overridden if a displaytracking element is used.
worldSize	Vec2f	Size of the image plane in the world.
worldOrientation	Vec4f	Orientation of the image plane in the world. Overridden if a displaytracking element is used.
eyeOffsetL	Vec3f	Offset of the left eye for a stereo display.
eyeOffsetR	Vec3f	Offset of the right eye for stereo displays.
viewpointPosition	Vec3f	The world position of the viewpoint. Overridden if a headtrtacking element is used.
mode		"AR" (default value) displays the actors of the presentation. "VR" also renders the stage elements for a virtual reality view onto the scene.
debug	xs:boolean	If set to true, debug information is shown in the display.
videoBackground	xs:boolean	

Allowed children

`headtracking?, displaytracking?`

headtracking
(of Type: `TrackableType`)

Configures the headtracking for the parent display element.

displaytracking
(of Type: `TrackableType`)

Configures tracking of the center of the display.

pointer
(of Type: `TrackableType`)

If used as a child element of a display element, it configures a pointing device for the display, which can be used to interact with widgets or point on real/virtual things. If used as a direct child of the host element, this is a general, tracked pointing device that can be used for all displays of the application.

Attributes

attribute	type	description
type		
debug	xs:boolean	

station
(of Type: `TrackableType`)

Defines a tracked artifact like a marker.

Attributes

attribute	type	description
debug	xs:boolean	

button
(of Type: `TrackableType`)

A hardware button that can be used for user input.

Attributes

attribute	type	description
key	xs:string	If no OpenTracker source for the button is given, this defines the keyboard key to use as an input for this button. If an OpenTracker source is given (by inline OT code or the ot-source attribute), it defines the button number to use (default=0)
function	xs:string	

B.5 Storyboard Elements

story

The content of this element specifies the logic (or the "story") of the presentation as a hierarchical state machine. The syntax is derived from XMI, the official UML-to-XML serialization syntax, but simplified for better readability.

cast

This element holds all references to used media resources (models, images, sounds, scripts etc.).

stage
(of Type: `VisibleActorType`)

Defines geometry as a proxy for real world objects. In augmented reality applications, this is not displayed, but used for correct rendering of intersections between virtual and real content, for example. In VR and authoring setups, these objects may be rendered as regular scene objects.

Attributes

attribute	type	description
position	Vec3f	Position of the object in world coordinates.
orientation	Vec4f	Orientation of the object in the world.
scale	Vec3f	Scale factor for this object.

scene
(of Type: `Scene`)

A state in a hierarchical state machine.

transition

A transition from one state to another, specifying the transition event name.

Attributes

attribute	type	description
event	xs:Name	Name of the event to trigger the transition.
source	xs:Name	Id of the source scene of the transition.
target	xs:Name	Id of the target scene of the transition.
guard	xs:string	A guard string, that can contian the id of another state. This means that the transition is only available if the story is currently in this state.

concurrentScene
(of Type: `Scene`)

Wrapper element to seperate multiple concurrent sub-engines inside a superstate.

annotation

Inside this element, any annotation might be added to describe what is happening inside this scene.

actor

Defines an actor of the presentation as an instance of an APRIL component. Inside the actor element, default values for its input fields can be set by using input elements.

Attributes

attribute	type	description
id	xs:ID	Unique id of the actor.
src	xs:anyURI	Source of the component definition file.

Allowed children
`input`

input

Defines the default value for a specific input for this presentation.

Attributes

attribute	type	description
id	xs:normalizedString	Id ofthe field to set.
value	xs:string	Default value of that field.

children

Attributes

attribute	type	description
id	xs:string	

B.6 Behaviour Types

ActionType

Base type for set, animate, connect and control elements.

Attributes

attribute	type	description
input		Which input field of the actor to set.
actor		The target actor.

LabelType

Base type for entry, exit, do elements.

B.7 Behaviour Elements

behaviors

The behaviours section binds behaviours of objects to states of the story logic.

Allowed children
`behavior`

behavior

A behaviour is a set of property changes bound to a state of the tory logic. Whenever the state is entered, the actions defined in the "entry" sub-element

are performed. If the story is still in this state after perfomring the entry actions, the "do" actions are performed, and can be interrupted at any time by leaving the state. Upon leving the state, the "exit" actions are guaranteed to perform.

Attributes

attribute	type	description
scene	xs:IDREF	Id of the scene this behaviour should be bound to.

