Collaborative Augmented Reality

Onur Mimaroğlu

Dissertation 2014

Erasmus Mundus MSc in Dependable Software Systems

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science

National University of Ireland, Maynooth

Co. Kildare, Ireland

A dissertation submitted in partial fulfilment

of the requirements for the

Erasmus Mundus MSc Dependable Software Systems

Head of Department: Dr Adam Winstanley

Supervisor: Dr John McDonald

June, 2014

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of the others save and to the extent that such work has been cited and acknowledged within the text of my work.

Onur Mimaroğlu

Acknowledgement

I would like to thank Dr. John McDonald and Dr. Rosemary Monahan from the National University of Ireland, Maynooth for all their help and support on this project.

Abstract

Over the past number of years augmented reality (AR) has become an increasingly pervasive as a consumer level technology. The principal drivers of its recent development has been the evolution of mobile and handheld devices, in conjunction with algorithms and techniques from fields such as 3D computer vision. Various commercial platforms and SDKs are now available that allow developers to quickly develop mobile AR apps requiring minimal understanding of the underlying technology.

Much of the focus to date, both in the research and commercial environment, has been on single user AR applications. Just as collaborative mobile applications have a demonstrated role in the increasing popularity of mobile devices, and we believe collaborative AR systems present a compelling use-case for AR technology.

The aim of this thesis is the development a mobile collaborative augmented reality framework. We identify the elements required in the design and implementation stages of collaborative AR applications. Our solution enables developers to easily create multi-user mobile AR applications in which the users can cooperatively interact with the real environment in real time. It increases the sense of collaborative spatial interaction without requiring complex infrastructure. Assuming the given low level communication and AR libraries have modular structures, the proposed approach is also modular and flexible enough to adapt to their requirements without requiring any major changes.

Contents

1. Introduction

We live in a physical world containing concrete and detectable objects with which we can interacted. With the help of technology, computers today are more capable of storing information about real world objects, as well as manipulating or modifying these objects. However, although programming paradigms, such as object-oriented approaches, have tried to represent and model the concept of real life objects, the gap between the real and virtual world has not yet been closed. What is needed is a new technology to enhance the user's sensory perception of the virtual information that they are seeing or interacting with in order to increase its connection to the real world: *Augmented Reality*. With the increasing processing power, more accurate sensors and high resolution cameras in mobile devices, augmented reality is no longer a science fictional concept as it was in Kubrick's *2001: A Space Odyssey*. The improvements of open source SDK's, furthermore, lead augmented reality to move from industrial niches to mass technology (Yang, 2011). Many commercial and off-the-shelf systems and applications now exist that permit the developers of collaborative augmented reality applications.

1.1. Augmented Reality

Augmented Reality (AR) is closely related to Virtual Reality (VR), since the concept of AR evolved as a variation of VR (Milgram, et.al, 1994). VR aims to present an artificial world that the user can explore interactively. Augmented Reality, on the other hand, supplements the real world with computer generated content, instead of creating an entirely virtual world.

Figure 1. *Reality-Virtuality (RV) Continuum (Milgram, et al., 1994)*

AR has been defined by Milgram as existing on a continuum of real to virtual environments, and is thus a subset of mixed reality (Figure 1).

The left extreme on the continuum is the real world, or the physical environment; which we all are familiar with. The opposite extreme is the virtual world, or virtual environment, in which all information is solely computer generated and nothing to do with real world objects. AR and VR exist between these two extremes. Augmented reality is often used to refer to manipulation of the real world, viewed through head mounted or mobile devices, with the additional information generated by the computer. An AR system adds or superimposes modifiable digital information directly on top of items in the real world.

Although it is a relatively new concept, the origins of augmented reality date back to the late 1950s, when Morton Leonard Heilig developed the *Sensorama,* a virtual reality based arcade game which gave the player the sense of real world interaction.

In 1966, Professor Ivan Sutherland of Electrical Engineering at Harvard University invented the Head-Mounted Display (Figure 2), which was a milestone in making AR a usable possibility (Sutherland, 1968).

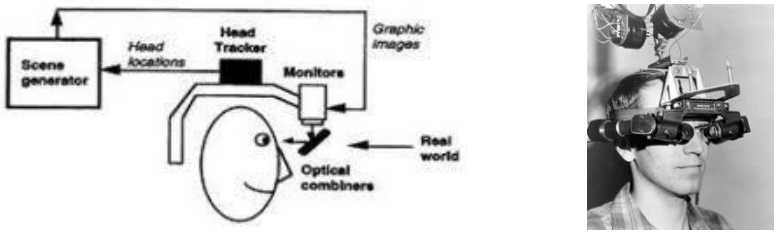


Figure 2. *Head Mounted Display (Azuma, 1995)*

It is believed that, Tom Caudell first coined the term *"Augmented Reality"* in 1990 while working in Boeing's Computer Services' Adaptive Neural Systems Research and Development in Seattle (Caudell, 1992). The complex software he came up with helped mechanics during assembly stages by showing the correct positions of where cables were supposed to go.

Despite all the improvements, AR remained pretty much a curiosity of the research domains until 1999, when Hirokazu Kato of the Nara Institute of Science and Technology released the ARToolKit[1] to the open source community. Together with the invention of smartphones and handheld devices with sensors and cameras, ARToolKit was what helped to bring AR to the masses and end users. Nowadays, numerous augmented reality applications have been released to the general public, mostly on Android and iOS platforms, thanks to open source and commercial AR tools and libraries developed by companies such as Qualcomm[2], Metaio[3] and ARMedia[4].

1.2. How does Augmented Reality work?

An augmented reality system basically consists of a camera, a processor and a display unit. The camera captures an image, and then AR software system superimposes virtual objects onto the image in real time and displays the result (Siltanen, 2012).

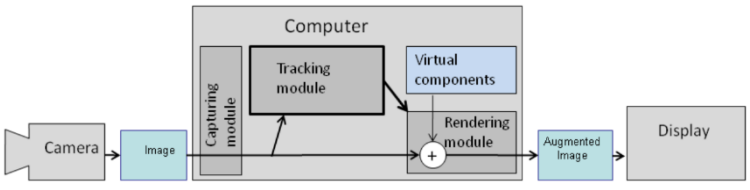


Figure 3. *Simple AR System (Siltanen, 2012)*

[1] http://www.hitl.washington.edu/artoolkit
[2] http://www.qualcomm.com/solutions/augmented-reality
[3] http://www.metaio.com
[4] http://www.armedia.it

As shown in the Figure 3, the current image viewed by the camera is captured by the image acquisition module. The tracker calculates the correct location and orientation for virtual overlay relative to the camera. Finally, the renderer puts the virtual image on top of the real image using the calculated pose and renders the augmented image onto the display.

The critical component here is the tracker, since its role is to compute the relative pose of the camera in real time. The term *pose* refers to the six degrees of freedom (DOF) position and orientation, as we will see it in detail later. As Sanni says, *"the fundamental difference compared to other image processing tools is that in augmented reality virtual objects are moved and rotated in 3D coordinates instead of 2D image coordinates"* (Sanni, 2012).

1.3. Types of AR

1.3.1. Marker Based Tracking

The most common type of augmented reality is marker based tracking, because calculating the pose of the camera, referred to as camera tracking, by using a complex marker is relatively simple.

As previously mentioned, an augmented reality systems displays additional information on top of real objects. To be able to do this, it has to know where the user, or camera, is and what the camera is looking at. Hence, computation of location and orientation of the camera relative to the real world is vital. In an unfamiliar environment, computing the pose is difficult as it takes some time to collect enough data. Adding an easily detectable marker in the real world makes the estimation easier. A marker is a sign, like QR code, which can be detected by a computer using image processing techniques. Once the marker is detected, the marker's pose defines the correct position and orientation of the camera. In other words, the marker is used as a reference point for computer generated graphics to be overlaid.

Figure 4. *Different Markers*

A good marker should be easily detectable. According to Hartley and Zisserman, differences in brightness are more easily detected than the differences in colour. In addition, four known points are sufficient to determine the pose of a camera (Hartley & Zisserman, 2000) and the simplest shape with four points is a square. That is the reason, most of the marker based systems use black and white square markers. In spite of its

advantages, such as easy implementation and faster detection, it is not suitable for all types of AR as it requires predefined markers to be placed within the scene. In some situations, for example, where we wish to augment a building, the lack of a predefined marker complicates the process.

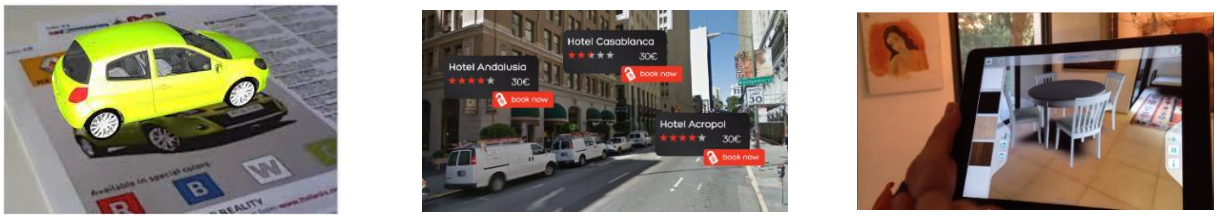


Figure 5. *A Multimedia marker (left), Location Based AR (middle), Markerless AR (right)*

1.3.2. Markerless Tracking

As its name suggests, markerless AR does not require a predefined marker to place the virtual objects with the real world. There are numerous subtypes of Markerless AR.

Location based tracking, for example, is one of them. With the help of a triangulation technology, such as an integrated GPS sensor and a digital compass in a mobile phone, location based AR helps to determine the current location and then displays some kind of additional information on the screen according to the location and position of the sensor.

Instead of placing a two dimensional predefined marker in the environment, a specific object can be modelled and stored in a cloud. Using image processing techniques (3D tracking), an AR system can search for the modelled object in the real world viewed through camera by comparing every frame with the existing object online.

Using advanced tracking algorithms is another way of providing markerless augmented reality. For example Pang suggests the following approach (Pang, et.al, 2006):

1. Kanade-Lucas-Tomasi (KLT) natural feature tracker is used to track corresponding points in two control images.
2. Four planar points are specified in each control image to setup the world coordinate system.
3. The affine reconstruction and reprojection techniques are used to estimate the image projections of these four specified planar points in the live video sequence.
4. These image projections are used to estimate the camera pose in real time.

In markerless tracking technique, a camera captures video frames and features in each frame are analysed. Then, correspondences between consecutive frames are found. The technique computes the camera's pose using these correlations. The features that has not been detected so far are stored and their three dimensional coordinates are computed. The system stores these new features to benefit from them for upcoming correspondence detections (Ziegler, 2009).

Even though markerless tracking allows more complex application of AR and thus gives more flexibility to the user, it is difficult to implement and requires a considerable amount of computational power because of complex calculations.

In this project, we use a hybrid approach in that we do not require a marker, but instead allows the user to define their own marker at runtime. A critical constraint of the system is that the marker region is assumed to be planar, and should be textured (i.e. to facilitate robust tracking). Once the marker region has been defined by the user it forms a planar map, and therefore the rest works similarly to the marker based tracking.

1.4. Our Motivation

"It is a human need to share and shape the world surrounding us", says Kasahara. (Kasahara, et.al, 2012). Several physiological studies have shown that the social interaction involved with other people constitutes a large motivational factor for gaming (Järvelä, et.al, 2014). Therefore, the need for synchronization (i.e., mutual variation) of interactions and sharing information can be considered as a human need. Sharing photos on social media platforms, such as Instagram, Twitter and Facebook is a good example of this need.

From this point of view, it can be said that, although single user augmented reality applications have shown great promise thus far; perhaps the greatest potential use for augmented reality is in developing new types of collaborative interfaces (Billinghurst & Kato, 2002). Augmented reality applications can be used to improve remote collaboration in a shared space.

Although there are currently some multi-user augmented reality applications in the market, most of these systems are based on complex infrastructure, such as optical motion capture systems, or predefined markers as mentioned above. These limitations are the main reasons why augmented reality is still considered as a niche technology.

Vicon[5] and OptiTrack[6], for instance, are both infrared marker tracking systems, which may be integrated into a collaborative augmented reality application (CAR). However, they are not suitable for everyday setup and are very costly. StudierStube[7], on the other hand, is an augmented reality interface which introduces a face-to-face collaboration in a shared physical workspace. However, it requires head mounted displays to allow users to collaboratively view 3D virtual models superimposed on the real world (Billinghurst & Kato, 2002). Since it is an old project, which was stopped in 2008, it is not supported anymore. Additionally, the native programming languages it uses, complicated setup and implementation details prevent it from being a suitable candidate for the rapid prototyping and mobile applications.

In this project, we address the following questions: *What elements are required in the design and implementation on collaborative augmented reality applications? How can we*

[5] http://www.vicon.com
[6] http://www.naturalpoint.com/optitrack
[7] http://studierstube.icg.tugraz.at

develop a collaborative augmented reality framework such that it allows developers to easily create multi-user mobile AR applications in which the users can cooperatively interact with the real environment in real time? How can we increase the sense of collaborative spatial interaction in a shared workspace without relying on complex infrastructure? How can our framework be flexible enough to adopt any type of AR engines and communication libraries when a replacement is needed? As such our aim is to create a framework that sits on top of existing augmented reality and communication frameworks, and makes use of their common behaviours.

Collaborative AR applications which can be developed using our framework will be mostly for entertainment purposes, such as games, shopping or social applications. Yet, we have identified a few other fields that can benefit from our CAR framework:

- **Advertisement:** Augmented Reality systems have been popular for commercial purposes since it was born. Commercial interest in AR is even increasing as AR has the power to bring consumers closer to products before purchase. Our framework suggests to use the real environment as a shared canvas, without any need for a complex setup. This can mean putting signatures, banners, product pictures or even live videos on trees or walls or wherever is suitable. It could dramatically change the advertising methods.
- **Education:** Smart boards have become a more popular way of teaching at schools (Lee K. , 2008). With the help of collaborative applications, a smart phone can work as a master which sends the information to students' tablets. Any student who has the right to manipulate the canvas can draw on his/her device and it simultaneously appears on the smart board.
- **Architecture and Design:** Augmented Reality has been widely used in video gaming and media entertainment to date. Architecture is another field that can benefit from the creative features of collaborative AR. Architecture and design firms can use it to show clients proposed designs in a collaborative way using mobile devices such that clients can make alterations on the proposed design in line with their requirements.

The rest of this paper is organized as follows: Section 2 gives an overview of the related work to our problem; more specifically we introduce previous attempts to integrate collaboration into augmented reality applications, although such attempts have been infrequent. Next, in Section 3 and in Section 4, we explain the techniques of pose tracking and content generation respectively in single user AR systems. Section 5 discusses how well known content distribution methods can be adapted into AR applications. These three sections together give the necessary background for our solution. Later in Section 6, we present our collaborative augmented reality solution together with some implementation details. Section 7 critically evaluates the solution in terms of its modularity, dependability and reliability and addresses its limitations. Finally, in Section 8 conclusions are drawn with an outlook into future research directions.

2. Related Work

In this chapter we critically analyse and refer to the other works that are related to collaborative augmented reality and compare our approach with theirs.

Recently, AR applications have become more popular due to the advancement in mobile device technology (Vuforia[8], Metaio[9], Layar[10], etc.). These mobile based applications allow programmers to apply augmented reality in different domains. Metaio and Layar allow users to show computer generated content on top of printed media. However, they both require a trackable printed surface to work. Scrawl[11] is a marker-based 3D augmented reality painting app by String. It requires users to print out markers to create interactive AR applications. Stiktu[12] allows users to scan everyday objects and virtually add AR stickers and images on surfaces. The created content may be captured as an image and posted to social networking platforms such as Twitter and Facebook (Kasahara, et.al, 2012). Wikitude[13] and Mixare[14] use sensors for location based AR. None of these examples support shared workspace for multiple users.

As mentioned in the previous chapter, although AR applications have been widely explored and applied in many domains to date, there has not been much work done in the collaborative AR field. We show that SecondSurface, developed by MIT Media Lab, is the closest and most comparable work to our solution. Studierstube, as described by Schmalstieg and others, was one of the first collaborative augmented reality systems (Schmalstieg, et al., 2002). Since our solution separates the collaboration module, the content generation module, and the AR engine; we will briefly look at some other works that embody peer-to-peer content distribution and mobile content generation.

2.1. SecondSurface

In 2012, MIT Media Lab introduced a multi user augmented reality system that takes real time interaction further by superimposing user generated contents on the physical environment (Kasahara, et al., 2012). The system consists of a mobile application and a server application. The mobile application runs on iOS and the server application runs on a Linux based machine.

Very similar to our approach, there is a shared spatial canvas on which users can create contents and share with each other. What differentiates our solution from theirs is that in their structure, the multiple iOS devices are connected to each other through a server application. When a user generates a new content on top of the canvas, the data (content itself and its pose matrix $\boldsymbol{M}_{\textit{virtual object}}$) is sent to the server (Figure 6). The server handles

[8] http://www.qualcomm.com/solutions/augmented-reality
[9] http://www.metaio.com
[10] https://www.layar.com/augmented-reality
[11] http://www.poweredbystring.com/showcase
[12] http://blog.stiktu.com
[13] http://www.wikitude.com
[14] http://www.mixare.org

the data and pushes it to all devices connected to each other. If any of the other devices are looking at the same object, the generated content appears on their screen.

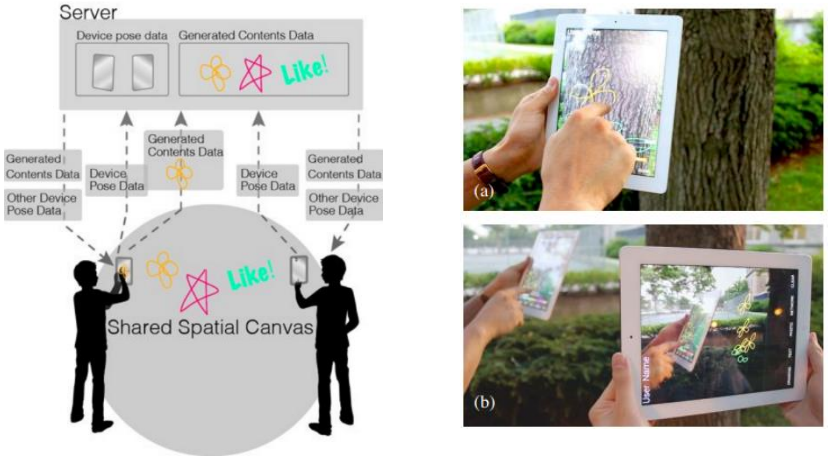


Figure 6. *SecondSurface Server System (left), Content Generation (right) (Kasahara, et al., 2012)*

SecondSurface[15] has been especially influential to our solution choice. It was the starting point of this project. As this project does, it also uses image based AR engine Vuforia, developed by Qualcomm, which recognizes a natural image as a target object with advanced registered dictionary data (Kasahara, et al, 2012). However, there are some major differences with our approach.

First of all, SecondSurface is an application rather than a framework that can be used to develop different collaborative AR applications. That means if they want to change the AR engine working under SecondSurface or the server design (for example, in order to allow additional functionalities such as sensor support for location based AR), most of the implementation has to be changed. By developing a flexible collaborative AR framework, this project aims to provide an easier way to create applications similar to SecondSurface, independently from the AR engine or content distribution design.

Secondly, using a server application to handle and distribute the generated content is a design choice here. For our project, it would double the time to share the content with other peers and increase the overall complexity of the framework. With the help of computational power of mobile devices today, our solution prefers to rely on each peer, instead of a server.

We note that it could be advantageous to integrate the server application into our system. Since the data would be stored in the server, it would provide a persistent layer. This means when a user views an object through his/her device's camera, all previous contents (i.e. information about that object) would appear on the object. By adapting peer to peer communication in our system, we lose the content when the current session between peers expires in favour of simplicity and faster distribution.

[15] http://tangible.media.mit.edu/project/second-surface

2.2. Studierstube

Studierstube[16] is a long running augmented reality research project carried out at Graz University of Technology. The maintenance of the project ceased in 2008. It is one of the first AR projects that allows multiple users to share the same virtual environment.

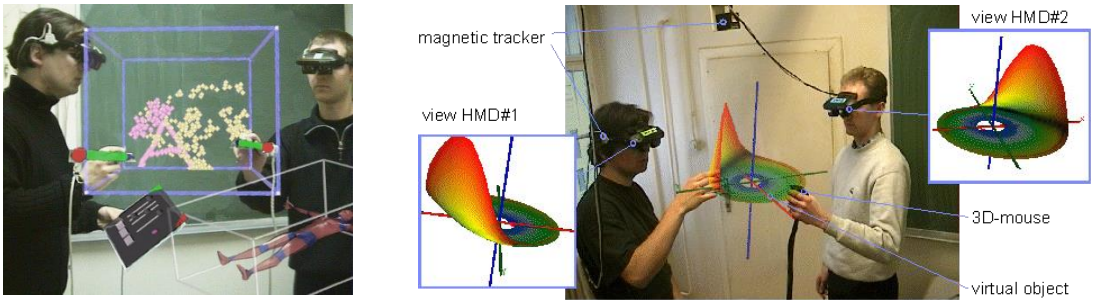


Figure 7. *Collaborative work in Studierstube: 3D painting application window (Billinghurst & Kato, 2002)*

At the heart of the Studierstube system, collaborative AR is used to embed computer generated images into a real world environment. Studierstube uses display technologies such as see through head mounted displays (HMDs), a virtual table (which could be interacted with a pen and a panel), or projection screens to combine computer graphics with a user's view of the real world (Schmalstieg, et. al., 2002). Differently from our solution, the project focuses on face to face collaboration rather than supporting remote collaboration. Another limitation of the project is that it is only available for Windows and Linux platforms and gives no support for handheld devices and mobile operating systems, such as Android and iOS.

