Object-Oriented Middleware Infrastructure for Distributed Augmented Reality

Eiji TOKUNAGA Andrej van der Zee
Makoto KURAHASHI Masahiro NEMOTO
Tatsuo NAKAJIMA

Department of Information and Computer Science, Waseda University
3-4-1 Okubo Shinjuku Tokyo 169-8555, JAPAN
{eitoku,andrej,mik,nemoto,tatsuo}@dcl.info.waseda.ac.jp

Abstract

The paper describes design and implementation of software infrastructure for building augmented reality applications for ubiquitous computing environments. Augmented reality is one of the most important techniques to achieve the vision of ubiquitous computing. Traditional toolkits for augmented reality provide the high level abstraction that makes it easy to build augmented reality applications. However, the applications programmers need to contemplate distribution and context-awareness that make the development of applications very hard, but they are necessary to build ubiquitous computing environments. Our infrastructure provides the high level abstraction and hides distribution and context-awareness from programmers. Therefore, the cost to develop augmented reality applications will be reduced dramatically by using our middleware infrastructure.

1. Introduction

Our daily life will be dramatically changed due to a variety of objects embedding computers. These objects behave intelligently to extend our bodies and memories[10]. A lot of research projects are working on attacking various problems to realize these computing environments. These computing environments are called *ubiquitous computing*[1, 19, 23, 24]. Also, other researchers have proposed similar concepts called *pervasive computing*[4], *sentient computing*[12], or *things that think*[10]. In *ubiquitous computing environments*, a variety of objects are augmented by containing computers. Since any programs can be executed on the computers, there are infinite possibilities to extend these objects by replacing the programs. Also, these objects have networks to communicate with other objects, thus respective objects will behave more actively by replacing programs at each other according to surrounding situations. For example, our environments may memorize what is going on in the world, or each object tells us where it currently exists.

This paper describes design and implementation of software infrastructure for building augmented reality applications in ubiquitous computing environments. Augmented reality[3] is one of the most important techniques to achieve the vision of ubiquitous computing. Traditional toolkits for augmented reality provide the high level abstraction that makes it easy to build augmented reality applications, but the applications programmers still need to take into account distribution and context-awareness that make the development of applications very hard, where distribution and context-awareness are inherent in ubiquitous computing, and programmers cannot avoid to take into account them. Our software infrastructure provides the high level abstraction. Also it hides distribution and context-awareness from applications programmers. Therefore, the cost to develop augmented reality applications will be reduced dramatically by using our software infrastructure.

The remainder of this paper is structured as follows. In Section 2, we show design issues of our infrastructure. Section 3 presents design and implementation of our middleware for distributed augmented reality. In Section 4, we describe the current status of our system. Section 5 presents several discussions about our current prototype system. Finally, we conclude in Section 6.

2. Design Issues

In this section, we describe the design issues involved for building our middleware. First, we present a brief overview of ubiquitous computing and augmented reality. Then, we show the requirements for building augmented-reality applications in ubiquitous computing.

2.1. Ubiquitous Computing and Augmented Reality

Augmented reality is a technology offering an augmented real-world to the user. More concretely, an augmented-reality application presents a view composed of the real-world and digital information managed by computers. Besides an augmented view of the real-world, it may provide a seamless human-computer interface as well.

Developing augmented-reality applications is not easy. Among other concerns, programmers must implement complex algorithms to detect visual markers. Some toolkits, like the ARToolKit [2], have been developed to equip the programmers with implementations of typical augmented reality problems.

In ubiquitous environments, computers and networks are accessed implicitly rather then directly. Most of the time, users are not even aware that they are connected to a network and accessing multiple computers at the same time. In the end, users want to deal with the real-world rather then with cyber-space. This requires a high level of transparency and makes ubiquitous environments even more complex.

2.2. Requirements

When developing ubiquitous augmented-reality applications, the developer is faced with the complexities inherent to ubiquitous environments. Existing AR toolkits are not designed for such environments and consequently do not address these complexities. We found it is necessary to meet the following three requirements when building augmented reality applications in ubiquitous computing environments.

High-Level Abstraction: Ubiquitous computing environments consist of various types of computers and networks. Networks may contain a mix of resource-constrained and specialized computers. Also, the existing augmented reality toolkits are platform-dependent. Consequently, application programmers must develop different software for each platform. A middleware to provide high-level abstraction to hide such differences from application programmers is nec2 essary[18, 20] in order to reduce the development costs.

