Design of a Component–Based Augmented Reality Framework

Martin Bauer, Bernd Bruegge, Gudrun Klinker, Asa MacWilliams,
Thomas Reicher, Stefan Riß, Christian Sandor, Martin Wagner
Technische Universität München, Institut für Informatik
(bauerma, bruegge, klinker, macwilli, reicher, risss, sandor, wagnerm)@in.tum.de

Abstract

In this paper, we propose a new approach to building augmented reality (AR) systems using a component-based software framework.

This has advantages for all parties involved with AR systems. A project manager can reuse existing components in new applications; an end user can reconfigure his system by plugging modules together; an application developer can view the system at a high level of abstraction; and a component developer can focus on technical problems.

Our proposed framework consists of reusable distributed services for key subproblems of AR, the middleware to combine them, and an extensible software architecture. We have implemented services for tracking, modeling real and virtual objects, modeling structured navigation or maintenance instructions, and multimodal user interfaces.

As a working proof of our concept, we have built an indoor and outdoor campus navigation system using different modes of tracking and user interaction.

1. Introduction

In this paper, we propose a new, framework-based approach to building augmented reality (AR) systems. We present the design of a flexible and modular software framework that allows components to be reused between applications, and describe a prototype system as a working proof of our development approach.

For AR in task-centered, well-structured activities such as aircraft maintenance, one traditionally chooses a different software architecture than one for AR in a ubiquitous computing environment. Indeed, the architectures of current AR systems [2, 3] are widely different. Each of these often monolithic and highly specialized systems uses a small demonstration setup to develop technology for a particular task such as tracking, calibration or human-computer interaction. Even though these tasks are essentially the same in other systems as well, reusing the technology in different architectures is quite difficult.

This notion of similar tasks is quite general. Trackers, for example, provide position and/or orientation information for a set of moving points with a certain speed and accuracy. The user's (virtual or real) environment can be represented in geometric and photometric models approximating the position, shape and appearance of real objects.

After identifying these and other common parts of AR applications as *software components*, we propose to build a *software framework* that can be used for a variety of different AR applications. We think that such a component-based framework can serve as a useful foundation for both research-oriented and real-world AR systems.

An AR system built with our framework consists of a set of *services*. Each service is a program that can potentially run on a separate hardware component with its own processor, memory, I/O devices and network connections. Different services connect dynamically to each other by revealing their needs and abilities on the network.

We present and examine our concept from four different points of view. We describe the framework from the standpoint of a project manager, the AR system's user, the AR application developer and the developer of individual technical components, and show how these groups' requirements can be satisfied with our concepts.

2. Related work

The primary goal of our framework is to develop a research platform for AR on wearable computers and in intelligent environments. Besides this research focus, the project aims to bring well-established parts of AR technology into

practical use within a short time frame. This idea was proposed last year by Dave Mizell [17].

Although concepts from the field of software engineering have been available, they are not yet widely used in the AR community. The *Studierstube* project at Vienna University of Technology is one of the research outfits that have been looking at software architectures more closely [24]. However, the architecture can so far be seen only partially as a framework. Reusability is mainly for testing new user interface paradigms [23] and configuration of the tracking subsystem [19]. In contrast, our approach modularizes entire AR systems as a set of services which connect dynamically at runtime.

Columbia University's Computer Graphics and User Interface Lab has also been involved in assembling different types of AR applications [7] from a common basis (*'Coterie'* [14, 15]). This approach is mainly based on providing a common and easy-to-use distributed graphics library rather than a framework of reusable AR components that can be configured on the fly.

Currently, a lot of the research effort in AR is going into developing and combining tracking technology. As such, it is not surprising that the most popular piece of reusable code, the *ARToolKit* [10], is aimed at tracking. This, however, is not a framework, but a software library. The same statement also applies to other graphics-related distributed research efforts, such as architectures of virtual reality caves or other virtual reality software libraries.

The philosophy and design of our framework is based on the concept of "AR-ready" intelligent buildings proposed by Klinker et al. [12].

3. Views of a component-based framework

This section examines the requirements for AR systems from several perspectives and proposes component-based solutions. The idea of different views of a system has been used, for example, for the description of distributed systems in the ISO standard ODP *(Open Distributed Processing)* [9].

3.1. The project manager's view: reuseable components

The manager of an AR project wants to get the job done. This means keeping down development cost and delivering on time. Let us now examine how a component-based framework can help the project manager reach these goals.

The components of the framework can be seen as black boxes that perform their specific tasks, interacting with the other components using well-defined interfaces. Once these interfaces have been specified, development can be distributed in time and space, and only after all components have been completed, interaction between the various development teams becomes necessary.

Components that are frequently needed in AR have to be kept general enough to allow easy incorporation into specific applications. These multi-purpose components, described in Section 4, form the core of our framework. They can be configured, for example, using XML files. Thus, the project manager can use different specialists for different tasks, such as cognitive psychologists and computer scientists for specifying and implementing user interfaces.