It gives support to many different markers (template markers, data-matrix markers, frame markers, split markers, grid markers), provides high performance (twice as fast as ARToolKit) and low memory consumption (5% of the memory usage of ARToolKit). However, despite its advantages, there are also some limitations. It does not read images from a camera, render anything, support hardware or track natural features. It only supports Local Area Network (LAN) communication between peers. Studierstube, furthermore, uses many different libraries and frameworks, such as ARToolKit as its core AR engine, QT[17] as its GUI toolkit, OpenVideo[18] as its video abstraction library and OpenTracker[19] to track the input data. It therefore requires complex configuration settings and high coding skills. This also prevents it from becoming a portable framework as it depends on too many external libraries and requires special settings. In contrast to Studierstube, our project makes use of the mobile devices that encompass all of the above (camera, touch screen, wireless networking) in one solution.

[16] http://studierstube.icg.tugraz.at
[17] http://qt-project.org
[18] http://www.open-video.org
[19] http://studierstube.icg.tugraz.at/opentracker

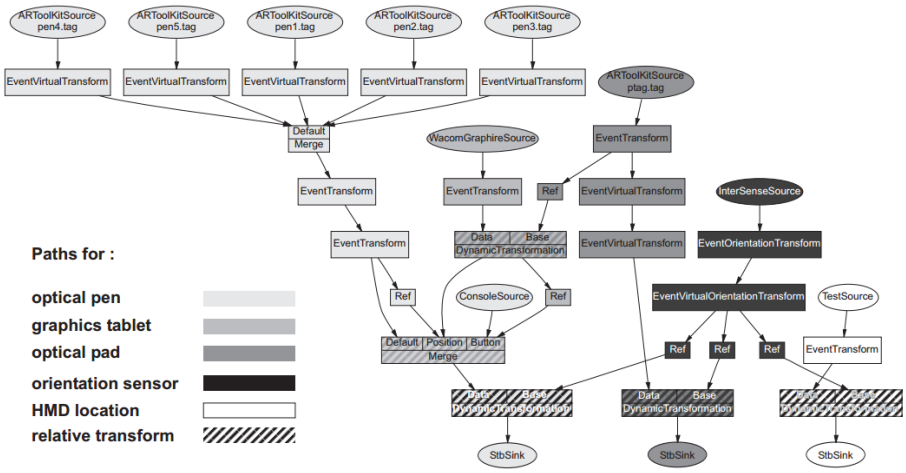


Figure 8. *The data flow graph of the tracking configuration for Studierstube AR setup (Schmalsteig & Reitmayr, 2005)*

2.3. Content Distribution in Mobile AR Systems

Content sharing is one of the key points that makes mobile devices so popular. Countless numbers of applications allow users to share information over a network. Social networking applications, such as Facebook, Twitter and Whatsapp, are good examples of content distribution. However, most of them provide this communication through servers. Whatsapp, for example, is an instant messenger application which uses a client-server model to provide connection between devices. A new text is created by a device and sent to the server, then the server handles the message and pushes it to the other peers. TextHer, on the other hand, is a peer to peer messaging application where messages are directly sent from one device to another without being routed through servers.

Muddleware is a networking solution for mobile devices, particularly for augmented reality applications (Wagner & Schmalstieg, 2007). There is a high speed XML server at its core which can handle large amount of queries. Muddleware uses XML DOM and XPath to support rapid prototyping.

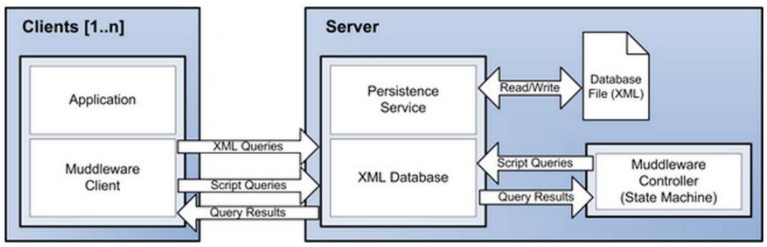


Figure 9. *Muddleware Components (Wagner & Schmalstieg, 2007)*

We want our solution to be independent from a server for the sake of overall simplicity of the framework, faster content sharing and to support more ad-hoc configuration. Our solution is based on Android devices. In Android OS there is not a default peer to peer

communication support. Android Wi-Fi P2P uses Wi-Fi Direct which is not the technology we want as it allows to send only a limited amount of data, and only primitive data types. Another limitation of Wi-Fi Direct is that multiple connections are not allowed. That means if a device is connected to a session, it cannot connect to another one until it leaves the current session. WifiShoot and SuperBeam are applications that use Wi-Fi Direct and allow users to share files between Android devices.

There are currently a few communication libraries for Android. Sip2Peer[20] is an open source framework that can interact with a bootstrap server to receive a list of active peers and exchange ping messages with them. It is a heterogeneous peer to peer system that is available for mobile devices.

PeerDroid[21] is another tool that is a porting of JXME, a P2P infrastructure, protocol to Android platform. It allows developers to create applications for Android that uses the features of JXTA (Juxtapose, an open source P2P protocol) system interacting with other mobile terminals and other traditional peers, such as PC (Farber & Picone, 2010).

AllJoyn[22] is an open source project developed by Qualcomm, which provides a universal framework that enables communication among connected products. It is a cross platform product that is compatible with Windows, Windows Mobile, OS X, iOS, Linux and Android. AllJoyn allows for proximity peer to peer over various transports. It is mainly written in C++ and provides multiple language bindings.

2.4. Content Generation

Content generation is a major topic for content rich applications. Large applications require smoother graphics, while simpler graphics would be enough for small applications. Especially for AR applications, content creation is a necessary part of the development and testing life cycle but doesn't have to be a visual feast for users (Wagner, 2007).

There are two types of content generation tools: creating/using professional tools and creating/manipulating predefined content in AR runtime. Virtual Reality Modelling Language (VRML) is a powerful 3D interactive representation language and is one of the most widely used graphic formats. Many AR and VR research projects still use VRML.

MARS[23] (Mobile Augmented Reality Systems) and DART[24] (Designers Augmented Reality Toolkit) (MacIntyre, et al., 2003) target graphic artists and designers, more than programmers. These types of graphic tools offer a timeline on which a designer can create non-linear content. The Authoring Mixed Reality (AMIRE) also does not require programming. Graphical designers can create and test AR scenes because a change instantly appears on the screen. APRIL is an XML based scripting language which uses the aforementioned Studierstube library and adds high level concepts on top of it (Wagner,

[20] https://code.google.com/p/sip2peer
[21] https://code.google.com/p/peerdroid
[22] https://www.alljoyn.org
[23] http://monet.cs.columbia.edu/projects/mars/mars
[24] http://ael.gatech.edu/dart

2007). Augment[25] is a commercial product that allows users to place pre-generated virtual objects on top of an AR scene in real time. These objects, then, can be manipulated with a touch gesture. Unity 3D can also be a good content generation tool with the help of an AR framework. Vuforia, for example, supports Unity 3D integration. Developers can create a 3D world and augment it with the model he/she defines. Changes immediately appear on the panel. It is also quite easy to create static AR markers with Unity 3D.

2.5. Conclusion

In this chapter, we have identified and critically analysed the existing work which this thesis touches. Since the project combines different areas of AR, only the most relevant aspects have been examined here. In Section 2.1 we discussed in detail MIT Media Lab's project SecondSurface, which is very close to our solution in terms of pose tracking and content creation. We showed that what differs with their solution to ours is the content distribution approach. In our project we propose to extend their work by creating a framework and by sharing the content in ad hoc manner. In Section 2.2 a long term project Studierstube has been examined. We then showed some of the content distribution methods on mobile devices in Section 2.3 and highlighted some popular content creation tools used in augmented reality applications in Section 2.4.

We believe that there is an increasing popularity of mobile collaboration among end users and existing collaborative tools and applications in the AR field are not enough to make use of the increasing capabilities of handheld devices. Collaborative mobile augmented reality is now more suitable for everyday usage by regular users and existing projects cannot target end users since most of them mandate special setup.

[25] http://augmentedev.com

3. Image Tracking

Augmented reality on mobile devices requires accurate 6DOF pose tracking of real world objects. 6DOF (Six degrees of freedom) refers to the freedom of movement of a rigid object in three dimensional space[26]. Specifically, the body is free to move forward/backward, up/down, left/right combined with rotation about three perpendicular axes. Pose tracking, especially for mobile devices, must not have high requirements and be able to adapt to different conditions.

A single camera mounted on a mobile devices is usually enough for mobile augmented reality configuration. The video stream from the camera is simultaneously used as a video background and for pose tracking of the camera relative to the environment. This inside out pose tracking needs to be executed in real-time with the limited computational resources of a mobile device (Wagner & Schmalstieg, 2009).

In order to achieve robustness and performance, tracking rectangular fiducial markers is a common approach. Vuforia is a marker tracking library that we use in our implementation as the AR engine. The reason we choose Vuforia is that it supports a wide range of mobile platforms.

Before tracking can be started, the camera on the device must be calibrated and passed to Vuforia so that an OpenGL projection matrix can be returned by Vuforia. We implemented our solution on Android. Beginning from Ice Cream Sandwich (API Level 14), Android provides a large set of camera calibration features.

```
CameraCalibration camCal = CameraDevice.getInstance().getCameraCalibration();
Matrix44F mProjectionMatrix = Tool.getProjectionGL(camCal, 10.0f, 5000.0f);
```

Camera calibration is passed to Vuforia API using the above code. The function **`getProjectionGL`** of the **`Tool`** class returns an OpenGL style projection matrix to be used later to multiply with the pose of detected trackable. The reason we need the projection matrix is that a smartphone's screen is a 2D surface. A 3D scene rendered by OpenGL must be projected onto the device's screen as a 2D image. **`GLProjectionMatrix`** is used for this transformation.

Other parameters of this function are respectively near plane and far plane as shown in the figure 9. The viewing frustum is the region of space in the modelled world that may appear on the screen. The goal of view frustum culling is to identify what is inside the frustum and what is not. Only the objects inside the frustum are drawn on the screen. Near and far planes are the boundaries of this frustum.

[26] http://en.wikipedia.org/wiki/Six_degrees_of_freedom

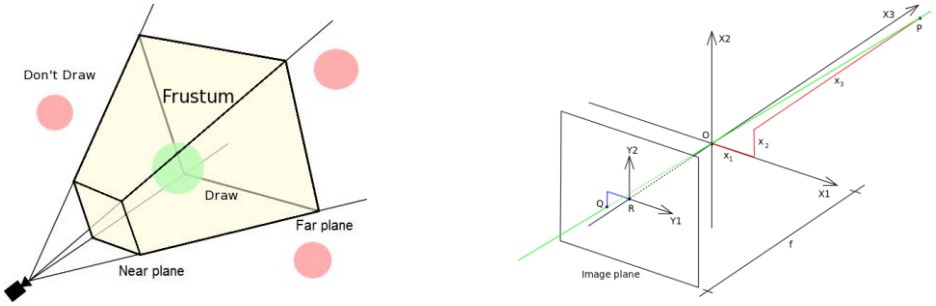


Figure 10. *Viewing Frustum (left), Geometry of Pinhole Camera (right)*

The `Matrix44F` in the above code snippet contains not only the perspective projection matrix but also focal length and principal point of the camera. As shown in the figure 10[27], the three axes of the coordinate system are X_1, X_2 and X_3. Axis X_3 is pointing in the viewing direction of the camera and is the principal axis. The point R at the intersection of the optical axis and the image plane is the principal point. Image plane is located at distance f from the origin in the negative direction and it is called the focal length.

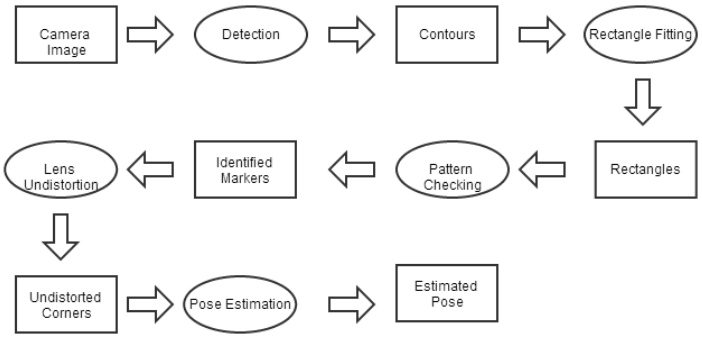


Figure 11. *Tracking Process*

After the camera is calibrated, the above tracking pipeline is executed for every new camera frame and gives an estimated pose, if a marker is detected. Despite the fact that Vuforia is partially an open source library, it is a bit like black box in the tracking side and doesn't reveal much how its tracking modules work. Therefore, the figure 11 shows a general tracking pipeline, not specific to Vuforia.

For image based tracking, fiducials are the real world objects that are used by the tracking system. Hence, first we need to detect these fiducials (Figure 12). There are various fiducial detection algorithms, but fundamentally they all work similarly. For example, Ababsa and Mallem (Ababsa & Mallem, 2008) present a robust circular fiducial detection technique that is based on edge following. The algorithm searches pixel by pixel for edges. To define an edge, they use constant thresholding which means less bright pixels than a certain threshold are considered as dark, and brighter ones as bright. If a dark-to-bright sequence is detected, this sequence is a candidate for a border. When the sequence closes a loop, borders are detected. Otherwise, the edge is eliminated.

[27] http://en.wikipedia.org/wiki/Pinhole_camera_model

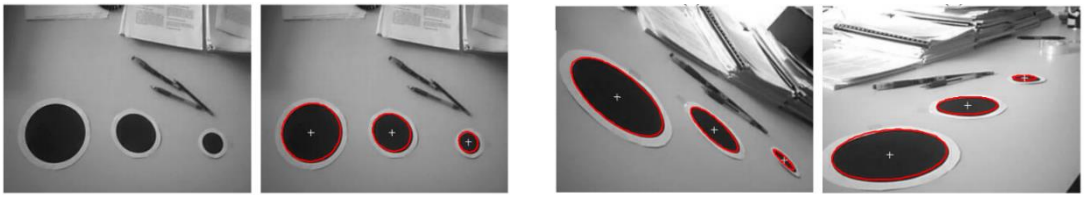


Figure 12. *Fiducial Detection*

Despite the fact that Vuforia makes an important contribution in terms of tracking, for a marker to be able to detectable later on, the marker should have high number of features. Vuforia defines a term augmentable rating which means how well an image can be detected and tracked using the Vuforia SDK.

Figure 13. *Marker Features*

For instance, the left image in the Figure 13 contains only two features for each sharp corner, since soft corners are not considered as features. The middle image contains no features as it contains no strong corners. The right image contains four features.

Figure 14. *Amount of Features in Markers*

Local contrast of the marker (Figure 15) is another feature that needs to be strong enough to be detected and tracked properly.

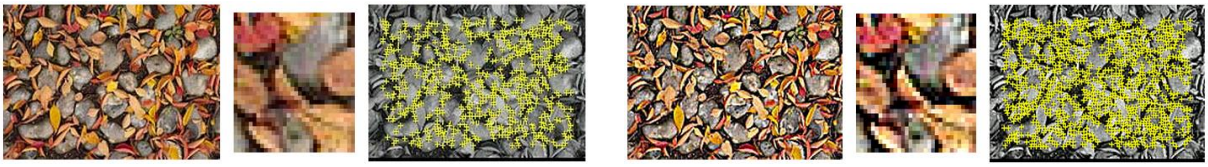


Figure 15. *Local Contrast of Markers*

Features should be evenly distributed, as shown in the Figure 16. The more balanced the distribution of the features in the image, the better the image can be detected and tracked.

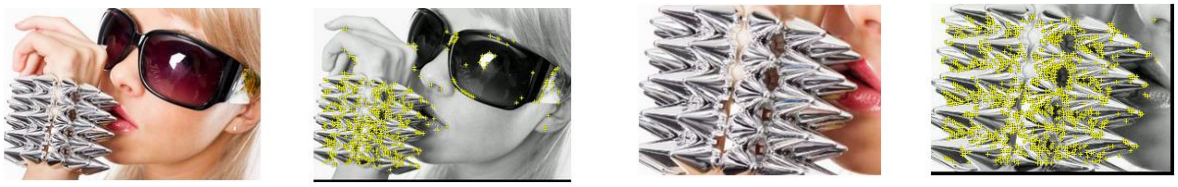


Figure 16. *Feature Distribution in Markers*

Even if an image contains high number of features and strong contrast, repetitive patterns, referred to as aliasing, may make the tracking harder since recurring features show no unique pattern to detect (Figure 17).

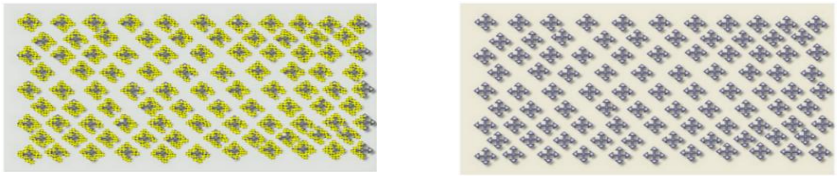


Figure 17. *Repetitive Features in Markers*

After fiducial detection step, what we have are closed polygons. However, they need to be rectangles to form a marker. In the next step, rectangles are detected using contours. For the pattern checking step, Vuforia provides three types of trackers (ImageTracker, MarkerTracker and TextTracker) which are extended from the base class Tracker.

```
State state = Renderer.getInstance().begin(); // get the current state from Vuforia
if(state.getNumTrackableResults() > 0) // contains a recently detected pattern
{
	// pattern detected, do something, render etc.
}
Renderer.getInstance().end(); // mark the end of the section (finish capturing)
```

The above code snippet is an example of pattern detection in Vuforia. For every camera frame, the `begin` function of `Renderer` class is invoked which returns a `State` object. The `State` shows the trackable objects currently being tracked by the tracker (i.e. ImageTracker). When we detect the pattern we are looking for, the final step before rendering is estimating the pose relative to the marker. Similar to fiducial detection and pattern checking, there are various pose estimation algorithms, which all use the same basic concept: First an initial guess is created that estimates the trackable's approximate position and orientation with respect to the camera. Next, this first estimate is refined iteratively until specific quality criteria are met or the maximum number of iterations is reached (Wagner, 2007).

```
State state = Renderer.getInstance().begin();
if(state.getNumTrackableResults() > 0)
{
	// estimate the pose
	TrackableResult tResult = state.getTrackableResult(0);
	Matrix44F modelViewMatrix =
	Tool.convertPose2GLMatrix(tResult.getPose());
	float[] modelViewProjection = modelViewMatrix.getData()
	// now render using modelViewProjection
}
Renderer.getInstance().end();
```

With Vuforia this can be done using the above code snippet. Vuforia estimates the pose for us so that we can get it with `getPose` function. The resulting matrix representation is then converted to an OpenGL matrix in order to be able to render the model into the user's view.

In this chapter, we discussed how tracking of fiducial trackables work with some example codes of Vuforia AR engine. We presented, by referring to Vuforia, the tracking pipeline of augmented reality applications in detail, camera calibration needed to start a proper AR session, fiducial marker creation, pattern identification and pose estimation of detected trackables. Tracking is the fundamental step for all augmented reality applications. Considering the limited power of mobile devices, Vuforia thus makes an important contribution for AR systems running on mobile platforms.

4. Rendering and Content Generation

Even though augmented reality systems does not focus too much on visuals, augmentation of reality is only possible with additional graphics. This makes rendering another important aspect of augmented reality.

In this chapter we outline the differences between software and hardware rendering, present the existing solutions for 2D and 3D graphics and then discuss how an augmented reality related content can be created with graphics libraries by showing example codes from our implementation.

4.1. Introduction

Computer graphics researchers mostly deal with low level graphic libraries such as OpenGL. The existing built in 3D graphic libraries on mobile platforms, such as Android Graphics, cannot usually meet the requirements of augmented reality systems. OpenGL ES and Direct 3D mobile has become indispensable for 3D applications on mobile platforms (Android, iOS and Windows Mobile respectively). They are the only effective low level graphics libraries as of yet.