Distribution: In ubiquitous computing environments, applications must be distributed over many processors. Since the environment usually consists of various types of computers, some may not be appropriate for heavy processing like video-data analysis. For example, cellular phones and PDAs are usually to weak for heavy processing, but they might want to utilize augmented-reality features. However, an application running on low CPU-resource could be distributed such that heavy processing is performed on strong computers. In ubiquitous computing, we think that such distribution needs to be hidden from the developer in order to keep development time and cost as low as possible.

Context-Awareness: In ubiquitous computing environments, applications must support context-awareness since users need to access computers and networks without knowing. It is required for an application to adapt itself to the users situation dynamically. However, implementing context-awareness in an application directly is very difficult. An application programmer does not want to be concerned with such complexities and we think that it is desirable to embed context-awareness in our framework and hide it from the developer.

3. Middleware supporting Augmented Reality

We have designed the middleware, which involves the design issues described in section 2, for supporting augmented reality in ubiquitous computing. The name of the middleware is TEAR: Toolkit for Easy Augmented Reality. In this section, we describe the design and implementation of TEAR.

3.1. Overview of Architecture

TEAR consists of two layers, as shown in Figure 1. The upper layer is the multimedia framework (see section 3.3) and the lower layer is a communication infrastructure based on CORBA (Common Object Request Broker Architecture). The support of context-awareness is handled by the communication infrastructure.

An augmented reality application using TEAR consists of an application composer and several multimedia components. An application composer is a user-side program that coordinates an entire application. It maintains references to objects contained by multimedia components, and configures them to build distributed context-aware applications. For example, as shown in Figure 1, a multimedia source component (a camera) and a multimedia sink component (a display) are connected. The setup is achieved by the application composer through the interface provided by the continuous media framework.

In TEAR, a proxy object in an application may hold several references to objects that provide identical functionality. In the example, there are two camera components and three display components. A proxy camera object in the application composer holds two object references to camera components, and a proxy display object holds three object references to display components. Which reference is used in an application is decided upon the context policies, specified in the application.

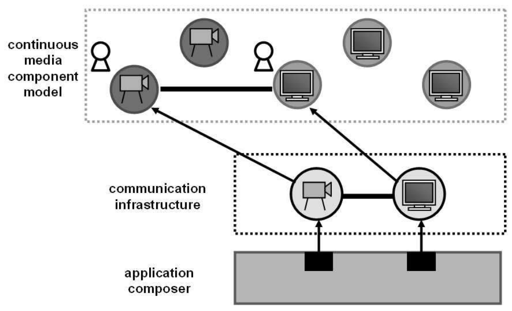
continuous media component model
communication infrastructure
application composer

Figure 1. Overview of TEAR Architecture

TEAR meets the requirements outlined in the previous section in the following way.

High-Level Abstraction: TEAR provides a multimedia framework for constructing augmented reality components in an easy way. Complex programs like detecting visual markers and drawing 3D objects are encapsulated in respective multimedia components. All the components offer AN IDENTICal CORBA interface for standardized inter-component access. In our framework, a complex distributed and context-aware AR application can be developed with the application composer that configures existing multimedia components. We describe details about the multimedia framework in Section 3.3.

Distribution: For composing multimedia components in a distributed environment, we have adopted a CORBA-based communication infrastructure. Each multimedia component is designed as a CORBA object. Since CORBA hides differences among OS platforms and languages, the continuous media components run on any OS platforms, and can be implemented in various languages.

Context-Awareness: In the TEAR framework, the communication infrastructure is designed as a CORBA compatible system that supports context-awareness. The infrastructure supports user mobility by automatically reconfiguring media streams. Also, the infrastructure allows us to select a suitable component to process media streams according to the condition of each computer and the situation of a user by specifying policies. We describe details about the communication infrastructure in Section 3.2.

3.2. CORBA-based Communication Infrastructure

As described in Section 2, context-awareness is one of the most important features for implementing augmented reality applications in ubiquitous computing. Therefore, a middleware supporting augmented reality must support context abstraction which allows us to specify application preferences about context information such as user location. We have designed a context-aware communication infrastructure based on CORBA which provides dynamic adaptation according to the current context.