To be able to develop effective components, we chose a two-track approach for the design of the framework and of applications [20]. Each new application uses components of the framework. When the new system is completed, a thorough analysis ensures that lessons learned from the application are integrated into the framework. This could mean generalizing a special application component to a multi-purpose framework component or extending an existing framework component's functionality. The framework will thus grow over time, leading to more powerful applications.

Another advantage of the framework is beyond the usual project-centered view. With a powerful framework that can be configured easily, new ideas in AR can be tested with very little effort. This kind of "rapid prototyping" is eminently useful for finding new applications for AR.

3.2. The user's view: pluggable modules

AR systems can be roughly separated into two classes: systems with high-precision tracking and high-end graphics in a controlled environment, and mobile, autonomous systems with limited processing power in a changing environment.

A controlled environment might be an operating room in a hospital, a television studio, or a research laboratory for the verification of crash test simulations [5]. High-end systems need a lot of computation power for tracking and rendering and often use a compute server (as in [22]). These systems have a lot in common with virtual reality systems. If wearable computers are used, there is no application logic on them—they are only used as multimedia terminals.

For the second class, the mobile, autonomous systems with the focus on the right information at the right place and the right time [6, 11], the wearable computers have to provide the entire functionality (as in [8]). The computation power of these systems is limited and only few percent of that of the high-end systems. Some systems cooperate with compute servers over a network for rendering and tracking, but most of them are standalone.

Let us focus first on these autonomous systems. From the user's perspective, a wearable AR system consists of a set of functional units: tracking the user's position and direc-

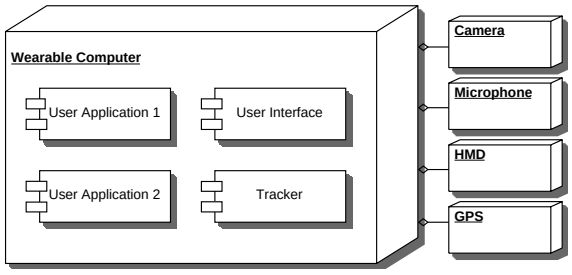

Figure 1. Hardware is separated from software

tion of view, rendering virtual objects, and the application itself. Usually there are additional units for speech recognition and synthesis, gesture recognition, and more. These units consist of hardware devices connected to the wearable computer and the associated software. Figure 1 illustrates this. For optical tracking, one or more video cameras and the tracking software are needed. The video cameras are connected to an I/O port of the wearable computer, and the camera driver and tracker software are installed on the wearable. The installation of the tracking software and the camera driver is too complicated for many end users, and requires a system administrator or system integrator. This approach is quite inflexible from the end users' perspective, since they cannot change the system configuration themselves.

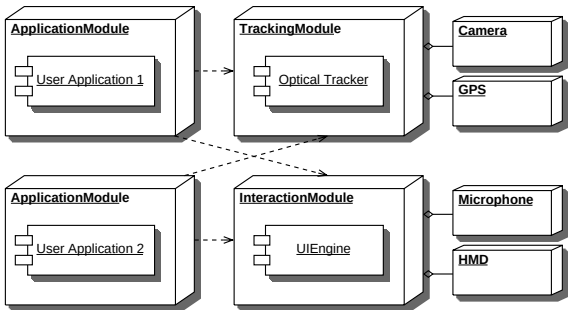

Figure 2. Modules combining software and hardware, communicating over the network

We believe that this inflexibility for the user is a consequence of the separation of functionality and hardware. For the user, the camera and the tracking software are seen as a unit and belong together. Therefore, we propose to bundle them into a tracking *module*, a hardware/software unit separated from the other parts of the system. The cooperation between the other parts and the module uses a communication network, as shown in Figure 2.

With this approach, the user has a set of tools that can be combined simply by plugging hardware modules together. We think that it is easier to understand how to use things if you can touch them. Therefore, the complete functionality of each tool should be encapsulated in a hardware box that can be be connected when needed.

We can apply the same concept of modularization to the class of high-end systems, as well: external trackers and computation servers are simply larger, non-wearable modules. This enables a distributed tracking concept as described in [12] and allows us to bridge the gap between high-end and mobile AR systems.

3.3. The application developer's view: layered services

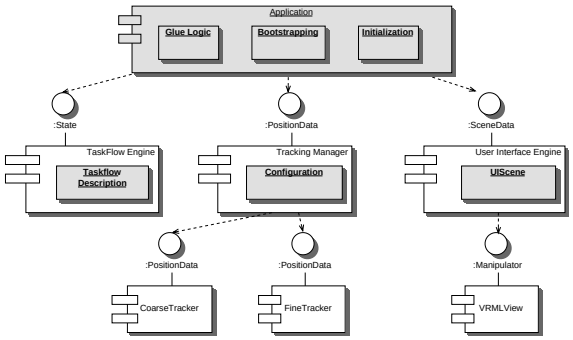

Figure 3. The application developer's view of the system (application shown in gray)

The application developer is responsible for the design and implementation of an AR system. Just as for the project manager, the framework helps him minimize development time and effort. However, the application developer also has to deal with implementation aspects.