Another solution for rendering on a mobile device is remote rendering. In remote rendering, a server with computationally high power undertakes the image generation task and sends final model to the mobile device (Wagner, 2007). Lamberti et al, for instance, proposed a system where a cluster of PCs is able to handle remote visualization sessions based on MPEG video streaming involving complex 3D models. The proposed framework allowed mobile devices such as smart phones to visualize objects consisting of millions of textured polygons at the server side and on multimedia capabilities at the client side (Lamberti & Sanna, 2007), assuming a very fast network connection. This is of course not very practical under low quality connection conditions. This thesis therefore focuses on native rendering.

4.2. Software vs Hardware Rendering

Rendering refers to the process of generating computer aided images with the help of computer programs. This can be performed through hardware or software rendering. Software rendering takes place exclusively via computer code or applications. Hardware rendering is performed through a computer chip that returns the images directly to the screen.

Software rendering is processed without the help of any kind of hardware, and performed in the CPU, whilst hardware rendering relies on a graphical unit (GPU). Software rendering is slower than hardware rendering (Figure 18) because GPU is a specialized computing architecture designed from the ground up with rendering in mind and it can process large data blocks in parallel, which is crucial for algorithms used in computer graphics. Ideally, software rendering algorithms should be translatable directly to

hardware. However, this is not possible because hardware and software rendering use two very different approaches. In more detail:

- Software rendering holds a 3D scene to be rendered, or some relevant sections of it, in memory, and samples it pixel by pixel. In other words, the scene is static and always present, but the renderer deals with one pixel at a time.
- Hardware rendering works the other way around. All pixels are present at all times, but the rendering sees the scene one triangle at a time, loading each one into the frame buffer. The hardware has no notion of other objects, only one triangle is known at a certain time.

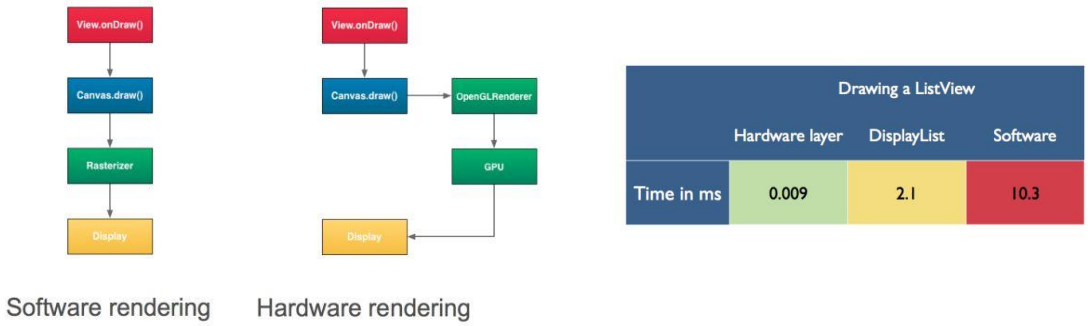


Figure 18. *Software vs. Hardware Rendering on Android (Romain Guy, Android Accelerated Rendering Google I/O, 2011)*

Since our implementation is based on Android, we had to use the features offered by the Android platform. Beginning in Android 3.0 (API level 11), the Android rendering pipeline supports hardware acceleration. This means that all drawing operations that are operated on a canvas use the GPU. Because of the increased resources required that we mention above, applications with enabled hardware acceleration will consume more memory.

4.3. OpenGL and OpenGL ES

In low level rendering, the developer has full control over the process (Wagner, 2007). Objects are defined as a set of vertices and redrawn on each frame. Whatever the rendering system is, this approach is the lowest level. OpenGL and Direct3D Mobile are the most widely used low level graphics libraries. Direct3D Mobile is only available for Windows Mobile platform, it is thus not our main interest.

Through OpenGL, 2D and 3D graphics can be created using a GPU. OpenGL benefits from the so called graphics pipeline, shown in the figure 19, to convert primitives (points, lines, etc.) into pixels. The main idea behind the pipeline is the following. First a number of vertices are entered from the left, loading them into the pipeline. Then, several intermediate steps are performed. At the end of the pipeline, we obtain the image we want.

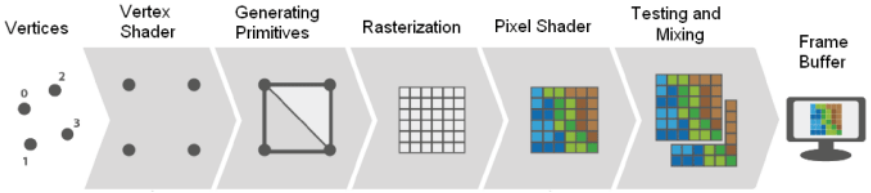


Figure 19. *OpenGL 2.0 Rendering Pipeline (taken from www.loria.fr)*

Vertex shader and Pixel shader are the only steps a developer can program in the pipeline. The rest of the steps in pipeline are done automatically. The shaders basically take the data to the GPU and tell it what computations would be carried out.

OpenGL ES (OpenGL for Embedded Systems) is a subset of OpenGL. Even though it offers most of OpenGL's functionality, it was designed to support embedded systems, such as smart phones, tablets, PDA's, and video game consoles, therefore it is more lightweight than its ancestor. For example, each function that it provides can be directly mapped to the underlying implementation. That simplifies the driver development and reduces the driver's code size (Lee & Baek, 2009). Moreover, some redundancies of OpenGL have been removed.

There are wide range of higher level graphics libraries working on top of OpenGL ES, such as libGDX[28] which is a very popular and up to date cross platform Java graphics framework, AndEngine[29] which is a broad 2D game engine. We do not use a higher level library in our solution, but OpenGL ES 2.0 natively, as it is widely supported by most of the mobile devices in the market.

```
public static final String CUBE_MESH_VERTEX_SHADER = " \n" + "\n"
       + "attribute vec4 vertexPosition; \n"
       + "attribute vec4 vertexNormal; \n"
       + "attribute vec2 vertexTexCoord; \n" + "\n"
       + "varying vec2 texCoord; \n" + "varying vec4 normal; \n" + "\n"
       + "uniform mat4 modelViewProjectionMatrix; \n" + "\n"
       + "void main() \n" + "{ \n"
       + "   gl_Position = modelViewProjectionMatrix * vertexPosition; \n"
       + "   normal = vertexNormal; \n" + "   texCoord = vertexTexCoord; \n"
       + "} \n";

public static final String CUBE_MESH_FRAGMENT_SHADER = " \n" + "\n"
       + "precision mediump float; \n" + " \n" + "varying vec2 texCoord; \n"
       + "varying vec4 normal; \n" + " \n"
       + "uniform sampler2D texSampler2D; \n" + " \n" + "void main() \n"
       + "{ \n" + "   gl_FragColor = texture2D(texSampler2D, texCoord); \n"
       + "} \n";
```

[28] https://github.com/libgdx/libgdx
[29] http://www.andengine.org

The above code snippet shows two shaders that we use in our application to render a cube when a trackable surface is detected.

Vertex shader: A vertex shader is run on each vertex to be rendered, meaning if we are rendering a sprite which contains only four vertices, a vertex shader will be run four times to compute the colour and other attributes for each vertex in the sprite.

Fragment shader: A fragment shader is run on each pixel on the screen, meaning if we are rendering a high resolution full screen on an Android phone, this shader will be invoked 1920×1080 times.

These shaders cannot exist alone, they need to be called together. They together form a program. First, a vertex shader defines the attributes for each vertex on the screen. Then, all pixels are divided into a subset of pixels which are executed with a fragment shader. Finally, resulting pixels are drawn on the screen.

```
public class CubeObject extends MeshObject
{
      private static final double cubeVertices[]  = { -1.00f, -1.00f, 1.00f, … };
      private static final double cubeTexcoords[] = { 0, 0, 1, 0, 1, 1, 0, 1, … };
      private static final double cubeNormals[]   = { 0, 0, 1, 0, 0, 1,  … };
      private static final short  cubeIndices[]   = { 0, 1, 2, 0, 2, 3, … };
      ...
}
```

In the above code snippet we show an example of how to define our model to be rendered on the screen. In this case, the model is a cube. The arrays that contain the cube's information will be passed to OpenGL ES Renderer class' `glVertexAttribPointer` function, which specifies the location and data format of the array of generic vertex attributes.

```
for (Texture t : mTextures)
{
  GLES20.glGenTextures(1, t.mTextureID, 0);
  GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, t.mTextureID[0]);
  GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MIN_FILTER,
            GLES20.GL_LINEAR);
  GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MAG_FILTER,
            GLES20.GL_LINEAR);
   GLES20.glTexImage2D(GLES20.GL_TEXTURE_2D, 0, GLES20.GL_RGBA, t.mWidth,
            t.mHeight, 0, GLES20.GL_RGBA, GLES20.GL_UNSIGNED_BYTE, t.mData);
}
```

After defining our cube model, we may optionally define a texture for that model. The `Texture` class in our application reads an image (.jpg or .png) from a file, converts it into a `Bitmap`, and then extracts the pixels in the `Bitmap` to an `Integer` array. Using this array, it creates a `ByteBuffer`, which then will be used by OpenGL ES Renderer to specify a two dimensional texture image with `glTexImage2D` function, and then bind it to our cube model with `glBindTexture` function, as shown in the above code snippet.

Next, since we are working on Android platform (4.4.2 more specifically), we need an interface between OpenGL ES and underlying native platform windowing system of Android. EGL (Embedded System Graphics Library) does this for us. It handles graphics context management, surface and buffer bindings, rendering synchronization and enables "*high-performance, accelerated, mixed-mode 2D and 3D rendering*" [30].

```
public class SampleApplicationGLView extends GLSurfaceView{
	...
	public void init(boolean translucent, int depth, int stencil){
		...
		setEGLContextFactory(new ContextFactory());
		setEGLConfigChooser(translucent ? new ConfigChooser(...));
		...
	}
	private static class ContextFactory implements
		GLSurfaceView.EGLContextFactory{

		public EGLContext createContext(EGL10 egl, EGLDisplay disp,
			EGLConfig eglConfig)
		{
			...
			EGLContext context;
			...
			context = egl.eglCreateContext(disp, eglConfig,
					EGL10.EGL_NO_CONTEXT, att_list_gl20);
			return context;
		}
	}
}

// In the main function (main Android activity)

SampleApplicationGLView  mGlView = new SampleApplicationGLView(this);
mGlView.init(translucent, depthSize, stencilSize);
mGlView.setRenderer(mRenderer);
addContentView(mGlView, new LayoutParams( // Layout Params ));
```

This is an example of how a `GLSurfaceView` can be created, initialized and added into the main activity's layout. It manages an EGL display, which enables OpenGL to render into our surface. Since it is an OpenGL related class, it accepts a user provided `Renderer` object, which does the actual rendering. The main advantage of this approach is that it renders on a dedicated thread to decouple rendering performance from the UI thread.

```
CubeObject mCube = new CubeObject();
GLES20.glClearColor(0.0f, 0.0f, 0.0f, Vuforia.requiresAlpha() ? 0.0f : 1.0f);
...
int shaderProgramID =
SampleUtils.createProgramFromShaderSrc(CubeShaders.CUBE_MESH_VERTEX_SHADER,
                                       CubeShaders.CUBE_MESH_FRAGMENT_SHADER);
int vertexHandle = GLES20.glGetAttribLocation(shaderProgramID, "vertexPosition");
int normalHandle = GLES20.glGetAttribLocation(shaderProgramID, "vertexNormal");
int textureCoordHandle = GLES20.glGetAttribLocation(shadrProgrmID,"vertxTexCord");
int mvpMatrixHandle = GLES20.glGetUniformLocation(shaderProgramID, "mVpMatrix");
int texSampler2DHandle = GLES20.glGetUniformLocation(shadrProgrmID, "txSamplr2D");
```

[30] http://www.khronos.org/egl

Now, before the final step that is the actual rendering of our cube model, what we have to do is initializing the rendering process. We do this by creating the cube object and initializing necessary handlers via the program that we create from shaders. These handlers will be used in the actual rendering, which is invoked on every frame. Realistic augmentation of a 3D environment can only be achieved if objects are continuously rendered in a manner consistent with their assigned location in a 3D space and the camera's viewpoint (Kutulakos & Vallino, 1996).

```
public class MyRenderer implements GLSurfaceView.Renderer
{
   @Override
   public void onDrawFrame(GL10 gl)
   {
       SampleApplicationSession vuforiaAppSession = new
                    SampleApplicationSession(this);
       Matrix.scaleM(modelViewProjection, 0, kCubeScale, kCubeScale, kCubeScale);
       float[] modelViewProjectionScaled = new float[16];
       Matrix.multiplyMM(modelViewProjectionScaled, 0,
              vuforiaAppSession.getProjectionMatrix().getData(), 0, mVProject, 0);
       GLES20.glUseProgram(shaderProgramID);
       GLES20.glVertexAttribPointer(vertexHandle, 3, GLES20.GL_FLOAT, false, 0,
              mCube.getVertices());
       GLES20.glVertexAttribPointer(normalHandle, 3, GLES20.GL_FLOAT, false, 0,
              mCube.getNormals());
       GLES20.glVertexAttribPointer(textreCordHandle, 2, GLES20.GL_FLOAT, false, 0,
              mCube.getTexCoords());
       GLES20.glEnableVertexAttribArray(vertexHandle);
       GLES20.glEnableVertexAttribArray(normalHandle);
       GLES20.glEnableVertexAttribArray(textureCoordHandle);
       GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
       GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, mTextures.get(0).mTextureID[0]);
       GLES20.glUniformMatrix4fv(mvpMatrixHandle, 1,
              false,modelViewProjectionScaled, 0);
       GLES20.glUniform1i(texSampler2DHandle, 0);
       GLES20.glDrawElements(GLES20.GL_TRIANGLES, mCube.getNumObjectIndex(),
       GLES20.GL_UNSIGNED_SHORT, mCube.getIndices());
       GLES20.glDisableVertexAttribArray(vertexHandle);
       GLES20.glDisableVertexAttribArray(normalHandle);
       GLES20.glDisableVertexAttribArray(textureCoordHandle);
    }
}
```

The code for the final step of rendering process is given above. Our `renderer` class implements `GLSurfaceView` which has the `onDrawFrame` function. This function is called on every frame and performs the AR content rendering. Inside the `onDrawFrame` function, if we find a detected pattern (we skip it above) the cube is drawn. A proper augmented reality system requires specifying three transformations that relate the coordinate systems of the virtual objects, the environment, the camera, and the image it produces (Kutulakos & Vallino, 1996).

Object to World ➔ World to Camera ➔ Camera to Image

$$\begin{bmatrix} u \\ v \\ h \end{bmatrix} = \mathbf{P}_{3\times4}\mathbf{C}_{4\times4}\mathbf{O}_{4\times4} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Figure 20. *Object to Image Transformation*

As we mentioned in the previous section, `modelViewProjection` is a 4×4 float matrix containing the pose of the trackable result. For each trackable detected and tracked by Vuforia, the SDK provides us with the pose of the trackable; the pose represents the combination of the position and orientation of the trackable local reference frame, with respect to the 3D reference frame of the camera (Vuforia API). We can extract this pose using the `getPose` function. The pose can be described by means of a rotation (3×3 matrix) and translation (1×3 matrix or vector) that brings an object from a reference pose to the observed pose in 3D space.

The rotation matrix determines the orientation of the camera and hence how the target is rotated with respect to the camera plane. The translation of the matrix describes the origin of the camera. This is the camera's position in the virtual 3D world. It says where the target is as seen from the camera. For example, a value of *<0,0,0>* means that camera and target are at the same position, whilst a value of *<0,0,5>* means that the target is 5 units away into the viewing direction of the camera.

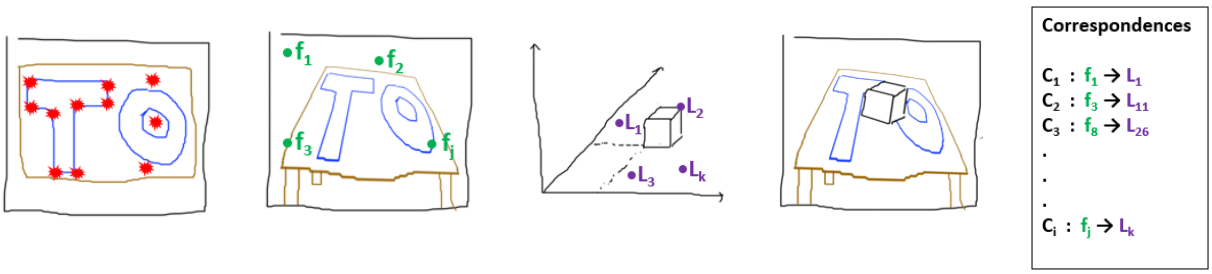


Figure 21. *Rendering Process*

This pose matrix specifies where the target is with respect to the camera and allows the system to render the AR content. However, it does not tell how the camera is placed with respect to the target (Vuforia API). If that is needed then the pose matrix needs to be inverted, which can easily be done by:

$[R \mid t]^{-1} = [R^T \mid -R^T t]$, *where R: Rotation matrix, t: Translation matrix.*

First, we scale the matrix with a constant value (`kCubeScale = 20.0f`) so that it fits nicely with the target size.

$$\begin{array}{c} \\ \text{x-axis} \\ \text{y-axis} \\ \text{z-axis} \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} \text{Orientation} & & & \text{Translation} \end{array} \\ \begin{bmatrix} \text{scaleX} & 0 & 0 & t_x \\ 0 & \text{scaleY} & 0 & t_y \\ 0 & 0 & \text{scaleZ} & t_z \\ 0 & 0 & 0 & t_w \end{bmatrix} \end{array}$$

The final step involves multiplying the scaled matrix `modelViewProjection` by `mProjectionMatrix` that is the projection matrix of the camera (pinhole camera matrix). This multiplication gives us an array of float values (4×4 matrix) that will be used to update the specified uniform variable in the line below.

```
GLES20.glUniformMatrix4fv(mvpMatrixHandle, 1, false,modelViewProjectionScaled, 0);
```

After this line, what we get is something like this:

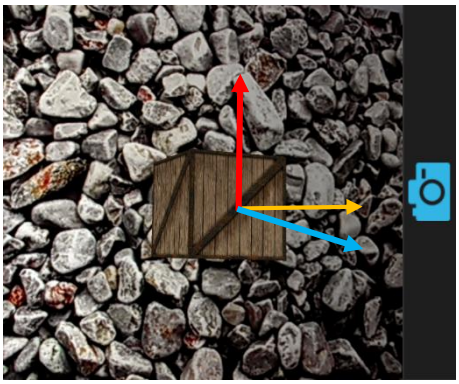


Figure 22. *Rendering A Static Model*

4.4. Content Generation

Content creation for augmented reality systems is somewhat different from typical game development process. As we highlight in Section 2.4, the contents in today's console games are a lot more extensive and complex compared to AR applications (Wagner, 2007). The example cube model that we use in our implementation is statically defined inside the application and is rendered on every frame when the defined pattern is detected.

Since this thesis focuses on the collaborative augmented reality (and thus involves multiple users), we need to add some kind of uniqueness to user behaviours. Our solution provides this by integrating a simple paint application to our AR process. User should be able to freely draw on the screen of their mobile devices and this path should appear as an augmentation on the target that the camera is looking for at that moment. Other users that are connected to the group should be able to see the generated content as the producer of the content sees it.

To generate a consistent shared space, it is important that the virtually generated content is rendered in a spatially consistent manner across all devices. Our solution alters the matrix generated with the pixels that form the path to provide a natural feeling relative to the physical scale of the real world and the AR target. For instance the generated path bounces from the position of the device's screen and slowly slides to the place where the trackable (virtual target) is located. This animation gives a feeling of transference from the screen to the physical world. To be able to do this, the generated pose matrix must be shared with other devices connected to the group.

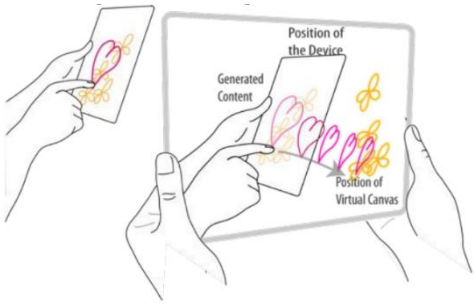


Figure 23. *Collaboration of Co-located Devices (Kasahara, et.al, 2012)*

Content generation can be achieved in different ways. One of them is to declare UI elements statically in an XML file. Even though it is a static approach to generate content, the elements can be accessed inside the code and changed programmatically. The advantage of this approach is that it separates the presentation of our visuals from the code that controls its behaviour. Another approach is to instantiate the elements at runtime. The application can create `View` objects and manipulate their properties dynamically. Whenever user interacts with the screen, we can update the XML and pass it to the renderer which first has to parse it and then render the content on the screen.

However, using these standalone approaches to generate content may suffer from performance issues that would make users feel the lack of synchronization and thus

degrade the feeling of naturalness because the camera surface is not a place that users can interact with. A better solution is to use them both to define a drawing layout on top of the camera surface. This layout and camera surface must be totally independent from each other and their behaviours must be controlled by different threads (Figure 23). When users interact with the screen, they will actually be interacting with this layout. The thread that controls the layout operations will pass the content data to rendering thread as it draws the content on the layout at the same time. When rendering is finished in the rendering thread, rendered content will be removed from the layout. All of these should be happen in a natural and very well synchronized way.