Allowed children
`entry?, exit?, do?`

entry
(of Type: `LabelType`)

The "entry" actions are performed when the associated state is entered. Actions defined here are guaranteed to perform, now matter how long the story remains in this state. Actions should therefore only be "set" and "connect", and very short "animate"ions.

Attributes

attribute	type	description
duration	xs:duration	Allows explicit setting of the duration of the entry substate. If this is not used, the dration is calculated from the set and animate elements inside.

do
(of Type: `LabelType`)

Actions in this group are performed as long as the story is in the associated state. Note that actions might not be called at all, if the state is immediately left. This is the only place where looped animations make sense, because they can be cancelled by leaving the state.

exit
(of Type: `LabelType`)

Actions defined here are executed when the associated state is left. They are guaranteed to perform.

set
(of Type: `ActionType`)

Sets the given input of a component instance to the given value.

Attributes

attribute	type	description
to	xs:string	New value for the input.
time	xs:duration	Time after enter/exit the change should happen.

animate
(of Type: `ActionType`)

Animates the given input. For further information, see the SMIL animation spec.

Attributes

attribute	type	description
begin	xs:duration	The start time of the animation, measured from the enter/exit time of the state.
duration	xs:duration	The duration of the animation.
to	xs:string	Target value of the input.
by	xs:string	Relative target value of the input. NOT IMPLEMENTED
calcMode	xs:token	CalcMode like defined in the SMIL animation spec. NOT IMPLEMENTED, always linear at the moment.
from	xs:string	

connect
(of Type: `ActionType`)

Connects an input of an actor to an output of another actor (master). The connection remains for the duration of the container state.

Attributes

attribute	type	description
master	xs:IDREF	Master actor to connect the field from.
output	xs:string	Which output of the master actor to use.

control
(of Type: `ActionType`)

Allows the user to control an input of an actor. If the id of a station is given, this station is used to control the input using its position or orientation output, depending on the dimensionality of the field; Float, Vec2f and Vec3f are using the position output, Rotation and Vec4f the rotation output), otherwise PUC is used to control the input.

Attributes

attribute	type	description
min	xs:anySimpleType	Lower limit of the value range that can be set.
max	xs:anySimpleType	Upper limit of the value range that can be set.
label	xs:string	A label shown to the user together with the control for changing the input.
station	xs:string	ID of a station (defined in the setup element) to be used to contol the property. Station inputs can only be used to control inputs with types SFVec3f or SFRotation.

B.8 Interaction Types

Interaction

Attributes

attribute	type	description
auto	xs:boolean	If set to true, the transition triggers automatically if it is the only available transition.

B.9 Interaction Elements

interactions

Allowed children
`event+`

event

Attributes

attribute	type	description
id	xs:ID	ID of the transition this interaction is mapped to.

Allowed children
`disabled?, always?, timeout+, ...`

disabled
(of Type: `Interaction`)

This disables a transition. Same as leaving it empty.

always
(of Type: `Interaction`)

This always triggers the transition immediately.

timeout
(of Type: `Interaction`)

This triggers the associated transition automatically after a specified time.

Attributes

attribute	type	description
time	xs:duration	Time after which the transition should trigger.

buttonaction
(of Type: `Interaction`)

This creates or uses a button when the transition becomes available, that can be pressed by the user to trigger the transition.

Attributes

attribute	type	description
virtual	xs:boolean	Whether to create a virtual button. DEPRECATED.
caption	xs:string	The label printed on the (virtual) button.
id	xs:string	

rayaction
(of Type: `Interaction`)

This allows the user to use a raypicker to point at objects in the scene. If the specified actor or stage object is selected, the transition fires.

Attributes

attribute	type	description
target	xs:IDREF	ID of the actor or stage object that must be selected to trigger the transition.

touch
(of Type: `Interaction`)

Attributes

attribute	type	description
source	xs:string	
target	xs:string	
radius	xs:float	

evaluator
(of Type: `Interaction`)

Evaluates an output of an actor or tracking element (station, head- or display-tracking) against a fixed value or another output. If the expression evaluates to true, the transition fires.

Attributes

attribute	**type**	**description**
source	xs:IDREF	Actor or tracking element that should be used as input.
output	xs:string	Output field of the source to use. For tracking elements, position and orientation are available outputs.
comparator		The kind of comparison to perform. One of equal (default), lessThan, greaterThan, lessOrEqual, greaterOrEqual, notEqual, isInside (bounding box check), notInside.
value	xs:string	The fixed value the source output should be compared to.
source2	xs:IDREF	A second actor or tracking element that delivers the output for comparison.
output2	xs:string	The output of the second source to be used.