For the application developer, the AR system has a three-layered architecture (Figure 3). The base layer consists of low-level services such as trackers, display devices or databases. These are completely encapsulated from the application by the second layer, the high-level framework services. These services provide general functionality needed by most AR systems. Examples are a *user interface engine* that allows the creation and control of user interfaces on different hardware devices, a *taskflow engine* that models task-centered activities as finite state machines and a *tracking manager* that combines tracking information from a variety of trackers. The main services that are currently provided by our framework are described in Section 4.2. The application itself is the third layer of the system.

Creating a new AR system involves selecting appropriate framework components (white in Figure 3) and writing the application (shaded gray).

The first step is to choose the high-level services from the framework that can best support the application's logic. For a maintenance system, for instance, we would use the taskflow engine to model the sequence of repair steps.

Second, we choose appropriate low-level services to support the higher ones. We could, for example, use two trackers: a GPS tracker for outdoor use and a high-precision indoor optical tracker.

Third, we can model the application logic using the abstractions offered by the high-level services, such as taskflows or user interface descriptions. The application developer does not have to worry about specific implementation details of, say, a taskflow representation, but can concentrate on the taskflow itself.

In real applications, there is always some functionality that cannot be fulfilled by the given framework components. To add this extra functionality, as a fourth step, the application developer writes new framework-compatible services, which include "glue logic" to initialize and coordinate the other components of the system.

As a fifth and last step, we can deploy and configure the framework services and the application on the target hardware. Many services are configured automatically using the middleware described in the next section. Others can be configured using XML files.

With this approach, application developers can view the entire system at a high level of abstraction and concentrate on modeling application logic. They do not need to worry about the services' implementation, or even about the fact that they are using a distributed system.

3.4. The module developer's view: needs and abilities

A module in the framework is a combination of hardware and software which provides a certain functionality to the user or to other modules in the system. Any necessary external devices like a camera or a GPS receiver are connected to the module. The module's software consists of *services*, which provide its core functionality; the drivers for external devices; and the *service manager*, a middleware component providing the communication with other modules in the system (Figure 4).

The service manager is identical on every module, even across hardware platform boundaries. Thus, the module developer has to concentrate only on problem-specific issues when writing the modules' services.

The service manager automatically connects matching services together, e.g. a tracker and a VRML display service. For this, services have several *needs* and several *abilities*. The abilities can be used by other services if and only if all needs of the service can be fulfilled by other services. Each need and ability has a specific *type*, and abilities can be fulfilled only by needs of the same type (see Figure 5).

At startup time of the module, each service describes its needs and abilities to the local service manager. The

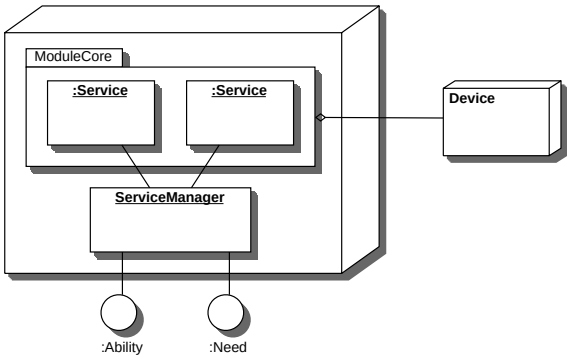

Figure 4. Software view of a module

different local service managers communicate over the network and distribute the information about the services with their needs and abilities. The service managers attempt to match the services' abilities and needs, taking quality-of-service information into account. Thus, tradeoffs in AR systems [13] can be adjusted dynamically.

When all needs of a service can be fulfilled by abilities of other services, dynamic connections are established between the corresponding needs and abilities. To connect the services of two different modules, no user interaction besides plugging the hardware together is necessary.

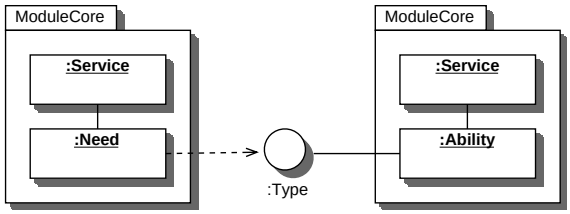

Figure 5. Connection between needs and abilities

Let us consider, for example, developing an optical tracking module. We need to implement the actual tracking and camera access code as a service. Basically, this constitutes 95% of the work. After this has been finished, we may end up with a continuing series of six-dimensional positions that are needed by the display module to render the images correctly. All we need to do now is to describe the service's needs and abilities to the module's service manager. This could be done using a configuration file as in Figure 6.

