AN OPERATING SYSTEM FOR AUGMENTED REALITY UBIQUITOUS COMPUTING ENVIRONMENTS

YEW WEIWEN, ANDREW

(B.Eng. (Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF MECHANICAL ENGINEERING
NATIONAL UNIVERSITY OF SINGAPORE

2014

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Yew Weiwen, Andrew
18 November 2014

Acknowledgements

I would like to express my sincerest gratitude to my thesis supervisors, Assoc. Prof Ong Soh Khim and Prof. Andrew Nee Yeh Ching, for granting me the opportunity and support to carry out this research. Their faith and guidance have been invaluable. Every endeavor they have made in making our laboratory a happy, clean and conducive environment for research, as well as their efforts in looking after my welfare, is greatly appreciated.

Sincere thanks also go to my fellow researchers in the Augmented Reality and Assistive Technology Lab past and present for their advice and friendship. I would like to make special mention of Dr. Shen Yan and Dr. Zhang Jie who helped me with a great many matters concerning my academic duties and settling into the laboratory, and of Dr. Fang Hongchao who has been a constant source of companionship, encouragement, and technical help. I would also like to thank the FYP students whom I have mentored who provided valuable assistance with this work.

Finally, I wish to thank my family for taking an active interest in my research work, and sometimes giving me wild ideas to ponder, and my parents for sacrificing so much in order for me to pursue this dream.

Table of Contents

Acknowledgements ……………………………………………... i

Table of Contents ……………………………………………… ii

List of Figures ………………………………………………… vi

List of Tables ………………………………………………… viii

List of Abbreviations ………………………………………… ix

Summary …………………………………………………… xi

Chapter 1. Introduction ……………………………………… 1

1.1 Ubiquitous Computing ………………………… 1

1.2 Augmented Reality ………………………………. 2

1.3 Research Objectives and Scope ………………… 4

1.4 Organization of the Thesis ……………………….. 7

Chapter 2. Literature Survey ………………………………… 8

2.1 Ubiquitous Computing Issues ……………………. 8

2.1.1 Heterogeneity and Spontaneous Interoperation … 8

2.1.2 Invisibility …………………………………… 9

2.1.3 Transparent User Interaction …………………. 10

2.1.4 Context Awareness and Context Management … 13

2.2 Augmented Reality Issues ……………………… 13

2.2.1 Tracking ………………………………………. 13

2.2.2 Display and Interaction Devices …………… 17

2.3 Ubiquitous Augmented Reality Frameworks …… 19

2.3.1 High-level Frameworks ……………………… 19

2.3.2 Component-based Frameworks ……………… 20

2.3.3 Standards-based Frameworks ………………. 24

2.4 Summary ……………………………………….. 26

Chapter 3. Design of the SmARtWorld Framework ……….. 28

3.1 Requirements …………………………………… 28

3.2 Overall Architecture …………………………… 29

3.3 Smart Objects …………………………………… 31

3.3.1 Smart Object Architecture ……………………. 31

3.3.2 Virtual User Interface ………………………… 33

3.4 Communications Protocol ………………………… 37

3.4.1 Messaging …………………………………… 37

3.4.2 Addressing and Routing ……………………… 40

3.5 Summary ……………………………………….. 46

Chapter 4. Implementation of a SmARtWorld Environment … 48

4.1 Basic Smart Object ……………………………… 48

4.1.1 Fundamental Layer ………………………….. 49

4.1.2 Functionality & Data Interface Layer ………. 50

4.1.3 Functionality & Data Access Layer ……….. 51

4.2 Primary Server …………………………………… 53

4.3 Landmark Server and Landmark Objects ……… 55

4.4 Object Tracker ………………………………….. 58

4.5 Summary ……………………………………….. 59

Chapter 5. User Interaction and Display Devices ……………. 61

5.1 Wearable System ………………………………….. 61

5.1.1 Pose Tracking ………………………………... 62

5.1.2 Rendering Virtual User Interfaces …………… 65

5.1.3 Bare Hand Interaction ……………………… 70

5.1.4 Occlusion of Virtual Elements by the Hand … 75

5.2 Tablet and Smartphone …………………………… 76

5.3 Device-less Interaction …………………………… 78

5.3.1 Sensors on a Wireless Sensor Network ………. 79

5.3.2 Gaze Tracking ………………………………... 80

5.3.3 Context Recognition …………………………. 82

5.4 Summary …………………………………………. 85

Chapter 6. Smart Object Representation ……………………… 87

6.1 Real and Virtual Objects …………………………. 87

6.2 Realistic Rendering ………………………………. 88

6.3 Physical Simulation ………………………………. 90

6.4 Sound Response …………………………………. 92

6.4.1 Sound Source …………………………………. 92

6.4.2 Sound Renderer ………………………………. 95

6.5 Summary ………………………………………… 96

Chapter 7. Manufacturing Applications ……………............ 98

7.1 Manufacturing Job Shop ……………………… 99

7.1.1 Smart CAD Object ………………………… 99

7.1.2 Smart Machining Object …………………… 101

7.2 Manufacturing Grid …………………………….. 104

7.2.1 Web Server ………………………………… 105

7.2.2 Cloud Gateway ……………………………… 107

7.3 Visual Programming …………………………….. 109

7.3.1 Robot Task Programming ………………….. 110

7.3.2 Programming Robot Safety Procedures ……. 113

Chapter 8. Conclusion ………………………………………. 118

8.1 Achievement of Objectives ……………………… 118

8.2 Contributions …………………………………… 122

8.3 Recommendations ………………………………. 126

Publications from this Research ………………………………. 128

References ……………………………………………………… 129

List of Figures

2-1	Coordinate transformations from virtual object to AR ………..	14
3-1	Architecture of a smart object ………………………………	31
3-2	Network connections in a SmARtWorld environment ……….	40
3-3	Propagation of smart object existence ………………………..	43
3-4	(a) Addresses used by hubs for objects hosted directly. (b) Addresses used by hubs for the same objects which are hosted directly or indirectly. (c) Addresses used by one of the objects to send messages to the other objects. (d) Routing of a message over multiple hubs. …	44
4-1	Architecture of a UAR environment …………………………..	48
4-2	Creation of a virtual user interface ……………………………	52
4-3	Virtual user interface definitions for the basic smart object …	53
4-4	Database of smart object information in the primary server …	55
4-5	Virtual user interface of a landmark object …………………...	57
5-1	A wearable system …………………………………………….	61
5-2	Flowchart of the wearable system program execution ………..	62
5-3	Occlusion of virtual objects by real objects …………………...	68
5-4	Texture-based font rendering …………………………………	68
5-5	Signed distance field representation of fonts …………………	69
5-6	Zoom-invariant font quality and font effects …………………	69
5-7	(a) Depth of a convexity defect indicates presence of fingers, (b) fingertip is the furthest point from the centroid of the hand …	71
5-8	The detection stages of different gestures ……………………..	72
5-9	Bare hand interaction with virtual user interface elements ……	74
5-10	Occlusion of virtual objects by the user’s hand ………………	75
5-11	Flowchart of the Android system program execution …………	77

5-12 Touch-screen interaction with virtual user interface elements … 78

5-13 Setup for object interaction using gaze tracking ……………… 81

5-14 Placement of smart objects for gaze tracker interaction ……… 82

5-15 Training an HMM-based context recognition object using a smartphone … 85

6-1 A virtual weather sensor object ……………………………….. 88

6-2 Shadows cast by virtual objects due to real light sources in the environment ……………… 90

6-3 A virtual object reflecting the real environment ……………… 90

6-4 Sound waves generated by two smart objects with different stiffness and natural frequency …………… 95

7-1 (Top) Smart CAD object creation tool, (bottom) SolidWorks part document converted into a smart CAD object. …………… 100

7-2 An interactive smart CAD object ……………………………… 101

7-3 Smart machining object: (a) Maintenance interface, (b) CAM interface, (c) Dragging a smart CAD object to the CAM interface, and (d) Smart CAD object loaded in the CAM interface ………... 103

7-4 Architecture of manufacturing grid of smart objects ………… 104

7-5 Smart machining object from a remote SmARtWorld environment ……… 107

7-6 Flow diagram of a program that stops a factory robot arm when a worker approaches it. ……… 115

List of Tables

3-1 Basic data and commands of a smart object 33

3-2 List of XML tags for interactive elements of a virtual user interface 35

3-3 List of standard commands and their parameters 39

4-1 Command and RPC handling procedures for a basic smart object 51

4-2 RPCs in a landmark server object 56

4-3 RPCs in a landmark object 57

7-1 Smart objects of a pick-and-place robot workspace 111

7-2 Smart objects for flow-based programming in a SmARtWorld environment 114

List of Abbreviations

2D	-	Two-dimensional
3D	-	Three-dimensional
AP	-	Access point
AR	-	Augmented reality
ARAF	-	Augmented Reality Application Framework
ARML	-	Augmented Reality Markup Language
ASCII	-	American Standard Code for Information Interchange
CAD	-	Computer-aided design
CAM	-	Computer-aided manufacturing
CNC	-	Computer numerical control
CV	-	Computer vision
FBP	-	Flow-based programming
GML	-	Geography Markup Language
GPS	-	Global Position System
GUI	-	Graphical user interface
HMD	-	Head-mounted display
HMM	-	Hidden Markov model
HTML	-	HyperText Markup Language
IP	-	Internet Protocol
KHARMA	-	KML/HTML Augmented Reality Mobile Architecture
KML	-	Keyhole Markup Language
LAN	-	Local area network
LED	-	Light-emitting diode
MGrid	-	Manufacturing grid

MRU	-	Most recently used
ODE	-	Open Dynamics Engine
OOP	-	Object-oriented programming
RF	-	Radio frequency
RPC	-	Remote procedure call
SDK	-	Software development kit
SNAP	-	Synapse Network Application Protocol
TCP	-	Transmission Control Protocol
TUI	-	Tangible user interface
UAR	-	Ubiquitous augmented reality
UbiComp	-	Ubiquitous computing
UDP	-	User Datagram Protocol
URI	-	Uniform resource identifier
URL	-	Uniform resource locator
VR	-	Virtual reality
WSN	-	Wireless sensor network
XML	-	Extensible Markup Language

Summary

The aim of ubiquitous computing is to shift computing tasks from the traditional desktops to the user's physical environment. Today, the manifestation of this vision can be seen in the proliferation of tablet devices and smartphones that provide access to services and applications. Everyday objects are transformed into smart objects, i.e., objects with computing and networking capability, which can sense and have rich contextual aware functionality. Everyday environments are transformed into smart environments that automatically monitor and adjust conditions, such as temperature and lighting for the inhabitants.

There are a number of limitations with current technologies. First, the user interfaces of smart objects and ubiquitous services are not intuitive and demand much focus from users. Second, the application development process requires expert knowledge, which means less fine control by users over their environment. Third, the types of applications and interfaces that can be implemented in a smart environment are limited by physical constraints. Augmented reality (AR) allows for computer generated graphics, sound and other sensory stimuli to be added into the user's experience of the physical world, therefore opening up many possible enhancements to ubiquitous computing.

In this research, a framework called SmARtWorld is proposed which aims to facilitate smart AR environments. SmARtWorld is designed for universal applications with a focus on intuitive and user-friendly interfaces to computer

applications. It is a component-based distributed system with smart objects as the building blocks of applications embedded into the physical environment. It incorporates AR technologies such that smart objects and their user interfaces can break physical boundaries and be created for maximum utility to the users.

Multiple research issues have been investigated. The basic architecture of a smart object and the networking infrastructure and protocols needed in order to create a ubiquitous AR environment have been developed and forms the foundation for subsequent developments. Various user interaction and display devices have been explored and integrated with SmARtWorld, demonstrating the separation of hardware and applications that the framework provides. As a result, a smartphone system and a wearable system have been developed that can be used with a SmARtWorld environment. The ways in which real and virtual smart objects can co-operate and co-exist in the same environment have also been studied. Finally, the potential impact that this research can make in the manufacturing industry has been studied in three areas, namely, as an interface for workers to access computer-aided manufacturing technologies in a job shop, as a basis for a manufacturing grid, and as a visual programming tool of manufacturing tasks.

The main contribution of the research is a new component-based framework for building UAR environments and applications, based on the novel idea that every component is a smart object with a virtual user interface to its data and functionality. All smart objects share the same architecture which includes a hardware abstraction layer. This allows for flexibility in the hardware and

software used to implement the smart object. A standard protocol for communication and a virtual user interface definition schema have been developed in this research so that smart objects can be accessed in any UAR environment. The implementation of smart objects that perform the fundamental functions needed for UAR applications, namely, the primary server, hubs that connect smart objects on different networks, viewing devices, landmarks for tracking and registration, and trackers for real objects. Smart objects that add interaction and rendering functionality to any UAR environment have also been investigated. These include context-sensing objects, environmental capture objects, light sources, and physics engine and sound rendering objects.

Issues that still warrant further development include error handling, network latency and tracking performance. The ergonomics of wearable systems is also an issue with the current hardware available, but it is hoped that this can be improved as technological advancement in this area is moving rapidly.

Chapter 1. Introduction

1.1 Ubiquitous Computing

The concept of ubiquitous computing (UbiComp) was formalized by Mark Weiser as he described its vision in a seminal paper, writing that technologies should "disappear into the background" so that users are "freed to use them without thinking" and are able to "focus beyond them on new goals" (Weiser, 1991). The problems that Weiser and other UbiComp researchers found with the traditional desktop model of computing relate to its computer-centricity and still hold true today. The computer screen becomes the focal point of the user's attention which interferes with the user's normal cognitive process when performing tasks and problem-solving. The act of interacting with a computer itself presents an overhead cost on effort. Furthermore, computers put information at our fingertips resulting in information overload, exacerbating the drain on the user's energy and time.

UbiComp has already made a significant impact on mankind. Ubiquitous computing literally means "computing everywhere". This has already been taken for granted with the proliferation of smartphones and tablets, interactive touchscreens and kiosks in public spaces, and smart household appliances. However, the problem of computer-centricity has merely been transferred to the individual devices, i.e., the problem with the modern model of computing is that it is now too device-centric. All of a person's software tools and information sources exist on a single device. Someone in need of information or location-specific information has to locate a kiosk before being able to

access the services. Smart household appliances can have many more functions than the users can conceive of and have time to discover.

UbiComp aims to move away from the problem of device-centricity altogether by granularizing computing resources into separate objects in the physical environment. Computer functions are presented and actuated through the user's interactions with the environment itself. It is arguable whether any of today's UbiComp systems have been completely successful in eliminating the problem of device-centricity.

1.2 Augmented Reality

Augmented reality (AR) refers to a perception of the real world where computer-generated graphics, sound and other sensory stimuli are added. It is often advocated as a natural complement to UbiComp because a key component of AR systems is the physical environment. AR systems started to appear in the 1990's. In 1992, a see-through head-mounted display (HMD) system was created by researchers at Boeing which could overlay diagrams on real-world objects during aircraft manufacturing operations (Caudell & Mizell, 1992). At the same time, a system of "virtual fixtures" was developed by Rosenberg (1992) which improved the performance of tele-operated tasks by augmenting the operator's vision with a view of the remote environment; this system has an exoskeleton to restrict the operator's motion and the audio overlaid on the operator's view of the remote environment aids in the perception of virtual objects.

AR works by tracking a user's view of the real environment, recognizing and estimating the pose, i.e., position and orientation, of known objects with respect to the user's point of view (via a camera), and rendering computer generated input spatially-registered around the detected objects. A key development in AR was the release of an open source tracking software library for PCs called ARToolKit (ARToolKit, n.d.) in 1999 which implemented computer vision (CV) functions for tracking square planar markers with known patterns efficiently and reliably. ARToolKit has allowed developers and researchers to develop AR applications more easily.

Within the next decade, research into AR applications had exploded as AR found its way into design and manufacturing (Nee, et al., 2012), medical, education, navigation, and entertainment applications (Krevelen & Poelman, 2010), etc. AR technology has rapidly advanced since then as markerless, non-optical-sensor-based, and sensor fusion techniques for tracking have been developed.

AR and UbiComp complement each other in several ways. AR can free UbiComp smart objects and interfaces from the confines of their physical configuration, and this enhances a smart environment in terms of its appearance and types of interaction. AR tracking technology adds fine location-awareness to smart objects which makes them intelligent and responsive to the needs of users. Without UbiComp, the scale and scope of AR applications may be limited. This is because as mere overlays, augmented objects have limited utility. However, if physical objects can be digitized and

become a part of the AR environment, more interactions and behaviors can be designed which can have actual effects on the real environment.

1.3 Research Objectives and Scope

As global knowledge and information grows and the world becomes more interconnected, it is becoming increasingly important to be able to present the knowledge and information intelligently and interactively to users. Packing services and data into individual devices will soon become impractical. Services and data should not be items that are sought after by the users when they feel they need it, but instead should be available wherever and whenever they are needed.

To remove this device-centric characteristic of computing is the main aim of this research. This is achieved by the development of a framework that facilitates AR applications that are embedded in large environments. There are three kinds of users who will benefit from this system, namely, environment developers, application developers, and end-users. Environment developers refers to the persons who set up the hardware infrastructure that turns the environment into a ubiquitous augmented reality (UAR) environment. Application developers are those who create smart objects which encapsulate the functions in an application. End-users are the persons who enter a UAR environment and make use of the smart objects. Therefore, the objectives of this research are as follows:

(1) A common framework for creating UAR environments that abstracts applications from hardware for tracking, interaction and display.
(2) Flexibility in the hardware and software used to implement context-aware smart objects with highly customizable behaviors, appearance and user interfaces.
(3) Flexibility in the hardware and software used to implement viewing and interaction devices.
(4) Recommended practices for AR application development using the proposed framework.
(5) A self-sustainable framework which continues to be relevant as technology evolves.

For objective (1), standard protocols and definitions for communication, interaction and object representation will be proposed. Furthermore, components of the framework will be defined to ensure that UAR environments will be able to provide fundamental AR, namely, tracking and interaction, so that application developers can focus on content.

For objective (2), the software architecture of a smart object will be defined and will incorporate hardware abstraction. Using this architecture, an exploration of the ways in which smart objects can be developed to have different behaviors, graphical properties, and interactive properties will be conducted.

For objective (3), the research will look into the implementation of viewing and interaction devices and to demonstrate the use of different platforms to achieve a variety of user experiences.

For objective (4), various ways with which smart objects can be designed to be more visible but also blend into their UAR environment, as well as their practicability in AR applications, will be explored.

For objective (5), two aspects of self-sustainability of the framework will be investigated. First is the ability for the framework to remain compatible with new hardware and devices. For this aspect, the framework will be designed with hardware-software abstraction at the level of smart objects, and, application-interaction abstraction at the level of applications. Second is the ability for the framework to maintain itself, i.e., creating new smart objects to encapsulate new technologies. For this aspect, the application of visual programming in a UAR environment will be explored.

As this is a wide topic, some important issues have not been included in the scope of this research including security, privacy, quality and reliability of service. The scope of this research has been limited to the following issues:

(1) Tracking of users and objects.
(2) Unifying heterogeneous objects and devices.
(3) User viewing and interaction.
(4) Ubiquitous AR application development.

1.4 Organization of the Thesis

The thesis is organized as follows. First, a comprehensive literature review on the state of art in UbiComp and AR technology as well as UAR frameworks is given in Chapter 2. Chapter 3 describes the SmARtWorld framework in detail, including its requirements, architecture, standards and protocols used. Chapter 4 describes the implementation of a basic UAR environment and its constituent smart objects using the SmARtWorld framework. Chapter 5 details the different implementations of SmARtWorld environments without a viewing device. Chapter 6 describes the different ways in which smart objects can be presented in a SmARtWorld environment. Chapter 7 describes three manufacturing applications of the framework, namely a manufacturing job shop, manufacturing grid, and visual programming. The thesis is concluded with the contributions of this research and recommendations for future work discussed in Chapter 8.

Chapter 2. Literature Survey

This chapter looks at the related research works that have been conducted for placing the research issues into context. Since the main contribution of this work is a framework for UbiComp applications, the review starts with examining relevant UbiComp issues and the systems that have been developed to deal with them. Next, as the framework incorporates AR, a survey on research on the main AR issues of tracking and display is presented. Finally, systems which combine AR and UbiComp will be explored to give an idea of how other researchers have approached this problem.

2.1 Ubiquitous Computing Issues

Costa et al. (2008) lists ten open issues in ubiquitous computing, namely scalability, dependability and security, privacy and trust, mobility (referring to applications that follow the user), heterogeneity, spontaneous interoperation, invisibility, transparent user interaction, context awareness, and context management. Of these, the last six issues are investigated in this research.

2.1.1 Heterogeneity and Spontaneous Interoperation

An UbiComp environment contains many different kinds of sensors, actuators, objects and services built on different technologies and protocols. Many UbiComp systems opt to wrap heterogeneous services and devices as web services as this unifies the representation of user interfaces (Sashimi, Izumi, & Kurumatani, 2005). Several systems take this a step further by proposing to make use of semantic reasoning and ontology structures like RDF (Resource

Description Framework) and OWL (Web Ontology Language) to describe heterogeneous services so that they can be universally understood by different devices (Singh, et al., 2006; Guo, 2008; Soylu & de Causmaecker, 2010). Other systems have proposed their own middleware for extracting meaningful output and control options to suit the application domain (Crepaldi, et al., 2007) so as to provide more suitable interfaces. The use of ontologies and middleware adds a layer of conformity requirement when applications are created and can add computational and memory overhead if a middleware solution attempts to unify many different communication and interoperability protocols.

2.1.2 Invisibility

Invisibility refers to computer hardware being hidden from the user in a UbiComp environment. This can be achieved by the use of wireless mesh networks like SNAP (*Synapse's SNAP Network,* n.d.) and ZigBee (*ZigBee Specification Overview,* n.d.). These networks are formed from tiny networked microcontrollers that can be used for sensing and control. The advent of wireless mesh networks have driven the development of smart buildings with automated lighting and climate control (*Occupying Yourself*, 2010; *LonWorks®-based Office Building*, n.d.) and The Internet of Things (*Synapse Wireless Drives,* n.d.).

SNAP and ZigBee nodes are suitable as agents for simple roles like user input and output, reasoning, learning, etc. (Jin, et al., 2010). However, as they are low-powered and greatly limited in memory capacity compared to a desktop

computer or even a smartphone, it would be difficult to implement sophisticated computer programs on these mesh networks. UbiComp frameworks try to bridge connectivity among different kinds of devices and appliances. The problem of invisibility then lies with the user interfaces and interaction methods that are used to control the functions that are provided in the UbiComp environment.

2.1.3 Transparent User Interaction

Transparent user interaction refers to making the user interface invisible to the user so that the user can focus on the task at hand. There have been reported research works on developing gesture recognition through sensors placed in the environment rather than worn by the user. Hand gesture recognition using CV is an extremely active area of research in user interaction (Rautaray & Agrawal, 2012) where cameras are used to detect hand gestures. This requires the user's hands to remain in the camera's field of view. There is non-vision gesture recognition research, such as through the use of electromagnetic interference (Kim & Moon, 2014) and Wi-Fi signals (Vyas, et al., 2013; Pu, et al., 2013).

Interaction methods that require an interaction device still remain in active development due to better recognition performance and different application requirements. Interactive surfaces are a familiar sight today in public places. These are typically flat screen displays with multi-touch gesture recognition. Over the last 20 years, there have been numerous research works on tabletop interactive displays, many of which do not have a fixed display orientation and

allow access to multiple simultaneous users (Muller-Tomfeld & Fjeld, 2012). Some tabletop interactive surfaces include tangible elements to represent graspable virtual objects (Ullmer & Ishii, 1997; Fjeld, et al., 1998) or recognize and interact with physical objects placed on them (Wilson & Sarin, 2007; Hincapie-Ramos, et al., 2011). A variant of this is the use of wall-mounted display projectors (Pinhanez, 2003; Song, et al., 2007) or user-carried portable projectors (Cao, et al., 2007; Willis, et al., 2011) to project user interfaces onto surfaces and made interactive using CV techniques.

There have been discussions on whether interactive displays can be classified as UbiComp user interaction. With good user interface design, user interaction can still be transparent. However, the heterogeneity of devices and services in UbiComp environments makes user interface design a challenging endeavor. A number of automatic user interface generation approaches for UbiComp environments have been proposed to allow for abstraction between applications and user interface design. Gajos et al. (2008) developed a method using decision-theoretic optimization to generate user interfaces for web browsers and PDAs based on user abilities, preferences, devices, and tasks. Automatic user interface generation based on semantic descriptions of interaction modality and types of service was proposed by Vanderdonckt & Simarro (2010) by adapting from a knowledge base of user interface models to generate an XML-based user interface. The problem with this approach is that even if the automatically-generated user interface is comprehensible by a user, it may not reflect the intention of an application designer in providing a user experience.

An alternative class of user interface is tangible user interfaces (TUIs). A TUI is made up of physical objects that are manipulated directly and intuitively in order to interact with a computer-aided task. Some TUIs are designed as application-specific systems where the modes of interaction with the physical elements correspond to the functionality of the system (Lee, et al., 2006; Nagel, et al., 2010). TUI implementation can also be approached generically with the use of standard interface devices, such as buttons, sliders and pointers, to interact with a UbiComp environment. An example is the iStuff framework (Ballagas, et al., 2003). With this generic approach, applications and system output are abstracted from the TUI so that any kinds of applications can be developed to work with the interaction objects. Short of labeling every interactive object, the TUI approach does not provide the awareness of functionality to the users. This means that the UbiComp environments utilizing TUIs require that users are familiar with the environments.

Wearable devices are another approach to user interaction that is sometimes employed in UbiComp systems. Park, et al. (2008) developed a wearable system consisting of a radio transceiver and GPS receiver worn on a vest, and three-axis accelerometer worn on the finger. The GPS receiver tracks the user's location while the accelerometer recognizes gestures made by the hand. The user points at an object to select it and then makes a gesture corresponding to the operation the user wishes to carry out. The radio transceiver transmits the recognized gestures as a command to the selected

object. Current technology remains an obstacle to widespread acceptance of wearable AR systems mainly due to the size and weight of the display users have to wear on the head and inadequate support for video output from mainstream mobile devices. However, mobile and wearable display technology is rapidly evolving to solve these issues.

2.1.4 Context Awareness and Context Management

Context awareness refers to the ability of the UbiComp environment to understand the state of the user as well as that of the environment, and context management refers to the way in which the UbiComp environment responds to these states. Context awareness therefore relates to sensing capabilities while context management relates to environment automation and responsiveness. Context management is important because it is the means by which information filtering takes place. Environmental and user-worn sensors are typically employed in order to achieve context awareness, together with algorithms, such as logic reasoning (Hunter, 2001; Haghighi, et al., 2008) and machine learning (Danylenko, et al., 2011; Ayu, et al., 2012), that process the data and extract meaning about the environment or a user's actions and intentions. These methods have frequently been applied to activity recognition tasks (Nguyen, et al., 2013; Zhan & Kuroda, 2014).