Bibliography

[1] ArgoUML website. http://argouml.tigris.org/, November 2003.

[2] The VRML97 specification. Specification 14772-1:1997, ISO/IEC, 1997.

[3] Authorware website. http://www.macromedia.com/software/authorware/, November 2003.

[4] Director website. http://www.macromedia.com/software/director/, November 2003.

[5] Dramatica website. http://www.dramatica.com/, December 2003.

[6] FMOD website. http://www.fmod.org, November 2003.

[7] OpenAL website. http://www.openal.org/, November 2003.

[8] OpenTracker website. http://www.studierstube.org/opentracker, May 2003.

[9] Dynasight webpage. http://www.orin.com/3dtrack/, January 2004.

[10] Aristoteles. *Poetik*. Philipp Reclam jun., Stuttgart, Germany, 1994.

[11] A.R.T. GmbH. ART website. http://www.ar-tracking.de/, January 2004.

[12] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent advances in augmented reality. *Computers & Graphics*, November 2001.

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

[14] S. Beckhaus et al. alVRed – Tools for storytelling in virtual environments. Technical report, Fraunhofer IMK, Sankt Augustin, 2002.

[15] T. Berners-Lee and D. Conolly. Hypertext markup language, version 2.0.

[16] M. Billinghurst, H. Kato, S. Campbell, D. Hendrickson, W. Chinthammit, I. Poupyrev, and K. Takahashi. Magic book: Exploring transitions in collaborative AR interfaces. In *Proceedings of the Siggraph 2000 Conference*, New Orleans, Louisiana, July 2000.

[17] O. Bimber and B. Fröhlich. Occlusion shadows: Using projected light to generate realistic occlusion effects for view-dependent optical see-through displays. In *Proceedings of the International Symposion on Mixed and Augmented Reality (ISMAR) 2002*, pages 186–195, Darmstadt, Germany, October 2002. ACM and IEEE.

[18] O. Bimber, B. Fröhlich, D. Schmalstieg, and L. M. Encarnação. The virtual showcase. *IEEE Computer Graphics and Applications*, 21(6):48–55, November 2001.

[19] O. Bimber, A. Grundhöfer, G. Wetzstein, and S. Knödel. Consistent illumination within optical see-through augmented environments. In *Proceedings of the International Symposion on Mixed and Augmented Reality (ISMAR) 2003*, pages 198–207, Tokyo, Japan, October 7–10 2003. IEEE.

[20] D. A. Bowman and L. F. Hodges. User interface constraints for immersive virtual environment applications. Technical Report GITGVU-95-26, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, 1995.

[21] D. A. Bowman and C. A. Wingrave. Design and evaluation of menu systems for immersive virtual environments. In *Proceedings of IEEE Virtual Reality*, 2001.

[22] T. Bray, J. Paoli, C. M. Sperberg-McQueen, et al. Extensible markup language (XML), version 1.0 – W3C recommendation. http://www.w3.org/TR/REC-xml/.

[23] J. Clark. XSL transformations (XSLT) version 1.0 – W3C recommendation. http://www.w3.org/TR/xslt, 1999.

[24] J. Clark and S. DeRose. XML path language (XPath), version 1.0 – W3C recommendation. http://www.w3.org/TR/xpath, November 1999.

[25] M. Conway, R. Pausch, R. Gossweiler, and T. Burnette. Alice: A rapid prototyping system for building virtual environments. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, volume 2, pages 295–296, April 1994.

[26] R. Dietz. CMIL specification, version 0.9. http://www.oacea.com/cmil, December 2003.

[27] ECMA International. ECMAScript language specification, 3rd edition. http://www.ecma-international.org/publications/standards/Ecma-262.htm, December 1999.

[28] L. Egri. *The Art of dramatic writing*. Simaon and Schuster, New York, 1946.

[29] L. M. Encarnação, A. Stork, D. Schmalstieg, and R. Barton. The virtual table - a future CAD workspace. In *Proceedings of Computer Technology Solutions conference*, Michigan, Detroit, USA, September 13-19 1999.

[30] T. Fahmy and I. Barakonyi. AR videoconferencing. *TODO*.

[31] D. C. Fallside. XML schema part 0: Primer – W3C recommendation. http://www.w3.org/TR/xmlschema-0/, May 2000.