Once the needs can be fulfilled, the service manager arranges for communication between the services. A connection for communication between two services is set up by two *connectors*, which are created by the two modules' service managers. These connectors configure the necessary communication resources such as event channels or shared memory blocks. The services can access the communication resources through the connectors and use them to com-

```
<module>
   <service name="OpticalTracker">
      <need name="world"
           type="WorldModel"
           minInstances="1"  maxInstances="1">
        <connector protocol="CorbaObjExport"
                   type="CorbaObjImporter"/>
      </need>
      <ability name="position"
              type="PositionData"
              attributes="accuracy=10,lag=30,
                          trackedThing=/User/Head">
        <connector protocol="NotifyStructuredPushSupply"
                   type="NotifyStructuredPushSupplier"/>
      </ability>
      <ability name="image"
              type="VideoData"
              attributes="resX=320,resY=240,fps=30">
        <connector protocol="SharedMemory"
                   type="SharedMemoryWriter"/>
        <connector protocol="udp"
                   type="udpSender"/>
      </ability>
   </service>
</module>
```

Figure 6. Example description of the optical tracker in XML format

municate with one another. Once the connection is set up, the service manager is not involved in the actual communication, although it can break connections or dynamically reconnect to other services.

Since the services specify their needs by type and by their preferred communication protocol, the service managers collectively act as a publish-and-subscribe mechanism for events or object references. Services wishing to receive position data from other services will receive it, using an appropriate communication protocol such as CORBA events or shared memory.

The advantage of this approach is that the communication partners and communication mechanisms are selected before the services actually start communicating. There is no communication overhead at application runtime.

4. The DWARF framework

Our concepts of a component-based framework for augmented reality systems were first tested within the DWARF *(Distributed Wearable Augmented Reality Framework)* project. DWARF [1] is an ongoing research project which started in the beginning of 2000 and has meanwhile resulted in the first prototype system.

4.1. Framework architecture

The framework we propose has three basic aspects: services, middleware and architecture. First, DWARF consists of software services such as trackers, running on hardware modules. Each service provides certain abilities to the user or to other services. Second, DWARF contains the middleware necessary to match these services dynamically and set up the communication between them, so that the system configuration can change at runtime. Third, the conceptual architecture of DWARF describes the basic structure of AR systems that can be built with it. This ensures that service developers agree on the roles of their own services within the system and on interfaces between them. The architecture is also easy enough for end users to understand, so that they can reconfigure their wearable system by simply plugging together the appropriate hardware modules.

Functionally, the services within the framework can be divided into four areas: modeling the world and things in it; accessing information; interacting with the user; and accessing external (non-DWARF) services. The services we have currently developed in these areas, together with an example application, are shown in Figure 7.

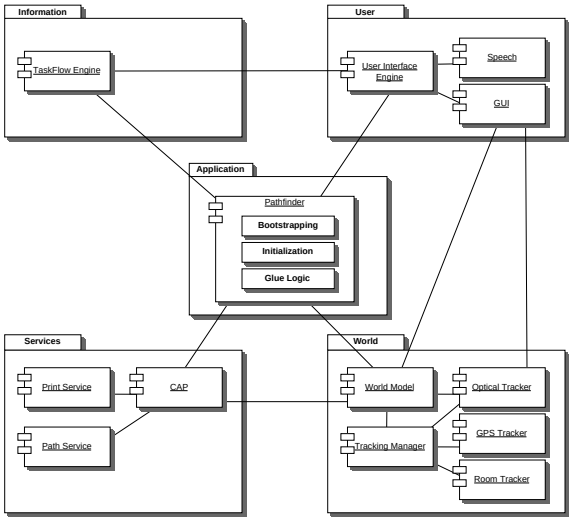

Figure 7. Services within the framework

The application is shielded from the low-level services, such as for user interface or tracking hardware, and accesses these at a higher level of abstraction using the various DWARF services. It includes a special service which provides bootstrapping functionality and “glue logic”. This provides the other services with models of the world and of tasks the user wishes to perform.

Connections for communication among the framework and application services are set up automatically by the middleware, and many of these bypass the application entirely. Thus, an optical tracker can provide position data to the head-mounted display, since it has an ability of type ‘PositionData’ which matches a need of the display service.

4.2. Components

In this section, we give an overview of the most essential DWARF services for building AR systems: tracking and modeling the user's environment, representing structured information, access to legacy services, and platform-independent representation of multi-modal user interfaces.

4.2.1. Tracking subsystem

The DWARF tracking subsystem consists of software services that can be combined dynamically. Its hierarchical architecture is based on low-level trackers which provide information of an object's position and orientation and a *tracking manager* which combines their output. Trackers can either use a single hardware device like GPS or a gyroscope, or already be hybrid trackers combining two or more different kinds of tracker hardware. The tracking devices provide position data in up to six dimensions, together with several quality-of-service parameters.