2.2 Augmented Reality Issues

2.2.1 Tracking

Tracking is used for computing a user's pose, i.e., position and orientation, in the environment as well as that of objects. There are a number of ways to

perform tracking. Thus far, CV is the most widely used tracking approach in AR systems because of its relative accuracy compared to other methods and low-cost as only a simple camera is needed.

ARToolKit (*ARToolKit*, n.d.) is one of the most widely used software in AR. The ARToolKit tracking module works by searching for square planar markers called fiducial markers with known patterns to obtain their 3D pose in the camera image (Kato & Billinghurst, 1999). CV algorithms are used to compute the pose so as to map the world 3D coordinates to coordinates with respect to the camera and then to the 2D image coordinates of the screen of a display device (Figure 2-1). A 3D coordinate system defined with respect to, for example, the top left corner of the marker as the origin can use the pose to render the virtual object, defined in the world 3D coordinates, at that location.

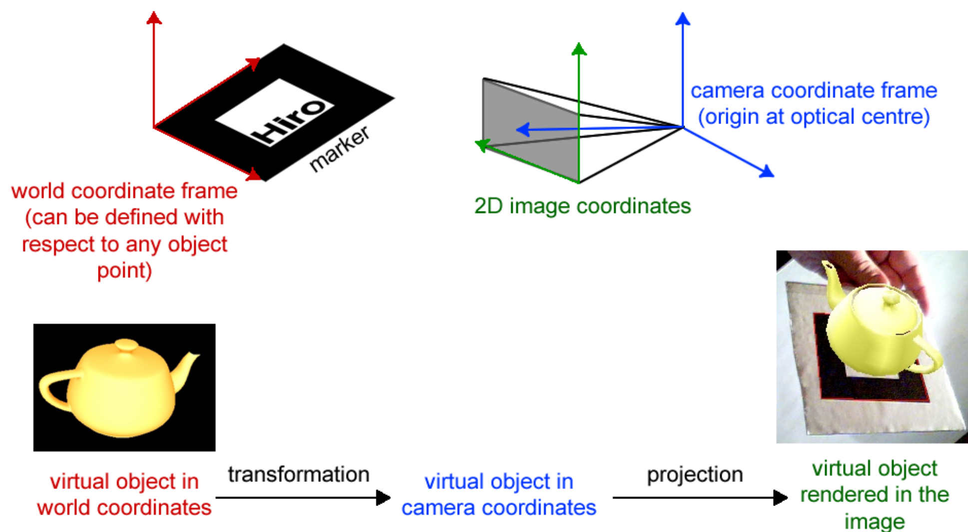


Figure 2-1. Coordinate transformations from virtual object to AR

While marker-based tracking remains widely used in AR applications because of the stability, accuracy and robustness of the algorithm, the main drawback for tracking in a large environment is the need to attach markers to it. Natural feature tracking eliminates the need for markers as it uses features found in the environment. Typical natural feature algorithms involve detecting feature points (points of high contrast change like object corners) in image frames of the scene and matching them to feature points which have been trained into the system. Many markerless AR systems make use of planar features (Wagner, et al., 2008; Fong, et al., 2009) or assume features are planar (Guo, et al., 2009) to reduce the complexity of the algorithm. Planar feature tracking makes use of CV techniques to extract the homography between the trained planar object and the object as seen by the camera. The pose of the object in the camera can then be extracted using the homography (Malis & Vargas, 2007).

Incremental tracking is sometimes used to supplement or enhance marker-based and markerless tracking in cases where continuous marker or natural feature tracking is not possible, such as outdoor and large area applications. There are vision-based methods like optical flow (Mooser, et al., 2007; Luo & Bhandarkar, 2007) and structure from motion (Mooser, et al., 2009), as well as inertial sensor-based methods that track a user's motions (Aron, Simon, & Berger, 2007). As inertial sensors are now commonly embedded in mobile devices along with cameras, a number of hybrid optical-inertial tracking systems have been researched for AR applications (Reitmayr & Drummond, 2006; DiVerdi & Hollerer, 2008). However, in practice, large and cohesive

AR environments with precisely-placed virtual objects are still challenging to implement. Miyashita et al. (2008) implemented an AR museum guide system using an ultra-mobile PC (UMPC) with a feature-rotation sensing hybrid tracking approach; however, whenever the system was switched to inertial tracking in the absence of features, the tracking result was inaccurate. They dealt with this problem by placing augmented information in floating balloons so as to hide the inaccurate tracking. The term “Swim AR” has been used to describe augmented graphics that float about a range of positions when accurate pose tracking cannot be obtained (*KHARMA Framework*, n.d.).

A system known as PTAM (Parallel Tracking and Mapping) does not restrict itself to tracking planar features. PTAM builds a map of features as the camera moves around the environment using SLAM (simultaneous localization and mapping) and simultaneously tracks its position using the map of features (Klein & Murray, 2007). A map contains feature points extracted from camera images localized in 3D space. By matching feature points detected by the camera with those in the map, the 3D pose of the camera is recovered. A map is initialized by obtaining two camera images that work as a stereo pair and using stereo vision to recover the 3D positions of the key feature points. This is done by the user translating the camera horizontally between a start and end point to simulate horizontal disparity between a pair of cameras. The initial feature points are used to estimate a dominant ground plane. As the camera moves, the mapping process tracks the position of the camera continuously and adds more feature points to the map. PTAM, however, suffers from drift, i.e., inaccuracies in the map build up as points further from the origin are

added. There is also a scale ambiguity when the map is initialized which makes virtual objects appear in the wrong size in the AR scene. Furthermore, the memory footprint of a map in PTAM is large which precludes the application of PTAM in large environments.

CV-based AR allows for very precise placement of virtual objects in real world locations. Geospatial AR is an alternative class of applications that uses geodetic coordinates to locate virtual objects on the Earth. The most widely used positioning system that obtains a user's geodetic coordinates is the Global Positioning System (GPS), but the accuracy of a regular GPS receiver is within a few meters. Geospatial AR is used for outdoor applications that encompass a very large geographical area as GPS receivers only work well outdoors. Until centimeter-accurate RTK satellite positioning systems (Meng, et al., 2008) become widely available in mobile devices, applications will be typically for providing coarse location-specific information and services through AR.

2.2.2 Display and Interaction Devices

A variety of display devices have been used in AR with the common ones being desktops, laptops, tablets, phones, and projectors. Desktops with simple off-the-shelf web cameras for tracking have been used in applications that only take place on a desktop. Tablets and phones allow for mobile AR applications, which use the embedded camera, sensors, and GPS receiver of these devices for tracking and the touchscreen for interaction and display. Wikitude (Wikitude App, n.d.) and Layar (Layar App, n.d.) started out as

applications for smartphones that displayed information and directional cues about places of interest using the GPS location of the device. These applications have since added CV-based tracking for viewing augmented graphics and videos on magazines.

As phones and tablets have small screens, it is difficult to view and interact with augmented graphics. Therefore, an alternative is wearable systems which typically consist of a HMD and laptop. The lack of a touchscreen means novel interaction methods have to be introduced. Schmalstieg & Reitmayr (2007) developed a backpack HMD system to view augmented information around the environment. A handheld spherical device called the iOrb was used with this system that allows users to issue commands and perform 3D selections on objects in the environment (Reitmayr, et al., 2005). The main drawbacks of wearable systems are their weight and ergonomics.

The use of projectors for AR display presents a unique set of challenges. The distortions arising from projecting at a non-planar surface or at an angle to a planar surface can be overcome by pre-distorting the projection image based on the surface geometry and tracking of the user's viewpoint (Park, et al., 2006; Krum, et al., 2012). However, one limitation is that this method needs a surface to project images onto, i.e., augmentations cannot occur in mid-air. Furthermore, mobile projectors cannot project in high light intensities, precluding their use in outdoor and bright environments.

2.3 Ubiquitous Augmented Reality Frameworks

UAR systems aim to provide universal access to heterogeneous objects and services, using AR mainly as a visualization mechanism for their information and user interfaces. Research in this area can generally be categorized as high-level frameworks, component-based frameworks or standards-based frameworks. High-level frameworks implement the low-level functions of the operating platform, such as network communications, tracking, rendering, and interaction, and allow creation of applications through scripts which can be plugged into the UAR infrastructure. Component-based frameworks treat all the low-level functions as abstractions and define a middleware to interface with their actual implementations. Standards-based frameworks only specify the data formats and messaging protocols to allow independently-developed systems to interoperate and to present UAR environments to users.

2.3.1 High-level Frameworks

Kimura et al. (2006) proposed an AR framework for mobile devices wherein mobile AR services in a ubiquitous computing environment are registered to visual tags in the environment, with the services stored as programs in remote locations. Therefore, when a mobile user discovers mobile AR services through tags, the user can choose to download and use the service. Furthermore, the framework under which the mobile AR services are to be created would also have access to the embedded sensors of the mobile devices so that natural interaction can be achieved.

A ubiquitous AR system prototyped by Li et al. (2009) employs a hybrid vision and inertial technique for tracking and registration, and connects to a wireless network of sensor nodes. The nodes are attached to objects of interest, so when a mobile computer carried by a user detects an object in its camera view, computer generated information based on the corresponding sensor and registered to the object is rendered.

High-level frameworks make AR application development very straightforward. Application developers would use development software specified by the framework to program the application and plug it into the infrastructure of the framework. However, the look and feel of the resulting UAR environment and the applications therein cannot be customized easily.

2.3.2 Component-based Frameworks

The DWARF framework (Bauer, et al., 2001) is based on interdependent services which collaborate to form the UAR environment. Each service displays its needs, abilities and connectors using XML scripts. The needs of a service refer to data that the service requires, abilities are the functions that the service offers, and connectors are the communications protocols used by the service. Each network node has one service manager which manages services at the node; there is no central control. The framework uses CORBA (*Documents Associated with CORBA 3.3*, n.d.) to enable different platforms and communication protocols to work with each other.

The Studierstube framework (Schmalstieg, et al., 2002) comprises application objects that contain application data, data operations, and the graphical representation of the data which acts as the user interface to the application. Graphical and application data are added to a distributed Open Inventor scene graph; thus a scene graph can be thought of as a set of application objects that make up an application. Application objects can be hosted by different network nodes, where each node contains a copy of the scene graph that is updated in real-time. Application objects are managed centrally by a session manager which maintains a list of application objects so that new objects and users can be aware of the existing objects. Distributed fundamental AR services such as tracking and video acquisition are accessible using the OpenTracker (*Studierstube project: Open Tracker*, n.d.) and OpenVideo (*OpenVideo Documentation*, n.d.) libraries, which allow for the configuration of custom tracking and video hardware to be configured to work in the Studierstube framework. Application objects are written in C++ as Open Inventor scene graph nodes and can be dynamically loaded during runtime (Kainz & Streit, n.d.). Interaction is achieved through a personal interaction panel (PIP) which consists of a pad on which virtual buttons and sliders are rendered and a pen to select and manipulate the virtual elements. The PIP also serves as a display for private information which can only be seen by the owner of the PIP.

The Tinmith evo5 framework (Piekarski & Thomas, 2003) uses a distributed object-oriented approach with four classes of objects, namely, data, processing (which outputs data), core (core features that other objects can inherit) and

helper (programming interfaces that help with application development). Data objects are used as input to processing objects which produce other data objects. Objects are programmed using C++ and inherit from one of the four classes; hence, tracking devices are implemented as a type of processing object which produces a data object that holds the position of a tracked object. Input devices use the keyboard model where all interactions are mapped to a unique identifier, while motion-based input devices use position offset data to represent motion. Other objects which perform other functions would similarly be implemented.

A component-based framework called VARU (Irawati, et al., 2008) is different from the frameworks that have been introduced as there are three interaction spaces in which objects can simultaneously exist, namely, AR, VR and UbiComp. This means that different users interacting in different spaces can collaborate on the same tasks. In VR, users can only interact with virtual objects, while users in UbiComp are able to communicate with physical smart objects like refrigerators and televisions. Users are able to communicate with both virtual objects and physical smart objects in the AR space. The VARU framework consists of a VARU server and a VARU client. Within the VARU server is an object database, object server and simulation server (for physics simulation of virtual objects). A VARU client implements the AR, VR and UbiComp rendering mechanisms and interaction devices. In the UbiComp and AR space, a middleware called CAIM (Ahn, et al., 2005) and the UPnP protocol (*UPnP*, n.d.) are used to allow physical objects to communicate on a VARU network.

The ARCS framework (Chouiten, et al., 2011) is based on components that use the signal/slot mechanism of the Qt framework (*Signals & Slots*, n.d.) to emit and respond to signals. As Qt is for non-distributed systems, a custom-built middleware is used to enable components on different network nodes to use the same signal/slot mechanism by the creation of proxy signal emitters and receivers. As a result, the granularity of component distribution is very fine, i.e., components can make very low-level function calls to different machines without prior knowledge of their location. Applications in ARCS are defined through the use of XML scripts which specify the signal/slot connections of different components. An application is a finite state machine and each XML script represents a state.

Most existing component-based frameworks provide flexibility by separating tracking and interaction implementation from application development. They typically use a middleware for connecting systems with different communication protocols and rely on specific APIs for application development. The APIs and development environments that must be used for application development may make it easy for programmers to create UAR applications. However, it is also a source of limitation in terms of the compatibility with other software libraries and programming mechanisms. Furthermore, many developers have already established tools and practices for developing applications in their field which may conflict with the ones specified by the component-based framework. Therefore, a more liberal type of framework uses standard definitions to allow interoperability of data and

functionality between components and leaves the implementation completely to the developer.

2.3.3 Standards-based Frameworks

The KHARMA framework (Hill, et al., 2010) is an extension of KML (Wilson, 2008). In KML, placemarks identify a location's name, description and WGS84 coordinates (*Department of Defense*, n.d.). The placement of 3D geometries like points, lines, polygons, and full 3D models on locations is also defined in KML. KHARMA extends the objects that can be placed in placemarks to labels, balloons, sounds, and trackers. HTML and JavaScript content can be placed in balloons. Sounds are defined in placemarks by adding a link to where a sound file is hosted. Trackers defined in a placemark indicate the specific trackers, identified using an ID string, that should be used in a location. For example, if a placemark uses fiducial marker tracking of a specific marker format, the client device would use the appropriate tracking algorithm to detect the placemark and render the graphical elements associated with it.

In the ARML framework (Lechner, 2013), a UAR environment consists of features, which are physical objects on which visual assets can be augmented. A feature defines an anchor, which is used by viewing devices for detection of the feature, and the visual asset to be augmented. An anchor can be a set of GPS coordinates (for geospatial AR), an image or a marker (for computer-vision AR), while a visual asset can be text, images, 3D models or video. There is some integration with GML (Portele, 2007), in particular its geometry

definitions, which are used in KML to define 3D geometry that can be used as anchors. This means that locations that are defined in KML or GML documents can have features defined in ARML attached to these locations. ARML uses ECMAScript (*ECMAScript Language Specification*, 2011), which must be supported by viewing devices if they are to access the dynamic elements of an AR scene. Trackers are defined using a uniform resource identifier (URI) to identify the type of trackers to be used in the AR scene, with remotely hosted tracking code linked to using a uniform resource locator (URL).

The ARAF standard (Preda, et al., 2013) defines a scene graph format where nodes can be of different basic types, such as media, script, sensor, actuator, scene animator, communication and compression. New node types can be defined based on the basic node types. Media nodes can be audio, image, video, text and 3D models. Sensor nodes generate data and allow for user interaction. Scene animators modify certain nodes by interpolating their orientation, scale, position, color, or some other value between a range over time. Script nodes can be programmed using ECMAScript to generate triggers to other nodes. Communication and compression nodes handle transfer and streaming of various kinds of data, e.g., playback of video. ARAF works in conjunction with the MPEG-V format (Han & Kim, 2014) which specifies the syntax and semantics of data and command representations to enable interoperability between virtual and real worlds. Thus, data formats for sensor nodes used for user interaction, virtual object data and properties, and command formats for the control of actuator nodes are all governed by

MPEG-V. Tracking for AR is accomplished using sensor nodes which can output position and orientation or GPS data. For computer-vision AR, the camera is implemented as a sensor node. Nodes are defined in XML files with the locations of raw data and scripts indicated by URLs. An ARAF device is a viewing device that interprets ARAF files and allows users to view and interact with nodes.

As discussed earlier, standards-based frameworks allow for more freedom in the implementation of objects in the UAR environment as well as input and output devices. The developer is able to use any CV-based tracking algorithms with the data provided through the standards. However, the standards reviewed are not for UAR applications per se but for distributed AR applications. The distinction between UAR and distributed AR is the lack of interoperability of physical objects and their integration in applications. A standards-based framework that can truly be classed as UAR is still not available.

2.4 Summary

The issues of UbiComp include hiding user interaction devices and computer hardware and enabling systems on different platforms and protocols to be interoperable. The fundamental challenges of AR are mainly in the tracking and display technology. The main thrust of this research is to achieve a UAR framework that handles all the afore-mentioned challenges while making application development rapid and easy. There are three main types of UAR frameworks, namely, high-level frameworks, component-based frameworks,

and standards-based frameworks. High-level frameworks are generally aimed at easing application development at the cost of application implementation flexibility. Component-based frameworks are highly successful in achieving interoperability of different systems and distributed computing but enforce the use of specific APIs and development environments. Standards-based frameworks, on the other hand, while allowing for implementation flexibility, do not have very good support for physical objects and do not ease application development.

Chapter 3. Design of the SmARtWorld Framework

3.1 Requirements

The framework proposed in this research, called SmARtWorld, is for the creation of UAR environments. UAR environments implemented with SmARtWorld will hereafter be called SmARtWorld environments. The basic requirements of UbiComp and AR which were discussed in Chapter 2, i.e., heterogeneity and spontaneous interoperation, invisibility of computer hardware, transparent user interaction, context awareness and management, and tracking of users and objects, will be considered in the design of the SmARtWorld framework.

User experience is another design consideration in SmARtWorld. Each of the UAR systems reviewed in Chapter 2 targets one or a combination of the following three groups of users, namely, end-users, application developers and resource creators. The SmARtWorld design, however, will account for the requirements of all three groups. For end-users, a SmARtWorld environment must provide accessibility to computer functions in a user-friendly way. Application developers should not have to be concerned with the implementation of the fundamental technologies of UbiComp and AR, and they should have the freedom to create any kind of application in a rapid and straightforward manner. Lastly, resource creators need to have assurance that the resources they create will be usable in any SmARtWorld environment while having complete flexibility in the way the resources would appear and behave.

Finally, with limitless potential applications of SmARtWorld environments and locations where they would be useful, SmARtWorld will be designed for universal access to all applications and SmARtWorld environments so that applications in general will not have different requirements of viewing and interaction hardware. However, application and resource designers will still have the freedom to cast such restrictions.

3.2 Overall Architecture

The main novelty of the proposed framework is that every resource, including a viewing device, is a smart object. Object-oriented programming (OOP) is a major influence on the design of this framework. Every smart object has data and behavior, with a basic set of data and functions found in every smart object, similar to the base "Object" type of some OOP languages like C# and Java. In bringing this concept into a SmARtWorld environment, smart objects can either be a physical or virtual object rather than a segment of computer code. The basic data and functions of smart objects are to facilitate basic UAR functionality such as communication, tracking and invocation of behavior. Smart objects are the very basic building blocks of applications in a SmARtWorld environment and can be implemented to encapsulate any kind of behavior and provide any kind of data. To allow for freedom of implementation, there are no typed objects which smart object implementations have to follow or inherit from.

A SmARtWorld environment is basically a physical location where smart objects are linked to a network and can be accessed by users. There is one primary server to which smart objects and viewing devices are generally connected. Once connected to the primary server, smart objects will become aware of each other and start waiting for invocation of their functions, requests for the data they provide, and start working together automatically. Smart objects can have functions which are invoked via remote procedure calls (RPCs). They can also have a virtual appearance that is seen in AR that can be used as a user interface for input to and output from the smart object. The addition of a virtual user interface can make complex computer interaction through smart objects more user-friendly and transparent than completely relying on physical interactive elements.

Application development at a high level is simply just the placement of smart objects in a SmARtWorld environment. Application developers can create new smart objects to fulfil some required functionality in their application or they can obtain smart objects created by resource creators. Resource creators can thus be product manufacturers, software developers, researchers, or hobbyists who create smart objects with specific specialized functionality which are provided to application developers. The various aspects of the SmARtWorld framework will be described in detail in the subsequent sections of this chapter.

3.3 Smart Objects

3.3.1 Smart Object Architecture

Every resource in a SmARtWorld environment is a smart object which can provide data and functions. Function calls are technically RPCs because they are typically invoked outside of the program that runs in the smart object. Each piece of data and RPC is associated with a data and RPC identifier respectively which are character strings allowing other objects to recognize data and RPCs that may be useful. There is also a set of basic commands common to all smart objects which facilitate basic UAR functionality.

The creation of a smart object involves the implementation of three main layers, as shown in Figure 3-1: the fundamental layer, which consists of the hardware used by the smart object and the native operating systems and libraries that are used to access the hardware; the functionality & data interface layer, which sends and receives messages using a communications protocol defined for SmARtworld and handles the messages appropriately; and the functionality & data access layer, which exposes the data and RPCs of the smart object to other objects.

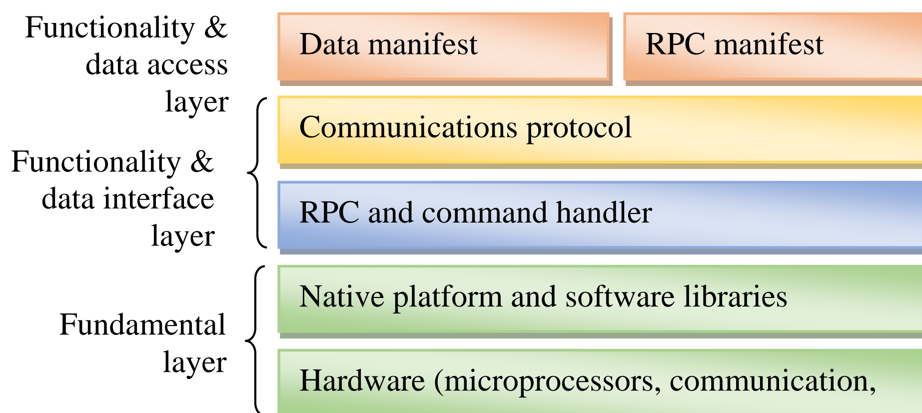


Figure 3-1. Architecture of a smart object.

There is virtually no restriction on the implementation of the fundamental layer. Key hardware elements are a microprocessor with memory to store and execute the programming of the smart object, and a network port to connect to the primary server. Any software libraries can be used in the fundamental layer, as long as the hardware platform can support the functions provided by the libraries. The underlying network protocol used by smart objects to communicate is not restricted, provided they have a communication route to the primary server via smart objects called hubs, which are able to relay messages between different networks. This will be explained further in Section 3.4.2. To users and other objects in the UAR, the fundamental layer is the hidden part of a smart object which quietly does its job. However, there may be physical elements in the fundamental layer which are perceptible by users in the UAR, like LEDs or sound.

The functionality & data interface layer implements the communications protocol and RPC and command handler. The communications protocol allows the smart object to interpret received messages on the network and format outgoing messages properly, and also conducts the necessary procedures when first joining a SmARtWorld environment. The communications protocol is described in detail in Section 3.4. The RPC handler invokes the corresponding behavior and internal function calls when an RPC is received from the network. The command handler takes care of the basic smart object commands that are received on the network. Table 3-1 lists some of the basic data and commands, some of which are mandatory, that a smart object can implement.

The data and RPC manifests are the only ways to access the appearance and functionality of a smart object, and this is the reason they are in the functionality and data access layer. In a SmARtWorld environment, data and RPCs can be used autonomously by smart objects or they can be explicitly accessed by users through the user interface of the smart object. The virtual interface of a smart object is an important aspect of the SmARtWorld framework. However, it is not mandatory for a smart object to have a virtual interface as viewing devices can generate ways for users to access RPCs and data using the identifiers stored in the manifests.

Table 3-1. Basic data and commands of a smart object

Data	Purpose
3D model and user interface	For rendering the virtual appearance and interactive elements of the smart object. Optional.
Data manifest	To expose data that other objects can retrieve. Mandatory, but can be empty.
RPC manifest	To expose the RPCs that other objects can invoke. Mandatory, but can be empty.
Pose	The position and orientation of the smart object in the SmARtWorld environment, for rendering and/or context-management. Mandatory, but can be unknown.
Command	**Purpose**
Get data	Allows other objects to request for a specific piece of data. Optional.
Get data manifest	Allows other objects to request for the list of data available from this smart object. Mandatory.
Get RPC manifest	Allows other objects to request for the list of RPCs available from this smart object. Mandatory.
Get pose	Allows other objects to request for the pose of this smart object. Mandatory.

3.3.2 Virtual User Interface

Interaction with objects is achieved either by direct manipulation of the objects, which can be the traditional mode of operation of the objects, or via

physical controls that are built into the smart objects to access their functions, or through the virtual user interface of the objects. The virtual user interface also provides information and visual output from smart objects as text or 3D graphics.

The 3D model of the virtual interface plays a crucial role in providing complex object-centric interaction methods with the object and is stored in a 3D assets file with the smart object. The 3D assets file includes the usual mesh data, such as vertices, normals and material properties. In addition, the definitions for the interface element, including the RPCs that they trigger, must be added to the 3D assets file. The interface element definitions determine the ways with which the element is interacted, e.g., whether the user simply has to touch to activate or whether it involves some other actions, and the data that is passed to the RPC of the object.

Due to its extensibility and comprehensiveness in describing graphical scenes, the COLLADA file format (Khronos Group, 2008) is chosen to store the virtual user interface in an XML file. In the COLLADA specification, a 3D scene is arranged into separate nodes representing separate virtual models. The node ID is thus used to reference nodes so that smart objects can alter their user interface dynamically. Each interactive element is defined as a separate node, using custom interaction tags and parameters that are defined outside the COLLADA XML schema for this research. Table 3-2 lists the tags and parameters that can be added to a COLLADA file to make nodes interactive and dynamic.