[32] M. Fiorentino, R. Amicis, and G. Monno. Spacedesign: A mixed reality workspace for aesthetic industrial design. In *Proceedings of the IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR) 2002*. IEEE Computer Society, 2002.

[33] A. S. Foundation. Apache xerces XML parser, version 2.5.0 – website. http://xml.apache.org/xerces-c/index.html, February 2004.

[34] A. Fuhrmann, G. Hesina, F. Faure, and M. Gervautz. Occlusion in collaborative augmented environments. In *Proceedings of the 5^{th} EUROGRAPHICS Workshop on Virtual Environments*, pages 179–190, Vienna, Austria, 1999. Springer.

[35] A. Fuhrmann, J. Prikryl, R. Tobler, and W. Purgathofer. Interactive content for presentations in virtual reality. In *Proceedings of the ACM Symposium on Virtual Reality Software & Technology*, 2001.

[36] D. Gulbransen et al. *Using XML*. Que Publishing, Indianapolis, Indiana, second edition, 2002.

[37] S. Güvem and S. Feiner. Authoring 3D hypermedia for wearable augmented and virtual reality. In *Proceedings of the 7th International Symposium on Wearable Computers*, pages 118–126, White Plains, NY, October 21-23 2003. IEEE.

[38] M. Haringer and H. T. Regenbrecht. A pragmatic approach to Augmented Reality authoring. In *Proceedings of ISMAR 2002*, Darmstadt, Germany, 2002. IEEE.

[39] G. Hesina, D. Schmalstieg, and W. Purgathofer. Distributed OpenInventor : A practical approach to distributed 3D graphics. In *Proceedings of the ACM VRST'99*, pages 74–81, London, UK, December 1999.

[40] W. Jobst. *Das Heidentor von Carnuntum*. Verlag der Österreichischen Akademie der Wissenschaften, Vienna, 2001.

[41] H. Kato and M. Billinghurst. Marker tracking and HMD calibration for a video-based augmented reality conferenencing system. In *Proceedings of IWAR'99*, pages 85–94, San Francisco, CA, USA, October 21–22 1999. IEEE CS.

[42] H. Kato and M. Billinghurst. ARToolKit website. http://www.hitl.washington.edu/artoolkit/, May 2003.

[43] H. Kato, M. Billinghurst, K. Morinaga, and K. Tachibana. The effect of spatial cues in augmented reality video conferencing. In *Proceedings of the 9th International Conference on Human-Computer Interaction (HCI International 2001)*, New Orleans, LA, August 5-10th 2001.

[44] H. Kaufmann, D. Schmalstieg, and M. Wagner. Construct3D: A virtual reality application for mathematics and geometry education. *Education and Information Technologies*, 5(4):263–276, December 2000.

[45] M. H. Kay. Saxon website. http://saxon.sourceforge.net/, December 2003.

[46] M. Keckeisen, S. L. Stoev, M. Feurer, and W. Straßer. Interactive cloth simulation in virtual environments. In *Proceedings of IEEE Virtual Reality 2003*, 2003.

[47] U. Kretschmer, V. Coors, U. Spierling, D. Grasbon, K. Schneider, I. Rojas, and R. Malaka. Meeting the spririt of history. In *Proceedings of VAST 2001*, Glyfada, Athens, Greece, November 28–30 2001. Eurographics.

[48] F. Ledermann, G. Reitmayr, and D. Schmalstieg. Dynamically shared optical tracking. In *Proceedings of ART'02*, Darmstadt, Germany, September 30 2002.

[49] B. MacIntyre and M. Gandy. Prototyping applications with DART, the designer's augmented reality toolkit. In *Proceedings of STARS 2003*, pages 19–22, Tokyo, Japan, October 7 2003.

[50] F. Mantovani. VR learning: Potential and challenges for the use of 3D environments in education and training. In *Towards CyberPsychology: Mind, Cognitions and Society in the Internet Age*, Amsterdam, NL, 2001. IOS Press.

[51] D. Norman. *The Design of Everyday Things*. Doubleday, New York, NY, 1990.

[52] Object Management Group. Unified modeling language (UML), version 1.5. http://www.omg.org/technology/documents/formal/uml.htm, June 2003.

[53] Object Management Group. XML metadata interchange (XMI), version 2.0. http://www.omg.org/technology/documents/formal/xmi.htm, July 2003.