Based on this information, the tracking manager combines and filters the outputs of the trackers and attempts to improve the results using prediction algorithms. This hierarchy can be extended to more layers; the tracking manager also makes dynamic handover between different trackers possible.

The DWARF tracking architecture has several advantages compared to a monolithic approach. First, adding a new tracker service to the framework is very easy. All that the new service has to do is to register with the service manager. This receives a service description of the tracker that can be stored in an external database, containing the quality-of-service parameters as properties. The tracker's ability to send out position data then becomes available to the rest of the system. Another advantage is the ability to dynamically add or remove tracking devices. With our framework architecture, the user of an AR system can use various trackers depending on his current location and seamlessly roam between them. As a third advantage, the logical separation of the tracking data's creation and combination allows developers to test tracking or prediction algorithms easily.

Currently, we have implemented a service that uses position data from a GPS device and orientation data from an electronic compass. In addition, a high-precision optical tracker based on the ARToolKit [10] was developed. The current implementation of the tracking manager is able to combine various tracking sources and allows the addition of various prediction algorithms at compile time.

4.2.2. World model

Tracking data is the dynamic information we need about the system's user and his environment. The static information about real objects such as rooms and virtual objects such as superimposed arrows is stored in a small specialized database called the *world model*.

Every real and every virtual object has an associated position and orientation, its *pose*. The world model provides means to compute the relative pose between any two objects. This is realized using a hierarchy of objects. All virtual and real objects are stored in a tree-shaped data structure, such that every object has a parent object associated with it. The child object's pose is given relatively to its parent and stored in 4×4 homogeneous matrices. Thus, all pose calculations can be done by simple matrix operations.

The world model is designed to deal with many different types of rapidly changing data. To allow a wide range of possible applications, it is possible to store arbitrary data—such as VRML descriptions—in named attributes associated with each object. To handle dynamically changing content, after every change, all interested services are notified by events. The world model subscribes to events indicating the change of an object's position or orientation, so that the pose of each object in the world model is always kept up to date by tracking data.

To facilitate the modeling of objects, we added the possibility to parse XML-based textual descriptions. This adds the ability to dynamically load information about the environment, for example, when the user enters a new building. This is a necessary feature for large-scale AR applications in intelligent environments.

4.2.3. Taskflow engine

The application domains for many current AR systems include navigation and maintenance aspects, which can be represented in the DWARF *taskflow engine*.

We define a *taskflow* to be the representation of a sequence of actions that have to be executed to complete a user's task (Figure 8). It may include branches based upon user input or other external events such as tracking or sensor information.

The taskflow engine parses a *taskflow description* and executes it. A taskflow description language (TDL) based on XML allows the AR application developer to easily specify how the framework should react and how the sequence of activities is influenced by the user's decisions or events from other services.

The TDL describes a finite state machines of activities connected by transitions. An activity's description contains a specification of what should happen upon entering it, such as displaying a supporting document to the user, firing a transition or sending an event to other framework components. It also contains a set of event handlers that define the reactions of the taskflow engine to user input or external events.

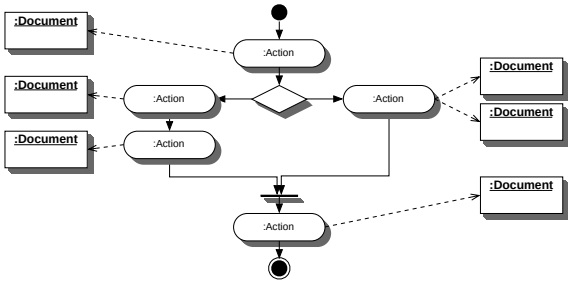

Figure 8. A simple taskflow

Additionally, the TDL offers a simple variable mechanism, conditional transitions and a mechanism for the modularization of taskflows. The TDL allows the development process to be divided into independent parts: creating the supporting documents and creating the taskflow.

First, information that supports the user during his task is prepared for displaying on the head-mounted display, using multimedia authoring tools. These documents may contain simple graphical or textual information but may also include virtual objects to be superimposed on and registered with the real world. The second step is to specify the flow of actions and assign the prepared supporting documents to the actions in the taskflow. The taskflow's description may be created using a text editor or a graphical user interface. Since the TDL is based on XML, it is possible for a legacy system to create the description from workflow, product management or other information.

This separation of content and application logic allows rapid development of application prototypes. Since a taskflow description may be loaded dynamically, it is possible to adapt the the system to new environments or reconfigure the taskflow at runtime.

4.2.4. Context-aware service access

Besides predefined taskflows, the user of a DWARF system should be able to spontaneously use external legacy services, such as connecting to a printer in an unknown environment. This is handled by the *context-aware packet* or CAP service. Here, the user defines a service he would like to use, such as "print out this document on the way to the meeting room", and the CAP service takes care of it.

The major problem for this task is the user's current printer configuration. Even technically simple tasks such as printing can lead to problems in unknown computing environments, since a lot of contextual information such as the preferred paper size has to be considered.