Table 3-2. List of XML tags for interactive elements of a virtual user interface

Tag	Parameters	Purpose
Click	Node ID, RPC identifier	Define element as a button
Drag	Node ID, minimum and maximum values, minimum and maximum translation, RPC identifier	Define element as a slider
Rotate	Node ID, minimum and maximum values, minimum and maximum rotation, RPC identifier	Define element as a knob
Object	Node ID, data identifiers, RPC identifier	Use a smart object as input
Translate	Node ID, x-displacement, y-displacement, z-displacement, animate (true/false), speed (units/second)	Translate the object
Rotate	Node ID, x-rotation y-rotation, z-rotation, animate (true/false), speed (units/second)	Rotate the object
Scale	Node ID, x-scale, y-scale, z-scale, animate (true/false), speed (units/second)	Change the scale of the object
Overlay	Node ID, colour	Change colour of the element
Text	Node ID, text, font colour, effect, effect colour	Render text on the element with optional glow or outline effect
Visible	Node ID, visibility (true/false), real (true/false)	Show or hide an element
Special	Node ID, special effect ID	Indicate that the element has some special rendering property (e.g. reflective, glows, etc.)

The first four tags in Table 3-2 are input tags. The first three tags allow for direct manipulation of the graphical element using a one or two-finger gesture. The last input tag allows for using smart objects as input to the RPC invoked by the corresponding graphical element. By defining the three gestures (Click, Drag, and Rotate) as input tags, these gestures are supported natively in the SmARtWorld framework, and viewing devices should ensure that these gestures are available as input methods. Smart object creators would be

assured that users would be able to use these four gestures to interact with their virtual user interface of the object. These four gestures are generally classified as direct gestures as opposed to symbolic or abstract gestures, because they involve direct manipulation of objects typically in 2D (though the objects themselves could be in 3D). They have been chosen for native support for two reasons. Firstly, these gestures have become pervasive in daily life as they are supported by almost all mobile devices and interactive screens. Secondly, direct gestures are much easier to learn and discover (Yee, 2009) and their intuitive meaning is generic and suitable for different applications.

All the input tags have the identifier of the RPC they are meant to invoke as a tag parameter. The Drag and Rotate tags have a “minimum and maximum value” parameter to define the start and end values of the dragging and rotary motions, and a “minimum and maximum translation” or “minimum and maximum rotation” to indicate how far each element is supposed to be translated or rotated from its original state. The latter parameters can also be used by viewing devices to display the dragged or rotated state of the interactive element accordingly, stopping when the maximum value is reached. The Object tag has “data identifiers” as one of its tag properties which indicates the data that is to be transferred from the object to the interactive element. It is then the responsibility of the viewing device to obtain the data and transfer them to the receiving object.

The last seven tags in Table 3-2 are output tags. They modify the graphical elements that they are applied to. The ability for objects to dynamically alter

the node definitions of their virtual user interface allows for output tags to reflect changing states or values held by the smart objects. The first three output tags can be animated. This is achieved by the viewing device which interpolates between the initial and final state of the element according to the speed property. The Visibility tag can be used to hide and show different interface elements. This feature can be used to realize context-aware user interfaces or interfaces akin to sequential menus. The visibility tag also has a "real" parameter that can indicate whether the mesh geometry is representing the physical geometry of the object, in which case viewing devices can hide the virtual mesh but use its geometry for occlusion of virtual objects (see Section 5.1). The special tag allows for special effects to be applied to the appearance of smart objects (see Section 6.2).

3.4 Communications Protocol

3.4.1 Messaging

A high-level communications protocol has been defined with a list of standard commands to facilitate the basic functionality that all smart objects need to possess in order to work in the SmARtWorld environment. This protocol is implemented on top of a lower level protocol, such as TCP (Cerf, et al., 1974), which takes care of data transmission, error detection, packet splitting and reassembly.

Majority of the standard commands can be categorized as a "GET" command for requesting specific data, or a "SET" command for transferring data to an object (Table 3-3). A message that is being sent between smart objects

typically consists of four main components, namely, the command, target address, sender address, and parameters. Commands broadcast to all objects do not include the target address in the message. The whole message is encoded as UTF-8 characters (Yergeau, 2003). Every message is concluded by the two-character sequence “<CR><LF>”, which are the carriage return and line feed characters respectively. They are used to indicate the end of a message; this allows the content of a message to be of any length. The general format of a message is thus “<command>, <target address>, <sender address>, <parameters><CR><LF>”.

Raw binary data is converted to Base64 encoding (Josefsson, 2006) before being transferred over the network so that raw bytes can be transferred as alphanumeric characters and thus will not be confused with the delimiting characters, such as commas and vertical bars, which are used to separate different parameters in the message. It is the responsibility of the receiving object to decode the raw data received and use it appropriately.

RPC parameters make up one component of a message. While the main components of a message are separated by commas, different parameters are separated by vertical bars (Unicode character 0x007C). As an example, a message from a smart object with address “3” informing a smart object with address “5” of its pose would be: Set My Pose,3,5,4.5|6.3|1.2|0|45|0.

Table 3-3. List of standard commands and their parameters

Command	Parameters	Purpose
New	Sender address	Announce the existence of an object
Disconnect	Sender address	Indicate that an object is leaving the environment
Get RPC Manifest	Target address, sender address	Request for the list of RPCs from the target object
Get Data Manifest	Target address, sender address	Request for a list of data that the target object provides
Get Data	Target address, sender address, data identifier	Request for a specific piece of data identified by the data identifier from the target object
Get Pose	Target address, sender address	Request for the pose of the target object in the environment
Set My RPC Manifest	Target address, sender address, list of RPC identifiers	Send a list of the sender object's RPCs to the target object
Set My Data Manifest	Target address, sender address, list of data identifiers	Send a list of the sender object's available data to the target object
Set My Data	Target address, sender address, data identifier, raw data in Base64 encoding	Send raw data to the target object
Set My Pose	Target address, sender address, translation vector, rotation angles	Send the pose of the sender object
Set My Virtual Interface Node	Target address, sender address, node ID, user interface tag, tag parameters, mesh	Send an alteration to the AR user interface of the sender object (see Section 3.2.2)
Set Your Data	Target address, sender address, data identifier, raw data in Base64 encoding	Set the values of a specific piece of data of the target object's
Set Your Pose	Target address, sender address, translation vector, rotation angles	Set the pose of the target object
RPC	Target address, sender address, RPC identifier, input data	Invoke an RPC in the target object

3.4.2 Addressing and Routing

Smart objects can connect to the SmARtWorld environment using any communication protocol. All connections must eventually lead to the primary server, which uses Ethernet LAN to achieve universal access and to allow viewing devices to connect wirelessly via a router. There is no distinction between viewing devices and smart objects, which means viewing devices can connect to the environment through any protocol and interact with smart objects. Smart objects using different communication protocols connect to the primary server via hubs. The hubs have both an Ethernet LAN connection as well as a connection to the communication protocol used by the smart objects it hosts. A possible SmARtWorld environment setup is shown in Figure 3-2.

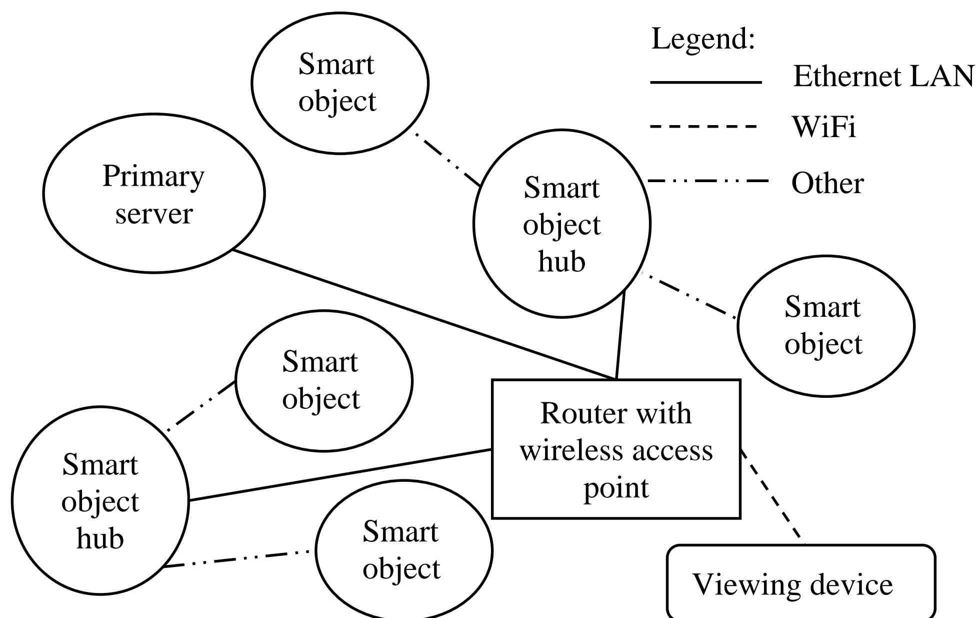


Figure 3-2. Network connections in a SmARtWorld environment.

In order for smart objects in a SmARtWorld environment to be uniquely identified so that messages can be routed to them, a smart object has two

address components. The first component is the network address; this is the address assigned by the networking hardware used by the smart object. An object using an Ethernet LAN connection would have a network address consisting of an IP address and a port number. The second component is an index number that uniquely identifies the smart object hosted at the network address. The second address component is hereafter known as the *internal address*. Smart objects which are not hosted by a hub do not have an internal address; they are simply identified by their network address. The target address and sender address components of a message in the SmARtWorld protocol refer to the internal addresses of the respective objects. Messages intended for smart objects which are not hosted by a hub would have an empty target address. The network address of the sender is assumed to be reported by the lower level communications protocol when a message is received; this is the reason that the network address is not included in a message.

This addressing scheme is important for allowing smart objects being hosted by hubs to have a presence in the SmARtWorld environment. Hubs are basically smart objects which act as a host to other smart objects. A notable feature of the hubs is that they hide the true network and internal addresses of objects from other hubs and assign their own internal addresses to these objects. The importance of this feature will be explained in later sections. The primary server is considered a hub as it acts as a host to all the smart objects in a SmARtWorld environment and it maintains connections to all the smart objects, either via hubs or directly. The primary server is specially designated because a SmARtWorld environment *must* have one (while other hubs are

optional) to serve as a common gateway for WiFi-based viewing devices to view and interact with a SmARtWorld environment.

When a new smart object joins the environment, it broadcasts a “New” message to report its internal address. The hub which the smart object is directly connected to receives the message and adds the smart object to a database of smart objects that it is aware of. Every hub has such a database which include the hub’s assigned internal address and the reported internal addresses of the smart objects. Smart objects directly hosted by the hub would not have a reported internal address, unlike smart objects from other hubs, so this value is empty in the database. The assigned internal address is given by the hub to ensure that it is a unique value not shared by other objects. The hub changes the sender address of the “New” message to the assigned internal address and forwards the message to all the smart objects that are directly and indirectly hosted by it (excepting the smart object that sent the message). Awareness of a new object thus propagates from its direct hub outwards to other objects. Each time the message encounters a hub, the sender address is changed to the one assigned by the hub (Figure 3-3).

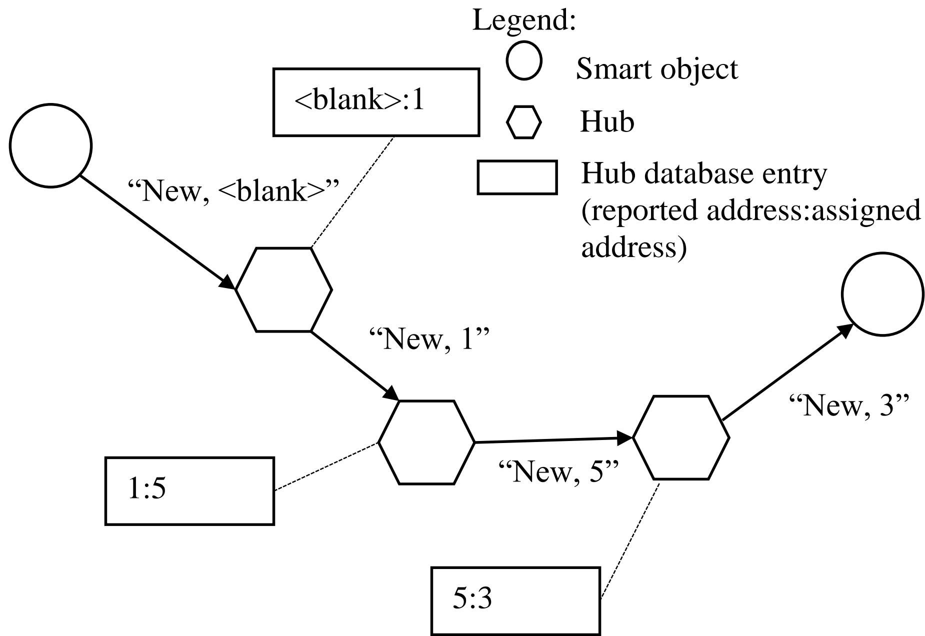
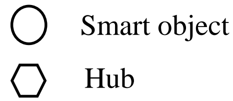


Figure 3-3. Propagation of smart object existence.

Hubs hide the network and internal address of smart objects and assign new addresses to the smart objects when propagating the existence of these smart objects. Figure 3-4 shows the address assignments and addressing changes that take place for routing messages between four objects. Two of the objects are hosted directly by hub A, the other two by hub C, and hubs A and C are connected to hub B. In Figure 3-4, the address of each object is denoted by the network address of its hub followed by its internal address. Figure 3-4a shows the addresses of smart objects assigned by the hubs that host them directly; these addresses reflect the true network locations of the objects.



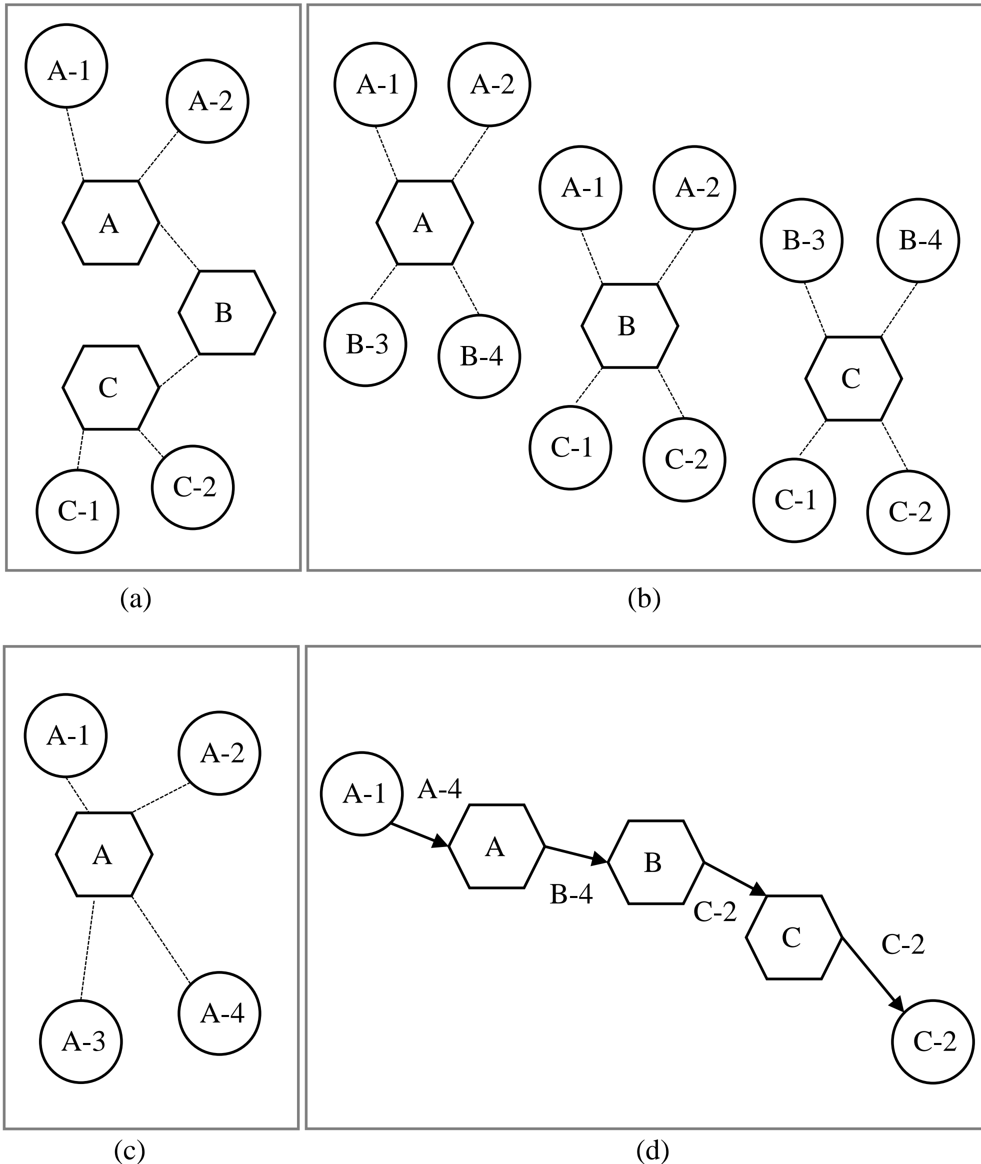


Figure 3-4. (a) Addresses used by hubs for objects hosted directly. (b) Addresses used by hubs for the same objects which are hosted directly or indirectly. (c) Addresses used by one of the objects to send messages to the other objects. (d) Routing of a message over multiple hubs.

In Figure 3-4b, the addresses assigned to the objects hosted by each hub directly and indirectly are shown. When a smart object sends a message to another smart object, it uses the internal address assigned by its hub as the

target address. This means that an object hosted at hub A uses the internal addresses assigned by A to send messages to other smart objects in the environment (Figure 3-4c). For smart objects directly hosted by the same hub, messages can be routed directly to the target object via the hub. For objects on different hubs, the message is routed to the next hub until the message reaches its target. Each time the message is relayed to the next hub, the target address of the message is changed to the internal address assigned by the next hub for the target object (Figure 3-4d).

Apart from connecting different network protocols, hubs serve other purposes. A hub could be used to allow smart objects to create and manage other smart objects. A complex smart object made up of a hierarchy of smart objects could be created using a system of nested hubs. However, for smart objects to join an environment automatically, there must be a mechanism for automatic hub discovery for the object to become aware of the network address of the hub. Automatic hub discovery can take place with a handshaking procedure, but this depends on the low-level communications protocol having a mechanism for broadcasting messages. The actual implementation of the handshaking protocol for smart objects with different networking hardware will be described in subsequent chapters. Alternatively, if there is no broadcasting mechanism for the hub discovery procedure to take place, the network address of the hub can be entered either manually into the smart object provided the object has a way to do this, or programmed into the smart object, in which case if the network address of the hub changes the smart object has to be reprogrammed.

3.5 Summary

In this chapter, the design of the SmARtWorld framework has been described. It is primarily a component-based framework, with a communication protocol and a schema for virtual user interface definition. The novelty of the framework is that every resource, whether it provides fundamental AR functionality or higher-level AR application functionality, is a smart object which uses the same architecture.

Smart objects can have virtual user interfaces which are displayed and interacted with in AR. Virtual user interfaces allow users to access the functionality of smart objects in a user-friendly manner. A high-level communications protocol has been designed which allows smart objects to be aware of each other, work together to provide basic UAR functionality as well as to invoke specially-programmed functionality in each other via RPCs. The use of smart objects as hubs to other smart objects allows for the unification of different networking protocols. There is one hub designated as the primary server which guarantees access to all smart objects and WiFi connectivity for viewing devices.

When compared to the other UAR frameworks reviewed in Section 2.3.2, there is more freedom in the implementation of the fundamental behaviour of a resource because there is no need to conform to a specific programming language or middleware, such as CORBA, which is used by the DWARF framework (Bauer, et al., 2001), Open Inventor, which is used by Studierstube (Schmalstieg, et al., 2002), UPnP, which is used by the VARU framework

(Irawati, et al., 2008) and Qt, which is used by the ARCS framework (Chouiten, et al., 2011). Instead, SmARtWorld utilizes a communication protocol to standardise data transfer and invocation of functions in smart objects, and an XML schema to standardise the definition of virtual user interfaces so that viewing and interaction devices will be able to access the functionality of smart objects in any UAR environment.

Based on the architecture and principles described in this chapter, the remaining chapters will discuss the implementation of smart objects that provide UAR functionality, viewing devices and smart objects for different purposes as examples of the potential of the SmARtWorld framework.

Chapter 4. Implementation of a SmARtWorld Environment

In this chapter, a demonstrative implementation of a UAR environment based on the SmARtWorld framework is described. The general architecture of a UAR environment is given in Figure 4-1. It consists of the basic smart objects that are required in order to realize an environment with smart objects that can be interacted with through their virtual user interfaces. This implementation consists of the following smart objects, namely, a basic smart object with simple behavior and a virtual user interface, a landmark hub which allows viewing devices to track their pose in the environment by serving data about physical features in the environment, an object tracker to track the pose of smart objects in the environment; and a primary server which links all the smart objects. These objects work together to realize UAR applications.

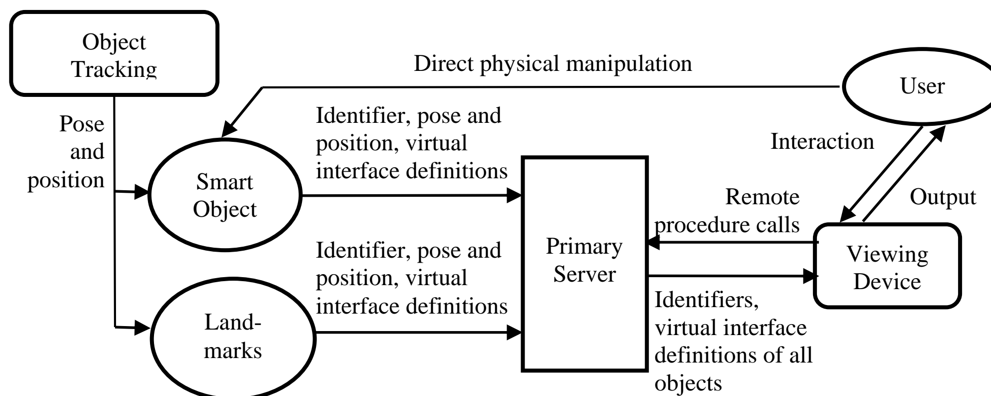


Figure 4-1. Architecture of a UAR environment

4.1 Basic Smart Object

In this section, the implementation of a basic smart object with two RPCs, namely, On and Off, is described. It runs on a PC which has an Ethernet LAN

connection. The smart object has a virtual user interface through which its RPCs can be invoked. The RPCs result in a change of color of the smart object.

4.1.1 Fundamental Layer

A basic smart object that runs on Windows can be implemented easily using the .NET Framework or Java. A computer with an Ethernet LAN network card, the Windows operating system and the .NET software libraries would thus form the fundamental layer of the smart object. Other operating systems which have Ethernet LAN interfaces can also be used.

For networking, the smart object is implemented as a client program (the server being an Ethernet LAN hub) using TCP sockets for data transfer. The first action of a smart object when its program is executed is to initiate the hub discovery protocol. As TCP sockets do not have a broadcast function, the smart object has to broadcast using a UDP socket to the subnet broadcast address to ensure that the hub will receive it. The hub sends a reply containing its network address and port number so that the smart object can establish a formal connection to the hub. To be able to receive the reply, the smart object has to be listening on a specific UDP port to which the hub can reply. A valid UDP port number must be specified by the smart object to avoid clashes with other programs which are on the same network host and listening on the same port number; thus, this allows for multiple independent smart objects (but not Ethernet LAN hubs) to be running on the same hardware. UDP sockets are not used for the other communication tasks because of the lack of reliability that

TCP can provide. If the UDP port number that the smart object broadcasts is lost in transmission or corrupted, or the hub is unavailable during the hub discovery procedure, the smart object can continue the discovery procedure until the procedure is successful. Once successful, the smart object will not need to be on a UDP port. To achieve this, a standard UDP port for Ethernet LAN hubs to use must be defined, and this is set as 2056 in this research.

The hub discovery procedure for Ethernet LAN devices can be summarized as follows. The smart object broadcasts the port number that it uses for hub discovery to UDP port 2056, and waits for a hub to reply to the port with its network address and port number for TCP socket communication. If there is no reply after a certain time period, the smart object repeats the procedure. Once hub discovery is achieved, the smart object establishes a persistent TCP connection with the hub, and sends a “New” command and receives the other “New” commands from the hub via the TCP socket.

4.1.2 Functionality & Data Interface Layer

The communications protocol is entirely based on the scheme described in Section 3.4. When data is received by the network card, it is interpreted as a stream of UTF-8 characters. When the two-character sequence “<CR><LF>” is encountered, the stream of characters read is treated as a single message. The message is split into its command, target address, sender address, and other parameters and sent to the appropriate handler code. Table 4-1 shows the basic handling procedures implemented for the relevant standard commands for this object. Other smart objects may have different ways of handling

commands or may lack some of the handlers, depending on the functionality they provide or way they work.

A simple smart object does not need to implement many of the standard basic commands, especially if it does not depend on other smart objects for its functionality. The five commands in Table 4-1 are sufficient for a smart object to have a virtual user interface displayed in the UAR environment and interactivity with users.

Table 4-1 Command and RPC handling procedures for a basic smart object

Command	Parameters	Procedure
Get RPC Manifest	Target address, sender address	Send a “Set My RPC Manifest” command with the “On” and “Off” RPCs listed (their identifiers are set as RPCStandalone_On, RPCStandalone_Off respectively).
Get Data Manifest	Target address, sender address	Send a “Set My Data Manifest” command with just one data identifier in the manifest, the one for virtual user interface (the data identifier is set as VirtualUserInterface).
Get Data	Target address, sender address, data identifier	Responds only if the data identifier is “VirtualUserInterface”, in which case a “Set My Data” command embedded with the smart object’s user interface COLLADA file is sent.
Get Pose	Target address, sender address	Send a “Set My Pose” command with the pose of the object as its parameter.
RPC	Target address, sender address, RPC identifier, input data	If the RPC identifier is “On”, send a “Set My Virtual Interface Node” command to change the color to green. If it is “Off”, change the color to red.

4.1.3 Functionality & Data Access Layer

As mentioned, the only RPCs that the basic smart object have are “On” and “Off”, and the only data that it provides is its virtual user interface. As these RPCs invoke standalone functions, i.e., there is no interoperability with other

smart objects, their RPC identifiers are not very important. However, they should not collide with RPC identifier definitions that have meaning. Such standalone RPCs can thus be assigned a special identifier prefix, which is given as “RPCStandalone” in this research. The identifiers for “On” and “Off” are thus “RPCStandalone_On” and “RPCStandalone_Off” respectively.

The basic smart object has a virtual user interface, which means it must have a COLLADA representation of its 3D mesh, materials, interactive and dynamic elements. The virtual user interface is created easily in any 3D authoring software and exported in the COLLADA format. The virtual user interface for the basic smart object is modeled using the free open-source 3D authoring tool Blender 3D (Figure 4-2).