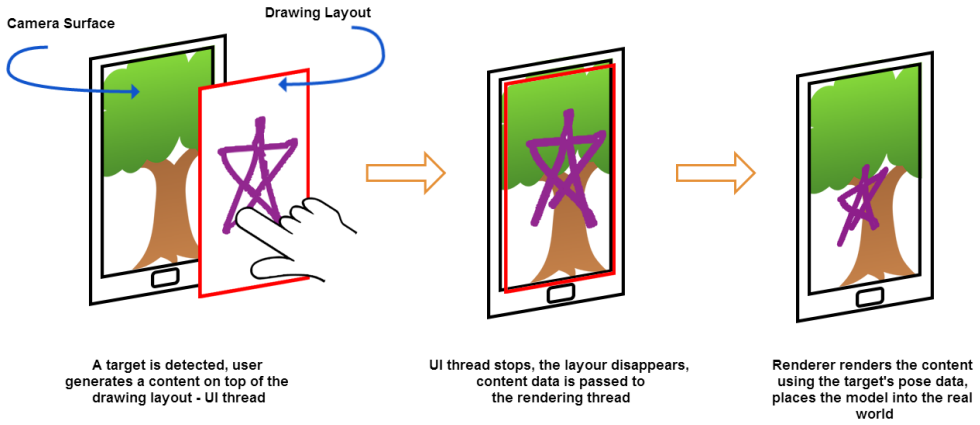


Figure 24. *Content Generation*

The generated content's (a path in our scenario) information should be sent to other co-located devices as well as to the rendering thread. Our solution has a `CustomPath` class which extends from Android Graphics libraries' `Path` class. The reason we do this is that the `Path` class cannot be serialized, as we will see its details later in this thesis. The `CustomPath` class basically has functions (`move`, `lineTo`, `quadTo`, etc.) related to movements of the path object.

The path that user draws is captured in the `onTouchEvent` function. First, all points (screen locations) are captured, and then these points are connected through lines to form a proper path object. `DrawingView` class, which extends from Android's `View` class, has a `Canvas` and a `Bitmap`. Every view class in Android has a default canvas which is drawn at the bottom of every other UI elements (but still on top of the camera view). `Bitmap` object can be considered as a pencil that draws on top of the canvas. Inside the `onDraw` function, which is called for every frame, we draw our default (blank) `Bitmap` object on the canvas, with the new paths if there is any. Below code snippet shows this process.

```
public class DrawingView extends View{
       public DrawingView(Context context){
              ...
       }
       public boolean onTouchEvent(MotionEvent event){
             float x = event.getX();
             float y = event.getY();
              if (event.getAction() == MotionEvent.ACTION_DOWN){
                    // User has started a touch action
                    path.moveTo(x, y);
                    ...
              }
```

```
            if (event.getAction() == MotionEvent.ACTION_MOVE){
                    // Action keeps going
                    path.quadTo(mX, mY, (mX + event.getX())/2, (mY + event.getY())/2);
                    ...
            }
            if (event.getAction() == MotionEvent.ACTION_UP){
                    // User has taken off his/her finger
                    path.lineTo(x, y);
                    ...
            }
        }
        @Override
        public void onDraw(Canvas canvas){
            canvas.drawBitmap(mBitmap, 0, 0, mPaint);
            for (CustomPath pp : pathList){
                canvas.drawPath(pp, mPaint);
            }
        }
        invalidate(); //Invalidates the whole view, and re-draws it according to changes
}
```

The problem with the path we form in the `DrawingView` class is that we cannot send it directly to the renderer since renderer expects a vertex list (an array of floats) relative to the world coordinates. We therefore need a mapping between the screen coordinates that the user touches and the world coordinates associated with the target. This transformation can be difficult to determine since screen coordinates can depend on the device's screen size and other UI layout elements. Thus, we need to get raw coordinates with the below code.

```
float xRaw = event.getRawX();
float yRaw = event.getRawY();
```

The `getRaw` functions returns the original raw X and raw Y coordinates of events. For touch events on the screen, in our case, this is the original location of the event on the screen, before it had been adjusted for the containing window and other views.

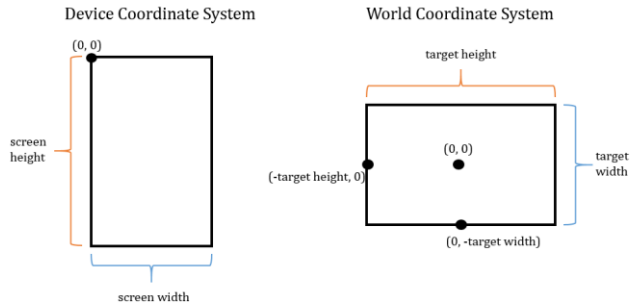


Figure 25. *Mapping from Android Screen Coordinate System to Vuforia World Coordinate System*

Another problem with the points we get is that the coordinate system that Vuforia uses to calculate the size of a target is different than the device's screen coordinate system as shown in the figure 25. The below code snippet shows an example of this conversion.

```
Display display = getWindowManager().getDefaultDisplay();
int width = display.getWidth();  // 1080.0f in our case
int height = display.getHeight(); // 1920.0f in our case

float xTransformed = y2 – height/2;
float yTransformed = x2 – width/2;
xTransformed = xTransformed * targetBuilder.getSceneWidth() / height; // 320.0f
yTransformed = yTransformed * targetBuilder.getSceneHeight() / width; // 240.0f
```

Before the final step which is rendering the path, we need to form the array of vertices in such a way the renderer expects. OpenGL ES Renderer draws a line by combining successive points (`glDrawArrays`). That means if we want to draw a rectangle, we must give the vertex array such that:

$$\{(x_0, y_0, z_0), (x_1, y_1, z_1), (x_1, y_1, z_1), (x_2, y_2, z_2), \ldots\}$$

Therefore, we add each point, which we get from the touch action and transformed into the world coordinates, into the vertex array twice, except the first one.

```
if(vertexList.size() != 0)
{
	vertexList.add(xRaw);
	vertexList.add(yRaw);
}
vertexList.add(xRaw);
vertexList.add(yRaw);
```

Since we are working on a 3D space but a line has no z-coordinate, inside the renderer class we add *z = 0.0f* for each vertex in the vertex list. So, the size of our paint vertex array would be (total size of vertex list × 3/2). Now we are ready to render the path as we do to render the cube (we also need to define two more shaders for line). The code below summarizes the process.

```
float paintVertices [] = new float[getVertexList().size() + getVertexList.size()/2];
for (int i = 0; i < getVertexList().size()/2; ++i){
    paintVertices[paintCounter] = getVertexList().get(i*2);
    paintVertices[paintCounter + 1] = getVertexList().get(i*2+1);
    paintVertices[paintCounter + 2] = 0.0f;
    paintCounter += 3;
}
public static final String LINE_VERTEX_SHADER = " \n"
	+ "attribute vec4 vertexPosition; \n"
	+ "uniform mat4 modelViewProjectionMatrix; \n" + " \n"
	+ "void main() \n" + "{ \n"
	+ "   gl_Position = modelViewProjectionMatrix * vertexPosition; \n"
	+ "} \n";

public static final String LINE_FRAGMENT_SHADER = " \n" + " \n"
	+ "precision mediump float; \n" + "uniform float opacity; \n"
```

```
        + "uniform vec3 color; \n" + " \n" + "void main() \n" + "{ \n"
        + "   gl_FragColor = vec4(color.r, color.g, color.b, opacity); \n"
        + "} \n";


GLES20.glUseProgram(vbShaderProgramID);
GLES20.glVertexAttribPointer(vbVertexHandle, 3, GLES20.GL_FLOAT, false, 0,
      fillBuffer(paintVertices));
GLES20.glEnableVertexAttribArray(vbVertexHandle);
GLES20.glUniform1f(lineOpacityHandle, 1.0f);
GLES20.glUniform3f(lineColorHandle, 1.0f, 1.0f, 1.0f);
GLES20.glUniformMatrix4fv(mvpMatrixButtonsHandle, 1, false, modelViewProj, 0);
GLES20.glLineWidth(lineWidth);
GLES20.glDrawArrays(GLES20.GL_LINES, 0, paintCounter/3);
GLES20.glDisableVertexAttribArray(vbVertexHandle);

UserDefinedTargets.mChatApplication.newLocalUserPaint(vertexList);
```

Parallel to the rendering process, we send our vertex list to other co-located devices, which will be discussed in detail in the next section.

Content Generation - while drawing on the drawing layout (left), content rendered (middle and right)

This chapter first discussed the existing solutions for rendering in an augmented reality system, showed low level graphical rendering libraries, and made comparisons between software and hardware rendering. We then presented some OpenGL features that can be used to render AR related models, and pointed out that OpenGL ES 2.0 is enough to meet the needs of augmented reality on mobile devices. We showed the close relation between pose tracking and the rendering process. Finally, we presented what the content generation methods are and how unique behaviours of users, such as drawing a path, can be rendered and augmented on an Android platform by giving example code snippets from our solution.

5. Content Distribution

Content management and distribution has always been a hot topic of conversation among computer researchers, especially in the entertainment field such as game technology, live streaming media or social networks. Researchers have shown that multiplayer gaming is the area of PC based entertainment where the highest revenue is made. However, while designing such collaborative applications, there are many technical aspects concerning users, user interfaces, networks, clients and servers to be considered. Early multiplayer games with a limited number of players used peer to peer solutions. Today's multiplayer computer games, on the other hand, have been designed for improved performance and scalable communication between collaborating players.

In this section, we first define the term collaboration in the context of an augmented reality system and then show that collaboration can be provided in different ways. Later, we discuss which approach is more suitable for our solution and what AllJoyn, an existing solution for communication, can offer us by providing some code samples from our solution.

5.1. Introduction

Although the term collaboration means "to work together" in a general sense, it can mean more for the field of computers. As Silverman says, it is a system in which both parties are sharing the task work load at an equal level of cognitive difficulty (Silverman, 1992). Terveen makes it more specific by saying "collaboration requires communication and it must involve at least one human and one computational agent" (Terveen, 1995).

Furthermore, the definition of the term becomes more specific when it comes to augmented reality. For example, there should be a shared space between co-located users where users can freely interact with a collective set of virtual objects in a 3D space.

Through a collaborative augmented reality application,

- Virtual objects that don't exist in the real world can be seen, modified and/or manipulated through natural gestures
- Real objects should be superimposed by virtual ones
- Co-located users should be able to see each other and/or each other's actions in a collaborative way
- Users can control only her/his movements from her/his point of view
- Generated content in the shared space must be the same for each user (single model)
- Displayed content may be different for different users (multiple views)

Within the boundaries of above key features of collaboration in AR systems, deciding how to provide a multiuser environment is a design choice. Although augmented reality focuses on visuals, collaboration can be done through, for example, audio. We can choose to distribute one or all of the following to provide cooperation between users:

- Static model (pre-loaded model, i.e. cube)
- Dynamic model (interactive manipulation of shared content, i.e. touch path)
- Target (trackable)
- Pose data of dynamic model
- Pose data of target

We will analyse each of these in detail later in this thesis.

As previously mentioned, there should be a real-time shared canvas to facilitate collaboration. Mobile devices become more suitable to generate this canvas as mobile technologies become more popular. However, there are some problems with mobile multi-user augmented reality applications, as Wagner says (Wagner, 2007).

- Communication is not guaranteed to be always available. The connection that the application relies on can be high or low quality. The system should be able to support different kinds of connections. When the connection is lost, users should be able to leave and enter a session at any time (Barkhuus, et al., 2005).
- Mobile devices that AR applications run on can differ greatly in terms of performance, user interface and portability. The variation of mobile devices makes it difficult to develop AR applications that rely on demanding networking modules.
- In certain cases, user generated virtual content and real world content needs to be permanently stored somewhere (and possibly mapped to each other). This is very difficult as it requires a huge content load.
- There shouldn't be a great workload difference between co-located devices.

An ideal network solution would allow mobile devices to use peer to peer connection as well as servers (Wagner, 2007). There are different network solutions that AR systems can use.

- GPRS is a packet oriented mobile data service based on GSM networks. Compared to other solutions, it has the highest network coverage. Its major disadvantage is low response time and high cost.
- Bluetooth is a wireless technology standard for exchanging data over short distances (up to 60 metres) between multi devices. Although there are no costs associated with running the network, range limitation may make it unsuitable for communication of devices.
- WiFi is a local area wireless technology that allows a mobile device to exchange data using 2.4 GHz radio waves. The greatest advantage of this solution is that the technology is fully compatible to regular Ethernet systems. Its range is usually enough to meet the needs of an AR system (20-50 metres).
- WiFi Direct, initially called WiFi P2P, is a WiFi standard that connects devices at typical WiFi speeds without requiring a wireless access point. The advantage of this infrastructure is that only one of the devices needs to be compliant with Wi-Fi Direct to establish a peer-to-peer connection.

The networking solution should provide high coverage, low response time, high bandwidth and have low cost for maintenance purposes. We choose WiFi and WiFi Direct for now, despite the fact that we know our choice may be influenced by restrictions of the communication library that we use.

5.2. Client Server Model

In the client server model, there is a server which handles the requests of clients and sends them back to the clients. For an augmented reality system, the tasks that the server handles may differ.

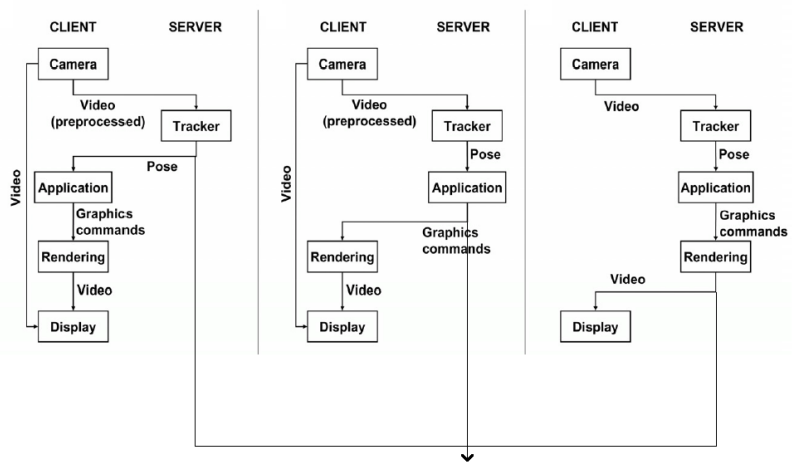


Figure 26. *Ways of using Client/Server Model in collaborative AR*

As it can be seen in the figure 26, client server model can be used in three different ways in the context of collaborative AR. At the right side, all of the important steps of augmentation are done in the server. Such a design does not only require connection on each frame, but also requires sending video stream in both directions plus sending to the other devices. It therefore requires a powerful network connection and cautious use of the available network bandwidth (Singhal & Zyda, 1999).

Despite the fact that today's mobile devices have high processing power (some even have graphics processing units), the rendering task alone can be offloaded to server, by sending the pose data pre-calculated natively. Server renders the model and sends the final image to all co-located devices.

The left figure shows an approach that the client sends the tracking task to the server, which returns the pose data when a pattern is detected. Other application related tasks and rendering are performed natively in the mobile device. This also requires frame by frame communication but relieves the burden of server and provides a more balanced network. Here, tracking load also can be divided. Client can handle predefined target tracking while server performs natural feature tracking.

Cloud Recognition technology offered by Vuforia is an example of this approach. Although our initial goal was to provide the collaboration in a peer to peer fashion, we also use the server client approach in our solution.

There are two different databases that Vuforia offers to developers.

- **Device Databases:** Device Databases are stored on the device and can either be packaged with the application, or loaded to device storage at runtime from the server.
- **Cloud Databases:** When there are large number of targets to track or targets are changing often, application can download them from the cloud database at runtime. It gives of course more flexibility since targets can be dynamic. Some extra information, such as metadata, can also be associated with targets.

```
private static final String kAccessKey ="56aa452bbfab85fa0d9f694941ef5df5ddfbd5e";
private static final String kSecretKey = "c4d3257640a39a24552a86ca156e9005bf1c15e";
```

When developers register the target manager system, Vuforia gives access keys. We first create a `TargetFinder` object and then using these access keys we can connect to our database.

```
TargetFinder targetFinder = imageTracker.getTargetFinder();
if (targetFinder.startInit(kAccessKey, kSecretKey));
{
    targetFinder.waitUntilInitFinished();
}
targetFinder.startRecognition(); // remote tracking starts
final int statusCode = finder.updateSearchResults();

// On every frame update, search if there is a new result
if (statusCode == TargetFinder.UPDATE_RESULTS_AVAILABLE)
{
    if (finder.getResultCount() > 0) // a pattern is detected online
    {
       TargetSearchResult result = finder.getResult(0);
       if (result.getTrackingRating() > 0)
       {
          Trackable trackable = finder.enableTracking(result);
       }
    }
}
```

This does not work if there is no target in our database. Uploading a target image (trackable) using the target manager in Vuforia website is not the approach we want as it dramatically slows down the collaboration. So, we want to post a target at runtime using Vuforia Web Services API, by making an HTTP POST request to a specific Vuforia link. The request header includes the authorization fields, and declares an application/Json content type. The body of the request is a JSON object that defines the properties of the targets. After uploading the image to the database, other co-located devices access the metadata (that contains the pose data of the target) and update the application logic (i.e. rendering) according to the new information (Figure 27).

Even though this seems a reasonable approach for distribution of the target, Vuforia Web Services does not immediately set the target status as active. That means when a client pushes the target it creates to the database, it is not instantly ready to be downloaded by co-located devices. In our experiments as of yet, it takes approximately 24 hours for a

target to be ready to be received again. Even if this duration was 5 minutes, for example, there would not be so much difference in terms of scientific point of view as long as it is not instant.

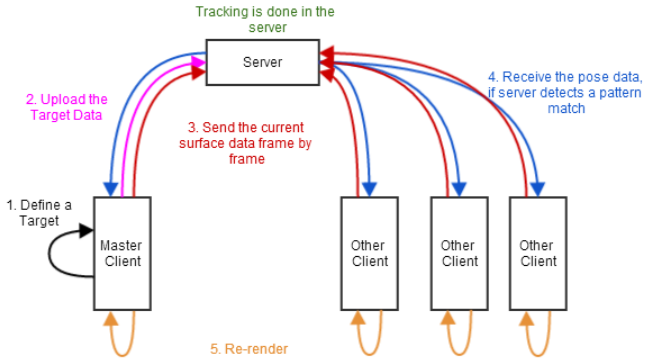


Figure 27. *Distributing Dynamic Target Using Vuforia Wev Services API (Cloud Recognition)*

Another approach, as shown in the figure 28, is to use device databases offered by Vuforia. This approach is different from Cloud Recognition such that the tracking step is done natively. The whole purpose of this method is to be able to send the target to other devices because we have some difficulties while multi-casting the target using P2P approach as we will see its details later. Even though, compared to the cloud recognition method, this approach gives better performance in terms of total download and upload time, it is still far from providing seamless real time augmentation effect.

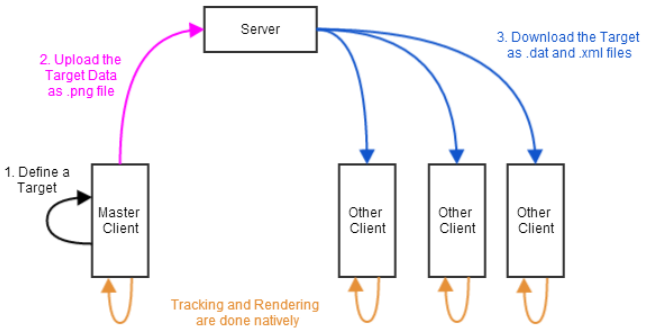


Figure 28. *Distributing Dynamic Target Using Vuforia Wev Services API (Device Database)*

The main advantage of client server model is that the server stores all generated content. This means the content in the database is persistent and when a device detects a pattern match, it gets provided with all information that is linked with the object (Kasahara, et al., 2012). Nevertheless, as we see using server client approach in a collaborative augmented reality application is not very useful to give users the feeling of naturalness as it may significantly suffer from the lack of instantaneous. Centralization of content is also another problem with that approach. When server fails none of the devices can neither upload nor receive the content data (target data, pose data, or static/dynamic models).

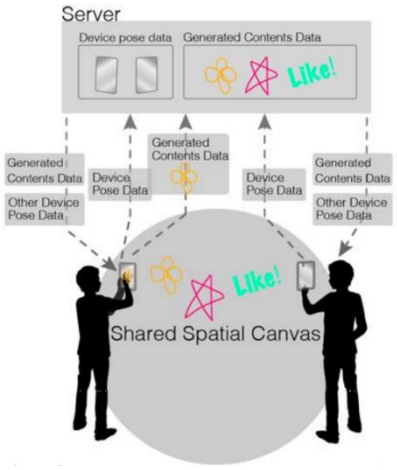


Figure 29. *SecondSurface Server System (Kasahara, et.al., 2012)*

These two approaches that we mention above consider distributing the target defined by a client device. Distribution of other AR related objects such as static or dynamic models, or pose data of target is not necessary to be considered because a target is typically more than 1000 times bigger than the others, and furthermore we don't have difficulties to distribute them using a peer to peer fashion, as we analyse next.