The basic idea of the CAP service is to encapsulate such information in packets that are further processed by software devices that route them in a suitable way. For the printing example, all of the user's configuration data, e.g. paper size, preferred color model etc., is stored in such as packet. The CAP service gathers all necessary information for an optimal fulfillment of the given task of printing from the other DWARF subsystems and executes the print job. Further details on the CAP service can be found in [16].

4.2.5. User interface engine

The services described up to now do not have any means of direct user interaction. This is provided at a high level of abstraction by the *user interface engine*. The design of this subsystem had two main requirements: platform independence of user interfaces, and multi-modal user interaction.

For wearable systems, the user interface consists of numerous I/O devices such as head-mounted displays, palmtops, and speech recognition systems. When the user can trigger the same action by different modes, i.e. voice or gestures, the user interface can be termed *multi-modal* [18].

We used UIML [4], the *User Interface Markup Language*, as a starting point. UIML can be used to develop generic user interface descriptions that can be transformed to markup languages for display in a browser (*views*) at runtime. With this approach it is possible to achieve platform independence for views. UIML, however, lacks support for multimodal user interfaces. Thus, we extended UIML and developed CUIML, for *Cooperative* UIML [21].

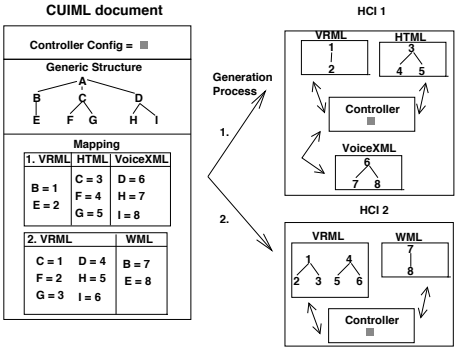

Figure 9. Creation of different multimodal user interfaces from a CUIML document

Figure 9 provides an overview of how human-computer interfaces are generated from a CUIML document. A multimodal user interface consists of several views. In CUIML it is possible to define a *controller* which keeps track of the state of the whole human-computer interface and determines which views have to be changed on user input or external events. For our prototype we reused a version of the taskflow engine as a controller. To alter the views, *manipulators* are generated as well. They separate the low-level access to a view from the specification of what to change on an event from the controller.

The first prototype of the user interface engine transformed generic CUIML descriptions to HTML and to VRML. We used a web browser and a VRML plugin to display the views. The VRML scene is updated by manipulators whenever the user changes her position.

The user interface engine has many advantages: Like UIML, it allows delegation of development tasks and rapid prototyping. Additionally, the user interface is highly flexible—when the user changes an I/O device at runtime, all that has to be done is to generate an appropiate view.

5. Developing applications with DWARF

As a working proof of our framework concepts and especially of DWARF's use for rapid prototyping, we developed a first prototype system. This is essentially a campus navigation system allowing the wireless use of services such as printers. A short video clip with a demonstration of our scenario can be downloaded from the DWARF web page [1].

5.1. A navigation scenario

We chose an AR campus navigation scenario similar to existing systems [8] but with an additional focus on seamless handover between tracking methods and on the use of location-based services.

Fred is invited to a meeting with some software engineering students at the TU München. He is equipped with a wearable computer and a head-mounted display. Fred has a PostScript handout on one of his laptops, and has already registered the handout to be printed as soon as he reaches the TUM main building.

Fred leaves the subway station. As he walks past an information terminal, his wearable system downloads personalized navigation instructions to the meeting room.

On Fred's head-mounted display, a three-dimensional map of the area enriched with navigation information appears that guides him to the TUM, where the print job for his handouts is sent off by wireless LAN.

Inside the building, he is guided by a schematic two-dimensional map to the hallway outside the meeting room.

Two printers are located there, and a three-dimensional arrow points to the printer his print job was printed on. He picks up the handouts and is now ready for the meeting.

5.2. Mapping onto the framework services

The scenario was, of course, chosen so that it could be implemented using the available DWARF components. Here, we show how the demonstration system took advantage of the high-level and low-level framework services.

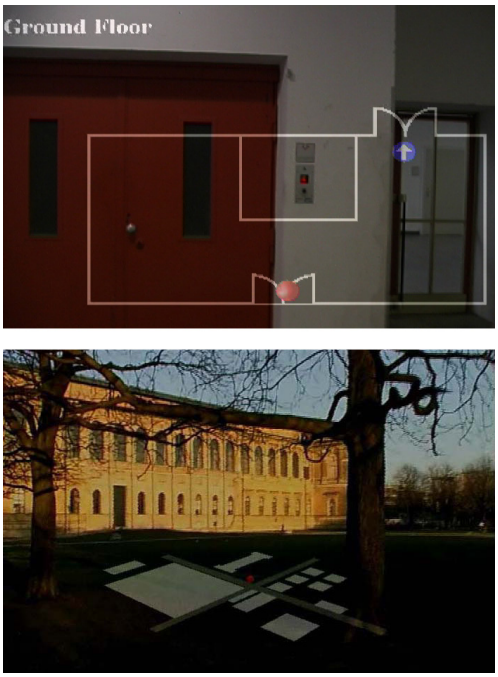

Figure 10. Indoor and outdoor navigation

High-level services The user navigates along a path determined for him by a route-finding service (which was outside the scope of the system). This path can easily be modeled using the taskflow engine, since it consists of a fairly linear sequence of landmarks with a few alternate routes.