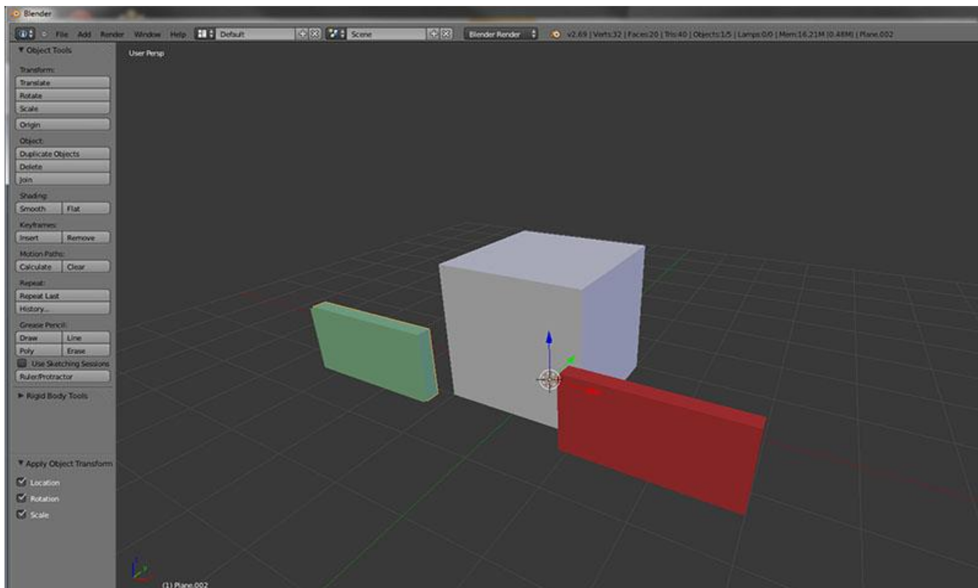


Figure 4-2. Creation of a virtual user interface.

Plugins can be created for most 3D authoring software to enable the definition of interactive and dynamic user interface elements as part of the modeling process. Otherwise, this can be achieved after the COLLADA file has been generated using third party tools or by adding the tags into the COLLADA file manually. The middle cube is the main body of the smart object while the green and red surfaces are buttons for the “On” and “Off” RPCs respectively. The salient definitions for these elements are given in Figure 4-3.

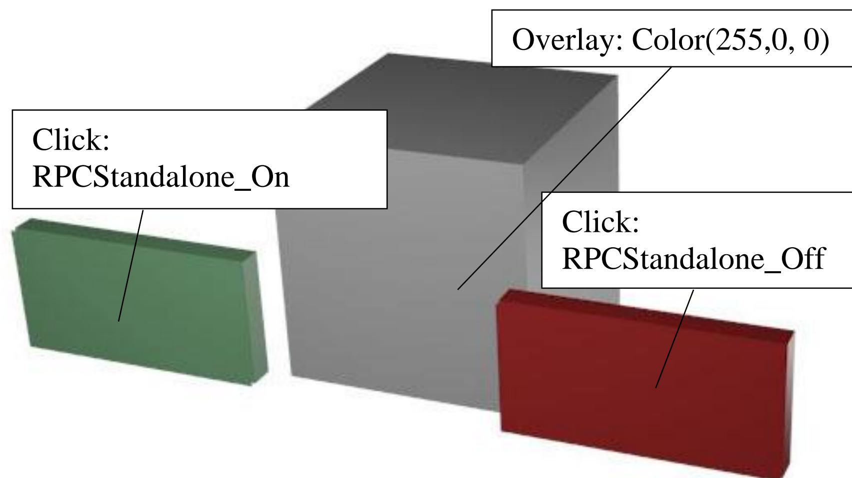


Figure 4-3. Virtual user interface definitions for the basic smart object.

4.2 Primary Server

The primary server is a hub that connects to all Ethernet LAN devices that are smart objects in the same subnet (which would be the case if all of them are connected to the same router). At its core, it is implemented similar to a basic smart object, except that it does not require any RPCs or a virtual user interface, i.e., it can perform its job without human interaction. It runs on a PC with an Ethernet LAN interface. Furthermore, TCP sockets are used for

communications with smart objects and the UDP port 2056 is monitored for the hub discovery procedure.

A significant aspect of the implementation of a primary server is its handling of basic commands. Since an ordinary hub relays messages between smart objects, it has to handle all the basic commands. As detailed in Section 3.4.2, its main responsibility as a hub in handling commands is to change the target and sender addresses from the received commands to those that it assigned and forward the modified commands to the target address. This means that it has to have a database of all the smart objects that are connected to it and this database stores their corresponding network and internal addresses. However, the primary server, and hubs in general, can help streamline communications by maintaining more data about smart objects so as to respond to “Get Data” and other requests immediately without having to relay the message to the actual target smart object. This means that when a new object is connected to the primary server, the primary server would start requesting for all of the object’s data to be stored in its database.

Figure 4-4 shows the server program that has been developed in this research for rapid prototyping of smart objects and UAR environments. It has a GUI with a table that represents the data stored in the primary server’s database of smart objects that are connected to it, namely, their reported internal address and the internal address assigned by the server (Hub ID), the RPCs, and data provided. There are four smart objects connected to the primary server as shown in Figure 4-4. The last object is the basic smart object that has been

described in Section 4.1. The first object is a landmark server, and the next two smart objects are landmark objects hosted by the landmark server which will be described in Section 4.3. As the network addresses of the objects indicate, the landmark server and landmark objects are hosted on the same remote machine while the basic smart object is hosted on the same machine as the primary server, but as a program running independently.

Figure 4-4. Database of smart object information in the primary server.

4.3 Landmark Server and Landmark Objects

The landmark server is like a hub except it does not host other objects on the LAN. Instead, it creates smart objects that are used as landmarks, i.e., landmark objects, for viewing devices to track their pose within the virtual coordinate system of the SmARtWorld environment. The landmark objects that are created are hosted by the landmark server. The core of the landmark server is similar to a basic smart object. It runs on a PC with Ethernet LAN

connection, and uses a TCP socket for communication and UDP port 2056 for hub discovery.

The landmark server stores features on planar surfaces in the environment as landmarks. Each landmark is represented as a rectangular image of the planar pattern. There are several benefits of selecting planar surfaces as landmarks. Firstly, planar patterns can be found naturally in many environments and can be put up easily as posters or decorations. Secondly, the specific algorithm for tracking the landmarks need not be defined. Thirdly, individual planar patterns can be added, removed and switched easily without affecting the ability to track the other patterns.

Like the primary server, the landmark server contains a database of smart objects. However, the database only contains landmark objects. Unlike the primary server, the landmark server provides RPCs that allow viewing devices and other smart objects to interact with it. These RPCs facilitate the creation of landmark objects out of physical objects in the environment as well as the deletion of the landmark object (Table 4-2).

Table 4-2. RPCs in a landmark server object.

RPC Identifier	Parameters	Procedure
LandmarkServerAdd	Image file of the planar pattern, other optional parameters (name, dimensions, pose)	Create a new landmark object and associate the image file as a piece of data identified as LandmarkPlanarImage
LandmarkServerDelete	Target address	Delete the targeted landmark object from the landmark server

Smart objects and viewing devices can interact directly with the landmark server using the RPCs in Table 4-2 to add and delete landmarks. As the landmark server does not have a virtual user interface, it depends on the viewing devices to provide a user interface for users to invoke its RPCs. Landmark objects, however, have virtual user interfaces which overlay the physical landmarks in the environment and their own RPCs so as to allow users to modify or delete the parameters of the landmark objects, namely the name, dimensions and pose of the landmark (Table 4-3). The virtual user interface is procedurally generated by the landmark server (Figure 4-5).

Table 4-3. RPCs in a landmark object.

RPC Identifier	Parameters	Procedure
LandmarkSetName	New name	Change the value of its ObjectName data to the new name
LandmarkSetWidth	Width in cm	Change the value of its LandmarkDimensionWidth data to the new width
LandmarkSetHeight	Height in cm	Change the value of its LandmarkDimensionHeight data to the new height
LandmarkDelete		Instruct the landmark server to delete it

Figure 4-5. Virtual user interface of a landmark object.

4.4 Object Tracker

The object tracker is a smart object that tracks movable objects in the environment. There is no restriction on the type of tracking methods used for tracking objects. There are many possible methods with varying degrees of accuracy and precision, which can vary depending on the conditions of the environment and the type of objects that are tracked.

The fundamental layer of the object tracker consists of the tracking hardware and algorithm. For the UAR implementation presented in this chapter, the most traditional method of tracking in AR is used, namely, fiducial marker tracking. This requires a camera to be attached to the object tracker. Objects that are to be tracked have a fiducial marker attached to them which the camera recognizes and uses to compute the pose of the objects with respect to the camera. An object tracker must know its own pose in the environment so that it can set the pose of the objects it tracks with respect to the environment. An object tracker sets the pose of the objects it tracks continuously using their "Set Your Pose" command. Objects that are tracked by object trackers need to provide some data that allows object trackers to recognize them. This depends on the tracking algorithm used. In this implementation, the Aruco tracking library (*Aruco*, n.d.) is used by the object tracker. The markers used by Aruco are a 5x5 grid of black and white squares which encode an ID integer number between 0 and 1023 in a format similar to Hamming Code. The data provided by objects to be tracked is the ID number of the marker attached to it. The data identifier of the ID number is given as "TrackingArucoMarker", so that when an object joins the environment and the object tracker sees that it has this data

identifier, it requests for the ID number and knows to start looking out for this object. Most other tracking methods require their tracked objects to have some sort of identification or descriptor which can be provided in this way.

A single fiducial marker tracking object will not be able to perform tracking over a large area, which means that objects outside of its tracking zone will not have their pose updated. A number of such tracking objects need to be placed in the SmARtWorld environment. In a large environment, it might become a challenge to ensure that dead tracking zones are minimized. Other tracking techniques which cover a larger area, such as magnetic field tracking and signal triangulation, might be useful for covering dead zones, albeit possibly with lower accuracy.

4.5 Summary

A basic implementation of a SmARtWorld environment has been presented in this chapter. A primary server and a landmark server are needed for smart objects to have an AR presence. Movable objects need to be tracked by an object tracker. For universally-accessible environment, the primary server should use an Ethernet LAN connection and the landmark objects should be based on planar patterns. Viewing devices can be created independently to be used in any SmARtWorld environment. The next chapter focuses on the implementations of different kinds of viewing devices and ways to interact with a SmARtWorld environment.

Compared to other UAR frameworks, such as Studierstube (Schmalstieg, et al., 2002) and ARCS (Chouiten, et al., 2011), where the components are programmed and linked to each other offline, a SmARtWorld environment is built up while in operation by adding landmark objects that are tracked by viewing devices and other smart objects that provide functionality to users for any application. The SmARtWorld environment begins working once the primary server is turned on. Further functionality is added via the smart object discovery procedure described in Section 3.4.2. Programming UAR applications can thus be done quickly and easily without having to write any programming code if suitable smart objects are available.

Chapter 5. User Interaction and Display Devices

5.1 Wearable System

Wearable computers have begun to gain traction in recent times and are the future of AR. While the use of a smartphone requires the user to hold and interact with the mobile device, a lightweight wearable system using a HMD that augments the full view of the wearer allows him/her to walk around unencumbered and hands-free while being able to view the UAR environment. Unfortunately, a lightweight and low-cost wearable system is unavailable thus far, though a number of companies are attempting to achieve this.

A wearable system has been implemented in this research for use with SmARtWorld environments. The program runs on the Windows platform and comprises a laptop which outputs video to a HMD with a camera mounted on it (Figure 5-1).

Figure 5-1. A wearable system.

The camera tracks its transformation by recognizing planar features in the environment and the AR view is achieved by rendering the virtual user interfaces over the camera feed as the background. Interaction is achieved through bare-hand interaction and gesture recognition. The program execution of the wearable system is given in Figure 5-2.

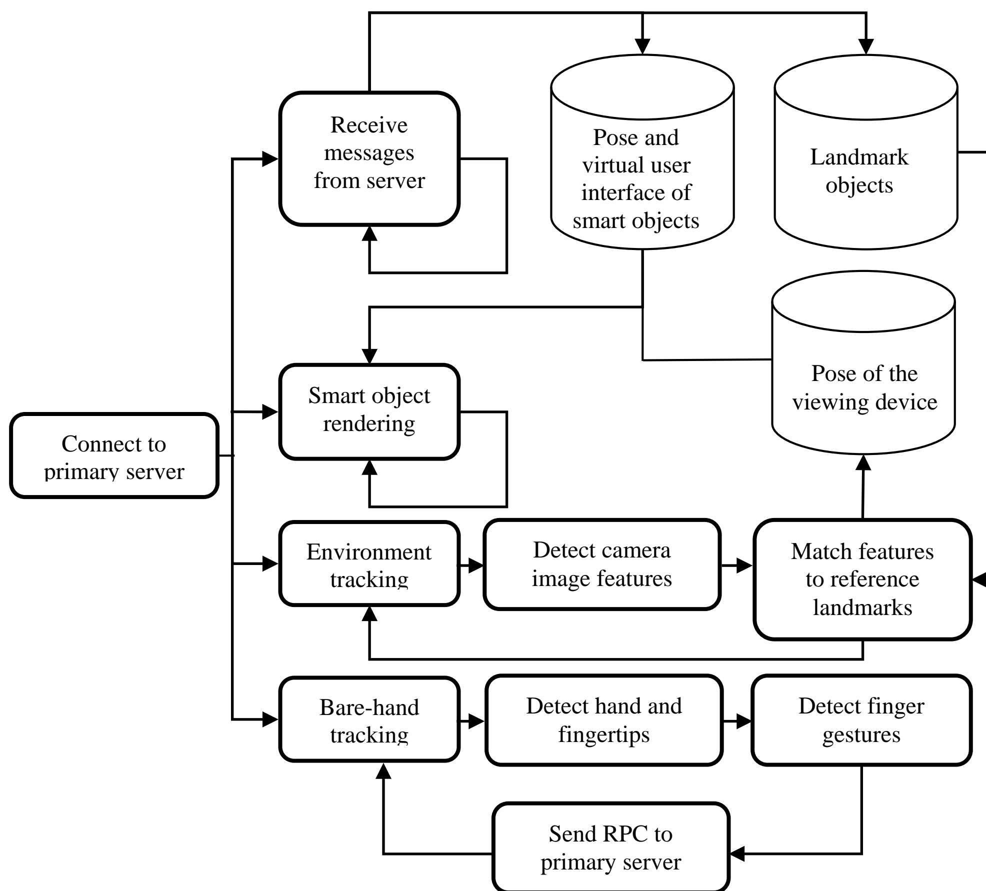


Figure 5-2. Flowchart of the wearable system program execution

5.1.1 Pose Tracking

The prototype wearable system relies on planar feature tracking to track its pose from the landmark objects in the SmARtWorld environment. When the viewing device first enters the environment, it detects the existence of the

landmark objects therein and requests for relevant data about them, such as their image file, dimensions and pose. As the landmark object data is received, the image files are trained to allow the system to track them. Training involves detecting the features in the image and computing descriptors for the features so that they can be recognized and matched to the features from incoming camera frames (Szeliski, 2011). The OpenCV library is used to provide the CV functionality needed in the system, including feature detection, descriptor computation and feature matching. ORB descriptors (Rublee, et al., 2011) are used as the feature descriptor type as it provides a good balance between computational efficiency and recognition performance. The planar feature tracking process is summarized as follows:

1. Detect features from the scene captured by the camera and compute descriptors for these scene features.
2. Match scene features to the trained features for each landmark.
3. Decide on a correct match and use the matched features to estimate a homography using a four-point homography estimation algorithm with RANSAC (Vincent & Laganiere, 2001).
4. Decompose the homography into the rotation and translation components and use these as the transformation of the landmark from the camera (Malis & Vargas, 2007).

A few issues related to tracking many landmarks in a wide area have to be addressed. Firstly, scale invariance is not guaranteed especially if a landmark could be seen from afar or at close proximity. This would be true for any

planar feature tracking algorithm. One approach to mitigate this problem lies on the training of landmarks. It is possible to train and map multiple reference images of the same landmark taken from varying distances as different landmark objects. A landmark captured further away can have more of the surrounding included in the reference image which will provide more features for the tracking algorithm to detect.

Another issue is the scaling of computational time as the number of landmarks grows. Three measures to improve the responsiveness of the tracking are used in the wearable system. First, an MRU (most recently used) list of the recently detected landmarks is maintained by the system. Scene features are always tested with the landmarks in the MRU list first. When looking at the same scene, the MRU effectively results in only one landmark being tracked every frame. If the landmarks in the MRU do not match the scene features, the remaining landmarks are tested. Therefore, the second measure is to pick intelligently the sequence of the remaining landmarks to be tested. The landmarks can be arranged by proximity to each other so that landmarks nearest to the recently matched landmarks are tested first. External sensor data, such as GPS and orientation sensors, can be used to make better decisions on the sequence of landmarks to be tested. Even with intelligent landmark sequencing, there may still be too many landmarks to be tested in a single frame and this reduces the screen responsiveness. Hence, the third measure caps the number of landmarks tested per frame to a maximum number. Tracking is performed in the same thread as graphics rendering since

rendering depends on the tracking result. The choice of this number is thus a balance between screen responsiveness and tracking accuracy.

The rendering framerate of the wearable prototype is about 12 frames per second, with the main bottleneck being the landmark tracking algorithm. Notwithstanding the measures to speed up landmark matching to a large database of landmarks, the computational time for each tracking cycle accounts for about 80 – 90 ms of processing time. The tracking performance can be improved by making use of the graphics processing unit (GPU) of the system's video card (Sinha, et al., 2011).

5.1.2 Rendering Virtual User Interfaces

Rendering is performed using the programmable pipeline of the rendering engine of the OpenGL library. The programmable pipeline is preferred over the traditional fixed pipeline for improved rendering performance as well as to allow for the implementation of certain effects that make the rendering of virtual graphics over a real scene clearer to the user. In the programmable pipeline, custom programs called shaders are loaded into graphics rendering hardware during the program's runtime and used to render 3D objects. Different shaders can be used to render different objects.

Before rendering the virtual user interfaces of smart objects, the camera image is rendered as a texture on a plane facing the OpenGL camera. Next, smart objects are rendered in front of the plane so that the texture forms the background.

The virtual user interface of a smart object is described as a 3D scene in the COLLADA format. From the COLLADA file and any accompanying files like texture images, the vertices, faces, normals, material properties, and texture data are extracted and saved in memory. The 3D scene consists of a collection of 3D elements in different poses with respect to the coordinate system of the smart object. Ordinary 3D elements in the scene use a typical shader for rendering, i.e., the object is rendered with colors, alpha transparency, light-shading and textures.

Some elements have a property of "real" (instead of "virtual") in their COLLADA definition, i.e., the geometry represents a physical part of a smart object. If an element has a "real" property, it indicates that the geometry should not be rendered but should still block virtual elements that are rendered behind it. This is achieved by setting the blending function used by OpenGL (*glBlendFunc function*, 2012). The blending function decides the way the pixels that are to be drawn (source pixels) interact with pixels which are already drawn (destination pixels). Its formula is as follows:

$$R = R_s s_R + R_d d_R$$

$$G = G_s s_G + G_d d_G$$

$$B = B_s s_B + B_d d_B$$

$$A = A_s s_A + A_d d_A$$

R, G, B, A denote the red, green, blue and alpha color components of the final pixel value, (R_s, G_s, B_s, A_s) and (R_d, G_d, B_d, A_d) are the source and destination color components respectively, and (s_R, s_G, s_B, s_A) and (d_R, d_R, d_B, d_A) are the scale factors for each color component.

When drawing smart objects, the destination pixels would contain the camera image texture and possibly other smart objects. When drawing a real element of a smart object, the scale factors for the RGB components of the source color are set to zero and that for the destination color are set to one. For the A components, the scale factor for the source alpha is set to one and that for the destination alpha is set to zero. The resulting RGBA values are as follows:

$$R = R_d$$

$$G = G_d$$

$$B = B_d$$

$$A = A_s$$

The RGB components of the final pixel remain the same as what it was before the smart object is drawn while the alpha component takes the source value. If the source alpha value is one (fully opaque), subsequent objects drawn behind it will be hidden. Therefore, if the virtual user interface of a smart object is superimposed on a physical object, the overall effect is that the elements that have the “real” property are not drawn but still block virtual elements behind it (Figure 5-3).

Textured fonts are used to generate and render dynamic text for virtual user interface elements which are given the text tag. In this approach, an image containing a collection of ASCII characters at known positions and with known dimensions is used as a reference to generate a texture of the required text to be rendered during run-time (Figure 5-4).

Figure 5-3. Occlusion of virtual objects by real objects.

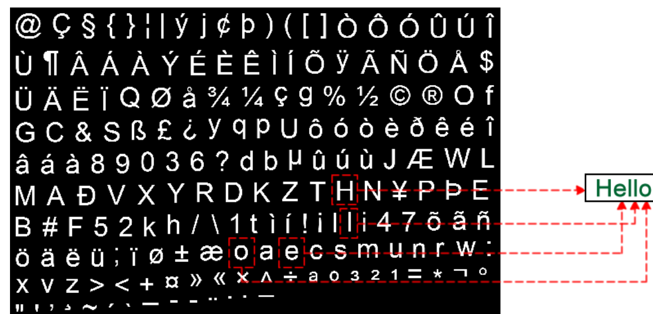


Figure 5-4. Texture-based font rendering

A font texture can be represented as a signed distance field (Green, 2007) where each pixel is a number indicating its distance away from a white pixel which is inside a character (Figure 5-5), rather than a bitmap of color intensities. This representation maintains a high quality of the font at any amount of zoom and supports computationally fast shader effects, such as anti-

aliasing, outlines, glow, and drop shadows (Green, 2007). In an AR application, the background colors are unpredictable, such that these effects are valuable for making text easier to read. Furthermore, text registered to 3D objects in the real scene can be at various distances from the camera, and hence maintaining the quality at different zoom levels is important.

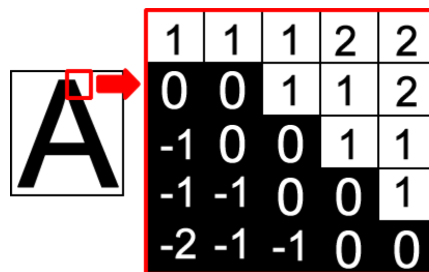


Figure 5-5. Signed distance field representation of fonts.

The “effect” and “effect color” properties of a text tag of a virtual user interface node indicate the effect to use for text rendering and the color to apply to the effect. Two effects are supported in the wearable system, namely, outline and glow (Figure 5-6). These two effects are implemented in the shaders that are used for text rendering and they make use of the signed distance field font representation to perform the calculations.

Zoomed out Zoomed in

Figure 5-6. Zoom-invariant font quality and font effects.

If the number of smart objects in the environment is large, and models are detailed and have high polygon counts, rendering performance may start to suffer. One optimization that can be done to mitigate this issue is to make use of a view-dependent level-of-detail algorithm (Hu, et al., 2010) to adjust the number of vertices to use for rendering smart objects.

5.1.3 Bare-Hand Interaction

The wearable system is designed to allow users to interact with the virtual user interfaces of smart objects using their bare hands. To detect the hand, the colors of the pixels are analyzed to determine skin-colored pixels. A black and white image is produced where pixels determined to be skin-colored are set to white and all other colors black. Next, contour analysis is performed on the black and white image to determine the most probable hand region based on its size and shape. The contour of this region is further analyzed to detect individual fingers.

Fingers are detected by obtaining the convex hull of the hand contour and finding “convexity defects”, which are the regions within the convex hull that are not within the hand contour. Only convexity defects of a minimum depth indicate the presence of a finger (Figure 5-7a). Individual fingers are classified as either a “pointer finger”, of which there can only be one, or an “additional finger”. If there is only one convexity defect of sufficient depth, there is only one finger which is classified automatically as a pointer.

It is important to obtain the screen coordinates of each fingertip for the later step of gesture detection. The tip of the pointer finger is defined as the farthest contour point from the centroid of the contour (Figure 5-7b). The tip of an additional finger is the second farthest contour point, and that of a second additional finger is the third farthest contour point, etc.

The possible gestures that can be used to interact with the virtual user interface elements in the wearable system are click, drag, and rotate. The click gesture is reminiscent of using the pointer finger to press a switch or button. The drag gesture resembles the pointer finger holding down a button and dragging it in one direction. The rotate gesture involves the pointer finger and another finger, most appropriately the thumb, rotating as if they are turning a knob. These are gestures that most users would be familiar with from their experience with using smartphones.

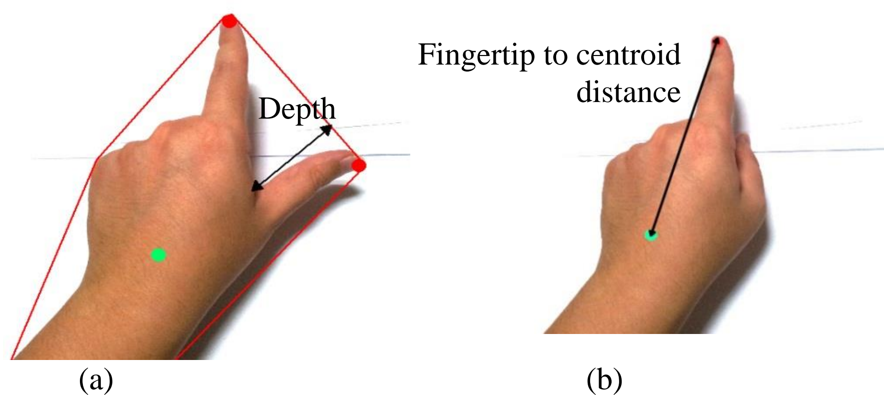


Figure 5-7. (a) Depth of a convexity defect indicates presence of fingers, (b) fingertip is the farthest point from the centroid of the hand.

The click gesture consists of four stages (top row of Figure 5-8). The first stage is the initialization stage, wherein the hand must remain in the same

position and pose for a fixed duration, e.g., one second. The blue dot depicts the position of the fingertip and the green dot the position of the centroid of the hand contour. The orange circles around the blue and green dots indicate the regions that these two points must remain within during the initialization stage. If the initialization is successful, the circles turn green. Otherwise, the initialization is restarted.

The second stage occurs after the initialization stage. The user can start performing a click action at any time during this stage. The system continues monitoring the fingertip and the center of mass during this stage. If the hand centroid moves out of the circle, the gesture is cancelled and the initialization stage is activated again.