5.3. Peer-to-Peer Model

In contrast to centralized client server model, individual nodes in the network act as both suppliers and consumers of generated content. Peer to peer networks are used by numerous applications, such as Torrent and Spotify. The most commonly known is file sharing, which made the model popular.

There are both advantages and disadvantages in a P2P networks related to data backup, recovery, and availability. For example, in a real P2P network, because each device might be being accessed by other co-located devices, it can slow down the performance. In order to overcome this, our solution blocks the access if there is a peer in the group with newly generated content that has not been distributed to all peers yet. That means until the content has received by all peers in the group, nobody can generate a new content in the shared space. Besides all these, it would be unreasonable to expect more than 10-20 users to join a group at the same time in a collaborative augmented reality application. For such a limited number of users, implementing a server application is redundant.

Despite its drawbacks, which we discuss in the evaluation section, its main advantages, such as easy setup, rapid prototyping, faster connection, and individual security permissions, make it a better candidate for a collaborative augmented reality system.

As it can be seen in the figure 30, deciding what AR related objects to send to peers in a group is a design choice. While sending all of these objects is considered as a collaborative AR system, sending one of them can also be considered the same. Although, in our solution we implemented the support for sending all of them for experimental reasons, choosing one or two of them and sticking to design choice makes the whole system more consistent.

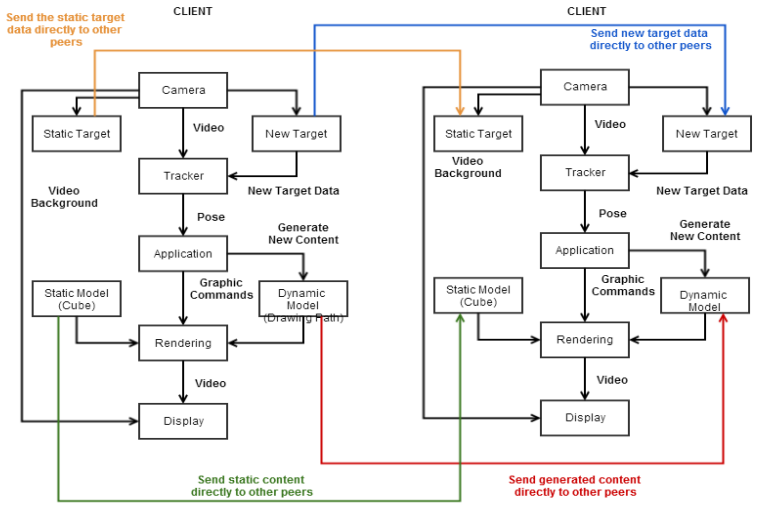


Figure 30. *Using P2P model to distribute AR related objects*

For example, if we allow both to send the static target and to generate a user defined target at runtime, design of the tracker might be too messy, and thus influences the renderer negatively as it cannot decide what to render. Likewise, if we allow to send both the static model and dynamically generated model at the same time, the renderer in the receiving client might not render properly.

Another design choice here is to determine which peer (or peers) defines the target that tracker track against. If we let all peers in a session (in a group) to define a new target, that makes the system too complex as there is no centralized system in the model that controls the data mapping between the peers and contents generated by the peers. A trivial solution to this problem is allowing only the group owner to define the target. This makes the system a bit different than pure peer to peer model. In a pure P2P model all peers are at the same hierarchic level and have equal workload, whereas in our solution group owner acts like a server that distributes the target. However, in terms of content distribution, each peers can independently generate content and distribute it as long as there is a defined target.

In this type of system, generating (and distributing) multi targets is also possible. Each peer can have a dataset that contains dynamically generated targets or static targets (a dataset containing both at the same time is a design issue that we discuss later). Each tracker of each peer recursively searches a match inside this target dataset.

Editing an already generated content is very difficult in this scenario. Each peer would have to control its own actions together with counting (or somehow remembering) other's actions since, as we discussed above, there is no centralized system that controls the mapping, but it is not our major concern.

Another important point with this approach is view permissions of generated content. More specifically, peers should be able to decide who would see the content generated by itself. Although we did not add this feature, this is not too difficult to implement since each peer can receive the list of group members from the group owner when a new peer joins the session. If a peer has well-known (unique) names of other peers, it can put the ones it wants into a block list and send the content to the rest.

Although the content is not stored in a database, we do not lose the content until a certain session is ended by the group owner. Until this time, peers should see all the content even if they leave and re-enter a session (unless they join another session within this period).

To provide these features, we use an open source framework AllJoyn, developed by Qualcomm, which allows peers in a same network (using the same access point) to do simple communication operations such as seeing each other, advertising, handshaking and transmitting data over an interface.

5.4. AllJoyn

The AllJoyn is an open source cross platform communication library that lets developers create peer to peer applications over various transports by enabling applications to connect, control and share resources with applications on nearby devices. It is written in C++ at its core, and provides multiple language bindings and complete implementations across various operating systems and chipsets. One of the reason we chose to use the AllJoyn framework is that it provides an object-oriented approach to make P2P easy avoiding the need to deal with low level network protocols and hardware. Furthermore, it is optimized for and fully compatible with mobile platforms, particularly with Android.

There are two major limitations of AllJoyn. The most important one is that although AllJoyn supports connections via WiFi, Bluetooth and 3G/4G, when WiFi is preferred, peers must be nearby, meaning they must be in the same network (they must use the same access point).

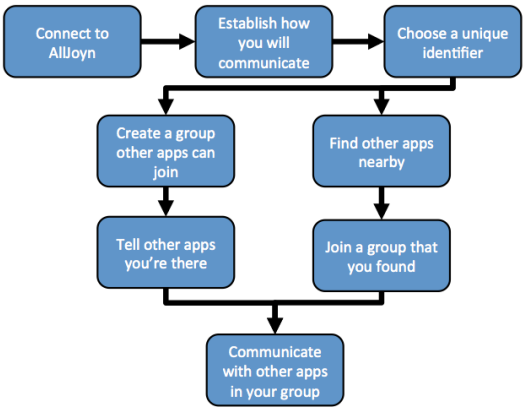


Figure 31. *AllJoyn Communication Process (taken from www.alljoyn.org)*

The reason for this is that each peer in the AllJoyn framework has a unique (well-known name) that is distributed to the other peers when advertising. This unique name is composed of the package name of the application, the channel name and the media access control address (MAC). It therefore only allows connection in a local network. The advantage is that the AllJoyn can perform device discovery without using the internet as long as there is no wireless isolation (which blocks multicast packets) on the WiFi network to which the peer is connected. AllJoyn does not create an explicit limit on the distance between the devices. The maximum distance between devices in the context of AllJoyn is the range of the WiFi network to which these devices are connected (AllJoyn).

Another limitation of AllJoyn in the context of this thesis is that it only allows sending of primitive types (or a container class that is made up of primitive types with annotations) over a bus interface. This means, for sending an object we first need to convert it into a primitive type, such as converting into a byte array, serializing, converting to a Json object, or to an XML string, etc.

Although it's not a major problem, AllJoyn can only handle a finite amount of combined data. This limit is defined to be $2^{\wedge 17}$ (131072 bytes, 128KB) (AllJoyn). If we want to send large amounts of data (e.g. more than one trackable at a time), we must divide the data into smaller sections.

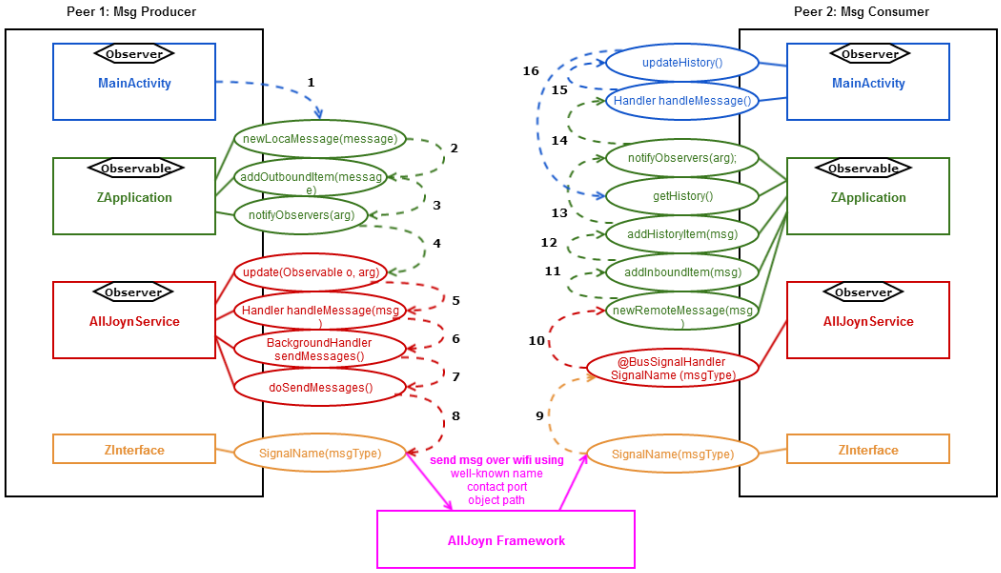


Figure 32. *Interaction sequence between peers*

The figure 32 shows how we integrated AllJoyn API into our solution. When a peer generates a new content, 16 steps are performed until another peer receives the content. This is the ideal modular structure of a communication system in which each module has specific responsibilities. `MainActivity` is the highest level structure whereas the `ZInterface` is the lowest level. As you see in the figure, when a new message is generated, it is sent to the AllJoyn Framework in a top-down manner, and delivered to the receiver in a bottom-up manner.

`MainActivity` is our Façade class. It handles every kinds of user commands. `ZApplication` serves as the Model of Model-View-Controller pattern for the application. It holds the global state for the application and starts the Android Service that handles the background processing relating to our AllJoyn connections. `ZApplication` always exists even if the application stops, as long as the application is not destroyed by Android OS. Therefore, it establishes a bridge between the `AllJoynService` and the `MainActivity`. AllJoynService handles the lower level networking processes that we do not want to re-implement but only use. Finally, the ZInterface is a typical Java interface that defines the types of messages that can be sent over a bus.

```
private BusAttachment mBus = new BusAttachment(app.PACKAGE_NAME, BusAttachment.RemoteMessage.Receive);
```

As shown in the above code snippet, there must be at least (usually at most) one `BusAttachment` object that a peer uses to connect with the other peers. This object must persist during the application lifetime. It is kind of a gateway between peers that handles all of the low level networking processes. This is the primary step to start advertising, discovering and communicating with other peers.

```
private static final String NAME_PREFIX = "com.zframeworkexample";
String wellKnownName = NAME_PREFIX + "." + mChatApplication.hostGetChannelName();
```

We need to have a unique name, the above code snippet, to avoid random behaviour that can occur if multiple peers use the same name. If there are more than one peer using the same name, another peer in a discovery state randomly selects the one to which it should connect. In our solution we are using a Globally Unique Identifier (GUID) to provide peer uniqueness.

```
Status status = mBus.advertiseName(wellKnownName, SessionOpts.TRANSPORT_ANY);
```

After we create a bus attachment and have a well-known name, we need to advertise as shown in the above code snippet. The purpose of advertising is to give information about the peers and to determine what peers it would connect to.

```
Status status = mBus.findAdvertisedName(NAME_PREFIX);
```

Peers should enter a discovery state to find other peers. AllJoyn triggers a callback whenever an advertisement is received. For peers to discover each other they have to agree on:

- Well-known name that will be advertised
- Object path
- Session port number that will be used

```
private class ChatBusListener extends BusListener{
	public void foundAdvertisedName(String name, short transport, String namePrefix) {
			ChatApplication application = (ChatApplication)getApplication();
			application.addFoundChannel(name);
	}

	public void lostAdvertisedName(String name, short transport, String namePrefix) {
			ChatApplication application = (ChatApplication)getApplication();
			application.removeFoundChannel(name);
	}
}
private ChatBusListener mBusListener = new ChatBusListener();
private void doConnect(){
	mBus.registerBusListener(mBusListener);

	...
}
```

`BusListener` listens for discovered peers and monitors sessions related to other peers. We first create a `BusListener` class that implements the necessary methods to find an advertised name. Next, the created object is registered with the `BusAttachment`. Calling the BusAttachment's `findAdvertisedName` function makes our system ready to discover nearby peers. If a peer is detected, which starts with the same prefix, `BusListener` class' `foundAdvertisedName` function is called. Even if the devices are running different Oss, they can discover each other. According to the AllJoyn API, "...because the AllJoyn framework supports multiple programming languages, applications can use the AllJoyn framework on different device operating systems and still be able to communicate" (AllJoyn).

```
class ZService implements ZInterface, BusObject
{
        public void Chat(String str) throws BusException {
         }
        @Override
        public void DrawingPath(double[] buff) throws BusException{
        }
        @Override
        public void TrackableSource(String ts) throws BusException{
        }
}
```

Our messages will be `BusSignals` multicasting in a specific session. To be able to send messages, we need a `BusObject` that will act as a signal emitter. `BusObject` allows be exchange of data and supports method interactions between connected peers. It is up to the developer what a BusObject should contain. The above code creates a `BusObject` that is aware of the bus signals in the ZInterface. A `BusSignal` defines a type of message to be sent. It is intentionally empty, since it acts as a signal emitter, it will never called explicitly.

```
@BusInterface (name = "com.zframeworkexample")
public interface ZInterface
{
        @BusSignal(name = "Chat")
    public void Chat(String str) throws BusException;

        @BusSignal(name = "DrawingPath")
    public void ChatPaint(double[] buff) throws BusException;

        @BusSignal(name = "TrackableSource")
    public void ChatTrackableSource(String ts) throws BusException;
}
```

Our `ZService` implements `ZInterface` that contains well-defined functions to provide the actual collaboration between peers. This interface may be defined in XML, inline code or a Java interface class (AllJoyn).

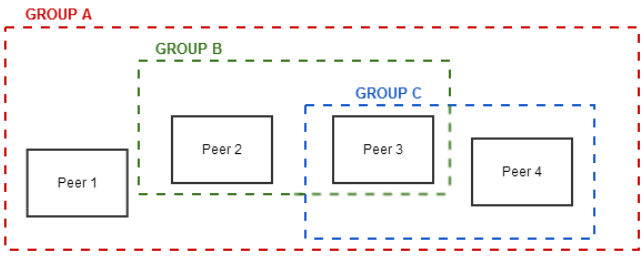


Figure 33. *Peer interaction in sessions*

The definition of a session means a group of peers connected to each other, as shown in the figure 33. The peer that creates a group (or a session) is the group owner (or the master), and the others are slaves. Each group has a group id. Peers can be a part of zero or more sessions at a given time. In the context of collaborative AR, let say peer 4 is the owner of Group C, and peer 3 is the owner of Group B. Peer 4 can define the target for Group C, but Peer 3 cannot, and vice versa. The distribution permission of a target (shared space) is controlled by the group owner. When it comes to distribution of content (dynamic or static content), the scenario gets more complex. Peer 2, for example, is both in Group A and in Group B. When it generates a new content, it must first select the group that it wants to distribute.

```
@BusSignalHandler(iface = "com.zframeworkexample", signal = "DrawingPath")
public void DrawingPath(double[] buff){
       String uniqueName = mBus.getUniqueName();
       MessageContext ctx = mBus.getMessageContext();

    if (ctx.sender.equals(uniqueName)){
        // dropped our own signal received on session + sessionId
              return;
    }
    if (mJoinedToSelf == false && ctx.sessionId == mHostSessionId){
       // dropped signal (not joined to self) received on hosted session + sessionId +
              return;
       }
    String nickname = ctx.sender;
    nickname = nickname.substring(nickname.length()-10, nickname.length());
        mChatApplication.newRemoteUserMessageDrawingPath(buff);
}
```

The code above defines a `DrawingPath` method which handles the received Android `Path` messages. `BusSignal` is executed on each peer that has already registered a `BusSignalHandler`. The execution of `BusSignal` is asynchronous, meaning there will be no response. If a peer is a part of a group, it receives the signal sent by co-located peers. Our `DrawingPath` function above invokes `newRemoteMessageDrawingPath` function, which is a part of an Observer Pattern.

```
TheFacade.mChatApplication.newLocalUserDrawingPath( // send vertex list );
public synchronized void newLocalUserDrawingPath(double[] buff){
       //addInboundItemPaint(buff);
       if (useGetChannelState() == AllJoynService.UseChannelState.JOINED){
              addOutboundItemDrawingPath(buff);
       }
}
private void addOutboundItemDrawingPath(double[] buff){//put the msg into “to be sent” list
       if (mOutboundDrawingPath.size() == OUTBOUND_MAX){
              mOutboundDrawingPath.remove(0);
       }
       mOutboundDrawingPath.add(buff);
       notifyObservers(OUTBOUND_CHANGED_EVENT_PAINT);
}
       // registered dependents will get notified
private void notifyObservers(Object arg){
     for (Observer obs : mObservers){
         obs.update(this, arg);
     }
}
```

As the above code snippet shows, when something happens in our main activity (i.e. a peer draws something on screen), the `newLocalUserMessage` function is invoked with parameters we want to send. As a typical Observer Pattern example, observers are notified with this new message.

```
public synchronized void update(Observable o, Object arg){
        String qualifier = (String)arg;
       ...
       if (qualifier.equals(ChatApplication.OUTBOUND_CHANGED_EVENT_DRAWING_PATH)){
           Message message
       mHandler.obtainMessage(HANDLE_OUTBOUND_CHANGED_EVENT_DRAWING_PATH);
            mHandler.sendMessage(message);
        }
       ...
}

private Handler mHandler = new Handler(){
      public void handleMessage(Message msg){
         switch (msg.what){
               ...
               case HANDLE_OUTBOUND_CHANGED_EVENT_DRAWING_PATH: {
                        mBackgroundHandler.sendMessagesDrawingPath();
                    }
               ...
       }
}
```

`AllJoynService` class, which handles the actual communication, is an observer and it also gets notified. We should have a `Handler` that runs at the background (the above code). It schedules (and enqueues) messages and runnables to be executed at some point in the future.

```
public void sendMessagesDrawingPath(){
        Message msg = mBackgroundHandler.obtainMessage(SEND_MESSAGES_DRAWING_PATH);
        mBackgroundHandler.sendMessage(msg);
}
public void handleMessage(Message msg){
       switch (msg.what) {
               ...
               case SEND_MESSAGES_PAINT:
                    doSendMessagesDrawingPath();
                    break;
               ...
       }
}
```

Both the above and the below codes show the final step which is to send the message using our `BusSignal` in the `BusInterface`.

```
private void doSendMessagesDrawingPath(){
        double[] buff;
        while ((buff = mZApplication.getOutboundItemDrawingPath()) != null){
               if (mJoinedToSelf) {
                       if (mHostZInterface != null) {
                              mHostZInterface.DrawingPath(buff);
                       }
               } else  {
                       mZInterface.DrawingPath(buff);
               }
         }}
```

The figure 32 summarizes all we discussed so far in this section. The design is flexible enough to support sending different message types, as we do in our implementation. Overloading `BusSignals` in the `BusInterface`, we are sending simple string messages for chat application, array of doubles that contains vertex list for drawing path (dynamic content), another array of doubles for cube (static content), (converted from byte array) for static target, integer commands for playing audio and so on. As we will see in the limitations section, we had difficulties while trying to send our dynamic target (user defined target at runtime) as the target is an object of Vuforia Trackable class. AllJoyn allows sending of send primitive data types without requiring extra setup. Although it also supports to send complex objects, all fields of these objects must be annotated so that the AllJoyn Framework knows what to send (and how). When tracker detects a pattern match against a dynamic target, what we can get from Vuforia API is a `TrackableSource` object. Unfortunately, this object can neither be serialized nor parcelized. We tried to convert it into a primitive data type using third party parser libraries, such as XStream to convert to XML, Gson and Jackson to convert to Json object, Mongo DB to convert to Bson object. Unfortunately none of these approaches succeeded due to the fact that Trackable and TrackableSource are closest to the client code, and therefore their internal state is inaccessible. Although this is a limitation of Vuforia, there is no reason that our framework could not support exchanging trackables when such an option become if such an option become available in future releases.

5.5. Conclusion

Collaboration is an important part in today's mobile applications such as games and social networks. It is foreseen by researchers (Billinghurst & Kato, 2002) that the popularity of collaboration in the context of augmented reality will keep being a paradigm shift.

In this chapter, we first defined what the term collaboration can mean for augmented reality applications and then discussed the advantages and the drawbacks of two well-known network models for our system: server-client and peer-to-peer. We showed how we could use AllJoyn, which is an open source communication framework that provides easy peer to peer collaboration, in our solution and gave AllJoyn related code samples from our implementation.