3.2.1 Dynamic Proxy Object

In our system, application programmers use a *dynamic proxy object* to access target objects, contained by multimedia components described in Section 3.3. The dynamic proxy object contains several object references to actual target objects, *context information*, and an *adaptation policy* for specifying how to adapt the invocation to a target object. A dynamic proxy object is a CORBA object like a multimedia component, and provides the same interface as actual objects. When a method in a dynamic proxy object is invoked, the request is forwarded to the most appropriate object according to the specified adaptation policy as shown in Figure 2.

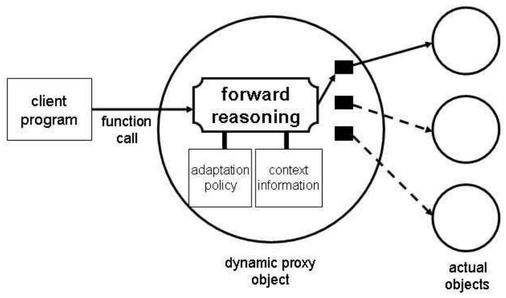

Figure 2. Dynamic Proxy Object

In the current design, an adaptation policy is specified as a set of location and performance policies. Examples of location policies are ”nearest object to me”, ”exact object connected to the camera” or ”any object”. Performance policies might be ”light loaded host” or ”any object”.

3.2.2 Context Trader Service

To create a dynamic proxy object described in the previous section, we we have developed a CORBA service called the *context trader service*. An application program can acquire a reference to the context trader by invoking the `resolve_initial_reference`-method provided by CORBA.

Figure 3 explains how a client program creates and uses a proxy object. (1) By invoking the `resolve` method on the context trader service a developer can acquire a reference to a proxy object. The method requires three parameters;

a type specifying the proxy object, an adaption policy and the scope for selecting the target objects. (2) The context trader service creates a proxy object of the specified type and registers a target object within the specified scope. (3) A reference to the proxy object is returned to the client program. (4) Callback handlers may be registered through the reference. (5) Context changes are reported to the *context manager*. (6) The context manager notifies the proxy object upon context change and (7) the client program is notified by invoking the registered callback handlers.

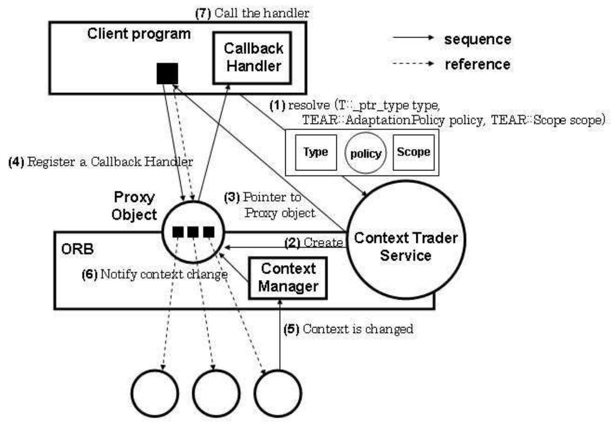

Figure 3. Resolving Dynamic Proxy Object

3.3. Multimedia Framework

The main building blocks in our multimedia framework are software entities that externally and internally stream multimedia data in order to accomplish a certain task. We call them *components*. In the following subsections we describe components in more detail and provide source code to illustrate how a developer can configure a component.

3.3.1 Components

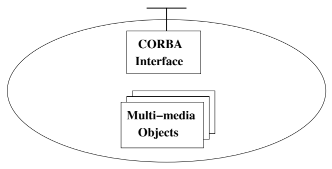

Figure 4. General Component

A continuous media component consists of a CORBA interface and a theoretically unlimited number of *subcomponents* or *objects* as shown in Figure 4. Video or audio data is streamed between objects, possibly contained by different components, running on remote machines. Through the CORBA interface virtual connections can be created in order to control the streaming direction of data items between objects. Components register themselves at the CORBA Naming Service under a user-specified name. Next, we will discuss the CORBA interface subcomponents, thread scheduling and virtual connections.

CORBA Interface

A component can be remotely accessed through one of three CORBA interfaces: `Component`, `Connector` and `Services`.