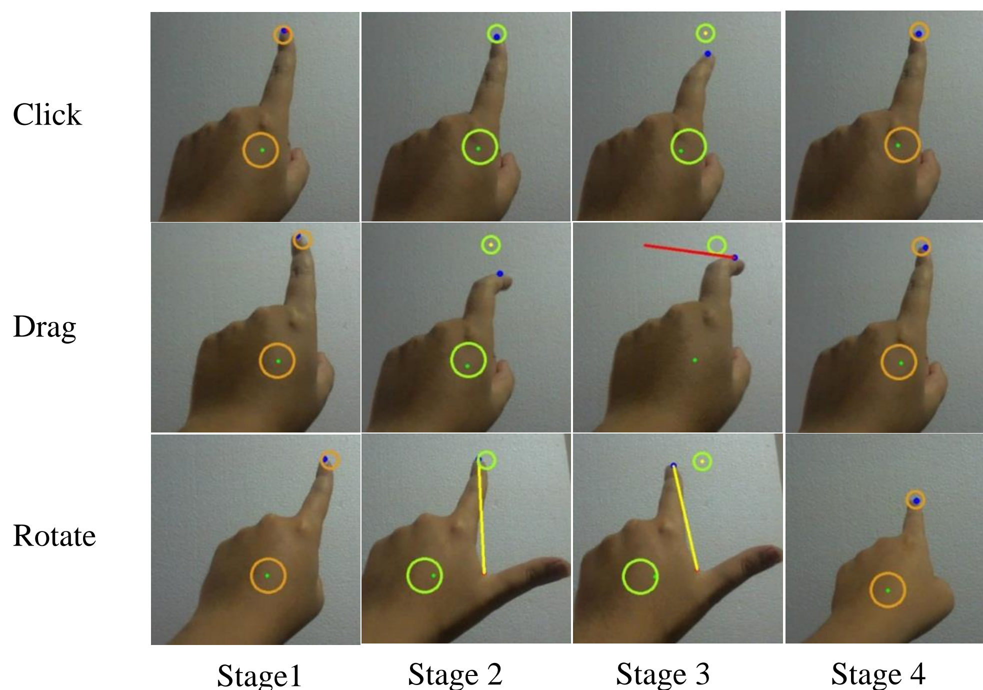


Figure 5-8. The detection stages of different gestures.

If the user bends their finger such that the fingertip goes out of the green circle which is around the fingertip and the distance between the fingertip and the centroid decreases, the system interprets this as a finger pressing action and the third stage is activated. A yellow dot indicates the original position of the fingertip before the pressing action started. During this third stage, as long as the centroid stays within the green circle, the click gesture is active.

If the click gesture is still active and the fingertip returns inside the green circle, the click gesture is completed. This is indicated by the yellow dot turning red momentarily before the initialization stage restarts. On completion of the click gesture, the screen coordinates of the yellow/red dot, i.e., the screen coordinates of the fingertip when the click gesture first started, is obtained and used to determine the object or element that has been clicked on.

The stages of the drag gesture are shown in the middle row of Figure 5-8. The first stage is the same initialization stage as for the click gesture. After initialization, when the user bends his finger downwards, this is still part of a click gesture. The click gesture turns into a drag gesture in the third stage, when the user moves his whole hand until the centroid leaves the green circle, which marks the region within which the centroid was originally located when the click gesture started. At this point, the click gesture is turned into a drag gesture. The drag distance is indicated by a red line joining the starting point of the drag gesture and the current position of the fingertip. During this stage, the distance dragged is continuously monitored and used as interaction input. The small green circle near the fingertip shows where the finger must return to

in order to complete the drag gesture (the finger release stage). The green circle translates by following the translation of the centroid. After the finger release, the drag gesture ends and the initialization stage is restarted.

The rotate gesture (bottom row of Figure 5-8) starts with the same initialization stage as in the click and drag gestures. The appearance of a second finger triggers the start of a rotate gesture. A yellow line is displayed to illustrate the rotation action as the user performs it. As the yellow line rotates from its previous orientation, the rotation angle is monitored and used as interaction input. The rotate gesture ends when the second finger disappears, i.e., the user retracts his finger back into the palm.

The three gestures supported by the wearable system correspond to the touch, drag, and rotate which can be defined in the virtual user interface of a smart object. Figure 5-9 shows a user interacting with different elements of a virtual user interface using the three gestures.

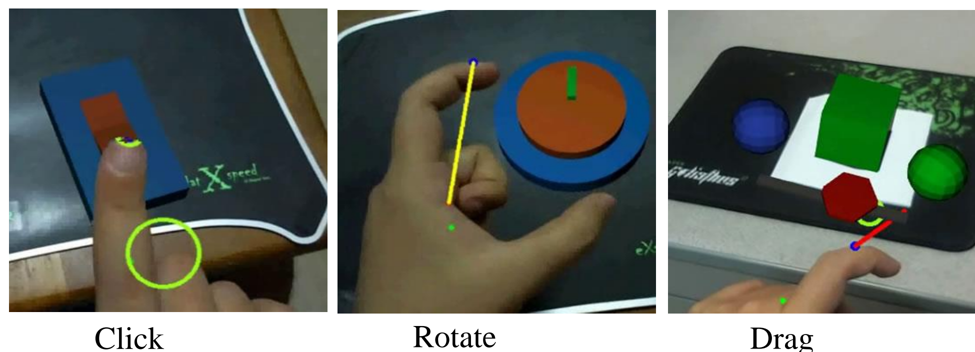


Figure 5-9. Bare-hand interaction with virtual user interface elements.

5.1.4 Occlusion of Virtual Elements by the Hand

As the AR view is achieved by overlaying virtual elements on the camera feed, every virtual element will appear on top of everything in the physical environment including the user's hands. However, to maintain a perception of depth and thus the immersiveness of the SmARtWorld environment, the user's hand should occlude virtual elements (Figure 5-10). This also makes it easier to interact with virtual elements because the user is able to see where they are pointing. This of course assumes that the hand is always nearer to the user than the virtual elements. However, this is a reasonable assumption because of the unlikelihood of a virtual element being between the user's face and hand.

Figure 5-10. Occlusion of virtual objects by the user's hand.

Making the user's hand appear to occlude virtual elements involves manipulating the graphics rendering pipeline. After the camera feed is drawn as the background of the screen, a depth mask is created from a black and white image of the user's hand in order to set the depth value of the pixels

belonging to the hand to a minimum value (these pixels of minimum depth represent objects that are nearest to the virtual camera of the graphics rendering engine). When the virtual elements are drawn, the graphics rendering engine performs a depth test, which means that any virtual element that occupies the same pixels as the hand pixels will fail the depth test and would not be drawn and displayed.

5.2 Tablet and Smartphone

Wearable systems are still too costly and uncomfortable to be used for long periods of time. The most viable device for use as a viewing device is therefore still the tablet or smartphone. Any average smartphone by current standards is sufficiently equipped to serve as a viewing device as long as it has an embedded camera.

In this research, an Android app was created to demonstrate the use of a smartphone as a viewing device and tested on a Samsung Galaxy S2 device. The Android viewing device app is largely a straight port of the wearable system. The program execution flowchart is given in Figure 5-11.

Tracking performance is worse than the wearable system, due to less powerful processing power in the mobile device. Apart from implementing GPU-based tracking, optimizations specific to mobile device hardware architecture can be done (Yang & Cheng, 2012). Some of the fundamental layer functions such as TCP and UDP socket communication were re-written to make use of functions

in the Android API. The main difference from the wearable system is in the implementation of the interaction method.

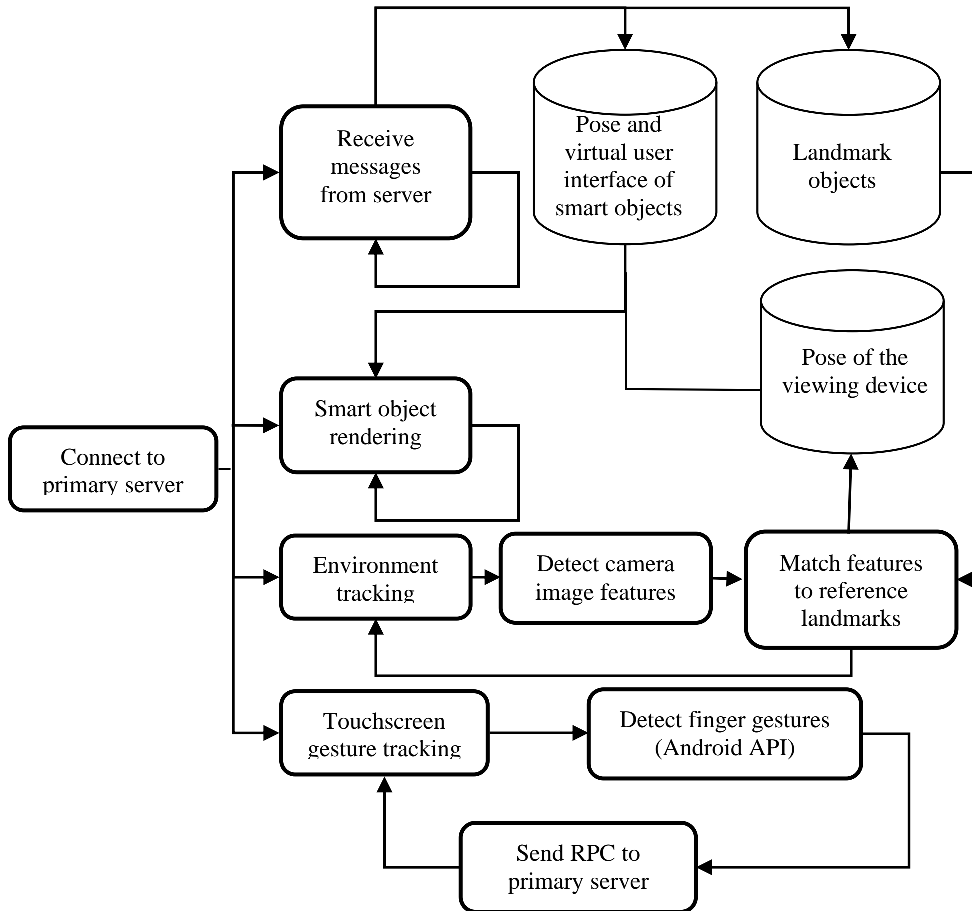
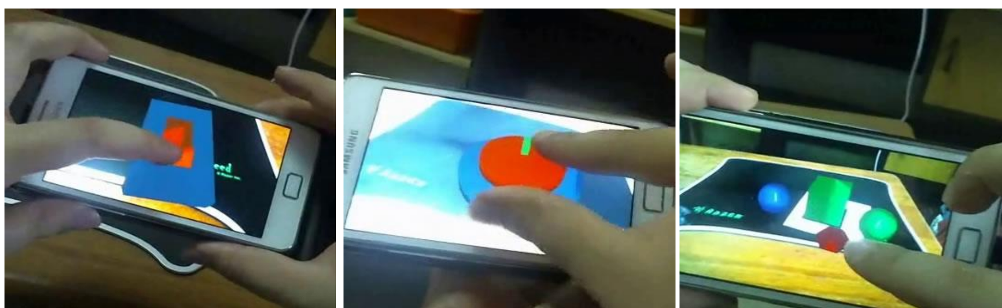


Figure 5-11. Flowchart of the Android system program execution

Interaction with virtual user interface elements is achieved through the touch-screen. The selection of 3D objects on the touch-screen is similar to using a mouse for picking 3D objects on a monitor, for which there are well-established methods like ray-casting and color picking. The user selects a user interface element by placing their finger down on it on the touch-screen. A touch operation is then performed by lifting the finger up from the screen. To perform a drag or rotate gesture, the finger is not lifted up immediately. A drag

gesture is performed on the interface element, if it accepts such input, by swiping the finger along the screen. A rotation gesture is performed by placing another finger down on the screen and moving one finger around the other. Interactions with each element will result in the corresponding RPC being sent to the smart object. Figure 5-12 depicts the three operations performed through the smartphone app on three different interface elements.



Click Rotate Drag

Figure 5-12. Touch-screen interaction with virtual user interface elements.

5.3 Device-less Interaction

Interaction with smart objects can also be achieved using the SmARtWorld framework without the use of any viewing device. This approach makes use of devices in the environment such as buttons, keyboards, touch-sensors, and voice recognition and context recognition sensors. Some of the input devices and sensors that will be described in this section are built using the SNAP system (*Synapse's SNAP*, n.d.) to form a wireless sensor network (WSN).

5.3.1 Sensors and Input Devices on a Wireless Sensor Network

A SNAP node is a low-cost and portable way to create a smart object with sensors and it is easily connected wirelessly to other nodes. A hub is needed to link the smart objects on the WSN to the other smart objects in the environment. A SNAP node is an RF transceiver with a microcontroller that runs Python code. SNAP nodes automatically form a mesh network when they are within range of each other, and they use the XML-RPC protocol (Winer, 2003) to communicate.

The SNAP hub program runs on a PC which has a SNAP node connected by USB. The hub program sees this node as an XML-RPC server and connects to it. Other SNAP nodes which join the SNAP network see the hub as another normal SNAP node, which means a hub discovery procedure is needed. SNAP does have broadcast capability which means the hub discovery procedure is straightforward. Like any other smart object, a new SNAP node broadcasts a "New" command to all nodes, and the information of existing smart objects are sent to the new node from the hub.

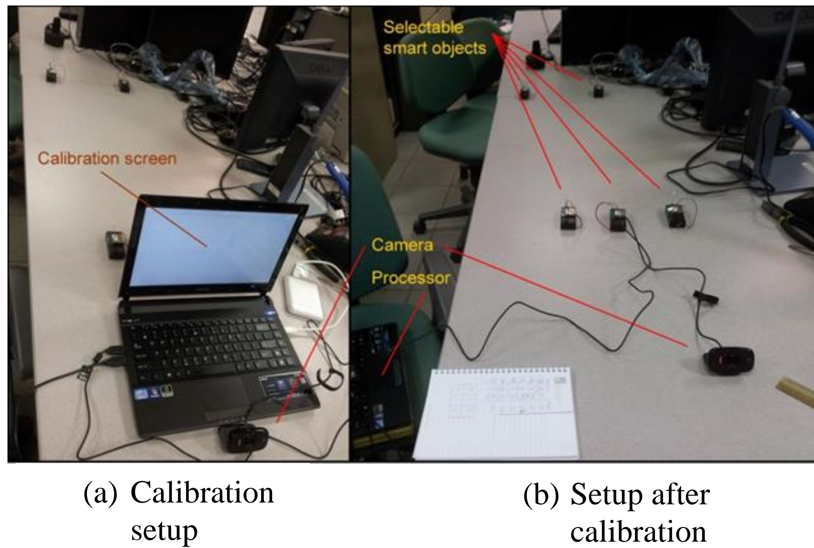
One of the input devices created in this research is a presence sensor. It is implemented as a smart object based on a SNAP node. The fundamental layer of this object consists of the RF transceiver and SNAP operating system, with an infrared presence sensor connected to its microcontroller. The microcontroller is programmed to read values from the presence sensor and infer if something is in front of the sensor based on the sensor readings. If the

sensor is obstructed, it invokes an RPC. If the object moves away from the sensor, another RPC is invoked.

Other input devices similar to the presence button are a door sensor, implemented using a flex sensor connected to a SNAP node, and a pressure sensor. The flex sensor is placed perpendicular to the hinge of a door so that when the door is opened, the flex sensor resistance changes. The door sensor interprets the flex sensor readings as whether the door is opening or closing and invokes different RPCs for each outcome. The pressure sensor is placed on a chair and invokes an RPC when a person sits on a chair and a different RPC when the person gets up from the chair.

5.3.2 Gaze Tracking

A gaze tracking sensor determines the direction of a person's gaze. One way gaze tracking can be used in a SmARtWorld environment for device-less input is to determine which smart object a user is looking at and invoke an RPC in that object. This is implemented with a camera connected to a PC, with the camera mounted facing where a user would be and smart objects placed behind the camera. The ITU Gaze Tracker library (San Augustin, et al., 2010) is used to process images from the camera and get the user's gaze point. There is a calibration step which involves the user looking at a computer screen while his gaze is tracked and focusing on different points on the screen as indicated by the program (Figure 5-13a).



(a) Calibration setup

(b) Setup after calibration

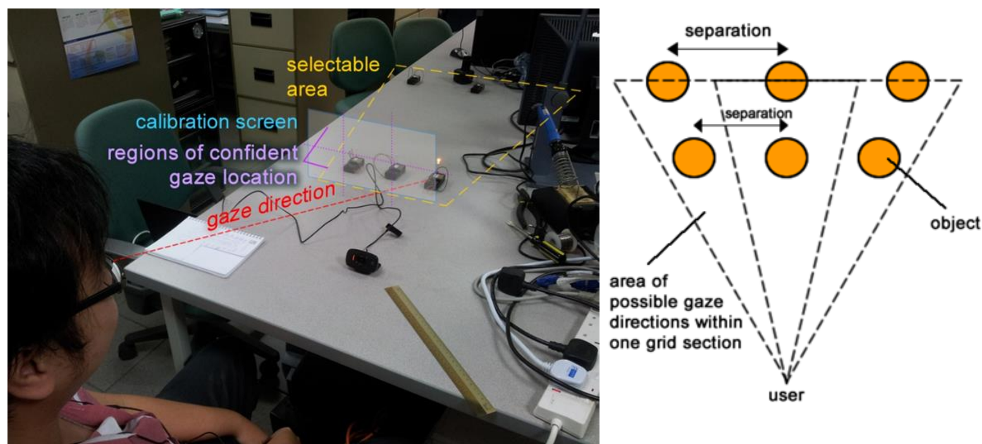
Figure 5-13. Setup for object interaction using gaze tracking.

Testing the gaze tracking accuracy of this system by moving the mouse cursor on the computer screen using eye gaze, it was found that eye gaze tracking was accurate to within each of the sections in a 3 x 2 grid dividing the screen. This means that a person's gaze cannot be determined accurate to a point on the screen but to a region where it is confident that the gaze tracking is within. Another point of note is that the maximum distance between the user and camera is about 45 cm before tracking deteriorated which means that the user must be within that distance to a cluster of smart objects in order to interact with them.

After the calibration and accuracy testing, the computer screen is removed so that only the camera is left (Figure 5-13b). Possible gaze directions are constrained to the 3 x 2 grid. The range of possible gaze directions for each grid section depends on the user's position and computer screen position during calibration (Figure 5-14a). Each range for each grid section thus

indicates that objects within that range are selected if the user looks in that direction. Therefore, in order for a unique object to be interacted with for a certain gaze direction, the objects must be separated according to Figure 5-14b.

With the above setup, the gaze tracking object allows interaction with up to six different smart objects by the user looking at them. This setup is quite limiting as the user cannot move his head or the tracking fails and the smart objects have to be quite well spaced-out. However, more than one gaze tracking objects can be used for different clusters of interactive objects.



(a) Selectable areas depends on the user's position and screen during calibration.

(b) The separation required between objects in different selectable areas.

Figure 5-14. Placement of smart objects for gaze tracker interaction.

5.3.3 Context Recognition

Context recognition refers to the detection of the intentions and activities of a person. This can be done on an abstract level, e.g., detecting activities like doing laundry and washing dishes (Pirsiavash & Ramanan, 2012; Hoque &

Stankovic, 2012), or a lower level with actions like sitting and standing (Kwapisz, et al., 2010).

Logic-based context recognition relies on understanding and reasoning of the observations that indicate with certainty the intention of a person. Wang et al. (2010) used a logic-based approach to infer high-level activities of users based on sets of low-level sensor output which frequently occur for a specific activity and rarely in other activities, e.g., a person is inferred to be brushing his hair as a result of holding a comb in the bathroom and moving it in certain directions while also holding a can of detangling spray. This approach can be implemented with smart objects in a SmARtWorld environment by channeling the outputs of low-level sensors like those described in the previous two sections to a smart object which has been trained to translate different sensor output combinations to specific user intention.

A probabilistic approach commonly applied to both high-level and low-level context recognition is the use of Hidden Markov models (HMMs). In this approach, each user intention is modeled as a single HMM which is trained from the data sequence observed when a person is carrying out the actions with the intention in mind. A trained HMM can then calculate the probability that a person has a particular intention based on newly observed data sequences.

The algorithms that are frequently used for training an HMM and evaluating the probability that a data sequence indicates a particular user intention are the

Baum-Welch algorithm the forward-backward algorithm respectively, and are described in detail by Rabiner (1989). An HMM can be implemented as a smart object which allows itself to be trained via an RPC that invokes the forward-backward algorithm. A trained HMM smart object performs context recognition by observing sensor data and computes the probability that the sensor data is a result of a particular user intention, invoking some behavior in other objects if the probability is above a threshold.

The training of an HMM object is done through the virtual user interfaces of the HMM object and the sensors in the environment (Figure 5-15). First, the user registers several sensors with the HMM object. The user then invokes a “record” RPC on the HMM object and carries out the actions for a certain user intention and invokes the “stop recording” RPC once the series of actions is completed. This can be carried out multiple times to improve the training results. After training is complete, the user registers an RPC to be invoked by the HMM object when it recognizes the user intention with a probability above a certain threshold. Finally, the HMM object is left to monitor the sensor objects registered to it.

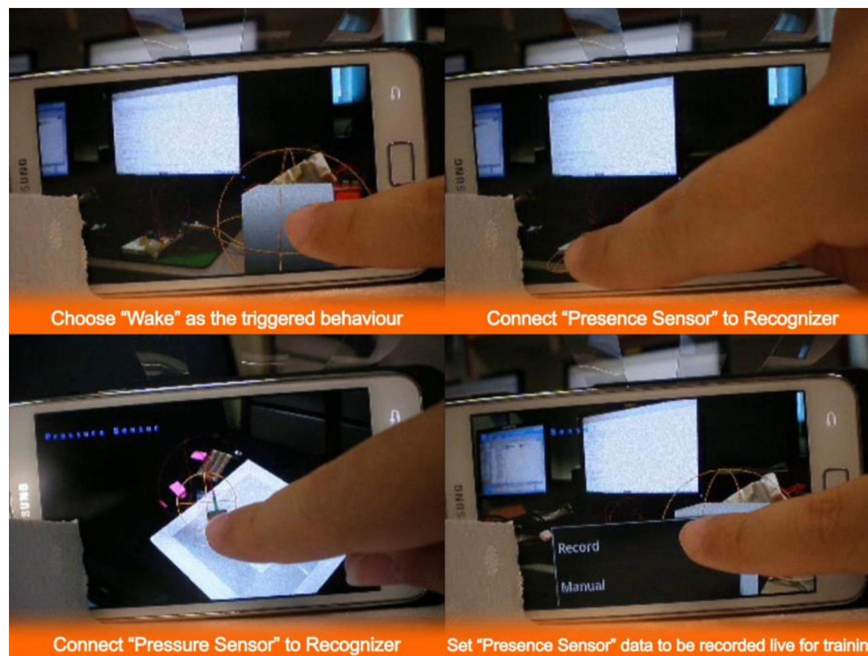


Figure 5-15. Training an HMM-based context recognition object using a smartphone.

5.4 Summary

In this chapter, viewing and interaction devices are described. With component-based UAR frameworks, such as Studierstube and Tinmith evo5 (Piekarski & Thomas, 2003), viewing devices are application-specific and may not work with other applications developed using the same tools. In SmARtWorld, generic viewing devices can be used to access the user interfaces and functions in a UAR environment. All the implementation architectures of these devices are based on that of a smart object, which means that these devices use the same procedures to connect to the network of any UAR environment and discover other smart objects.

The SmARtWorld framework is designed to be used with current devices as well as future devices. Wearable systems may still be too costly and cumbersome, but they are a rapidly evolving technology. Context recognition

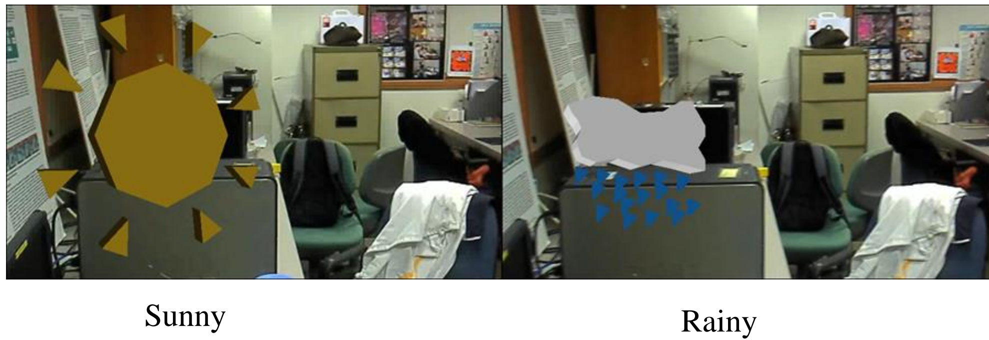
is a possible means of interaction without the use of devices. This provides access for the elderly or disabled; they are also significant for use in conjunction with viewing devices that do not provide a method for interaction with smart objects through direct gestures. By placing different types of sensors in the environment, different modes of context recognition such as symbolic gesture recognition, speech recognition, and high-level activity recognition can augment the interaction of users with smart objects. The separation of RPCs and interaction devices in the framework makes it compatible with almost any interaction method as long as it can be mapped to the RPCs of smart objects. This idea of dynamic creation of new interaction methods in a UAR by users by making use of smart objects, which function as sensors and as data processors, is not seen in other UAR frameworks.

Chapter 6. Smart Object Representation

6.1 Real and Virtual Objects

The SmARtWorld framework allows for both real and virtual objects to co-exist in a UAR environment. Real objects are smart objects with a physical form embedded with their own fundamental layer (i.e., computing and networking hardware). Their behavior is programmed in their own memory and their pose must be tracked to allow viewing devices to augment their virtual user interfaces over their physical location. Real objects can be directly handled and may have physical controls to invoke RPCs.

Virtual objects do not have physical form and are perceived only by their virtual user interfaces. Their hardware is not at the same location as their pose. They can come into existence in the environment through smart object hosts, e.g., landmarks which come from landmark servers, or exist as independent running programs like the basic smart object in Section 4.1. Virtual objects can be very useful in bringing abstract functionality into an environment, such as the HMM objects for context recognition as described in Section 5.3. As an example, a virtual weather sensor can be created as a smart object that retrieves the weather conditions at a particular geographic location from a cloud weather service (Figure 6-1). Users in the SmARtWorld environment would be able to make use of this object to make travel plans.



Sunny Rainy

Figure 6-1. A virtual weather sensor object.

6.2 Realistic Rendering

The appearance of an object is defined by the 3D model data that defines its virtual user interface. The vertices of the 3D model define the shape of the object while the surface normals and material properties, namely, diffuse, ambient and specular color, and shininess which determines the size of the specular highlights, determine the color and shading of the object. Texture mapping affects the color of specific points along the surface of the object, while normal, displacement, and bump maps give the appearance of geometry that is not shaped by the vertices of the object model. These data parameters affect the appearance of the material of the object. External parameters which change the appearance of the object come from environmental factors, such as ambient light, light sources, and reflection of the environment on the object surface by the use of reflection maps.