[54] W. Piekarski and B. Thomas. Tinmith-metro: New outdoor techniques for creating city models with an augmented reality wearable computer. In *Proceedings of the 5th International Symposium on Wearable Computers (ISWC)*, Zurich, Switzerland, October 2001.

[55] I. Poupyrev, M. Billinghurst, S. Weghorst, and T. Ichikawa. The go-go interaction technique: Non-linear mapping for direct manipulation in VR. In *Proceedings of ACM UIST 1996*, pages 79–80, Seattle, WA, USA, 1996. ACM.

[56] D. Raggett, A. L. Hors, and I. Jacobs. Hypertext markup language (HTML), version 4.01 – W3C recommendation. http://www.w3.org/TR/html4/, December 1999.

[57] R. Raskar, G. Welch, M. Cutts, A. Lake, L. Stesin, and H. Fuchs. The office of the future : A unified approach to image-based modeling and spatially immersive displays. *Computer Graphics*, 32(Annual Conference Series):179–188, 1998.

[58] R. Raskar, G. Welch, K.-L. Low, and D. Bandyopadhyay. Shader lamps: Animating real objects with image-based illumination. In *Proceedings of the 12th Eurographics Workshop on Rendering*, London, June 2001. Eurographics.

[59] B. Reitinger, A. Bornik, R. Beichel, G. Werkgartner, and E. Sorantin. Augmented reality based measurement tools for liver surgery planning. *Bildverarbeitung in der Medizin (BVM)*, March 2004.

[60] G. Reitmayr and D. Schmalstieg. Mobile collaborative augmented reality. In *Proceedings of the 2nd ACM/IEEE International Symposium on Augmented Reality (ISAR'01)*, pages 114–123, New York, Oct. 29-30 2001.

[61] G. Reitmayr and D. Schmalstieg. OpenTracker – an open software architecture for reconfigurable tracking based on XML. In *Proceedings of IEEE Virtual Reality 2001*, pages 285–286, Yokohama, Japan, March 13–17 2001.

[62] G. Reitmayr and D. Schmalstieg. Collaborative augmented reality for outdoor navigation and information browsing. In *Proceedings of the Symposium on Location Based Services and TeleCartography*, Vienna, Austria, January 2004.

[63] M. Roussos, A. Johnson, J. Leigh, C. Vasilakis, C. Barnes, and T. Moher. NICE: Combining constructionism, narrative, and collaboration in a virtual learning environment. *Computer Graphics*, 31, 1997.

[64] S. Sauer and G. Engels. Extending UML for modeling of multimedia applications. In *Proceedings of the IEEE Symposium on Visual Languages (VL'99)*, pages 80–87, 1999.

[65] S. Sauer and G. Engels. UML-based behavior specification of interactive multimedia applications. In *Proceedings of the IEEE Symposium on Visual/Multimedia Approaches to Programming and Software Engineering*, Stresa, Italy, September 2001. IEEE.

[66] D. Schmalstieg, A. Fuhrmann, and G. Hesina. Bridging multiple user interface dimensions with augmented reality. In *Proceedings of ISAR 2000*, pages 20–29, Munich, Germany, October 5–6 2000. IEEE and ACM.

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

[68] R. Schönfelder, G. Wolf, M. Reeßing, R. Krüger, and B. Brüderlin. A pragmatic approach to a VR/AR component integration framework for rapid system setup. In *Proceedings of the Workshop "Augmented und Virtual Reality in der Produktentstehung"*, pages 67–79, Paderborn, Germany, June 11–12 2002. Heinz Nixdorf Institut.

[69] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems*, 11(3), July 1993.

[70] R. Splechtna, A. L. Fuhrmann, and R. Wegenkittl. ARAS - augmented reality aided surgery system description. Technical Report TR VRVis 2002 040, VRVis, Vienna, Austria, 2002.

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

[72] R. M. Taylor II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. VRPN: A device-independent, network-transparent VR peripheral system. In *Proceedings of VRST 2001*, pages 55–61, Banff, Alberta, Canada, November 15–17 2001. ACM.

[73] H. Tramberend. Avocado: A distributed virtual reality framework. In *Proceedings of IEEE Virtual Reality 1999*. IEEE, IEEE Press, 1999.

[74] J. Zauner, M. Haller, and A. Brandl. Authoring of a mixed reality assembly instructor for hierarchical structures. In *Proceedings of ISMAR 2003*, pages 237–246, Tokyo, Japan, October 7–10 2003. IEEE.