6. Solution

In this chapter, we present our collaborative augmented reality solution. First, in section 6.1 we compare some of the well-known augmented reality tools and highlight their similarities and differences regarding their suitability for our solution. In section 6.2, we give the overall structure of the solution together with its requirements. In the next four sections, from 6.3 to 6.7, we closely look at the modules in the structure with explanations of some important classes and functions.

6.1. Comparisons of AR Libraries

	Vuforia	Metaio	Layar	ARToolKit	Studierstube	ARMedia	Wikitude
Type	Free+Commercial	Commercial	Free+Commercial	Free+Commercial	Open Source	Free+Commercial	Free+Commercial
PLATFORMS							
iOS	✓	✓	✓	✓	✓	✓	✓
Android	✓	✓	✓	✓	✓	✓	✓
Windows Mobile					✓		
Web		✓				✓	
PC/Mac/Linux		✓				✓	
FEATURES							
3D Object Tracking						✓	
Natural Feature	✓	✓	✓	✓	✓	✓	✓
GPS		✓	✓			✓	✓
IMU Sensors		✓	✓			✓	✓
Marker	✓	✓		✓	✓	✓	✓
Visual Search		✓	✓				
Face Tracking		✓					
Content API	✓	✓	✓		✓		✓
Framework	✓	✓			✓		✓
PLUGIN COMPATIBILITY							
Unity 3D	✓	✓				✓	
PhoneGap							✓
Appcelerator							✓

Figure 34. *Comparison of AR SDKs*

Our solution relies on an augmented reality engine which works under our framework together with a communication library, such as AllJoyn. The figure 34 compares the most popular AR libraries according to the features they offer. For experimental purposes, we mostly use Vuforia in our solution.

Figure 35. *Wikitude World Browser in iOS*

Wikitude[31] is a mobile AR software which is developed by the Austrian company Wikitude GmbH and was first published in October 2008 as freeware. It displays information about the users' surroundings in a mobile camera view, including image recognition and 3D modelling. Wikitude was the first publicly available application that used a location-based approach to augmented reality. For location based AR, the position and the orientation of virtual objects on the screen of the mobile device are computed using the device's position (through GPS or WiFi), the direction in which the user is facing (using the compass) and accelerometer. The key element in the Wikitude World Browser is the location. In contrast to regular web pages, Wikitude is optimized for mobile location based usage. Since August 2012, Wikitude also offers image recognition technologies. Although Wikitude shows a great promise, as previously mentioned, location based augmented reality is not a major interest of this thesis.

Figure 36. *ARMedia Applications*

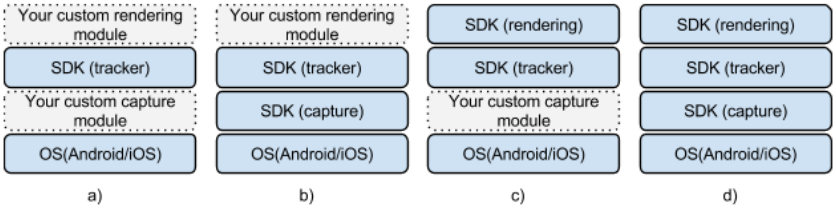


Figure 37. *ARMedia Architecture[4]*

ARMedia 3D SDK is a commercial augmented reality product that is based on a 3D model tracking approach. It does not just assume a planar scene (as is the case with a number of

[31] http://en.wikipedia.org/wiki/Wikitude

platforms), but also complex 3D objects. It is fully compatible with advanced authoring environments such as Unity 3D or with advanced graphic libraries such as OpenSceneGraph[32]. This gives developers a great flexibility to create high quality 3D content at runtime, unlike other AR libraries. As shown in the Figure 37, it has a modular architecture which separates image capturing, tracking and rendering modules. Developers can use only the tracking module or any combination of modules. As Vuforia does, The ARMedia SDK also allows to capture frames from the device camera and add them into the dataset which the tracker tracks against. With its modular structure and Android support (although minimum version supported is 4.0.3), it is a suitable AR engine for our framework.

```
public ARMedia3DTracker(Activity ctx) // 3D tracker constructor

public boolean isValid() // whether the tracker is valid or not
public boolean isReady() // if the tracker is ready to track (ready to process any frame passed to
                                                                    the track(Frame) method)
public boolean initTracker() // init the tracker with the previously chosen tracking configuration file
public void cleanupTracker() // reset the tracker
public void startTracker() // put the tracker in a state ready to track any provided frame
public void stopTracker() // put the tracker in a paused state, the tracker ignores any passed frame
public void track(frame) // examine the passed frame in order to retrieve a valid pose for 3D target
public boolean isTracking() // if the tracker is currently tracking
public void getPose(double[] pose_matrix) // retrieve the latest available pose for the 3D target
public void getProjectionMatrix(double[] projection_matrix) // retrieve the camera proj matrix
```

For example, some important tracking related methods in the ARMedia3DTracker module are given above. These methods look very similar to the ones in the Vuforia API (figure 38), which we discuss in the Image Tracking Section. The only difference is that the track function expects to get a frame as a parameter, whereas each tracker in Vuforia has an associated dataset that the tracker tracks against. Instead of explicitly passing each frame to the tracker, we put them in datasets and the rest is handled by Vuforia API. Of course ARMedia's approach gives more flexibility to developers since frames viewed through the camera can be interrupted and modified. Another advantage of the ARMedia framework is that there are lots of setter methods both in the tracking and the capturing modules, which enable the modification of target and content objects.

```
if(armedia3DTracker.isTracking())
{
  // tracking, get pose...
  armedia3DTracker.getPose(pose);
  // show virtual content update its pose...
}
else
{
  // not tracking hide virtual content...
}
```

```
State state = Renderer.getInstance().begin();
if(state.getNumTrackableResults() == 0){
  // tracking, get pose...
 TrackableResult tr = state.getTrackableResult(0);
 tr.getPose()
  // show virtual content and update its pose...
} else{
  // not tracking, hide virtual content...
}
```

Figure 38. *Pattern Detection code samples ARMedia (left), Vuforia (right)*

[32] http://www.openscenegraph.org

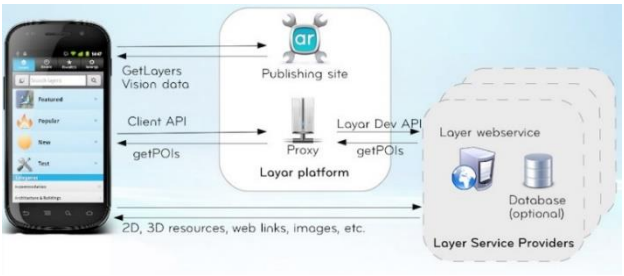


Figure 39. *Layar Platform Architecture[33]*

Layar[33] is a mobile augmented reality browser, like Wikitude, which is created by a Dutch company called Layar. Together with location based AR, it supports natural feature tracking. The architecture has five components: The Layar browser, which works on the mobile device of the user, The Layar server, The Layar publishing website, The Layar service providers and content sources (Figure 39). The Layar API is the interface between the Layar Server and the Layar Service Providers. Developers can create their own layers and submit them via the Layar Publishing Website to be added to the Layar service. The API is used to fetch live data about the layer (Layar). Although it is a very popular framework among creative advertisement companies, its structure is not compatible with our solution as it mostly depends on external services.

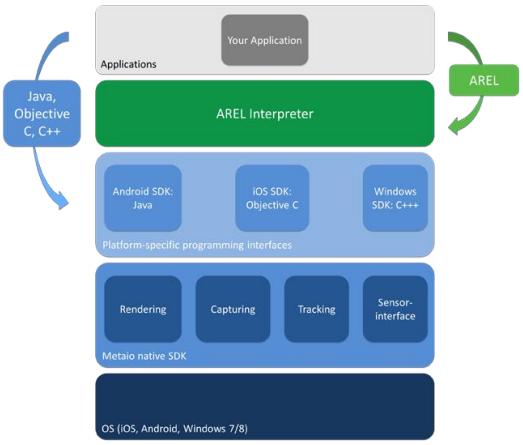


Figure 40. *Metaio Platform Architecture[34]*

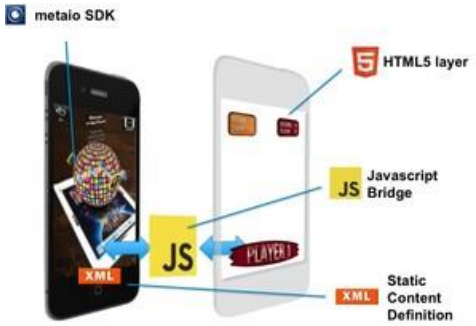


Figure 41. *How AREL Works[34]*

Metaio GmbH[34] is a Germany based company that developed Metaio SDK, which is one of the most popular and advanced AR libraries, and Junaio, which is an advanced (location based) mobile augmented reality browser. Unfortunately, however, there is not a free option of Metaio SDK.

At the core of Metaio SDK there is AREL (Augmented Reality Experience Language) interpreter (Figure 40). It is a JavaScript binding of Metaio SDK's API in combination with a static XML content definition (Metaio). With the help of AREL, creating platform

[33] https://www.layar.com/augmented-reality
[34] http://www.metaio.com

independent AR applications is possible. AREL puts HTML5 overlays into AR applications and provides easy GUI development.

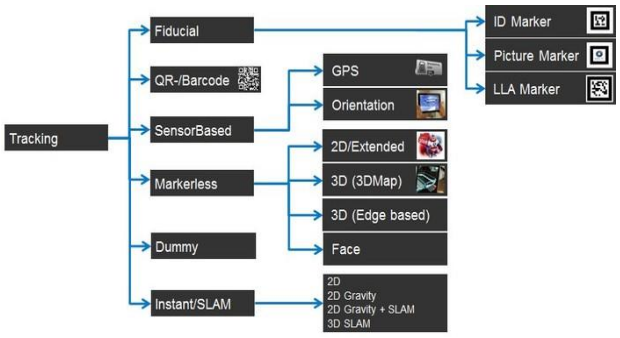


Figure 42. *Types of Markers supported by Metaio API[34]*

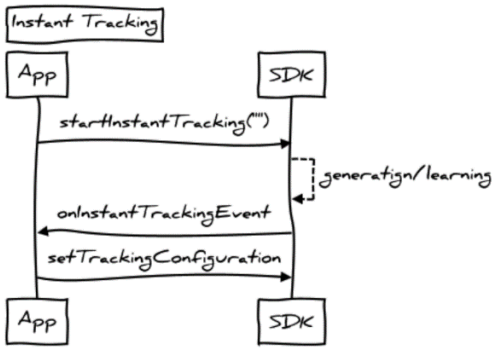


Figure 43. *Instant Tracking Sequence Diagram from Metaio[34]*

```
IGeometry createGeometryFromImage(String filepath, boolean displayAsBillboard) // Loads an image from a
        given file and places it on a generated 3D plane
boolean loadEnvironmentMap(String folder) // Creates an environment map, which can be seen as a
        reflection on 3D geometries.
boolean setTrackingConfiguration(String trackingConfig, boolean readFromFile) // Load a tracking
        configuration from an XML file.
void setCoordinateSystemID(int coordinateSystemID) // Bind the 3D model to a specific coordinate system
void getTrackingValues(int coordinateSystemID, float matrix, boolean preMultiplyWithStandardViewMatrix)
        // Allows to get state of tracking system for a given coordinate system compatible with OpenGL
void startInstantTracking(String trackingMode, String outFile) // instantly start creating a tracking
        configuration based on camera image
Vector2di startCamera(int index, int width, int height, int downsample) // Start capturing on a camera
TrackingValues invertPose(TrackingValues inPose) // Invert the pose in the given TrackingValues.
void getProjectionMatrix(float matrix, boolean rightHanded) // Allows to get the OpenGL projection
        matrix retrieved from camera calibration.
Vector2d getScreenCoordinatesFrom3DPosition(int coordinateSystemID, Vector3d point) //Converts the
        given 3D point to screen coordinates.
```

Some of the important functions from Metaio API are given above. The closely related signatures of these functions in Metaio SDK demonstrate its similarity to both the API's of Vuforia and ARMedia. For example, `startInstantTracking` function in Metaio serves the same purpose as `userDefinedTargets` in Vuforia both of which create a new target on the fly. Metaio also allows the creation of a 3D target, commonly referred as a SLAM technology. As we see in the figure 40, Metaio has also a modular architecture, like Vuforia and ARMedia. Rendering, tracking and capturing modules are separated from each other.

```
final TrackingValues trackingValues = metaioSDK.getTrackingValues(1);
if (trackingValues.isTrackingState()){
      // We detected a pattern, do something, render, play audio, etc.
      float[] modelMatrix = new float[16];
      metaioSDK.getTrackingValues(1, modelMatrix, false, true);
      metaioSDK.getProjectionMatrix(projMatrix, true);
      ...
      // Metaio does the mapping from world-to-camera coordinates for us, Vuforia
      does not do this, see Section 4.4
      projMatrix[0] *= mCameraImageRenderer.getScaleX();
      projMatrix[5] *= mCameraImageRenderer.getScaleY();
      mCube.render(gl);}
```

Figure 44. *Pattern Detection code sample Metaio, see Figure 38*

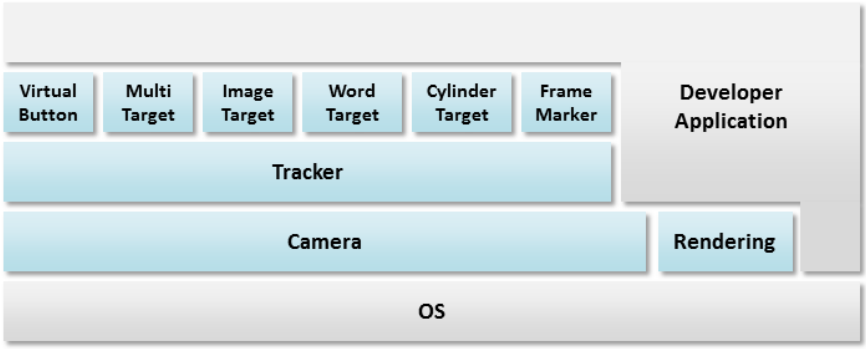


Figure 45. *Vuforia Architecture*[8]

As we discussed, Wikitude and Layar mostly focus on location based augmented reality and Studierstube has not a modularized structure (see Section 2.2). In addition, Studierstube and ARToolKit does not support different language bindings. Therefore, they are not suitable for mobile AR development. Vuforia (Figure 45), Metaio and ARMedia are the most suitable AR libraries for our solution because of their modularized structure and mobile platform support.

In conclusion, considering all the features offered by these libraries, it is fair to say that any augmented reality library that uses image based tracking and separates the image capturing, tracking and rendering modules is a good candidate to fit into our solution without requiring major changes.

6.2. Overall Structure

As we see in the previous section, most of the AR libraries have (at least) a capturing module, a tracking module and a rendering module. In our solution, the central component is the communication agent which sits on top of these modules and interact with them as shown in the figure 45. The major restriction of the collaboration agent is that these modules must be separated.

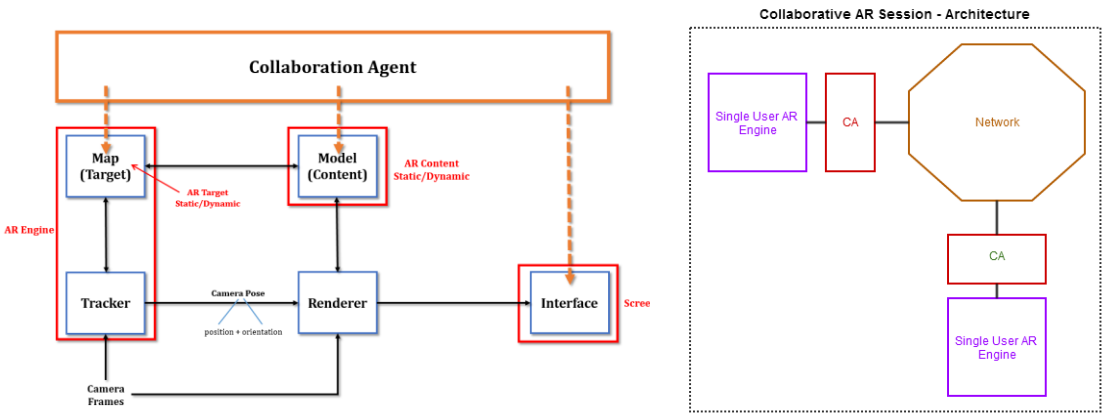


Figure 46. *Interactions with Collaboration Agent (CA) (left), Collaborative Architecture (right)*

Since the collaboration agent interacts with the target (by creating and tracking against it), the content (by generating it) and the screen (by rendering on it), it consists of four modules:

- Group (Session) Management
- Map (Target) Management
- Model (Content) Management
- Interaction Management

Figure 47. *Structure of the Collaboration Agent (CA)*

If we can ensure that these four modules of collaboration agent work in a modularized fashion, we can create a flexible collaborative mobile augmented reality framework that adapts different AR libraries such as Vuforia, Metaio and ARMedia.

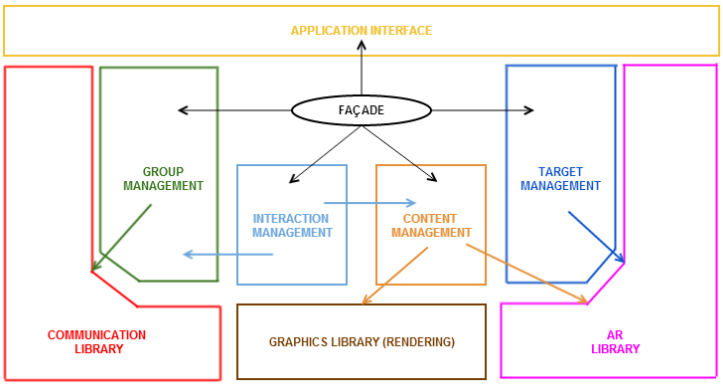


Figure 48. *Overall Structure of the System (Green, Light Blue, Orange and Dark Blue represent the CA)*

The Group Management Module is the interface between the communication library and the rest of the application. It interacts with the Façade, the Interaction Management Module and the Communication Library. Management of sessions between peers is the main responsibility of this module.

The Interaction Management Module is the interface between the Group Management Module and the Content Management Module. It manages the interactions with the real world, such as generating a new content on top of a target. Whenever a new user generated content is created, letting the Group Management Module know about this content is its main responsibility. Besides this, it also forms the content data such that the renderer in the receiving peer does not require any additional OpenGL conversion and use the data as it is.

The Content Management Module is the interface between the low level graphics library and the rest of the application. Rendering is the only responsibility of this module.

The Target Management Module is the interface between the AR library and the rest of the application. It interacts with the Façade, the Content Management Module and the AR library. Its main responsibility is on the fly target creation and detection (tracking).

The Façade is the connector. It interacts with all these modules and acts as the bridge between them by transporting requests and responses. It handles user actions if it can, otherwise delivers them to the necessary units.

In the collaboration agent pipeline, there are 17 steps.

1. A device starts a session (becomes the group owner)
2. Group owner advertises its unique name
3. A peer discovers the group owner
4. Handshaking is performed
5. Group owner distributes the static target to peers (Optionally to a server)
6. Group owner distributes the dynamic target (Optional)
7. Group owner distributes the static content (Optional)
8. Peers receive the static target
9. Peers receive the dynamic target (If there is)
10. Peers receive the static content (If there is)
11. Each peer starts local tracking
12. Each peer starts remote tracking (tracking in the server)
13. If a peer detects (locally or remotely) a target (static or dynamic), it renders the content associated with the target
14. Peers interact with a target (generate a dynamic content)
15. The Peer, in the step 14, distributes the dynamic content and its pose data
16. Peers receive the dynamic content and its pose data
17. Go to step 11

In order to understand how our structure works, consider the scenario below:

Let's say we have three peers Peer 1, Peer 2 and Peer 3 in the same network (GSM, WiFi or Bluetooth), but a session between these peers has not formed yet. Peer 1 has a static target and a static content. It starts a room and advertises its unique name so that other peers in the same network can find it. Peer 2 is in the discovery stage right now. It wants to see the room list currently available in the network and selects a room from the list. As soon as Peer 2 selects the room that the Peer 1 created, Peer 1 gets notified. Handshaking is done, Peer 2 joins the room and a session is formed between Peer 1 and Peer 2. Peer 1 sends the static target (and optionally the static content) to Peer 2. At this point, Peer 1 can also choose to upload the static content and static target into the server. When Peer 2 receives the static target, it automatically starts the local tracking (Peer 1 has already started). Peer 2 is also connected to the server and remote tracking process is also started in the server. Right now both Peer 1 and Peer 2 both natively and remotely is tracking against a possible target. Remote tracking is done by sending each camera frame to the server and receiving an answer from the server. If tracker in the server detects a pattern match, Peer 2 (or Peer 1) natively renders the static content on top of the target. Likewise, if the tracker in the Peer 2 detects a pattern match, again the same static content is rendered on top of the found target (Here we can also render a different content). If Peer 2 generates a new content on top of the static target, the content (and its pose data) is sent to Peer 1 and the renderer in the Peer 1 updates itself according the generated content. At this point, Peer 1 can choose to create a new target (dynamic target) at runtime. The new target is sent to Peer 2. Since there has not been an associated content with the new target, even if the tracker in Peer 2 detects this target, the renderer renders nothing until one of the peers create a new content on top of the new target.