Texture mapping can be achieved easily. The wearable system described in Section 5.1 uses the programmable pipeline of OpenGL to render the AR scene, in which texture mapping involves copying the texture to video memory and calling the texture mapping functions in the shaders. To realize

effects, such as shadows and environmental reflection requires more effort. There are generally two approaches. The direct approach is to track light sources and real objects to calculate object lighting and shadows (Haller, et al., 2003), and extract an environmental map for reflecting the real environment on virtual objects (Ropinski, et al., 2004). The second approach is to use indirect methods, such as image-based lighting and shadowing of virtual objects which simulate global illumination and shadows based on real-time images of the environment (Supan, et al., 2006).

Using the direct approach, viewing devices must implement the appropriate algorithms for shadow and environmental reflection rendering, and there must be support from the SmARtWorld environment. Figure 6-2 shows a virtual object casting shadows in a SmARtWorld environment using a shadow mapping technique implemented on a viewing device. This technique involves two passes. First, the scene is rendered from the perspective of the light source. Next, the proper scene is rendered from the perspective of the virtual camera and the result of the first pass is used to add shadows into the scene. In an AR environment, the pose and properties of light sources must be known. In Figure 6-2, the light source is a smart object which provides data about its color and intensity. The viewing device uses this data with the pose of light sources to perform the first pass.

For environmental mapping on reflective smart objects, viewing devices can use images of the environment as textures to wrap around the smart object (Figure 6-3). If an environment is assumed to be a box, the environmental map

would consist of six images, i.e., one for each face of the box. Alternatively, in the absence of a true environmental map, a viewing device can use its own camera image as an environmental map. This will, however, give a less accurate reflection of the environment.

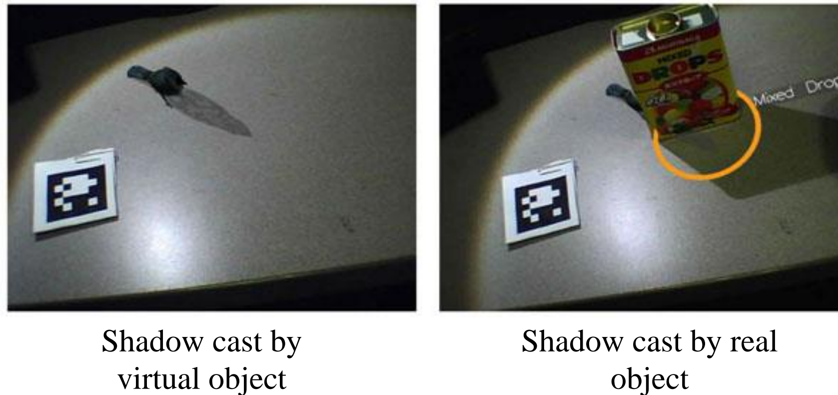


Figure 6-2. Shadows cast by virtual objects due to real light sources in the environment.

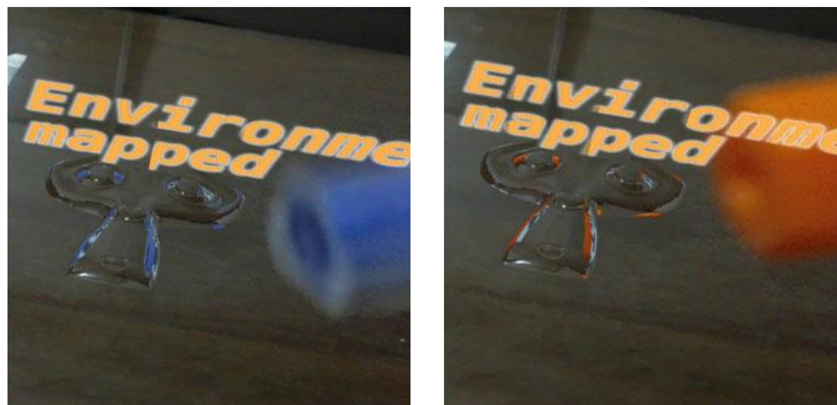


Figure 6-3. A virtual object reflecting the real environment.

6.3 Physical Simulation

A physics engine object can be added to a SmARtWorld environment to add realism to the physical interactions between smart objects. The physics engine

object monitors the physical interactions of smart objects with each other in real-time and updates their states when interaction occurs.

In this research, a physics engine object was created using the Open Dynamics Engine (ODE) library (Smith, 2007) to add collision detection between objects in a SmARtWorld environment. Collision detection requires the geometry of smart objects to be modeled using primitive geometries defined by the physics engine. This "ODE model" of the smart object is made available as part of the data of the smart object. Therefore, when a new smart object joins the environment and has an ODE model, it can be added automatically to the physics simulation through the physics engine object. Apart from the ODE model, the smart object must have its mass and center of gravity defined.

At every time step of the collision detection simulation, physics simulation is performed. For collision detection, the ODE library returns information about the contact points between colliding objects, penetration depth of one object into another, and the normal vector of the penetration. For the physics simulation, forces on each object can be obtained. The physics engine object calculates the velocities of each virtual object involved in the simulation and makes it move by invoking its "Set Your Pose" RPC.

Real objects will not be affected by collisions with virtual objects unless they have actuators to move them. To achieve accurate physics simulation in such cases, real objects can be set as kinematic bodies in ODE which will effectively make them impervious to any forces applied on them. The physics

engine object determines whether an object is real or virtual by inspecting its virtual user interface to determine if there are any elements with the “real” property applied on them.

6.4 Sound Response

Realistic and positional sounds that are made by objects in response to interactions by users and other objects would be very desirable in UAR environments. First, it can bring to attention to smart objects that are not in the user’s field of view. Secondly, it can create a more immersive and realistic atmosphere of a mixture of real and virtual objects. Having positional and realistic sounds may allow a visually impaired user to locate and understand the functionality of smart objects without seeing them. For the rendering of sound in a SmARtWorld environment, two kinds of objects are needed, namely, a sound source and a sound renderer.

6.4.1 Sound Source

The sound source object streams digital sound data to a sound renderer to be played. Uncompressed sound data is digitally represented by sampling the sound wave at regular intervals and getting discrete values of the wave. The sound is re-created by the sound renderer object from digital form by demodulating the digital signal and shifting the output signal to speakers.

Smart objects can be packaged with recorded sound files by their developers to output sounds intentionally with certain events. In this approach, the triggering of a sound is controlled in the fundamental layer and is programmed

by the developer. When a sound is triggered, the sound file is streamed via an RPC in the sound renderer object. Streaming is done by sending chunks of the sound data to the sound renderer, which concatenates the chunks into a long sound wave. The size of each chunk is decided by the smart object that sends the sound.

Smart objects without packaged sounds can still make sounds in the environment. This can happen if an environment consists of objects for applications which do not have sounds but the environment owner would still like them to stimulate the aural senses. Two ways to achieve this are to define the object's material and shape class which it provides as data so that a suitable sound can be selected from a library, and to model the sonal response of an object accurately such that tapping the object at different locations and with different force can result in different sounds. For the first approach, an intermediary smart object would be trained to detect certain events (e.g., from a context recognition object or collisions detected by a physics engine object) and a sound based on the materials and shapes of the colliding objects is generated by it. In the second approach, there is a body of research work dedicated to the real-time synthesis of realistic sound made by virtual objects due to contact (Liu, et al., 2011; Chadwick, et al., 2012).

In physics-based sound modeling, object sounds which are to be simulated are treated as vibrating objects, and a commonly used model for this is the exciter/resonator model (Avanzini, et al., 2003). Each object has one or more resonators, i.e., vibration modes which are to be modelled mathematically.

The exciter transfers energy to the object that results in a vibration response. For example, a guitar string is an exciter while the body of the guitar is a resonator. The basic parameters which could affect the character of a sound are pitch, loudness and timbre. Pitch and loudness depend on the frequency and amplitude of the wave respectively, while timbre, which is described as the perceptual difference in two sounds which have the same pitch and loudness, is attributed to the waveform of the sound.

A simple model used for physics-based sound synthesis is a spring-mass vibration model represented as:

$$x = e^{-\alpha t} \cos wt$$

where α is the stiffness and w the natural angular frequency.

To apply physics-based sound synthesis to objects in a SmARtWorld environment, smart objects need to provide the vibration model that is used to represent their sonal response when they hit each other. As a proof-of-concept, two virtual objects are placed in a SmARtWorld environment, along with a sound renderer and physics engine object. The physics engine detects when there is a collision and triggers the sound response in each object. The objects then stream the digitized sound data to the sound renderer. Both objects use the simple spring-mass vibration model shown above but with different parameters. One object has a low stiffness value and high natural frequency, resulting in a metallic "ting" sound when hit, and the other has a high stiffness value and low natural frequency, resulting in a short tap sound. The waveforms of the two sounds are shown in Figure 6-4.

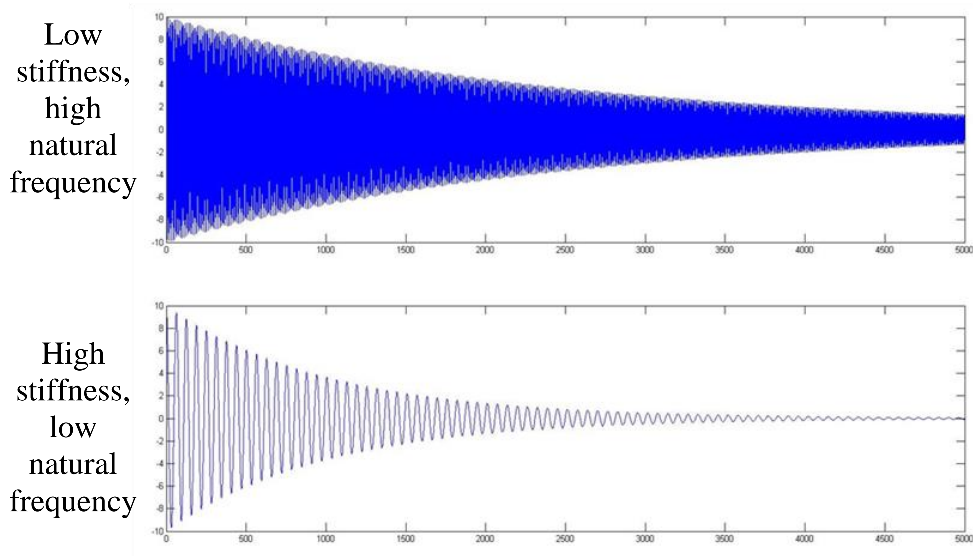


Figure 6-4. Sound waves generated by two smart objects with different stiffness and natural frequency.

6.4.2 Sound Renderer

In a UAR environment, positional sounds come from objects in a 3D space. Positional sound can be achieved using sound rendering software libraries such as OpenAL (OpenAL Soft, n.d.). The OpenAL SDK defines three entities, namely the listener, sound source, and buffer. The sound renderer adapts to different speaker configurations to generate positional sounds based on the poses of the listener and sound sources. A buffer holds the actual sound data and is associated with a sound source. Sound sources can be data streams with continuous incoming data.

To achieve positional sound in a SmARtWorld environment, a sound rendering object is connected to a surround sound speaker system. The sound rendering object uses OpenAL to play positional sounds with other smart

objects as sound sources which stream sound data. The 3D poses of smart objects are retrieved by the sound rendering object for positional sound generation, but in a multi-user environment, the positions of users must be assumed at one fixed location. Alternatively, sound rendering functionality can be embedded with viewing devices in which case every user will hear positional sounds from smart objects with respect to the user's true position.

6.5 Summary

In this chapter, the different ways in which a smart object can be represented in a SmARtWorld environment have been described. Environment developers can choose to support more advanced rendering effects by placing smart objects, such as environmental map sources or sound renderers into the environment. In the absence of these objects, the smart objects would be represented by the basic features where at least the 3D geometries and colors of virtual user interfaces can be seen and interacted with. With physics-based simulation and sound synthesis, even greater realism and blending between real and virtual objects can be achieved, regardless of the intended application of the smart objects.

The methods for achieving realistic rendering of virtual objects that have been used in this chapter are based on published AR rendering and computer graphics algorithms, but implemented using distributed smart objects under SmARtWorld. These rendering algorithms would be implemented on viewing devices but rely on data that is available from various smart objects in a UAR environment. For shadow mapping, a viewing device would have to discover

light source objects in the UAR environment while for environment mapping, the viewing device could obtain the required data from an environment map object that maintains a 360º image of the room. This reinforces the goal of abstracting applications from lower-level tasks.

Another idea that is introduced in this chapter is the interaction between real and virtual objects, particularly what happens when they collide with each other. The implementation of a physics engine and sound rendering engine as smart objects is given, along with the data that smart object developers have to define, allows their physical and aural properties to be derived by other objects. These objects can be used in any UAR environment in conjunction with any other smart objects, as long as data describing their physical properties are included. The generalization of this concept of mixing real and virtual objects is not demonstrated in other UAR frameworks, i.e., UAR developers would have to program such behavior manually.

Chapter 7. Manufacturing Applications

Modern manufacturing today is characterized by temporary virtual enterprises comprising different manufacturing companies and the sharing of their resources. Monolithic factories built for very specific products have given way to smaller and more flexible facilities. Modern factory architecture, commonly known as "smart factories", employ UbiComp technology to embed machines and sensors with intelligence and connect them to a network so that workers can retrieve real-time production information through mobile devices that they carry or computers on the facility (Hessman, 2013). AR has commonly been applied in the manufacturing sector to enhance specific human-centric activities like factory layout planning (Pentenrieder, et al., 2007), product design (Lee & Park, 2005; Ng, et al., 2013), assembly (Caudell & Mizell, 1992; Hou, et al., 2013) and CNC machining (Olwal, et al., 2008).

In this chapter, the application of distributed computing and virtual user interfaces of smart objects in manufacturing activities is presented. The first section presents the application of SmARtWorld to a small job shop. The second section presents the integration of smart objects in SmARtWorld environments around the world through a manufacturing grid (MGrid). The last section briefly discusses visual programming in a SmARtWorld environment which is applicable to manufacturing as well as numerous other disciplines.

7.1 Manufacturing Job Shop

7.1.1 Smart CAD Object

The understanding of part designs and the use of computer-aided design (CAD) files is an integral part of a job shop. Therefore, a smart CAD object has a virtual user interface that allows for interactive features which help aid in the understanding of features and dimensions of a design. It also encapsulates CAD data of a design that other computer-aided technologies will use.

To demonstrate the creation of a smart object from a SolidWorks CAD model, a tool to extract features from a part document and generate a smart CAD object based on this part document is described here. The smart CAD object creation tool creates and acts as a hub to smart CAD objects in the AR environment. The CAD feature extraction functionality of the tool is based on the SolidWorks API which is used to extract data from the active document of a running instance of SolidWorks. The tool extracts the features, including its geometry and annotations that are attached to the part. Next, the user can associate annotations with features that will be useful for the smart CAD object (Figure 7-1). When the extracted data is saved and stored as a smart CAD object, a COLLADA file for the AR user interface of the object is generated automatically. The COLLADA file includes the geometry of the features, i.e., the vertices and normal vector of each vertex for rendering the shaded model in the AR environment, as well as 3D models of the annotations which are generated procedurally from the properties of the annotation elements, e.g., lines and arrows, that are extracted through the SolidWorks

API. Only the texts of the annotations are not converted to a 3D mesh as they are dynamically generated by a viewing device when viewed in the AR environment.

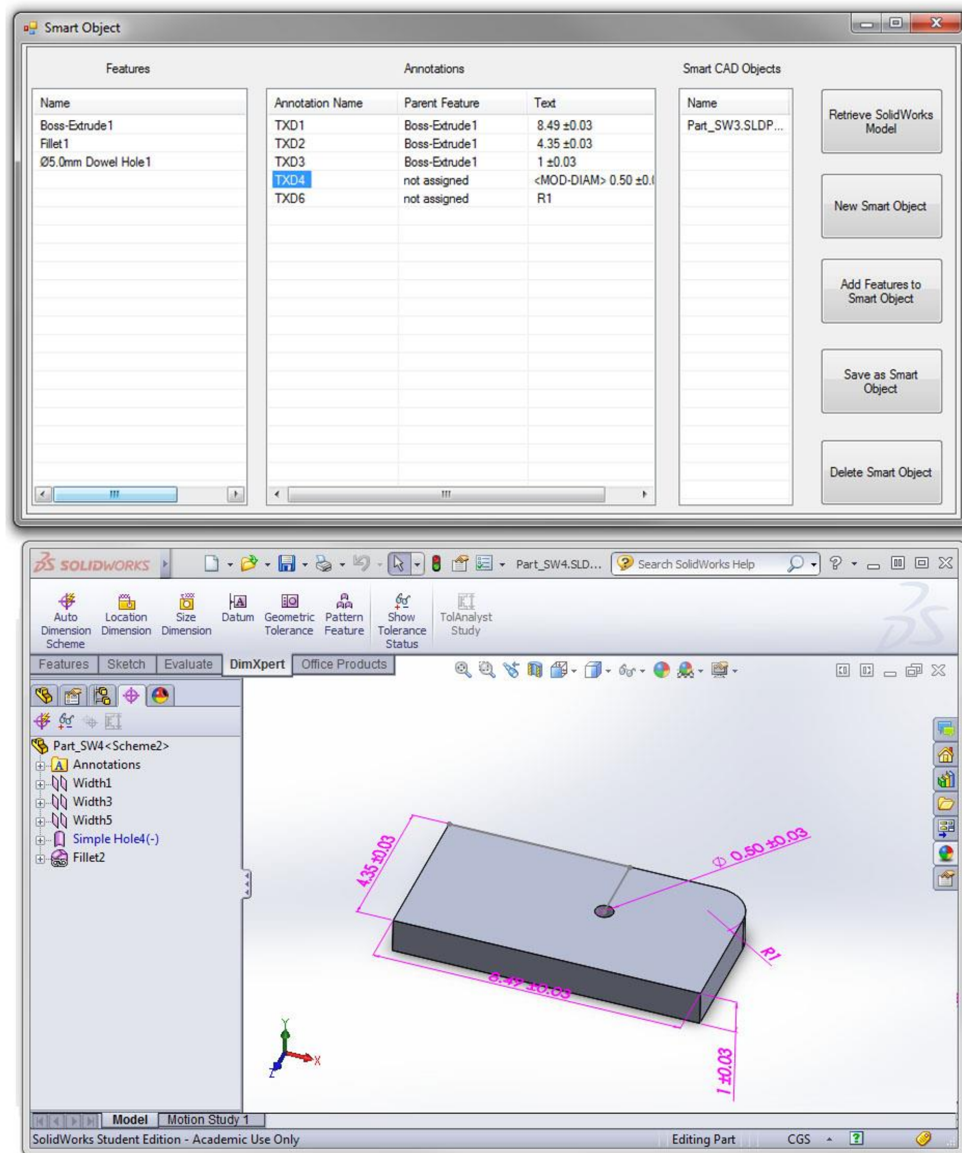


Figure 7-1. (Top) Smart CAD object creation tool, (bottom) SolidWorks part document converted into a smart CAD object.

A smart CAD object enhances the understanding and evaluation of a part design by utilizing the design-by-features philosophy, with each feature of the CAD model making up a single interactive element of the virtual user interface. The virtual user interface of the whole model is a visualization of the product or part, with interactive features that allow the user to hide and show specific features and to display parameters, such as dimensions, tolerances, etc., of each feature, which is similar to the kind of functionality one would find within the SolidWorks environment. Figure 7-2 shows the smart CAD object in an AR environment with different feature annotations shown or hidden as the user clicks on the feature.

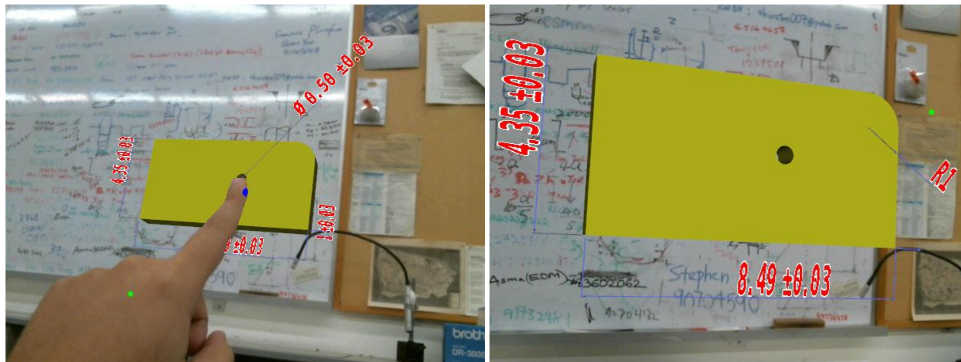


Figure 7-2. An interactive smart CAD object.

7.1.2 Smart Machining Object

A smart machining object, which overlays a CNC machine, is an example of a smart object that augments a physical object existing in the environment with a virtual user interface and added functionality. A smart machining object has different use contexts, such as machining and maintenance.

In the machining context, a smart machining object provides computer-aided manufacturing (CAM) functionality through RPCs, similar to the way SolidWorks functionality is implemented in the smart CAD object. For example, MasterCam has C-Hooks and NET-Hooks SDK for providing MasterCam functionality respectively through C/C++ and .NET programs. A smart machining object can have CAM functionality by creating RPCs which call the functions provided by CAM software SDKs. With an accurate model of the CNC machine, the graphical output from the CAM software can be augmented directly on the CNC machine, thus making the physical CNC machine part of the CAM software interface. If the machining table of the CNC machine is defined as an interactive element in its virtual user interface file with the tag “Object”, it can be used to load a CAD model into the CAM interface of a smart machining object by the user by placing a smart CAD object on the table. Once the CAD model is loaded, the smart machining object can proceed to generate and simulate machining toolpaths.

In the maintenance context, smart machining objects can display real-time sensor data and maintenance information so that maintenance personnel can inspect the conditions of the machining resources promptly. Furthermore, animated graphical maintenance instructions can be superimposed on the actual machine parts to provide easy-to-understand guidance on performing maintenance on machines. As different machines have different designs and parts, each smart machining resource object would have its own individual sensor outputs and maintenance instructions superimposed accordingly through the use of the AR user interface. However, making use of the capacity

of smart objects for independent behavior can allow for smarter maintenance systems. For example, an AR maintenance system that displays information according to the type of maintenance and the user's skill level and allows authoring of AR content to improve maintenance guidance (Zhu, et al., 2013) can be implemented by programming the smart machining resource object to activate different AR user interface elements according to a user's skill level setting that is input to the smart object. Figure 7-3 shows the virtual user interface of a smart machining object for different tasks and gives an idea of how users might interact with it.

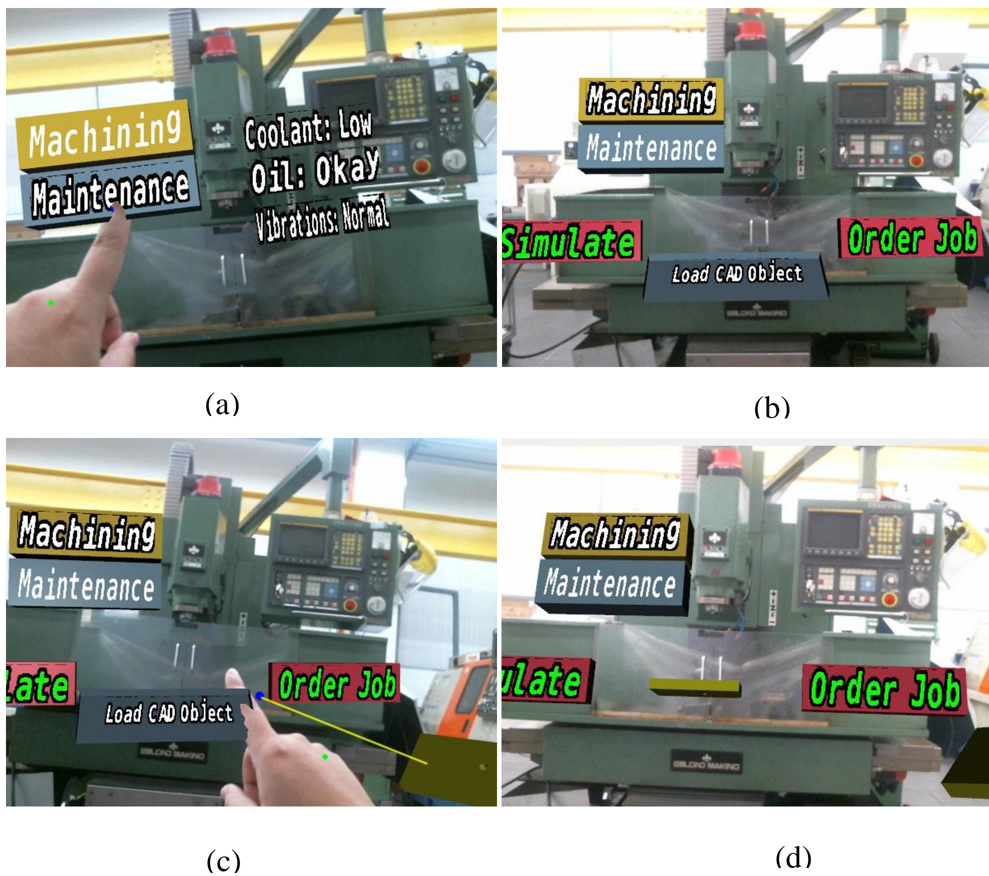


Figure 7-3. Smart machining object: (a) Maintenance interface, (b) CAM interface, (c) Dragging a smart CAD object to the CAM interface, and (d) Smart CAD object loaded in the CAM interface.

7.2 Manufacturing Grid

MGrid research often encapsulates manufacturing services as web services (He, et al., 2006; Dong, et al., 2008) in order to make use of the connectivity and homogeneity of the web. Therefore, it is useful to make use of web protocols to link smart objects to a manufacturing grid so as to make it side-by-side compatible with web service-oriented manufacturing services.

A possible approach for putting local smart objects on the internet is to set up a web server which provides remote access to smart objects using HTTP protocols and generates HTML forms (World Wide Web Consortium, n.d.) based on the RPCs of the smart objects. For users in their local SmARtWorld environment to interact with a remote smart object, the HTML code received from the remote web server needs to be interpreted and the RPCs packaged as HTML form submissions. For this, a smart object called a cloud gateway that mediates a web server and local smart objects is needed (Figure 7-4).