The structured map of the TUM campus and the rooms in it fit well into the world model.

The user interface was specified abstractly with the user interface engine, which received indoor and outdoor navigation scenes to display when certain landmarks had been reached in the navigation taskflow.

Finding a nearby printer that matches the user's preferences was managed by the context-aware packet service.

Low-level services For indoor and outdoor navigation, we needed different types of tracking. Outdoors, we used GPS and an electronic compass. Indoors, we used an optical tracker when accurate three-dimensional registration was required (i.e. highlighting the printer), and a simulated ID-tag tracking system identifying doorways when it was sufficient to know which room the user was in. Note that the higher DWARF services did not have to worry about which tracker to use where, as these were dynamically chosen by the middleware based on their availability.

The low-level user interface services included an HTML and a VRML display for two- and three-dimensional navigation, and a speech recognition system for user input.

Downloading navigation instructions from the informa-

Figure 11. DWARF demonstrator

tion terminal and sending the print job used wireless network technologies such as bluetooth and wireless LAN.

5.3. Modeling the application

Most of the application could be modeled using the high-level services as described above. Still, some of the application's logic had to be programmed separately.

One key responsibility of the application in DWARF is bootstrapping. Here, the application had to ensure that the taskflow, world model and user interface descriptions for navigation were correctly sent to the high-level DWARF services, when the user confirmed the download via a scene in the user interface engine.

Once the system was started, the application provided glue logic. This involved, for example, translating the position data from the tracking subsystem into more high-level events that were significant for the taskflow, such as "entering room 3175". To accomplish this, the application used the representation of the rooms in the world model. Additionally, the application had to trigger the sending of the print job when the user had selected this using the user interface engine.

5.4. Deployment

For our prototype, we used standard PC laptops mounted on a fixed frame backpack as shown in Figure 11. In the future, we hope to use one of several powerful wearable hardware platforms that are currently under development.

The first version of DWARF was developed on a variety of platforms, including Linux for Intel and for PowerPC, Windows 2000, and Macintosh. The demonstration system was deployed on two laptops running Windows 98 and Windows NT, connected with standard ethernet cables. Additional peripheral devices included a FireWire camera for optical tracking, a GPS receiver and a head-mounted display. All devices are battery-powered, and the time of operation is more than two hours.

5.5. Results

The results of the demonstration were very encouraging. First of all, the implementation of the scenario took only three weeks' time, which is rapid prototyping at its best. The concept of using loosely coupled services for AR proved quite workable in practice.

Moreover, the performance of the distributed wearable system was sufficient for AR, even for the indoor navigation using optical tracking. The lag in transmitting position data between tracking and display services, which were on different computers and communicated using a connection that had been set up dynamically by the middleware, was quite tolerable and remained below 10 ms.

6. Conclusion

We believe that the technology to solve individual problems within augmented reality is becoming mature enough to consider the more general problem of building whole classes of AR applications. In our opinion, this can best be solved by a component-based framework.

6.1. Progress to date

We have designed and implemented the first version of a flexible framework with modular software services that address the key areas of functionality in general AR systems. Such a component-based framework has advantages for all the people involved in AR systems: the project manager, application developer, component developer, and end user.

As a proof of concept, we have built a demonstration system for an indoor and outdoor navigation scenario. This proved quite workable, allowing sufficient performance despite the architectural flexibility and rapid application development.

6.2. Future work

With the first version of DWARF, we have a platform for research and development which is extensible in four directions: technology for individual components, middleware, new applications, and new types of services.

First, each framework component is a target for future research and development—for example, more advanced tracking methods and better display hardware. New technologies and algorithms, e.g. for tracking, can easily be tested in existing systems, since we use standardized interfaces and a middleware system to shield the services from the complexity of distributed programming.

Second, the DWARF middleware can be extended in order to support richer kinds of quality-of-service informa-

tion, additional communication protocols, and better error handling in the case of network failures.

Third, new AR applications can quickly be prototyped using available components, taking advantage of high-level application descriptions such as taskflows, world models and multimodal user interface specifications.

Fourth, when developing new applications, we will be able to identify new types of useful services and, using the middleware, migrate them into the framework. For example, we could extend the abstract models of applications in the taskflow engine and the world model, and develop a powerful model of the user (or multiple users), including preferences, history and security information.