The `Component` interface is added to the component to provide a single object reference through which references can be obtained to other CORBA interfaces. The benefits of adding such an interface is to give clients access to all functionality through a single reference. Such a reference can be published in the Naming or Trading Service [11]. In addition, the `Component` interface provides functions to query individual objects and the component as a whole. The `Component` interface is identical to all components.

The `Connector` interface provides methods to establish virtual connections between objects, possibly contained by different components, running on remote sites. More specific, the interface provides functions to access and update routing information of individual source objects. The `Connector` interface is identical to all components.

The `Services` interface provides methods for controlling specific objects within a component. Clients may find it useful to query and/or change the state of a multimedia object. For example, a client may want to query a display object for the resolutions it supports. The `Services` interface varies from component to component, depending on the internal objects it contains.

The interfaces are part of the module IFACE and are written in CORBA IDL [11, 15]. Here follows a snapshot of the `Connector` and `Component` interface[1]:

```
interface MConnIface
{
   void
   addRoutingSeq(in ObjectId id,
                 in RoutingSeq seq)
   raises(InvalidObjectId);

   boolean
   removeRoutingSeq(in ObjectId id,
                    in RoutingSeq seq)
   raises(InvalidObjectId);
};

interface MCompIface
{
   MConnIface
   getConnIface();
```

[1]The Services interface is not included since it varies for different component configurations.

```
    MServIface
    getServIface();

    boolean
    isInput(in ObjectId id)
    raises(InvalidObjectId);

    boolean
    isOutput(in ObjectId id)
    raises(InvalidObjectId);
};
```

Subcomponents or Objects

Typically, within a component, several objects run in separate threads and stream data in one direction. For example, a camera object may capture images from a video device, and stream the video data to a display object through a red-blue swapper that swaps the red and blue values of a video frame as shown in Figure 5.

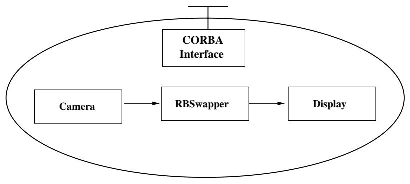

Figure 5. Example Component

In our approach, the central focus is the stream of data from data producers to data consumers through zero or more data manipulators [14]. Data producers typically are interfaces to video or audio capture hardware or media storage hardware. In our framework we call them *sources*. Data manipulators perform operations on the media-data that runs through them. Data manipulators get their data from sources or other data manipulators and stream the modified data to a consumer or another manipulator. In our framework we call them *filters*. Data consumers are objects that eventually process the data. Data consumers typically interface to media storage devices. In our framework we call them *sinks*. In our example from Figure 5, data is streamed from our camera source object, through the red-blue swapper filter object, into the display sink object.

Objects are categorized as *input* and/or *output* objects. For example, a filter object is both an input and an output object, meaning it is capable of respectively receiving and sending data. Clearly, a source object is of type output and a sink object of type input.

More concrete, our framework provides the abstract classes `MSource`, `MFilter` and `MSink`[2] written in C++. Developers extend the classes and override the appropriate hook-methods [9] to implement functionality. Multimedia objects need only to be developed once and can be reused in any component.

[2]The M preceeding the class names indicate that they are part of the framework and stands for multimedia.

Components know two specialized objects for handling inter-component data streaming, namely *rtp-in* and *rtp-out*. An rtp-in object is a source object, consequently of type input, that receives data from remote components over a RTP connections. Semantically this is not strange at all, since from the components point of view, data is produced by means of receiving it from another component. Similarly, rtp-out is a sink object that is responsible for sending data to other components.

3.3.2 Stream Reconfiguration

Supporting context-awareness by multimedia applications requires not only dynamic adaptation of object references, but also dynamic re-direction of continuous media streams. When the current object reference of a dynamic proxy object is changed, continuous media streams must be reconnected dynamically to change the current configuration of continuous media components according to the current context information. To achieve this, a callback handler described in Section 3.2.2 is used. It is registered to a dynamic proxy object by an application, and the handler is invoked when the current context is changed. Next, we discuss how our system reconfigures the connections among continuous media components by using the example described in the previous section.