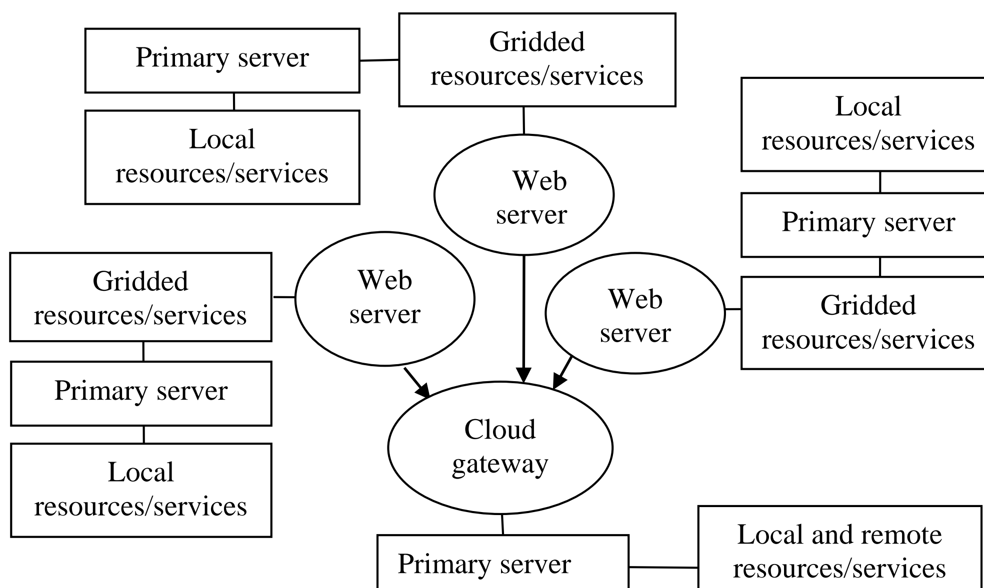


Figure 7-4. Architecture of manufacturing grid of smart objects.

7.2.1 Web Server

The main index page of a web server would list the smart objects available, its RPCs and the URL to retrieve the virtual user interfaces. The HTML code below illustrates the manner in which a smart machining object called "Makino Milling Machine" is publicized through the index page. Custom HTML tags are used to indicate to a remote client information about the RPCs, data and virtual user interface of a smart object, with standard HTML tags interspersed to allow normal web browsers to display information about the smart object:

```
<smartobject>
<u>Tom's Workshop</u><br/>
<i>Name: </i>
<smartobject-name>Makino Milling Machine</smartobject-name><br/>
<i>Description: </i>
<smartobject-description>Smart Machining Object</smartobject-
description><br/>
<a href="1/resource_index.html">Access Resource</a><br/>
<div id="hidden" style="display:none">
<smartobject-interface>1/MakinoMilling.dae</smartobject-interface>
<smartobject-
rpc>1/Load_Model.html,1/Simulate.html,1/Order_Job.html</smartobject-
rpc>
<smartobject-data>Available:Yes</smartobject-data>
```

```
</div>
</smartobject>
```

The custom HTML tags are those with the “smartobject-” prefix and encapsulate the information which is hidden from display on normal browsers, so as not to look out of place on the website. RPCs are provided as HTML forms serving as web interfaces to the specific functions of the manufacturing resource. Following the above example, the Makino Milling Machine has three RPCs, namely, Load Model, Simulate, and Order Job, as specified within the <smartobject-rpc> tag. The “Load Model” RPC is for a remote user to load a CAD model into the smart machining object for machining simulation using the “Simulate” RPC, and finally to request for the manufacturing resource to carry out a production run of the part the “Order Job” RPC is used. As a smart object, these RPCs are specified in the COLLADA file of the virtual user interface of the object. The HTML form for each RPC is defined in its own HTML file. The HTML code for the “Load Model” RPC is as follows:

```
<form action="Load_Model" method="post" enctype="multipart/form-data">
<label for="file">Upload a SolidWorks part file (.sldprt) or STL file (.stl):</label>
<input type="file" name="CadModel" id="file"><br>
<input type="submit" name="submit" value="Submit"><br>
</form>
```

In the HTML form, the “action” attribute of a “form” tag is used as the RPC identifier, and the “name” attribute of an “input” tag is used as the data identifier. When the “Load Model” RPC is invoked, the CAD model, labeled with the data identifier “CadModel”, is transferred using HTML as a form submission regardless of whether the source of the RPC invocation is a smart object or web browser.

7.2.2 Cloud Gateway

Web browsers will be able to interact with remote smart objects immediately by communicating with the remote web server, but this bypasses their virtual user interfaces and uses HTML forms as the user interface instead. The cloud gateway serves as the link between a remote web server and local smart objects so that a remote smart object can be treated as a normal smart object in the local AR environment. In the local AR environment, the cloud gateway is seen as a host to smart objects which users can interact with via their virtual user interfaces (Figure 7-5).

Figure 7-5. Smart machining object from a remote SmARtWorld environment.

The cloud gateway allows users to browse different web servers, allowing proxy versions of smart objects found on the web server to be hosted on the cloud gateway. RPCs from viewing devices and other objects are sent to the remote objects via the cloud gateway. The cloud gateway formats RPCs as HTML form data as follows:

```
------------8d185dce20123be
Content-Disposition: form-data; name="CadModel"; filename="Part_SW4.sldprt"
Content-Type: application/octet-stream

<binary data as Base64 encoded string>
------------8d185dce20123be--
```

In the form submission, each piece of data is bounded by a random string prefixed by a series of dashes. The data identifier of each piece of data is entered in the “name” attribute, and the form submission is sent towards the URL of the RPC’s corresponding HTML form hosted on the remote web server. The web server extracts the data and sends the corresponding RPC to the object in its local environment. The web server can send a response to the form submission and include a standard command, such as to update certain pieces of data or the virtual user interface, for the object that sent the RPC in the first place.

One limitation of this approach of using HTML forms alone is that the remote object is not able to send updates on its status to local objects after the initial response to the form submission unless the cloud gateway automatically queries the remote object at intervals. However, this can be overcome by adding server-side scripting to the web server and client-side scripting to the cloud gateway.

7.3 Visual Programming

Visual programming is a programming paradigm in which programming tasks are performed graphically by making connections and defining relationships between different functional entities rather than by writing code. In the manufacturing industry, it has been applied in areas like control and mechatronic system design and simulation (Effen, 2001; Sakairi, et al., 2013), resource planning (Dorner, et al., 2009), robot programming (Schlette, et al., 2014), and production control (Köhler, et al., 2000). In addition, AR has been used as a child's tool for creating AR applications through visual programming (Radu & MacIntyre, 2009), demonstrating how easy programming can be in an AR environment. Two examples of visual programming in a manufacturing environment are given below. The first shows the use of high-level programming to enhance robot task programming, and the second makes use of low-level programming to make a robot consider human safety during its operation.

7.3.1 Robot Task Programming

The programming of industrial robots such as pick-and-place and painting applications, usually involves writing codes in a programming language that is interpreted by the robot controller. For example, the tasks for ABB robots are programmed using the RAPID code. This requires the task engineer to be familiar with programming and RAPID code. Sometimes, an engineer from ABB is hired to help with the programming. This process is often both time-consuming and costly.

To program a robot to pick a workpiece from a worktable and place it on a conveyor belt, the robot programmer has to define a waiting position for the robot, command the robot to pick the workpiece when it is ready, define a path from the worktable to the conveyor belt, and command the robot to release its gripper. In addition, a presence sensor is needed to be placed on the worktable and connected to the robot controller so that the robot program will be able to read its signal. Assuming the workpiece is always picked up from the same position and orientation on the worktable, the program for the robot, written in pseudo-code, is as follows:

```
1    Wait for presence signal input
2    Move along Path 1 (waiting position to
     workpiece position)
3    Close gripper
4    Move along Path 2 (from workpiece to conveyor
     belt)
5    Release gripper
6    Move to Path 3 (from conveyor belt to waiting
     position)
7    Goto line 1
```

Each path in the pseudo-code is made up of a sequence of positional and orientation targets in the workspace of the robot. Each target has to have its position, orientation, speed, zone (how close the robot must pass through the target), and robot configuration defined in the code. In addition, the signal input from the presence sensor must be given a name and associated with the input port on the robot controller that it has been connected to. This is done through the user interface on a teach pendant, which is a tablet device that is used to manually move the robot and create targets based on the pose of the robot.

Turning the robot's workspace into a UAR environment and the task objects, like the workpiece, worktable and conveyor belt, into smart objects, the task of the robot can be programmed visually through the virtual user interfaces of the smart objects. Table 7-1 shows the smart objects that are used in the pick-and-place scenario and the relevant RPCs.

Table 7-1. Smart objects of a pick-and-place robot workspace

Smart Object	RPCs and Data
Robot	Create new looped task Finalize task creation
Workpiece	Pick up Release
Worktable	Wait for ready signal Move to waiting position Ready signal (data)
Conveyor belt	Place object

The "Create new looped task" RPC creates a task that repeats the first action after the last action has been performed. Invoking this RPC puts the robot in a programming state where the robot waits for messages from other smart objects that indicate actions to perform for the task. Each RPC accessible on

the task objects sends a signal to the robot, which generates the RAPID code to be sent to the robot controller. To create the same task as the pseudo-code given above, the RPCs are triggered in the following order:

1. Move to waiting position
2. Wait for ready signal
3. Pick up
4. Place object
5. Release

Targets and paths are generated automatically by the robot by using knowledge of the pose of the task objects, which is the fundamental information that smart objects in a SmARtWorld environment provide. For example, the waiting position is defined by the worktable object with respect to its own coordinate frame. The robot computes the waiting position with respect to the UAR environment by transforming the waiting position with respect to the worktable by the pose of the worktable with respect to the UAR environment. The robot then computes the waiting position with respect to its own coordinate frame by inverse-transforming the waiting position with respect to the UAR environment by the robot's own pose in the UAR environment. This enables the robot to move to the waiting position using its own coordinate frame.

The smart objects in Table 7-1 have to use pre-determined RPC identifiers in order that the robot smart object can understand the action indicated by the RPCs of the task objects. For example, when the "Wait for ready signal" RPC is invoked, the worktable object must send a message that the robot will

understand to inquire on the “ready” status of the worktable via its “Ready signal” RPC. This requires cooperation between the robot and the task object manufacturers to come up with standard RPC identifiers for robot programming.

7.3.2 Programming Robot Safety Procedures

The object-oriented SmARtWorld architecture is open to certain programming paradigms, such as object-oriented programming (OOP) and flow-based programming (FBP), which can be applied in a SmARtWorld environment for visual programming at a lower level.

In OOP, objects contain functions and variables. Variables can be assigned values while functions are called to invoke certain program behavior. FBP is component-oriented rather than object-oriented. Components are “black boxes” which process data from its input streams to generate output data. Applications are built by connecting the inputs and outputs of different independent components. FBP can be implemented over OOP languages and can help make the logical flow of information in applications easier to visualize. A smart object in a SmARtWorld environment can directly wrap objects or components that are implemented in an OOP language and provide access to their functions and variables via RPC and data identifiers. By establishing connections between different smart objects through their virtual user interfaces, high-level programs and behavior can be created visually.

Table 7-2 shows five smart objects which can be used as components for FBP in a SmARtWorld environment. The first object, the Smart Object Creator, is used to create a new smart object that encapsulates the behavior that is to be programmed for. The Conditional More-than object has a data identifier representing its current output state. It takes an input value and compares it to a comparison value that is set by the programmer. Its output state is set as "true" if the input value is more than the comparison value, and "false" otherwise. Additionally, it can be set to trigger a different RPC in another object for each state. The Logical AND object also has an output state which is "true" if both its inputs are "true" and "false" otherwise. The While-loop object keeps triggering the RPC which is set as its output as long as its input value is "true". Other conditional and logic components can be implemented similarly.

Table 7-2. Smart objects for flow-based programming in a SmARtWorld environment.

Smart Object	RPCs	Remarks
Smart Object Creator	Create New Object	
New Smart Object	Start Finalize	This is the object created by the smart object creator
Conditional More-than	Set Input Source Set Comparison Value Set True Output Set False Output	Data identifiers for a Boolean value representing its current output state and a number representing the current comparison value.
Logical AND	Set Input Source 1 Set Input Source 2 Set True Output Set False Output	Data is a Boolean value representing its current output state.
While-loop	Set Input Source Set Input Value Set Output	Data is a Boolean value representing its current input state.

To illustrate the use of visual programming in a factory where robots and humans work in close proximity, the objects in Table 7-1 are used to create an application that monitors a robot workspace and stops the robot if a human worker is about to cross its path. The rotation angle of the robot arm is output by the robot and connected to the Conditional More-than object. A rotation of more than 90 degrees means the robot arm is about to swing into the path of a walkway for workers. The Conditional More-than object outputs "true" to a Logical AND object which also receives a true/false signal from a presence sensor set up near the walkway. If both inputs to the Logical AND object are "true", a pause signal is sent to the robot to halt the robot arm until the walkway is clear. The flow diagram of the system is shown in Figure 7-6.

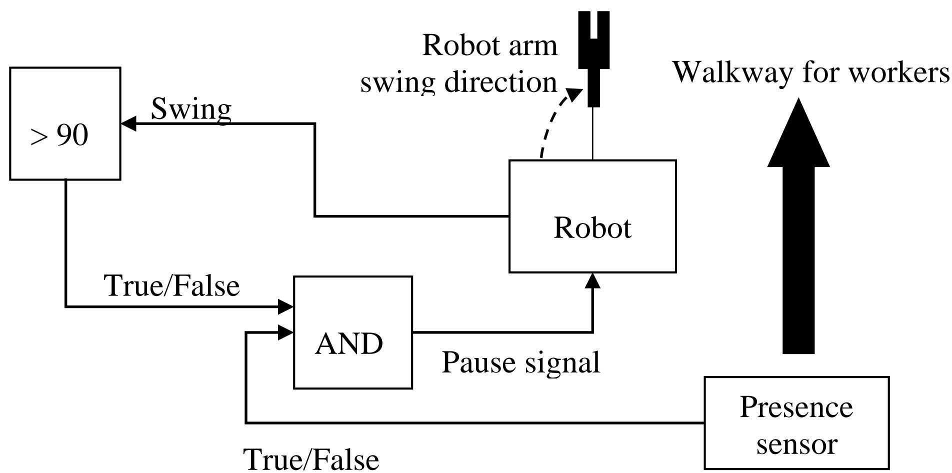


Figure 7-6. Flow diagram of a program that stops a factory robot arm when a worker approaches it.

For the program shown in Figure 7-6, the smart object only executes its behavior once. The user may wish to execute the program continuously. To achieve this, another smart object is created and the While-loop object is used

to continue execution of the RPC of the first smart object. The input value of the While-loop object is set to “true” to make it an infinite loop, and its output is set to the “start” RPC of the first object.

The implementation of the objects in Table 7-1 and the visual programming process in a SmARtWorld environment is described next. The Smart Object Creator is a hub which hosts the new objects that it creates. The programmer creates a new object which comes with a Start RPC and Finalize RPC. The virtual user interface of the new object is by default a cube with buttons for the RPCs. The Conditional More-than, Logical AND and While-loop objects are hosted by the smart object creator. When a new smart object is created, the smart object creator and all its constituent objects enter into the programming mode, which means any connections made during this mode will be applied as behavior triggered by the new RPC. The Finalize RPC is used to exit the programming mode and store the behaviors that have been programmed. Connections and values are remembered by the default component objects, i.e., the Conditional More-than, Logical AND and While-loop objects, for each smart object that has been created. Thus, when a new smart object executes its behavior, the default component objects work independently to fulfill the functionality of the new smart object.

The system described is a highly simplified environment for FBP. A more complete implementation of an FBP environment would allow more complex behavior to be programmed, more RPCs to be added and virtual user interfaces to be user-modeled. Furthermore, to achieve a self-sustaining UAR

framework, visual programming should be done at an even lower-level where new smart objects that run independently from the Smart Object Creator are built. This would entail source code generation and compilation into an executable program. However, current approaches in visual programming paradigms have not been proven to be efficient methods of low-level programming as simple functionality that would take only a few lines of codes to implement might require very complex component connections, thus making this still an open issue.

Chapter 8. Conclusion

8.1 Achievement of Objectives

The goal of this research is to develop a framework that facilitates highly interactive and intelligent UAR environments. The result is SmARtWorld, a component-based framework that transforms effectively a physical environment into an operating system for UAR environments and relies on independent smart objects to provide basic UAR functionality as well as UAR applications. The novelty of SmARtWorld is that every component is essentially an independent smart object that can have a dynamic virtual user interface in AR for user-friendly interactivity with users. Another unique feature of SmARtWorld is that, as a component-based framework, it does not require any middleware software as interoperability between components is achieved through the use of standard definitions of RPC and data identifiers and formats. The balance between the enforcement of standards and the goal of universal access has been considered carefully throughout this research. Five objectives of this research were stated in Chapter 1. The achievement of each objective is explained below.

Objective 1: create a common framework for UAR environments that abstracts applications from hardware for tracking, interaction and display. Chapter 4 has given an implementation of a basic UAR environment. Based on this, applications such as a UAR manufacturing environment and manufacturing grid have been demonstrated in Chapter 7. These applications are made up of individual smart objects that can be developed independently.

They work together by using a common set of identifiers for sharing data and triggering remote procedure calls. The applications available in a UAR environment thus depend on the smart objects that are placed in the environment. The different network protocols, embedded hardware and operating systems used by individual smart objects are hidden, while functionality of the smart objects are provided to users through the virtual user interfaces that are defined using a common schema.

Objective 2: allow for flexibility in the hardware and software used to implement context-aware smart objects with highly customizable behaviors, appearance and user interfaces. In Section 3.3.1, it was shown that any hardware and software libraries can be used in the fundamental layer of the smart objects, as long as the hardware and software libraries are compatible and the functionality and data access layer, i.e., the middleware layer, is written to interface with the fundamental layer. In addition, virtual user interfaces allow any part of the 3D model of a smart object to be interacted with to trigger any behavior that has been defined in its functionality and data access layer.

Objective 3: allow for flexibility in the hardware and software used to implement viewing and interaction devices. The abstraction of hardware from the programming of smart objects has been emphasized in the framework, as illustrated in Section 3.3.1. Fundamental functions like object and environmental tracking can be performed by different types of sensors, while the data output can be formatted in the same way and use the same data

identifiers. Graphical rendering and tracking hardware and algorithms are always improving, and so the framework will be able to make use of new technologies by formatting data transfer according to the communications protocol defined in Section 3.4.

User interaction methods are also not strictly enforced in the framework. In Sections 5.1 and 5.2, the implementation of three hand gestures, which are commonly used today on the majority of touch-screen systems, to interact with smart objects through different display devices are given. However, different sensors and gesture recognition methods can also be used to detect the three hand gestures. User interaction is not limited to the three hand gestures because, as presented in Section 3.4, specific smart object behavior can be triggered remotely using the "RPC" command, and this command can be triggered via any gesture as long as it is recognized by the target smart object. Furthermore, as illustrated in Section 5.3, user interaction need not be implemented in viewing devices.

Objective 4: recommended practices for AR application development using the proposed framework. The SmARtWorld framework gives developers greater freedom in the implementation of smart objects, viewing devices, and applications. However, there are some considerations raised in various chapters to ensure user-friendly to smart object functions using generic viewing devices. In Section 5.1.2, a number of graphical rendering effects were described that would enhance the visibility of virtual graphical elements overlaying a scene of the physical environment. For example, applying the

“real” property to the physical parts of smart objects so that they occlude the virtual parts, and specifying textual elements with an outline or glow property. Furthermore, in Chapter 6, several ways to help a virtual smart object blend in with the physical environment were described. These involve the use of several different types of smart objects, such as physics engines and light sources. These objects are not required for a basic UAR environment. However, AR applications would certainly benefit from making use of them.

Objective 5: a self-sustainable framework. Self-sustainability of the framework is achieved firstly by the abstraction of hardware from the programming of smart objects, which gives rise to flexibility of hardware and software used in the fundamental layer of smart objects. This means that as new types of sensors and other embedded technologies become available, the same smart object architecture can be used to implement smart objects to be used in a UAR environment. Therefore, as technology progresses, new smart objects can be created to encapsulate old and new functionalities using new technology.

Secondly, the framework features abstraction of behaviors from interaction methods, allowing for new interaction devices and technologies to be encapsulated in viewing devices and smart objects. As explained in Section 3.4, behaviors are triggered via remote procedure calls (RPCs) using a unique RPC identifier for each function, and new RPC identifiers can be defined for new functions and new smart objects. Smart objects can be programmed to invoke RPCs in other smart objects when they detect user input. A few

examples of device-less interaction methods are given in Section 5.3. It is possible that during the initial adoption phase of the framework, most interaction will take place via viewing devices using the three hand gestures described in Sections 5.1 and 5.2. However, as developers who specialize in user interaction start creating smart objects to encapsulate their technology, more UAR frameworks will start to utilize device-less interaction methods. This allows for simpler viewing devices for users and more customized user experiences in different UAR environments.

Another aspect of self-sustainability is the ability for the framework to create smart objects which make use of new technologies in order to “update” UAR environments. The vision of this research is to replace traditional desktop computing with “UAR computing”. Ideally, this includes the development of smart objects in a UAR environment. This aspect was briefly discussed in Section 7.3.2. However, the creation of smart objects from a low level, i.e., the generation of programming code in the fundamental layer of the smart object to allow the functionality and data interface layer to access the fundamental behavior of a smart object, has not been comprehensively investigated in this research. This is a very profound and extensive topic and would likely involve a fundamental change in the way software libraries which interface with hardware would need to be implemented.

8.2 Contributions

The main contribution of the research is a new component-based UAR framework for building UAR environments and applications. Every

component is a smart object that performs all fundamental functions as well as high-level application functions. A standard protocol for communication and a standard user interface definition schema have been proposed in this research so that smart objects can be used in any UAR environment. Smart objects have a virtual user interface that gives users access to their functionality and data through AR. The specific contributions are highlighted below.

Communications protocol. The communications protocol is one of the few standards enforced in SmARtWorld that allows smart objects to communicate with each other regardless of networking protocol, while the basic set of RPCs allows for basic UbiComp functionality in the environment. More functionality and content can be added to a SmARtWorld environment by bringing more smart objects into the environment with their own RPCs and data.

Virtual user interface definition. The virtual user interface forms the appearance and interactive elements of a smart object in AR. It is defined in a standard COLLADA schema with a few definitions added for the SmARtWorld framework to represent interactive elements and special rendering properties for virtual user interfaces.

UAR implementation. An implementation of a UAR environment using the SmARtWorld framework involves the development of several crucial types of smart objects, namely, the primary server, landmark server, landmark objects and object tracker. These objects can be used by adopters of the SmARtWorld

framework to create their own UAR environments. Some potential applications in manufacturing based on this implementation have been given in Chapter 7.

Viewing and interaction devices. Several methods of viewing and interacting with smart objects have been explored in this research. Viewing devices are smart objects which look for landmark objects in a SmARtWorld environment to track their pose and download the virtual user interfaces of smart objects to display them to the user in AR. The framework is designed to be able to work with current mainstream devices as well as emerging technology like wearable systems. The distributed nature of SmARtWorld means that a few less crucial rendering effects and behavior, such as reflectivity, physical and sound response, can be added with the support of other smart objects in the environment rather than requiring the viewing device to handle the possible ways that smart objects can be represented in the environment.

Universal access to independently-developed UAR environments in any location. The SmARtWorld framework provides universal access to the functions of UAR environments and smart objects through user-friendly virtual user interfaces, though specialized viewing devices can also be created that might cater customized functions and interaction methods for specific UAR environments.

Bridged gap between real and virtual objects. It has also been shown how the framework can facilitate blending and interaction between real and virtual

objects so as to reduce the distinction between the physical and digital domain through the development and use of particular smart objects in the UAR environment. The real and virtual objects blend smoothly with each other in terms of appearance and functionality. Real objects can occlude virtual objects and collide with virtual objects under the framework. Real and virtual objects can work together, e.g., to integrate virtual and real sensor data together in one application.

Abstraction of applications from fundamental functionality. There is significant flexibility in the potential behavior, AR appearance and virtual user interfaces of smart objects. Any hardware and software libraries can be used in the implementation of the fundamental behavior of a smart object. As a result, application developers can focus on creating smart objects that provide specific applications without having to consider the implementation of fundamental functionality, and without having to consider the specific details about the environment they would be used in. Environment developers can easily set up a UAR environment and the applications therein without having to perform low-level programming. When smart objects, which encapsulate basic functionality become well-established and shared around the world, developers will be able to use visual programming techniques to create smart objects and applications without having to learn programming languages.

8.3 Recommendations

There are a few issues that have not been considered in this research. First, the scalability, quality of service, and error handling issues have not been investigated. Before this framework can be widely deployed and adopted, possible failures and network latency issues need to be handled.

Secondly, the tracking accuracy and speed have not been completely optimized for the wearable and smartphone systems that are used to evaluate the SmARtWorld environment execution based on the framework. This disturbed users during tests. Dedicated research on the issues and methods of tracking in a large environment should be conducted with the aim of minimizing jitter and blind zones (areas where no tracking takes place), and maximizing frame rates.

Thirdly, the weight and cumbersomeness of the wearable system prototype have negative impacts during test. This problem is caused by the hardware required in order to have the hands-free display capability and enough computational power for tracking and interaction. However, mobile and display technology is rapidly evolving and will enable much lighter and comfortable wearable systems to be built in the near future.

Lastly, while virtual object animation was superficially investigated in this research, it does not have the animation quality that can be found in modern video games. For proper gaming, entertainment and media applications to be possible, this aspect has to be developed further.

Once the afore-mentioned issues are resolved, a clear documentation of all the standards and protocols that have been used must be made so that developers can use the framework. There are a few AR and UbiComp standards that might be worth integrating with the framework, e.g., vision-based descriptors proposed by MPEG (*Compact Descriptors for Visual Search*, 2011).

Publications from this Research

Yew, A.W.W., Ong, S.K., & Nee, A.Y.C. (2010). *SmARt World – Ubiquitous Augmented Reality Computing*. Paper presented at the International Workshop on Mobile Collaborative Augmented Reality, 13-16 October 2010, Seoul, South Korea.

Yew, A.W.W., Ong, S.K., & Nee, A.Y.C. (2011). SmARt World - User-Friendly Mobile Ubiquitous Augmented Reality Framework. In L. Alem & W. Huang (Eds.) *Recent Trends of Mobile Collaborative Augmented Reality Systems*. Berlin: Springer.

Yew, A.W.W., Ong, S.K., & Nee, A.Y.C. (2013). Adding Augmented Virtual Interfaces to Simple Smart Objects for Ubiquitous Computing Environments. *Proceedings of the IADIS International Conferences on Interfaces and Human Computer Interaction 2013 and Game and Entertainment Technologies 2013* (pp. 37 – 44). IADIS Press. Paper presented at the IADIS International Conference on Interfaces and Human Computer Interaction, 22 - 24 July 2013, Prague, Czech Republic.