6.3. Group Management

The Group (Session) Management Module manages the sessions between peers. The first four steps of the collaboration agent pipeline are performed under the control of the group management module. It is the interface between the communication library (AllJoyn) and the rest of the application. It therefore:

- Interacts with the Façade by sending the requests coming from the Façade class (such as view changes, initialization of a new group, a new peer joining the room, playing audio, etc.), and sends the responses back to the Façade class.
- Interacts with the Interaction Management Module by transferring mostly user generated content to the AllJoyn API.
- Interacts with the AllJoyn API by managing the communication logic between peers (such as connecting to a network, advertising the unique name of the group owner, discovering a unique name, determining the connection state (initialized or lost), agreeing on port numbers, deciding the types of messages that can be sent, sending the messages, sequencing the messages coming from different peers, etc.).

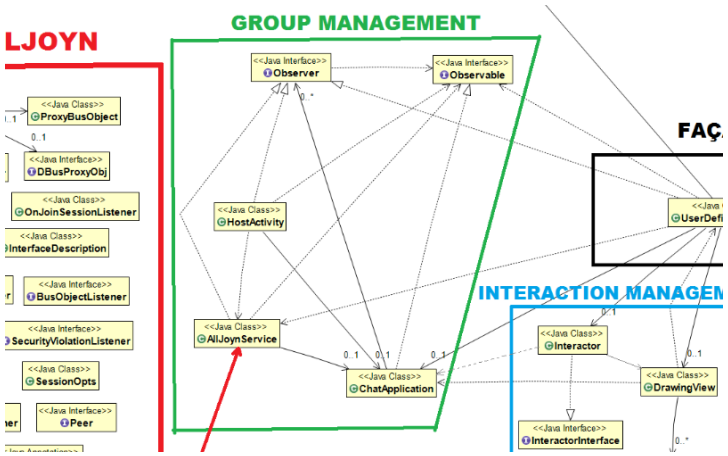


Figure 49. *Group Management Module*

Considering all of the above, it is almost obvious that there should be an Observer Pattern inside this module. There should be an `observer` and an `observable` interface, a `hostActivity` to start, stop, advertise the room, and an `AllJoynService` class that deals with all the lower-level network operations. Fundamentally, it is the *Controller* for the application. In our solution, we have the fifth class, which is called `ZApplication`. It serves as the *Model* for the application and what we end up with is the typical user interface design pattern *Model-View-Controller* (MVC) (*View* is the Façade class which we discuss later). This allows the UI to be independent from the `AllJoynService` (The *Controller* in *MVC* pattern).

Android Activities, which are usually UI related classes and operated by Android OS, can be created and destroyed in unusual ways, such as when the device is rotated. `ZApplication` class extends from a special Android class `Application` and connects the whole application to the outside world. The importance of `ZApplication` class is that its lifecycle is consistent with the lifecycle of the whole application, meaning even if the activities are destroyed and recreated by the Android OS, the application object still exists.

We need this persistent `Application` object because Android activities cannot be saved (i.e. by serializing) and we are deeply paranoid that Android OS might interrupt the process by destroying and recreating our `Activity` objects.

The `Façade`, the `ZApplication` and the `HostActivity` classes implement the `Observer` interface, whereas the `AllJoynService` class implements the `Observable` interface (Figure 49). `Observers` are registered with the `Observable` and are notified by the `Observable` when a state is changed. More specifically, the `AllJoynService` class maintains its dependents.

6.4. Target Management

The Target (Trackable/Map) Management Module manages the target creation and target exchanges between peers. Steps 5 to 12 in the collaboration agent pipeline are performed under the control of the target management module. It is the interface between the augmented reality library (Vuforia) and the rest of the application. It therefore:

- Implements the tracking related functions in the Vuforia API and interacts with the Façade class by performing these functions through the commands of the Façade (such as loading and executing the tracker, initializing, starting, stopping, pausing, resuming the AR tracking operations, handling the camera initialization, setting the camera projection matrix, configuring the video background, etc.).
- Interacts with the Model Management Module by sending the rendering related commands (such as saying when and where to render, which textures to be drawn on, initializing the screen configurations and dealing with screen orientation changes, etc.)
- Interacts with the Vuforia API by making Vuforia perform the tracking properly. Since Vuforia is a large framework, the Target Management Module offloads to it, almost every piece of tracking task (such as complex math operations, matrix multiplications, conversion from pose data to matrices, rotation, translation and scaling camera matrices, dataset initialization, the current state information, pixel format, video mode, etc.).

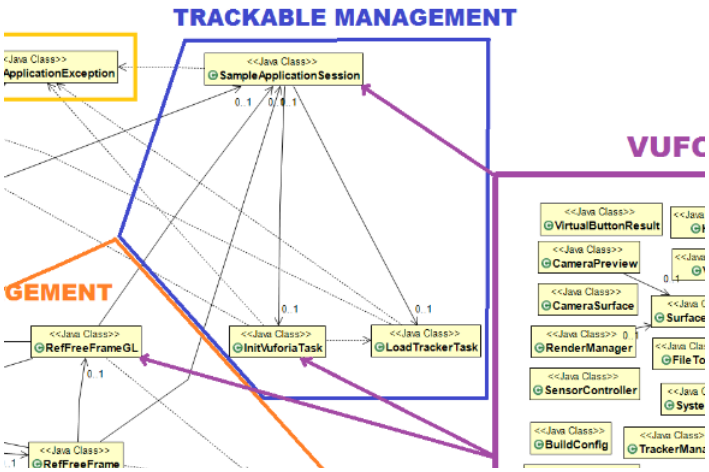


Figure 50. *Target Management Module*

Since Vuforia handles most of the tracking tasks, only one class is enough between the Vuforia API and the rest of the application. The `SampleApplicationSession` class standalone acts as the interface that tells Vuforia what to do, together with its two inner classes: `InitVuforiaTask` and `LoadTrackerTask`. An object of this class is passed both to the renderer and to the Model Management Module for proper screen initialization and rendering of content. Some of the important operations of the `SampleApplicationSession` class are given below.

```
public class SampleApplicationSession implements UpdateCallbackInterface{
       public void initAR(Activity activity, int screenOrientation)
       public void startAR(int camera) throws SampleApplicationException
       public void stopAR() throws SampleApplicationException
       public void resumeAR() throws SampleApplicationException
       public void pauseAR() throws SampleApplicationException
       public Matrix44F getProjectionMatrix()
       public void QCAR_onUpdate(State s)
       private void storeScreenDimensions()
       private void updateActivityOrientation()
       private void setProjectionMatrix()
       private void stopCamera()
       private void configureVideoBackground()
}
```

The most important function is `QCAR_onUpdate`, given below. This function is called by the Vuforia API implicitly on each camera frame. To put it more specifically, when the tracker in Vuforia finishes its tracking against a particular frame, it invokes this function by passing the current state information, `State`. The `State` object contains the tracking information (i.e. a pattern is detected or not) according to the latest camera frame. The Target Management Module passes this state to the Model (Content) Management module so that the Model Management Module updates the screen. Also, there are two inner classes in this class that deal with initializing Vuforia components and loading the tracker data.

```
private SampleApplicationControl m_sessionControl;
...
@Override
public void QCAR_onUpdate(State s)
{
   m_sessionControl.onQCARUpdate(s);
}
```

6.5. Content Management

The Content (Model) Management Module manages only rendering tasks. Thus, only the step 13 in the collaboration agent pipeline is performed under the control of the Content Management Module. It is the interface between the lower level graphics library (OpenGL ES 2.0), Vuforia API, and the rest of the application. It therefore:

- Interacts with OpenGL ES by initializing the rendering tasks, telling it how to render the content (and by compiling the shaders associated with the content).
- Interacts with the Façade class by getting the content related information from it and sending the current rendering state. For example, when a target is detected, the Content Management Modules informs the Façade so that the Façade can perform additional operations (such as playing audio, or updating the layout). Since none of the classes in the module is an Android Activity, it cannot change the view or the layout of the application, instead this is the responsibility of the Façade.
- Interacts with the Interaction Module by getting the generated contents (dynamic contents) and responding back.
- Interacts with the Vuforia API by using its complex math and rendering tools.

The `SampleApplicationGLView` class handles the initialization, configuration, and the maintenance of `GLSurfaceView` of the application. `GLSurfaceView` is an OpenGL view that is overlaid onto the existing activity's layout and it is the surface that the renderer uses to render on. There are also static content related classes in this module: The `MeshObject` the `CubeShaders`, the `CubeObject` and the `Plane`. The purpose of these classes is defining the static content compatible with the OpenGL library, such as vertex count, shape and colour of the content, as we discuss in the Section 4.3. The `Texture` class loads an image from a file and converts it into a `ByteBuffer` so that the renderer can use it to texture map an object. The `SampleUtils` class contains some useful functions such as for initializing and compiling the shaders, for checking OpenGL errors, for getting the orthogonalization matrix (used for converting from fractional to Cartesian coordinates). The `RefFreeFrameGL` class is used to render not the AR content but the layout related images, such as camera frame images, UI elements, and so on. It is therefore independent from the tracker.

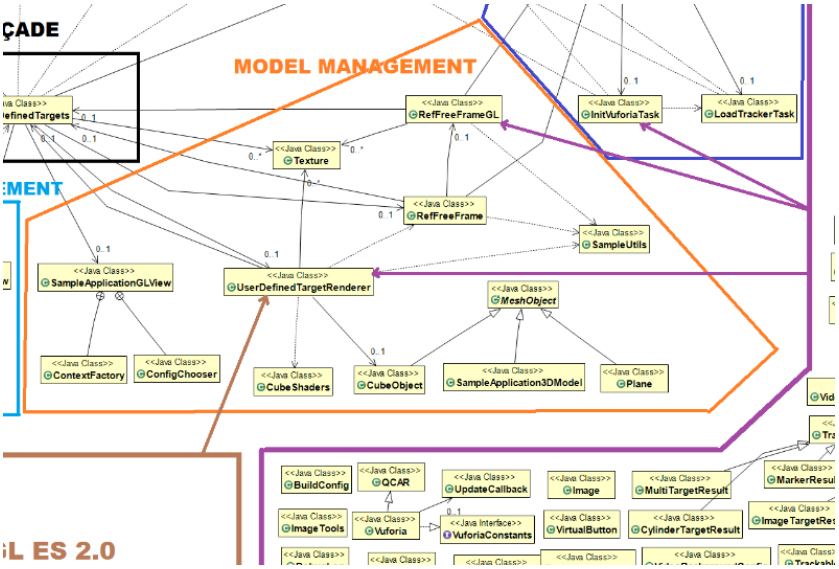


Figure 51. *Content (Model) Management Module*

The `RefFreeFrame` class is the intermediary class between the Façade, the Target Management Module and the renderer. During the program lifecycle, the tracker can be in different states: idle, scanning, creating, success, and so on. For example, when a new target needs to be created (in the Façade), the `RefFreeFrame` class receives this command and updates the tracking status so that a new target can be properly created. The most important function in this module is the `UserDefinedTargetsRenderer` class. It deals with all the rendering logic by directly talking with the OpenGL library. It renders (or makes the OpenGL render) the static contents (the cube) as well as the dynamic ones (drawing paths), binds the textures with the contents. Since it is not an Android Activity class, it cannot directly change the application's view. Just before it renders a content on the screen, it notifies the Façade (because the tracker detected a pattern) so that the Façade can do additional operations (such as playing audio).

6.6. Interaction Management

The Interaction Management Module manages both the static and user generated content and the users' interactions with them. Thus, steps 14 to 16 in the collaboration agent pipeline is performed under the control of the Interaction Management Module. It is the interface between the Group Management Module and the Content Management Module. It therefore:

- Interacts with the Façade by getting the content generated (or interacted with) by peers.
- Interacts with the Group Management Module by passing it the user generated content data, its pose matrix and the target associated with the content.
- Interacts with the Content Management Module by telling it the changes on the content.

The interaction managements could be done in the Façade class, however, for the sake of modularization, we decided to separate it. It is the middle layer between the session and the content. When a peer wants to create a new content (or alter the existing one) on top of a particular target, the new content has to be distributed to co-located peers and eventually reaches to the Content Management Module. The request first comes to the Façade. It is then sent to the Group Management Module through the Interaction Management Module because we may want to change the request so that it is fully compatible with the Content Management Module.

The `CustomPath` class extends from the Android `Path` class which encapsulates geometric paths consisting of straight line segments. The reason that we created a new class is that existing Android `Path` class is not serializable and AllJoyn API does not support distributing complex objects over the `BusInterface`. That means either we had to convert it to a primitive type, or serialize it somehow. We failed to use external parser libraries, such as Gson, Bson, XStream, for this purpose because all we got after the conversion were pointers pointing to junk memory locations. The serializable `CustomPath` class is like a container that simulates the actions of real path objects. Although the Android `Path` class contains many functions, we need only three of them.

Since we know which functions are needed to draw a path to the screen, we can store these functions in an array and simulate their behaviour with serializable inner classes.

The `DrawingView` class is an Android `View` class which is responsible for drawing and event handling. For every touch gesture of the peer on the screen, a new `CustomPath` object is created. As long as the peer keeps drawing, the path segments are added successively and a line is formed. They are drawn on a `Bitmap`, then the `Bitmap` is drawn on the `Canvas` that is the background of every Android `View` object.

The `Interaction` interface defines the type of interactions that can be performed on a user generated content, such as dragging and dropping, editing, deleting, rotating, and scaling, and the Interactor class implements these operations. When a peer interacts with a content, the updated version of the content (and its pose data) is sent to the Group Management Module so that the renderer updates the model.

A crucial point here, as we discuss in the Section 4.4, we need a conversion from the screen coordinates to the world coordinates so that the renderer receives the correct vertex locations of the path. This conversion is also done in this module.

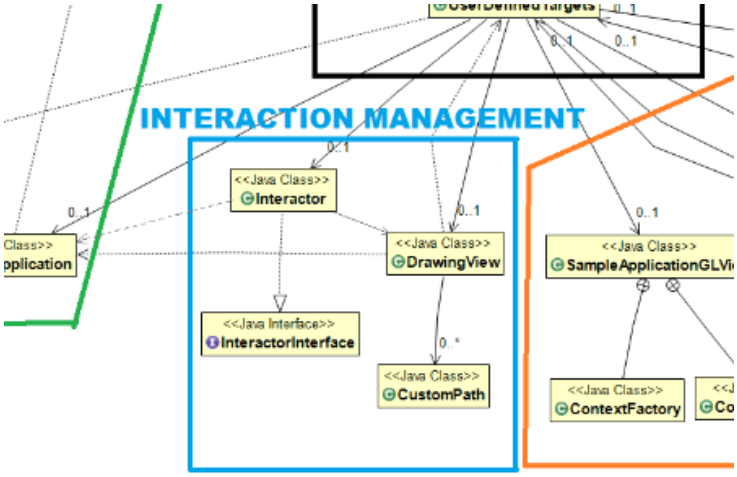


Figure 52. *Interaction Management Module*

6.7. Façade

As its name suggests, The Façade is the front side of the application. It provides a simplified interface both to the modules and to the outside world. It can be considered as the Façade in the commonly used Façade Pattern. There is only one class in the Façade. Since it is the responsibility of the Façade to handle user interface related tasks, it extends from the Android Activity class, which is the only class that can change an application's view. The Façade class deals with the application's overall lifecycle, meaning the application is launched, created, started, resumed, run, paused, restarted, stopped and destroyed through the Façade class.

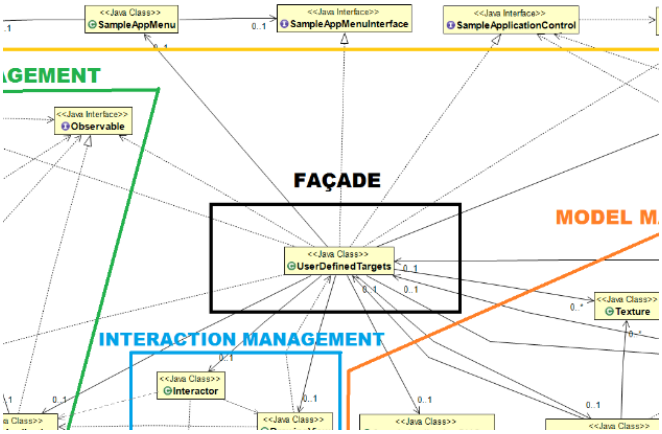


Figure 53. *The Façade*

Although there are some exceptions in our solution, ideally, it is the junction point that each request and response coming from/to the aforementioned modules must stop by. In other words, none of the modules should directly talk to each other, but indirectly through the Façade. In addition to organizing the communications between modules, some other responsibilities of the Façade are:

- Deciding what to be done when the application is launched, created, started, resumed, run, paused, restarted, stopped and destroyed
- Deciding what to be done when the device's screen configuration is changed
- Loading textures
- Listening gestures
- Handling events (touch events, button clicks, etc.)
- Showing error dialogs
- Initializing AR tasks (initializing, deinitializing, loading, unloading, starting, stopping the trackers)
- Connecting to the remote server and starting the tracker in the server
- Deciding what to be done when AR is initialized
- Deciding what to be done on each camera frame (i.e. when a pattern is detected)
- Adding overlay views to the screen
- Starting and updating the renderer
- Starting building a target
- Starting building a content
- Deciding what to be done when a new content or a new target is received
- Telling AllJoyn to update the channel state
- Setting the application's menu options and getting user commands
- Playing sounds

6.8. Conclusions

This chapter showed the approaches we took to the problem. It first critically compared the popular augmented reality libraries and stated that any modern AR library with the modularized capturing-tracking-rendering structure is a suitable candidate to fit into our solution. It gave the overall structure of our solution by explaining what are needed to create a collaborative mobile AR framework. Then, it described the each module in our solution and their interactions with each other by presenting some screenshots, class diagrams and code snippets from our application.

7. Evaluation

In this chapter, we identify some metrics and evaluate our solution in terms of modularity, reusability, extensibility and reliability. We then show the limitations of current system.

7.1. Modularity and Extensibility of the Resulting Code

In the context of software design, much work has been done so far to define the modularity of a software system. Modularity simply refers to the way that a software design is decomposed into different subparts or modules. Sarkar, for example, states that modularity can be provided by dividing the software into logical modules, publishing APIs for each module, and then guarantying that the modules access each other's resources only through the published interfaces (Sarkar, et.al., 2005). While authors vary in their definitions of modularity, they tend to agree on the concepts that lie at its heart; the notion of interdependence within modules and independence between modules (MacCormack, et.al., 2007). The former refers to high cohesion, whereas the latter refers to low coupling. However, the main problem with all the studies is the lack of a validated metric of modularity (Fenton, 1994).

Although modularity is an important design notion for every kind of software system, it is vital for a framework. Since a software framework is designed to provide a generic functionality, it must be extensible and extensibility can be provided if and only if there is a low dependency between its modules and if each module has a specific responsibility. Even though the notions of low coupling and high cohesion apply more to classes, we can use them to evaluate our proposed solution's modularity.

Sarkar notes that modularization quality is not synonymous with modularization correctness (Sarkar, et.al., 2005). The correctness can be established by checking function call dependencies at compile time and at run time. If all intermodule function calls are routed through the published API, the modularization is correct.

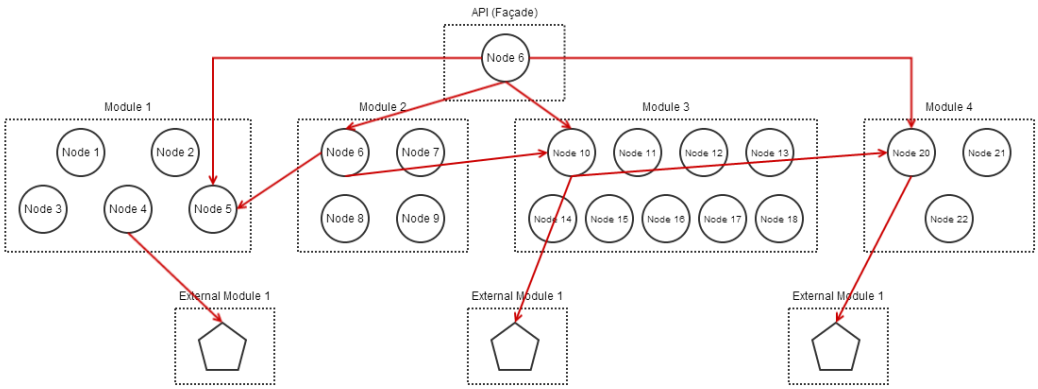


Figure 54. *Dependencies between modules*

For this purpose, we check if the modularization of our proposed solution is correct. We mentioned that each of the four modules, which are explained in the Chapter 6, has a specific task. Figure 54 shows an abstraction of the proposed solution's class diagram. Nodes represent classes, dashed rectangles represent modules, and pentagons represent external libraries. For each module, each incoming response is delivered to a specific node whilst each outgoing request is routed through a specific node. So, assuming that the modularizations of external libraries are correct, modularization of our solution is correct.