Suppose a context change is reported to the context manager and a notification is triggered to the proxy object holding a reference to the red-blue swapper. In response, the proxy object might want to change its internal reference to the red-blue swapper in order to adapt to the new context. If so, its registered callback handlers are invoked. Typically, one of the callback handlers is concerned with updating routing information of affected source objects. Such handlers expect a parameter holding a reference to the new target object. In the example, the reference to the red-blue swapper is used to construct a new routing list, and the routing information of the camera source object is updated to reflect the new configuration.

By updating the routing information of source objects virtual connections are added and deleted. Subcomponents that do not appear in routing information of any source object are not presented any data and consequently reside in an idle state. By using virtual connections, no notification messages have to be sent to any filter or sink object to hold them from processing any data. Solely updating the routing information of source objects is sufficient.

3.3.3 Components for Augmented Reality

Among others, TEAR provides augmented reality components for the detection of visual markers in video frames and superimposing 3D objects at a specified location within a frame. Such components are implemented as objects contained by multimedia components as described in subsection 3.3. They use the ARToolkit to implement functionality.

A detection filter object expects a video frame as input and looks for visual markers. Information about visual markers, if any, is added to the original video frame and send as output. Since different types of visual markers will be available, the format of the marker information must be defined in a uniform way. Consequently, filter components detecting different types of visual markers can be used interchangeably .

A super-imposer object expects video frames with marker information as input, superimposes additional graphics at the specified location, and outputs the augmented video frame.

Figure 6 shows how the two components can be used in sequence to enhance a video stream with augmented reality. In this configuration, video frames are captured by an input device and sent to the output device through the detection filter and super-imposer. As a result, visual markers are replaced by digital images.

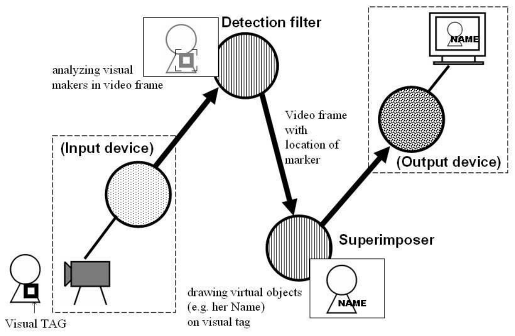
Detection filter
analyzing visual makers in video frame
NAME
Video frame with location of marker
(Input device)
(Output device)
Superimposer
Visual TAG
drawing virtual objects (e.g. her Name) on visual tag
NAME

Figure 6. Components for Augmented Reality

4 Current Status

In our current prototype of TEAR, we use omniORB [15] for our CORBA-based communication infrastructure. OmniORB is open source and considered very efficient. Above that, in our design of continuous media components, the contained objects run in separate threads. Therefor, a fully multi-threaded OMG CORBA compliant ORB is needed. Obviously, omniORB supports all multi-threaded features.

We have currently developed some C++ classes for building continuous media components described in Section 3, and We have implemented several subcomponents for supporting augmented reality features. The implementation uses ARToolkit[2] and TRIP[16]. We have currently implemented two detection filter objects. One is adopted in ARToolkit, and another is based on TRIP. The superimposer object have adopted the OpenGL technology similar to ARToolkit.

The picture shown in Figure 7 shows a demonstration of our prototype system. The laptop computer in the right side is IBM ThinkPadX23 which has Pentium III 866MHz and SDRAM 256MBytes. In the left side, a DV camera is connected to a high-end desktop computer which has Pentium4 1.9GHz and RDRAM 512MBytes. The desktop computer captures visual images, which include visual tags from the DV camera, and transmits superimposed images to the laptop computer(Figure 8). On the desktop computer, a continuous media component using our framework is running. This component consists of our two detection filter objects described above to detect visual tags, the superimposer object for constructing 3D virtual images, the capturing object for capturing video images from the DV camera and the RTP-out object for transmitting video images using the RTP protocol. On the other hand, on the laptop computer, another component which consists of the RTP-in object for receiving video images via the RTP protocol, and the gtk-display object for displaying received video images is running. Most of heavy processing like visual tag detection and superimposing 3D images is executed on the high-end desktop computer. Thus, the component running on the laptop computer requires less resources to run it.