We think the time has come for AR frameworks. Therefore, an important goal of this paper is to encourage discussion in the AR community on how these frameworks should be designed, what components are needed in general, and what interfaces are useful in particular. We hope that in the not-too-far-off future, we will be able to combine new technology developed by other research teams and by us into flexible, yet powerful wearable AR systems.

Acknowledgements

The authors would like to thank Christoph Vilsmeier, Florian Michahelles, and Bernhard Zaun for their work with us on the concepts, design and implementation of the first version of DWARF. Additionally, we would like to thank Siemens AG for the requirements of the first application to be built with DWARF, and for their support in the related *AiRGuide* project.

References

[1] DWARF Project Homepage. Technische Universität München, `http://www.augmentedreality.de`.

[2] *Proceedings of the IEEE International Workshop on Augmented Reality – IWAR 1999*, San Francisco, 1999.

[3] *Proceedings of the IEEE and ACM International Symposium on Augmented Reality – ISAR 2000*, Munich, 2000.

[4] M. Abrams, C. Phanouriou, A. Batongbacal, S. Williams, and J. Shuster. UIML: An Appliance Independent XML User Interface Language, 1999. `http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html`.

[5] T. Alt and S. Nölle. Weite Welt — Augmented Reality in der Automobilindustrie. *iX Magazin für Professionelle Informationstechnik*, pages 142–145, Jun. 2001.

[6] Internet Presentation of the ARVIKA Project, 2001. `http://www.arvika.de`.

[7] S. Feiner, B. MacIntyre, and T. Höllerer. Wearing It Out: First Steps Toward Mobile Augmented Reality Systems. In *First International Symposium on Mixed Reality (ISMR'99)*, 1999.

[8] S. Feiner, B. MacIntyre, T. Höllerer, and T. Webster. A Touring Machine: Prototyping 3D Mobile Augmened Reality Systems for Exploring the Urban Environment. In *Proceedings of ISWC 1997*, Boston (MA), 1997.

[9] ISO/IEC. Basic Reference Model of Open Distributed Processing - Part 1: Overview and Guide to Use, 1994.

[10] H. Kato, M. Billinghurst, R. Blanding, and R. May. *ARToolKit PC version 2.11*, 1999.

[11] G. Klinker, O. Creighton, A. Dutoit, R. Kobylinski, C. Vilsmeier, and B. Brügge. Augmented Maintenance of Power Plants. In *Proceedings of ISAR 2001*, New York, 2001.

[12] G. Klinker, T. Reicher, and B. Brügge. Distributed User Tracking Concepts for Augmented Reality Applications. In *Proceedings of ISAR 2000*, Munich, 2000.

[13] G. Klinker, D. Stricker, and D. Reiners. Augmented Reality: A Balancing Act Between High Quality and Real-Time Constraints. In *Mixed Reality - Merging Real and Virtual Worlds*, pages 325–346. Springer-Verlag, 1999.

[14] B. MacIntyre. *Exploratory Programming of Distributed Augmented Environments*. PhD thesis, Columbia University, 1999.

[15] B. MacIntyre and S. Feiner. A Distributed 3D Graphics Library. In *Computer Graphics Proceedings of SIGGRAPH '98*, pages 361–370, Orlando, Florida, 1998.

[16] F. Michahelles. Designing an Architecture for Context-Aware Service Selection and Execution. Master's thesis, Ludwig–Maximilians–Universität München, 2001.

[17] D. Mizell. Augmented Reality Applications in Aerospace. In *Proceedings of ISAR 2000*, Munich, 2000.

[18] L. Nigay and J. Coutaz. A design space for multimodal systems - concurrent processing and data fusion. In *INTERCHI '93 - Conference on Human Factors in Computing Systems*, Amsterdam, 1993. Addison Wesley.

[19] G. Reithmayr and D. Schmalstieg. OpenTracker – An Open Software Architecture for Reconfigurable Tracking based on XML. Technical report, TU Wien, 2000.

[20] D. Roberts and R. Johnson. Evolving Frameworks. A Pattern Language for Developing Object-Oriented Frameworks. University of Illinois, `http://st-www.cs.uiuc.edu/~droberts/evolve.html`, 2001.

[21] C. Sandor and T. Reicher. CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces. In *Proceedings of the European UIML conference*, 2001.

[22] F. Sauer, F. Wenzel, S. Vogt, Y. Tao, Y. Genc, and A. Bani-Hashemi. Augmented Workspace: Designing an AR Testbed. In *Proceedings of ISAR 2000*, Munich, 2000.

[23] D. Schmalstieg, A. Fuhrmann, and G. Hesina. Bridging Multiple User Interface Dimensions with Augmented Reality. In *Proceedings of ISAR 2000*, Munich, 2000.

[24] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavári, L. Miguel Encarnação, M. Gervautz, and W. Purgathofer. The Studierstube Augmented Reality Project. Technical report, TU Wien, 2000.