Yew, A.W.W., Ong, S.K., & Nee, A.Y.C. (2014). Augmented Reality Interfaces for Smart Objects in Ubiquitous Computing Environments. In P. Isaías, & K. Blashki (Eds.) *Human-Computer Interfaces and Interactivity: Emergent Research and Applications* (pp. 208-229). Hershey, PA: Information Science Reference.

Yew, A.W.W., Ong, S.K., & Nee, A.Y.C. (2014). Towards A Griddable Distributed Manufacturing System with Augmented Reality Interfaces, *Robotics and Computer Integrated Manufacturing*, submitted.

References

Ahn, S., Kang, D., Kim, H., & Ko. H. (2005). Ubiquitous Smart Interaction Space. *Proceedings of the First International Workshop on Personalized Context Modeling and Management for UbiComp Applications 2005* (pp. 122 – 124). Aachen, Germany: CEUR Workshop Proceedings.

ARToolKit. (n.d.). Retrieved August 10 2014 from http://www.hitl.washington.edu/artoolkit/

Aruco: A Minimal Library for Augmented Reality Applications Based on OpenCV (n.d.). Retrieved 12 August 2010 from http://www.uco.es/investiga/grupos/ava/node/26

Avanzini, F., Rath, M., Rocchesso, D., & Ottaviani, L. (2003). Low-level Models: Resonators, Interactions, Surface Textures. In D. Rocchesso & F. Fontana (Eds.) *The Sounding Object* (pp. 137 – 165). Italy: Phasar

Ayu, M., Ismail, S., Matin, A., & Mantoro, T. (2012). A Comparison Study of Classifier Algorithms for Mobile-phone's Accelerometer Based Activity Recognition. *Procedia Engineering*, 41(0), 224 - 229.

Ballagas, R., Ringel, M., Stone, M., & Borchers, J. (2003). iStuff: A Physical User Interface Toolkit for Ubiquitous Computing Environments. *Proceedings of SIGCHI Conference on Human Factors in Computing Systems* (pp. 537 – 544). New York, NY: ACM.

Bauer, M., Bruegge, B., Klinker, G., MacWilliams, A., Reicher, T., Riß, S., Sandor, C., & Wagner, M. (2001). Design of a Component-Based Augmented Reality Framework. *Proceedings of the Second IEEE and*

ACM International Symposium on Augmented Reality (pp. 45-54). Washington, DC: IEEE Computer Society.

Cao, X., Forlines, C., & Balakrishnan, R. (2007). Multi-User Interaction using Handheld Projectors. *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (pp. 43 - 52). New York, NY: ACM.

Caudell, T., & Mizell, D. (1992). Augmented Reality: An Application of Heads-up Display Technology to Manual Manufacturing Processes. *International Conference on System Sciences (Vol. 2)* (pp. 659 - 669). Washington DC: IEEE Computer Society.

Cerf, V., Dalal, Y., & Sunshine, C. (1974). *Specification of Internet Transmission Control Program*. RFC 675. Retrieved 18 August 2014 from http://tools.ietf.org/html/rfc675

Chadwick, J., Zheng, C., & James, D. (2012). Precomputed Acceleration Noise for Improved Rigid-Body Sound. *ACM Transactions on Graphics (TOG) - SIGGRAPH 2012 Conference Proceedings*, 31(4), Article 3.

Chouiten, M., Didier, J-Y., & Mallem, M. (2011). Proceedings of the 5th International Conference on Communication System Software and Middleware (pp. 3:1 – 3:7). New York, NY: ACM.

Compact Descriptors for Visual Search. (2011). Retrieved 5 Sep 2014 from http://mpeg.chiariglione.org/standards/mpeg-7/compact-descriptors-visual-search

Costa, C., Yamin, A., & Geyer, C. (2008). Toward a General Software Infrastructure for Ubiquitous Computing. *IEEE Pervasive Computing*, 7(1), 64 - 73.

Crepaldi, R., Harris, A., Kooper, R., Kravets, R., Maselli, G., Petrioli, C., & Zorzi, M. (2007). Managing Heterogeneous Sensors and Actuators in Ubiquitous Computing Environments. *Proceedings of the First ACM Workshop on Sensor and Actuator Networks* (pp. 35 - 42). New York, NY: ACM.

Danylenko, A., Kessler, C., & Löwe, W. (2011). Comparing Machine Learning Approaches for Context-Aware Composition. In S. Apel, & E. Jackson (Eds.), *Software Composition* (pp. 18-33). Berlin: Springer Berlin Heidelberg.

Department of Defense World Geodetic System 1984. (2000). Springfield, Virginia: National Geospatial-Intelligence Agency (NGA).

Documents Associated With CORBA, 3.3. (2012). Retrieved 20 August 2014 from http://www.omg.org/spec/CORBA/3.3/

Dong, B., Qi, G., Gu, X., & Wei, X. (2008) Web Service-Oriented Manufacturing Resource Applications for Networked Product Development. *Collaborative Design and Manufacturing,* 22(3), 282 – 295

Dorner, C., Draxler, S., Pipek, V., & Wulf, V. (2009). End Users at the Bazaar: Designing Next-Generation Enterprise Resource Planning Systems. *IEEE Software,* 26(5), 45 – 51.

ECMAScript Language Specification. (2011). Geneva: ECMA International.

Effen, M.C. (2001). A Mechatronics Library for SIMULINK. *Proceedings of the 2001 IEEE International Conference on Control Applications* (pp. 121 – 124). Washington, DC: IEEE Computer Society.

Fjeld, M., Bichsel, M., & Rauterberg, M. (1998). BUILD-IT: An Intuitive Design Tool Based on Direct Object Manipulation. *Proceedings of the International Gesture Workshop on Gesture and Sign Language in Human-Computer Interaction* (pp. 297 - 308). London: Springer-Verlag.

Fong, W., Ong, S., & A.Y.C., N. (2009). Computer Vision Centric Hybrid Tracking for Augmented Reality in Outdoor Urban Environments. *Proceedings of 16th Symposium on Virtual Reality Software and Technology* (pp. 185 – 190). New York, NY: ACM.

glBlendFunc function. (2012). Retrieved 25 August 2014 from http://msdn.microsoft.com/en-us/library/windows/desktop/dd318368(v=vs.85).aspx

Guo, J., Wang, Y., Chen, J., Lin, J., Wu, L., Xue, K., Liu, W., & Zhang, J. (2009). Markerless tracking for augmented reality applied in reconstruction of Yuanmingyuan archaeological site. *Proceedings of 11th IEEE International Conference on Computer-Aided Design and Computer Graphics* (pp. 324 – 329). Washington DC: IEEE Computer Society.

Guo, Y. (2008). Reasoning with Semantic Web Technologies in Ubiquitous Computing Environment. *Journal of Software*, 3(8), 27 - 33.

Haghighi, P., Krishnaswamy, S., Zaslavsky, A., & Gaber, M. (2008). Reasoning about Context in Uncertain Pervasive Computing Environments. In D. Roggen, C. Lombriser, G. Tröster, G. Kortuem, & P. Havinga (Eds.), *Smart Sensing and Context* (pp. 112 - 125). Berlin: Springer Berlin Heidelberg.

Haller, M., Drab, S., & Hartmann, W. (2003). A Real-time Shadow Approach for an Augmented Reality Application Using Shadow Volumes. *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (pp. 56 - 65). New York, NY: ACM.

Han, J.J. & Kim, S.-K. (2014). *Text of White Paper on MPEG-V*. International Organization for Standardization.

He, Y., Yu, T., Liu, L., Shen, B., & Sun, H. (2006). A WSRF-Based Resource Management System of Manufacturing Grid. *Sixth IEEE International Symposium on Cluster Computing and the Grid Management System of Manufacturing Grid,* (pp. 174-177). Singapore.

Hessman, T. (2013, February 14). *The Dawn of the Smart Factory.* Retrieved November 10, 2013, from Industry Week: http://www.industryweek.com/technology/dawn-smart-factory

Hill, A., MacIntyre, B., Gandy, M., Davidson, B., & Rouzati, H. (2010). KHARMA: An open KML/HTML architecture for mobile augmented reality applications. *Proceedings of the 9th International Symposium on Mixed and Augmented Reality* (pp. 233 – 234). Washington, DC: IEEE Computer Society.

Hincapie-Ramos, J., Tabard, A., & Bardram, J. (2011). Mediated Tabletop Interaction in the Biology Lab: Exploring the Design Space of the Rabbit. *Proceedings of the 13th International Conference on Ubiquitous Computing* (pp. 301-310). New York, NY: ACM.

Hou, L., Wang, X., Bernold, L., & Love, P. (2013). Using Animated Augmented Reality to Cognitively Guide Assembly. *Journal of Computing in Civil Engineering*, 27(5), 439 – 451.

Hoque, E. & Stankovic, J. (2012). AALO: Activity Recognition in Smart Homes Using Active Learning in the Presence of Overlapped Activities. *Proceedings of the 6th International Conference on Pervasive Computing Technologies for Healthcare* (pp. 136 – 146). Washington, DC: IEEE Computer Society.

Hunter, A. (2001). A Default Logic Based Framework for Context-Dependent Reasoning with Lexical Knowledge. *Journal of Intelligent Information Systems*, 16(1), 65 - 87.

Irawati, S., Ahn, S., Kim, J., & Ko, H. (2008). *IEEE Virtual Reality Conference 2008* (pp. 201 – 208). Washington DC: IEEE Computer Society.

Jin, Y., Wang, R., Huang, H., & Sun, L. (2010). Agent-Oriented Architecture for Ubiquitous Computing in Smart Hyperspace. *Wireless Sensor Network*, 1(2), 74 – 84.

Josefsson, S. (2006). *The Base16, Base32, and Base64 Data Encodings (RFC 4648).* Retrieved 18 August 2014 from https://tools.ietf.org/html/rfc4648.

Kainz, B. & Streit, M. (n.d.). *How to Write an Application with Studierstube 4.0.* Retrieved 20 August 2014 from http://studierstube.icg.tugraz.at/doc/pdf/Stb4AppWriting.pdf

Kato, H., & Billinghurst, M. (1999). Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System. *Proceedings of 2nd IEEE and ACM International Workshop on Augmented Reality* (pp. 85 - 94). Washington DC: IEEE Computer Society.

KHARMA Framework. (n.d.). Retrieved 20 August 2014 from https://research.cc.gatech.edu/kharma/content/kharma-framework

Khronos Group. (2008). *COLLADA – Digital Asset Schema Release 1.5.0.* Retrieved 18 August 2014 from http://www.khronos.org/files/collada_spec_1_5.pdf

Kim, Y.C. & Moon, C.H. (2014). Non-Contact Gesture Recognition Using the Electric Field Disturbance for Smart Device Application. *International Journal of Multimedia and Ubiquitous Engineering*, 9(2), 133 – 140.

Kimura, H., Tokunaga, E., & Nakajima, T. (2006). Building Mobile Augmented Reality Services in Pervasive Computing Environments. *Proceedings of ACS/IEEE International Conference on Pervasive Services* (pp. 285 - 288). Washington, DC: IEEE Computer Society.

Köhler, H.J., Nickel, U., Niere, J., & Zündorf, A. (2000). Integrating UML Diagrams for Production Control Systems. *Proceedings of the 22nd International Conference on Software Engineering* (pp. 241 – 251). Washington, DC: IEEE Computer Society.

Krevelen, D.W.F. van & Poelman, R. (2010). A Survey of Augmented Reality Technologies, Applications and Limitations. *The International Journal of Virtual Reality*, 9(2), 1 – 20.

Krum, D., Suma, E., & Bolas, M. (2012). Augmented Reality using Personal Projection and Retroreflection. *Personal and Ubiquitous Computing*, 16(1), 17 - 26.

Kwapisz, J.R., Weiss, G.M., & Moore, S.A. (2010). Activity Recognition Using Cell Phone Accelerometers. *ACM SIGKDD Explorations Newsletter*, 12(2), 74 – 82.

Layar App. (n.d.). Retrieved January 25, 2014, from Layar: https://www.layar.com/products/app/

Lechner, M. (Ed). (2013). OGC Augmented Reality Markup Language 2.0 (ARML 2.0) [Candidate Standard]. Open Geospatial Consortium Inc.

Lee, E.S., Hong, S., & Johnson, B.R. (2006). Context Aware Paper-Based Review Instrument: A Tangible User Interface for Architecture Design Review. In G.A. Luhan (Ed.), *Proceedings of the 25th Annual Conference of the Association for Computer-Aided Design in Architecture* (pp. 317 – 327). Louisville, KY: Association for Computer-Aided Design in Architecture.

Lee, W., & Park, J. (2005). Augmented Foam: A Tangible Augmented Reality for Product Design. *Proceedings of the 4th IEEE and ACM International Symposium on Mixed and Augmented Reality* (pp. 106 - 109). Washington DC: IEEE Computer Society.

Li, X., Chen, D., & Xiahou, S. (2009). Ubiquitous Augmented Reality System. *Proceedings of Second International Symposium on Knowledge Acquisition and Modeling* (pp. 91 – 94). Washington DC: IEEE Computer Society.

Liu, S., Cheng, T.-W., & Hsieh, Y.-C., (2011). Synthesizing Physics-Based Vortex and Collision Sound in Virtual Reality. *Lecture Notes in Computer Science*, 6939(1), 190-198.

LonWorks®-based Office Building Cuts HVAC, Lighting Costs 80% (n.d.). Retrieved October 2, 2013 from http://www.echelon.com/customers/smart-buildings/bob.htm

Meng, X., Yang, L., Aponte, J., Hill, C. Moore, T., & Dodson, A.H. (2008). Development of Satellite Based Positioning and Navigation Facilities for Precise ITS Applications. *11th International IEEE Conference on*

Intelligent Transport Systems (pp. 962 – 967). Washington DC: IEEE Computer Society.

Malis, E., & Vargas, M. (2007). *Deeper Understanding of the Homography Decomposition for Vision-based Control*. Nice: Institut National de Recherche en Informatique et en Automatique (INRIA).

Muller-Tomfelde, C., & Fjeld, M. (2012). Tabletops: Interactive Horizontal Displays for Ubiquitous Computing. *Computer*, 45(2), 78-81.

Nagel, T., Heidmann, F., Condotta, M., & Duval, E. (2010). Venice Unfolding: A Tangible User Interface for Exploring Faceted Data in a Geographical Context. *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries* (pp. 743 – 746). New York, NY: ACM.

Nee, A.Y.C., Ong, S.K., Chryssolouris, G., & Mourtzis, D. (2012). Augmented Reality Applications in Design and Manufacturing. *CIRP Annals - Manufacturing Technology, 61*(2), 657–679.

Ng, L., Wang, Z., Ong, S.K., & Nee, AY.C. (2013). Integrated Product Design and Assembly Planning in an Augmented Reality Environment. *Assembly Automation,* 33(4), 345 - 359.

Nguyen, T. A., Raspitzu, A., & Aiello, M. (2013). Ontology-based office activity recognition with applications for energy savings. *Journal of Ambient Intelligence and Humanized Computing*, doi: 10.1007/s12652-013-0206-7

Occupying Yourself with Energy and Occupancy (2010). Retrieved 2 October 2013 from

http://www.automatedbuildings.com/news/aug10/articles/sinopoli1/100728121202sinopoli.htm

Olwal, A., Gustafsson, J., & Lindfors, C. (2008). Spatial Augmented Reality on Industrial CNC-Machines. *Proc. SPIE 6804, The Engineering Reality of Virtual Reality 2008*, 680409, doi:10.1117/12.760960

OpenAL Soft. (n.d.). Retrieved January 27 2014 from http://kcat.strangesoft.net/openal.html

OpenVideo Documentation (n.d.). Retrieved 20 August 2014 from http://studierstube.icg.tugraz.at/openvideo/

Park, H., Lee, M.-H., Kim, S.-J., & Park, J.-I. (2006). Surface-Independent Direct-Projected Augmented Reality. *Lecture Notes in Computer Science*, 3852, 892 - 901.

Park, K., Park, K.-W., Lee, J., Yoo, J.-W., Lim, S.-H., & Choi, H.-J. (2008). U-TOPIA: A Ubiquitous Environment with a Wearable Platform, UFC and Its Security Infrastructure, pKASSO. *Communications in Computer and Information Science*, 11(1), 183 - 193.

Piekarski, W. & Thomas, B. (2003). An Object-Oriented Software Architecture for 3D Mixed Reality Applications. *Proceedings of the Second IEEE and ACM International Symposium on Mixed and Augmented Reality* (pp. 247 – 256). Washington, DC: IEEE Computer Society.

Pinhanez, C. (2003). Creating Ubiquitous Interactive Games Using Everywhere Displays Projectors. *The International Federation for Information Processing*, 112(3), 149 - 156.

Pirsiavash, H. & Ramanan, D. (2012). Detecting Activities of Daily Living in First-person Camera Views. *Proceedings of the 2012 IEEE Conference*

on Computer Vision and Pattern Recognition (pp. 2847 – 2854). Washington, DC: IEEE Computer Society.

Portele, C. (Ed). (2007). *OpenGIS® Geography Markup Languag (GML) Encoding Standard*. Open Geospatial Consortium Inc.

Preda, M., Choi, B.S., & Anh, M. (Eds.). (2013). *WD of the 2nd Edition of ISO/IEC 23000-13, Augmented Reality Application Format*. International Organization for Standardization.

Pu, Q., Gupta, S., Gollakota, S., & Patel, S. (2013). Whole-home Gesture Recognition using Wireless Signals. *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking* (pp. 27 – 38). New York, NY: ACM

Rabiner, L. (1989). A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2), 257 – 286.

Radu, I. & MacIntyre, B. Augmented-reality Scratch: A Children's Authoring Environment for Augmented-reality Experiences. *Proceedings of the 8th International Conference on Interaction Design and Children* (pp. 210 – 213). New York, NY: ACM.

Rautaray, S.S. & Agrawal, A. (2012). Vision based hand gesture recognition for human computer interaction: a survey. *Artificial Intelligence Review*, doi: 10.1007/s10462-012-9356-9

Reitmayr, G., Chiu, C., Kusternig, A., Kusternig, M., & Witzmann, H. (2005). iOrb - Unifying Command and 3D Input for Mobile Augmented Reality. *Proceedings of the IEEE VR Workshop on New Directions in 3D User Interface*s (pp. 7 - 10). Washington DC: IEEE Computer Society.

Ropinski, T., Wachenfeld, S., & Hinrichs, K. (2004). Virtual reflections for Augmented Reality Environments. *Proceedings of the 14th International Conference on Artificial Reality and Telexistence* (pp. 311 – 318). Berlin: Springer.

Rosenberg, L. (1992). The Use of Virtual Fixtures as Perceptual Overlays to Enhance Operator Performance in Remote Environments. Dayton, Ohio: Wright-Patterson Air Force Base.

Rublee, E., Rabaud, V., Konolige, K., & Bradski, G. (2011). ORB: An efficient alternative to SIFT or SURF. *Proceedings of the 2011 IEEE International Conference on Computer Vision (ICCV)* (pp. 2564 – 2571). Washington DC: IEEE Computer Society.

Sakairi, T., Palachi, E., Cohen, C., Hatsutori, Y., Shimizu, J., & Miyashita, H. (2013). Model Based Control System Design Using SysML, Simulink, and Computer Algebra System. *Journal of Control Science and Engineering*, Vol. 2013, Article ID 485380, 1 – 14.

San Augustin, J., Skovsgaard, H., Mollenbach, E., Barret, M., Tall, M., Hansen, D., & Hansen, J. (2010). Evaluation of a Low-Cost Open-Source Gaze Tracker. *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications* (pp. 77-80). New York: ACM.

Sashima, A., Izumi, N., & Kurumatani, K. (2005). Agents That Coordinate Web Services in Ubiquitous Computing. *Lecture Notes in Computer Science*, 3598, 131 - 145.

Schlette, C., Losch, D., & Rossmann, J. (2014). A Visual Programming Framework for Complex Robotic Systems in Micro-Optical Assembly.

Proceedings of the 41st International Symposium on Robotics (pp. 1 – 6). Berlin: VDE.

Schmalstieg, D., Fuhrmann, A., Hesina, G., Szalavári, Z., Encarnação, L. M., Gervautz, M., & Purgathofer, W. (2002). The Studierstube Augmented Reality Project. *Presence: Teleoperators and Virtual Environments*, 11(1), 33 - 54.

Schmalstieg, D., & Reitmayr, G. (2007). The World as a User Interface: Augmented Reality for Ubiquitous Computing. In G. Gartner, W. Cartwright, & P. P. Michael (Eds.), *Location Based Services and TeleCartography* (pp. 369-391). Berlin: Springer Berlin Heidelberg.

Signals & Slots. (n.d.). Retrieved 20 August 2014 from http://qt-project.org/doc/qt-5/signalsandslots.html

Singh, S., Puradkar, S., & Lee, Y. (2006). Ubiquitous computing: Connecting Pervasive Computing through Semantic Web. *Information Systems and e-Business Management*, 4(4), 421 - 439.

Sinha, S.N., Frahm, J., Pollefeys, M., & Genc, Y. (2011). Feature Tracking and Matching in Video Using Programmable Graphics Hardware. *Machine Vision and Appliances*, 22(1), 207 – 217.

Smith, R. (2007, May 28). *Open Dynamics Engine*. Retrieved January 27 2014 from http://www.ode.org/

Song, P., Winkler, S., & Tedjokusumo, J. (2007). A Tangible Game Interface Using Projector-Camera Systems. *Proceedings of the 12th International Conference on Human-computer Interaction: Interaction Platforms and Techniques* (pp. 956 - 965). Berlin: Springer-Verlag.

Soylu, A., & de Causmaecker, P. (2010). Ubiquitous Web for Ubiquitous Computing Environments: The Role of Embedded Semantics. *Journal of Mobile Multimedia*, 6(1), 26 - 48.

Studierstube project: Open Tracker (n.d.). Retrieved 20 August 2014 from http://studierstube.icg.tugraz.at/opentracker/

Supan, S., Stuppacher, I., & and Haller, M. (2006). Image Based Shadowing in Real-time Augmented Reality. *International Journal of Virtual Reality*, 5(3), 1 - 10.

Synapse's SNAP Network Operating System (n.d.). Retrieved 20 August 2014 from https://www.synapse-wireless.com/documents/whte_paper/Synapse-SNAP-OS-White-Papper.pdf

Synapse Wireless Drives Internet of Things Innovation (n.d.). Retrieved 20 August 2014 from http://harborresearch.com/wp-content/uploads/2014/04/HRI_Synapse-Paper_2014.pdf

Szeliski, R. (2011). Feature Detection and Matching. In *Computer Vision: Algorithms and Applications* (pp. 181 – 234). Berlin: Springer-Verlag.

Ullmer, B., & Ishii, H. (1997). The metaDESK: Models and Prototypes for Tangible User Interfaces. *Proceedings of the 10th Annual ACM Symposium on User interface Software and Technology* (pp. 223-232). New York, NY: ACM.

UPnP. (n.d.). Retrieved 20 August 2014 from http://www.upnp.org/

Wagner, D., Reitmayr, G., Mulloni, A., Drummond, T., & Schmalstieg, D. (2008). Pose Tracking from Natural Features on Mobile Phones. *Proceedings of the 7th IEEE/ACM International Symposium on Mixed*

and Augmented Reality (pp. 125 - 134). Washington, DC: IEEE Computer Society.

Wang, L., Gu, T., Chen, H., Tao, X., & Lu, J. (2010). Real-Time Activity Recognition in Wireless Body Sensor Networks: From Simple Gestures to Complex Activities. *Proceedings of the 16th International Conference on Embedded and Real-Time Computing Systems and Applications* (pp. 43 – 52). Washington, DC: IEEE Computer Society.

Weiser, M. (1991). The Computer for the 21st Century. *Scientific American*, 265(3), 94 - 104.

Willis, K., Poupyrev, I., Hudson, S., & Mahler, M. (2011). SideBySide: Ad-hoc Multi-user Interaction with Handheld Projectors. *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (pp. 431 - 440). New York, NY: ACM.

Wilson, A., & Sarin, R. (2007). BlueTable: Connecting Wireless Mobile Devices on Interactive Surfaces Using Vision-based Handshaking. *Proceedings of Graphics Interface 2007* (pp. 119 - 125). New York, NY: ACM.

Wilson, T. (Ed). (2008). *OGC® KML*. Open Geospatial Consortium Inc.

Winer, D. (2003). *XML-RPC Specification*. Retrieved 25 August 2014 from http://xmlrpc.scripting.com/spec.html

World Wide Web Consortium (n.d.). *Forms*. Retrieved 18 August 2014 from http://www.w3.org/TR/html4/interact/forms.html

Vanderdonckt, J., & Simarro, F. (2010). Generative Pattern-based Design of User Interfaces. *Proceedings of the 1st International Workshop on*

Pattern-Driven Engineering of Interactive Computing Systems (pp. 12 – 19). New York: ACM.

Vincent, T. & Laganiere, R. (2001). Detecting Planar Homographies in an Image Pair. *Proceedings of the 2nd International Symposium on Image and Signal Processing and Analysis* (pp. 182 – 187). Washington, DC: IEEE Computer Society.

Vyas, K.K., Paree, A., & Tiwari, S. (2013). Gesture Recognition and Control Part 3 – WiFi Oriented Gesture Control & its application. *International Journal on Recent and Innovation Trends in Computing and Communication*, 1(9), 682 – 685.

Yang, X. & Cheng, K. (2012). Accelerating SURF Detector on Mobile Devices. *Proceedings of the 20th ACM International Conference on Multimedia* (pp. 569 – 578). New York: ACM.

Yee, W. (2009). Potential Limitations of Multi-touch Gesture Vocabulary: Differentiation, Adoption, Fatigue. *Lecture Notes in Computer Science*, 5611, 291 – 300.

Yergeau, F. (2003). *UTF-8, a Transformation of Format of ISO 10646 (RFC 3629).* Retrieved 18 August 2014 from http://tools.ietf.org/html/rfc3629.

Zhan, Y., & Kuroda, T. (2014). Wearable sensor-based human activity recognition from environmental background sounds. *Journal of Ambient Intelligence and Humanized Computing*, 5(1), 77-89.

Zhu, J., Ong, S.K., & Nee, A.Y.C. (2013, June). An Authorable Context-Aware Augmented Reality System to Assist the Maintenance Technicians. *The*

International Journal of Advanced Manufacturing Technology, 66(9-12), 1699-1714.

ZigBee Specification Overview (n.d.). Retrieved 20 August 2014 from http://zigbee.org/Specifications/ZigBee/Overview.aspx