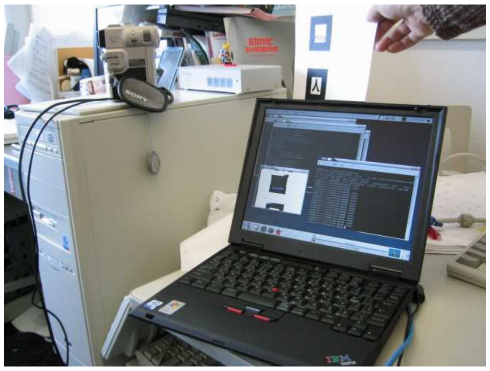

Figure 7. Prototype Demonstration

The TRIP tag detection requires more resources than for the ARToolKit's tag detection. Therefore, it is hard that the laptop detects and superimposes by using the TRIP's tag. This means that the TRIP's tag detection is not suitable for PDAs and cellular phones. However, the TRIP tag is useful for augmented reality in ubiquitous computing environments because it is easy to encode some inforation like the bar code and it allows us to calculate the tag's location

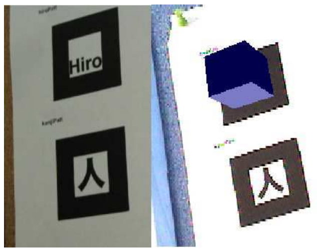

Figure 8. Superimposed Image

and angle accurately. Also, we may want to use more advanced visual tag technologies which require more resource than the current detection filters in the future. Our system enables us to adopt these new technologies very easily by replacing objects by using the component based framework in our approach.

5. Discussions

We believe that our design is very effective. Especially, the automatic reconfiguration of an application to reflect context change seams very promising. However, in our approach, a number of issues still need to be addressed. In this section, we will discuss strengths and weaknesses of our current design.

In our framework, we assume that continuous media components do not have state. Consequently, if multimedia components are switches as a result of context change, restoring state information of new target components is not necessary. However, we found that components controlling devices might hold some state information to configure device parameters. In our approach, the application composer restores such parameters after receiving a change-of-context notification.

Regarding context change, application behavior is handled in the communication infrastructure rather than put the responsibility with the programmer. In our approach, an application programmer needs to specify context policies to reflect application behavior to his or her desire. Most infrastructure software for ubiquitous computing adopt a different approach. For example, the Context Toolkit [8] offers a mechanism to deliver events upon context change, making the development of context-aware applications more difficult. In our approach, we choose to hide as much detail as possible from the programmer in order to reduce the development costs of ubiquitous computing applications. In our future design, we want to adopt QoS parameters in order to provide the application programmer more flexibility in specifying the adaption policy of an application in response to context change.

Specific to our multimedia framework, we designed components as self-describing software entities. Consequently, a client can retrieve a description of all components registered at the Naming or Trading Service at run-time. In this way, a client can query and/or change component configuration dynamically. Also, information about virtual connections between components can be retrieved and updated. Visual tools can be developed to provide easy means to (re)configure a set of available components.

Objects contained by multimedia components can be developed fast and easy. Once an object is developed it can be reused in other components. In addition, components can be composed by utilizing services provided by other components in order to provide more complex services. By such composition, components can be reused as a whole and the development of new components becomes even less time-consuming.

In our framework, a programmer specifies a policy to control the behavior of an application according to current context. Some policies are very difficult to implement. For example, assume a location policy that always chooses a nearest service to user location. The presence of a wall between a user and a server inflicts complications regarding to implementation. Clearly, the nearest server might not be the most suitable one. In the future, we need to investigate a way to specify a policy that does not depend on sensor technologies to monitor context information.

Currently, virtual connections between components need to be updated to reflect a context change. Suppose a target object in a proxy is changed, then the routing information in source objects containing the old object reference need to be updated in a callback handler registered for the proxy object. Note that only the routing information of source objects initialized by the respective application needs to be updated, since other applications might still use the old reference and therefor are not affected by the context change. Though, it might be desirable to put the responsibility of reconfiguring routing information in our middleware and completely hide the burden of context change from the programmer. Consequently, information about virtual connections need to be managed by the communication infrastructure. This approach requires more complex middleware and we might need support of a more complex CORBA implementation like OpenORB2 [6]. The above described issue will likely be studied in the near future.

6. Conclusion

In this paper, we have described our middleware framework to support augmented reality for ubiquitous computing. We have described the design and the implementation of our system, and shown some experiences with our current prototype system. Our experiences show that our system is very useful to develop several augmented reality applications for ubiquitous computing.

References

[1] G.D. Abowd, E.D. Mynatt, “Charting Past, Present, and Future Research in Ubiquitous Computing”, ACM Transaction on Computer-Human Interaction, 2000.

[2] ARToolkit, http://www.hitl.washington.edu/people/grof/SharedSpace/Download/ARToolKitPC.htm.

[3] R.T. Azuma, “A Survey of Augmented Reality”, Presence: Teleoperators and Virtual Environments Vol.6, No.4, 1997.

[4] G.Banavar, J.Beck, E.Gluzberg, J.Munson, J.Sussman, D.Zukowski, “Challenges: An Application Model for Pervasive Computing”, In Proceedings of the Six Annual International Conference on Mobile Computing and Networking, 2000.

[5] Martin Bauer, Bernd Bruegge, et al.: *Design of a Component-Based Augmented Reality Framework*, The Second IEEE and ACM International Symposium on Augmented Reality, 2001.

[6] G.S.Blair, et. al., “The Design and Implementation of Open ORB 2”, IEEE Distributed Systems Online, Vol.2, No.6, 2001.

[7] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, Daniel Villela, “A Survey of Programmable Networks”, ACM SIGCOMM Computer Communications Review, Vol.29, No.2, 1999.

[8] A.K.Dey, G.D.Abowd, D.Salber, “A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications”, Human-Computer Interaction, Vol.16, No.2-4, 2001.

[9] Erich Gamma, Richard Helm, Ralph Johnson, John Flissides: *Design Patterns, Elements of Reusable Object-Orientated Software*, Addison-Wesley Publishing Company (1995), ISBN 0-201-63361-2.

[10] N. Gershenfeld, “When Things Start to Think”, Owl Books, 2000.

[11] Michi Henning, Steve Vinoski: *Advanced CORBA Programming with C++*, Addison-Wesley Publishing Company (1999), ISBN 0-201-37927-9.

[12] Andy Hopper, “Sentient Computing”, In *the Clifford Paterson Lecture*, volume 358, pages 2349-2358, Phil. Trans. R. Soc. Lond., September 1999

[13] R.Koster, A.P. Black, J.Huang, J.Walpole, and C.Pu, “Thread Transparency in Information Flow Middleware”, In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, 2001.

[14] Christopher J. Lindblad, David L. Tennenhouse: *The VuSystem: A Programming System for Compute-Intensive Multimedia*, In Proceedings of ACM International Conference on Multimedia 1994.

[15] S Lo, S Pope, “The Implementation of a High Performance ORB over Multiple Network Transports”, In Proceedings of Middleware 98, 1998.

[16] D.Lopez de Ipina, “Visual Sensing and Middleware Support for Sentient Computing”, PhD thesis, Cambridge University Engineering Department, January 2002

[17] Diego Lopez de Ipina and Sai-Lai Lo, “LocALE: a Location-Aware Lifecycle Environment for Ubiquitous Computing”, In Proceedings of the 15th IEEE International Conference on Information Networking (ICOIN-15), 2001.

[18] T.Nakajima, “System Software for Audio and Visual Networked Home Appliances on Commodity Operating Systems”, In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, 2001.

[19] T.Nakajima, H.Ishikawa, E.Tokunaga, F. Stajano, “Technology Challenges for Building Internet-Scale Ubiquitous Computing”, In Proceedings of the Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems, 2002.

[20] T.Nakajima, “Experiences with Building Middleware for Audio and Visual Netwoked Home Appliances on Commodity Software”, ACM Multimedia 2002.

[21] OMG, “Final Adopted Specification for Fault Tolerant CORBA”, OMG Technical Committee Document ptc/00-04-04, Object Management Group (March 2000).

[22] C.Pinhanez, “The Everywhere Display Projector: A Device to Create Ubiquitous Graphical Interfaces”, In Proceedings of Ubicomp’01, 2001.

[23] K.Raatikainen, H.B.Christensen, T.Nakajima, “Applications Requirements for Middleware for Mobile and Pervasive Systems”, Mobile Computing and Communications Review, Octorber, 2002.

[24] M. Weiser, “The Computer for the 21st Century”, Scientific American, Vol. 265, No.3, 1991.