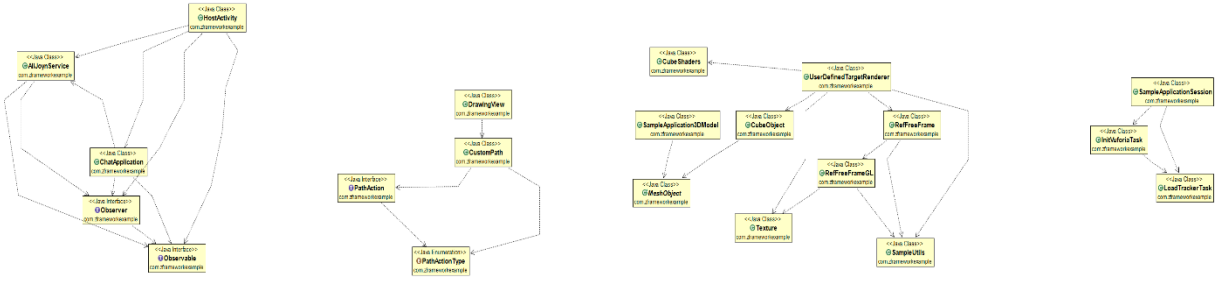


Figure 55. *Interdependencies within the modules starting from left, module 1..4*

Since the Façade is an API, we can consider it as a hot spot, meaning the users can add their code (Pree, 1994). Therefore, if we ignore it, the rest of the modules have at most one dependency. This also proves the loose coupling between the modules.

The figure 55 represents the internal connections within the modules. In order to measure the coupling value, we can consider completed graphs where every node is connected to every other node. A completed graph with n nodes is the most cohesive system possible with n nodes, where $number\ of\ edges = \frac{n(n-1)}{2}$. Our proposed solution's average cohesion value is 0.745, which is not undesirable.

	# nodes	*# edges*	*Max # edges*	*Ratio*
Module 1	5	10	10	1
Module 2	4	4	6	0.67
Module 3	9	11	36	0.31
Module 4	3	3	3	1
Average				0.745

In order to see the quality of the resulting code's base, we use the following metrics:

NOM: Number of methods – less is better
LCOM: Lack of Cohesion of Methods – measures the correlation between and the local instance variable of a class. High cohesion indicates good class subdivision. Low cohesion increases complexity.
HEFF: Halstead Effort – determines the effort required to maintain a program. The lower, the simpler the program is to change.
UWCS: Unweighted Class Size – number of methods + number of attributes of a class. Smaller class size usually indicates a better designed system with better distributed responsibilities. There are no strict rules, but above 100 is not good.

RFC: Response for Class – number of distinct methods and constructors invoked by a class. If it is high, it means there is a high complexity. Usually it should not exceed 50.
MI: Maintainability Index – determines how easy to maintain a body of code. 89 and below for low maintainability, 123 and above for high maintainability
LOC: Lines of Code – less is better

	NOM	LCOM	HEFF	UWCS	RFC	MI	LOC
Group Management Module							
AllJoynService	25	0.10	150197	68	29	81.60	490
ZApplication	49	0.02	73919	84	7	140.42	391
HostActivity	6	0.29	40213	21	7	93.09	173
Observable	2	0.00	70	2	2	198.68	5
Observer	1	0.00	54	1	1	146.31	4
Target Management Module							
SampleApplicationSession	20	0.05	73354	33	21	111.46	256
InitVuforiaTask	3	0.50	14347	4	3	101.57	66
LoadVuforiaTask	2	0.00	4081	2	2	111.09	128
Interaction Management Module							
DrawingView	11	0.11	55400	18	12	108.30	122
CustomPath	29	0.08	17290	42	24	162.96	156
Interactor	7	0.29	28717	22	8	111.08	110
InteractorInterface	5	0.02	33	5	5	193.46	7
Content Management Module							
UserDefinedTargetsRenderer	8	0.23	503170	27	8	62.27	254
RefFreeFrame	16	0.15	40908	27	16	110.55	203
RefFreeFrameGL	9	0.21	182597	30	9	90.24	216
SampleApplicationGLView	10	0.33	2236	19	10	117.97	116
SampleUtils	5	0.50	129687	6	5	38.25	133
Texture	2	0.14	43427	9	2	35.15	67
MeshObject	10	0.00	6809	10	10	180.83	55
CubeObject	4	0.42	71526	12	4	76.45	93
CubeShaders	0	0.00	2526	2	0	171.00	24

According to Pree, software frameworks consist of frozen spots and hot spots (Pree, 1994). Frozen spots remain unchanged in any instantiation of the application framework, whilst hot spots represent the parts which the users (developers) are allowed to change by adding their own code, or overriding the existing code. To understand the extensibility of our solution, we have identified these hot & frozen spots.

The Group Management Module is somewhat extensible. Most of the functions in this module are fundamental components (frozen spots) of the collaboration. However, users can define new message types (i.e. bus signals) to be sent over a bus interface and add new functionalities to sessions.

The Interaction Management Module is fully extensible. Users can define new interactions in the interaction interface and implement them in the Interactor. Our custom path class is just an example. The solution allows to define new content types.

The Content Management Module is fully extensible. Since the rendering tasks are done here, changing the existing OpenGL code is straightforward. Also the super class mesh object allows to create new models to be rendered. Since it is not dependent on the Group Management and Interaction Management Modules, changes in this module do not require changes in the other ones.

The Target Management Module is somewhat extensible. Since it depends on the features offered by Vuforia, it is not easy to make major changes.

The Façade is fully extensible. It is the actual part of the solution where the user can do whatever he/she wants.

7.2. Reliability

Reliability refers to a measure for the probability of a software failure occurring. Measuring the reliability of a framework is not an easy task because the metrics usually are not well defined. Therefore, we measure the reliability of an application that is developed by using our solution.

It is a P2P drawing application where multiple users can draw on the screen. A target is created together with the content when the user finishes drawing. The target and the content is distributed among co-located peers. To measure the reliability of this application, we identified some metrics:

- Time to distribute the content (+ ACK) – TDC (ms)
- Time to render the content – TRC (ms) (after tracking is started)
- Time to distribute to target (+ ACK) – TDT (ms)
- Number of peers - NOP
- Content size – COS (total number of vertices, may be multiple contents)
- Target size – TAS (number of pixels)
- Number of targets – NOT
- Number of contents – NOC (drawing path)

We ran the application on 5 different Android devices. Our results showed that:

- TDC is decreasingly growing as NOP grows
- TDC is linearly growing as COS grows
- TDC is increasingly growing as NOC grows
- TRC is decreasingly growing as COS grows
- TRC is increasingly growing as NOC grows
- TDT is decreasingly growing as NOP grows
- TDT is decreasingly growing as TAS grows
- TDT is increasingly growing as NOT grows

Although we observed some performance issues as number of peers and/or number of contents increase, we did not report major functional problems while running our sample drawing application (figure 56).

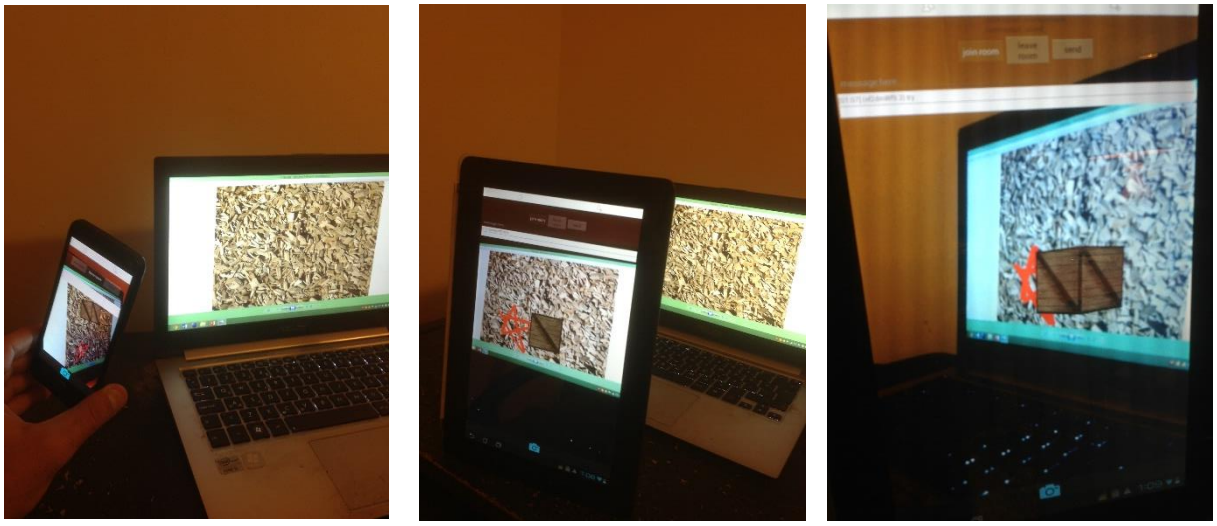


Figure 56. *Three Android devices running our sample drawing application*

7.3. Limitations

We have identified the following limitations of the current solution:

- Although distributing the on-the-fly created target was one of our primary goals, we failed to do that, because of the low level reasons mentioned in the section 5.4 and 6.6. However, this is due to the restrictions of the AR library (Vuforia) that we use in our implementation.
- Our solution does not make the rendering easier because there is no high level interface above OpenGL library. This requires users to have an OpenGL background to some extent.
- The solution requires Android API Level 14 and above.
- The Interaction Management Module currently supports only two types of interactions. Some natural effects, such as drag and drop, are missing.
- Our solution does not completely remove the need to understand the low level AR and communication libraries.
- The communication library that we use (AllJoyn) allows to send only primitive types or containers containing primitive types. Therefore, just as we failed to send user defined targets, it is also not easy for developers to send complex objects. Some third party libraries, such as serialization, parcelization, converting to XML, or to a JSON object, should be used.
- Editing an already generated content is currently not supported.
- The tracker assumes that the real world is planar. Therefore, 3D tracking is not supported.

8. Conclusion and Future Work

Although the popularity of augmented reality is increasingly growing among researchers and companies, developers and end users still cannot properly benefit from it because of the lack of tools and applications. This prevents AR from being a developer toy and thus it is considered as a niche technology. Existing AR systems, on the other hand, cannot target end users because of their complex setups and high requirements. For example, it is still quite intimidating for programmers who do not have computers graphics background or for those who do not want to deal with complicated AR components, such as image capturing, tracking or rendering.

Due to the advancements in mobile technologies, mobile devices today are much more capable of running AR applications. We believe that collaborative mobile applications play a great role in the increasing popularity of mobile devices. Therefore, in this thesis, we stated that augmented reality has also a high potential to be combined with collaborative systems. Although single user AR has shown a great promise so far, just as it did to other technologies, collaboration can take AR one step further.

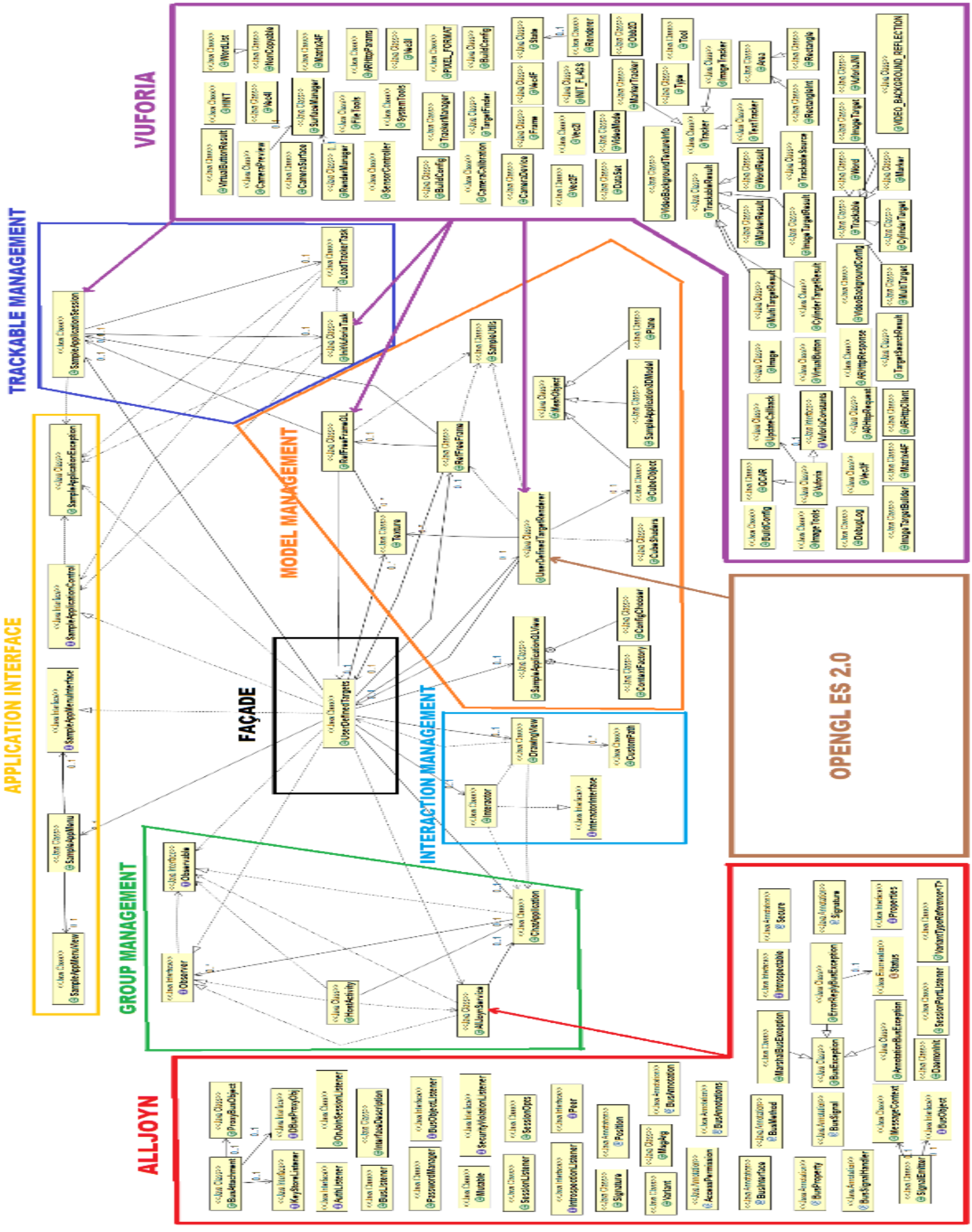
For this purpose, we proposed a way to create a mobile collaborative augmented reality framework. We identified the elements required in the design and implementation on collaborative AR applications.

Chapter 1 introduced the term augmented reality, gave some historical background, described the types of AR and presented our motivation. Chapter 2 closely looked at the existing works that are related to collaborative AR. Chapter 3 explained the pose tracking methods used in single user AR systems by often referring to Vuforia. Chapter 4 emphasized some of the popular graphics libraries for augmented reality applications and showed how they can be used to generate runtime contents. Chapter 5 analysed the most suitable distribution methods for a collaborative AR system and explained how we used AllJoyn to provide ad hoc communication. Chapter 6 presented our solution by clarifying the each module of a collaborative AR application in detail. Chapter 7 identified the metrics to evaluate our approach in terms of modularity, extensibility, reusability, reliability and finally outlined the limitations of the proposed solution.

We believe that our approach allows developers to easily create multi-user mobile AR applications in which the users can cooperatively interact with the real environment in real time. The proposed approach can increase the sense of collaborative spatial interaction without requiring complex infrastructure. Most importantly, assuming the given low level communication and AR libraries have modular structures, the proposed approach is also modular and flexible enough to adapt to their requirements without requiring any major changes.

The proposed approach enables future work to be developed on top of our solution. It is extensible to the extent of communication and AR libraries. Future work may involve 3D tracking, natural feature tracking, natural interactions with the model, different types of content generation, target distribution and server implementation.

Appendix


VUFORIA
TRACKABLE MANAGEMENT
APPLICATION INTERFACE
MODEL MANAGEMENT
FAÇADE
INTERACTION MANAGEMENT
GROUP MANAGEMENT
OPENGL ES 2.0
ALLJOYN

References

Ababsa, F., & Mallem, M. (2008). A Robust Circular Fiducial Detection Technique and Real-Time 3D Camera Tracking. *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on* (pp. 1159 - 1164). Montreal, QUE: IEEE.

Azuma, R. T. (1995). *Predictive tracking for augmented reality.* North Carolina: University of North Carolina, NC.

Barkhuus, L., Chalmers, M., Tennent, P., Hall, M., Bell, M., Sherwood, S., & Brown, B. (2005). Picking pockets on the lawn: the development of tactics and strategies in a mobile game. *UbiComp'05 Proceedings of the 7th international conference on Ubiquitous Computing* (pp. 358-374). Heidelberg: Springer-Verlag.

Billinghurst, M., & Kato, H. (2002, July 7). Collaborative augmented reality. *Communications of the ACM - How the virtual inspires the real*, pp. 64-70.

Caudell, T. (1992). Augmented reality: an application of heads-up display technology to manual manufacturing processes. *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on* (pp. 659 - 669 vol.2). Kauai, HI: IEEE.

Clouth, R. (2013). Mobile Augmented Reality as a Control Mode for Real-time Music Systems. *M.Sc Thesis*. Barcelona: Universitat Pompeu Fabra.

Ekengren, B. (2009). Mobile Augmented Reality. *M.Sc Thesis*. Stockholm: Royal Institute of Technology.

Farber, B., & Picone, M. (2010, November 3). *PeerDroid Presentation.* Retrieved from Distributed Systems Group: http://dsg.ce.unipr.it/

Fenton, N. (1994, March). Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering, 20*(3), 199-206.

Hartley, R., & Zisserman, A. (2000). *Multiple View Geometry in Computer Vision.* Cambridge University Press.

Höllerer, T. H., & Feiner, S. K. (2004). Mobile Augmented Reality . In H. A. Karimi, & A. H. (eds.), *Telegeoinformatics: Location-Based Computing and Services.* CRC Press.

Järvelä, S., Kivikangas, J. M., Kätsyri, J., & Ravaja, N. (2014). Physiological Linkage of Dyadic Gaming Experience. *Simulation and Gaming*, 25-40.

Kalkofen, D., Mendez, E., & Schmalstieg, D. (2007). Interactive Focus and Context Visualization for Augmented Reality. *ISMAR '07 Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality* (pp. 1-10). Washington, DC: IEEE Computer Society.

Kasahara, S., Heun, V., Lee, A. S., & Ishii, H. (2012). Second surface: multi-user spatial collaboration system based on augmented reality. *SA '12 SIGGRAPH Asia 2012 Emerging Technologies.* Singapore: ACM.

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

Kutulakos, K., & Vallino, J. (1996). Affine Object Representations for Calibration-Free Augmented Reality. *VRAIS '96 Proceedings of the 1996 Virtual Reality Annual International Symposium (VRAIS 96)* (p. 25). Washington, DC: IEEE Computer Society.

Lamberti, F., & Sanna, A. (2007). A streaming-based solution for remote visualization of 3D graphics on mobile devices. *Visualization and Computer Graphics, IEEE Transactions on. 13*, pp. 247-260. IEEE Computer Society.

Lee, H.-Y., & Baek, N.-H. (2009). OpenGL ES 1.1 Implementation Using OpenGL. *The Kips Transactions:parta*, 159-168.

Lee, K. (2008, March/April). Augmented Reality in. *AECT International Convention*, *56.* Jacksonville, FL.

MacCormack, A., Rusnak, J., & Baldwin, C. (2007). The impact of component modularity on design evolution: Evidence from the software industry. *Harvard Business School Working Paper*.

MacIntyre, B., Gandy, M., Bolter, J., Dow, S., & Hannigan, B. (2003). DART: The Designer's Augmented Reality Toolkit. *ISMAR '03 Proceedings of the 2nd IEEE/ACM International Symposium on Mixed and Augmented Reality* (p. 329). Washington, DC: IEEE Computer Society.

Milgram, P., Takemura, H., Utsumi, A., & Kishino, F. (1994). Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum. *Proc. SPIE 2351, Telemanipulator and Telepresence Technologies.* Boston.

Nadalutti, D., Chittaro, L., & Buttussi, F. (2006). Rendering of X3D content on mobile devices with OpenGL ES. *Web3D '06 Proceedings of the eleventh international conference on 3D web technology* (pp. 19-26). New York, NY: ACM.

Pang, Y., Yuan, M. L., Nee, A. Y., Ong, S.-K., & Youcef-Toumi, K. (2006). A markerless registration method for augmented reality based on affine properties. *User Interfaces 2006, 7th Australasian User Interface Conference (AUIC 2006)* (pp. 25-32). Hobart: Australian Computer Society.

Pree, W. (1994). Meta Patterns - A Means For Capturing the Essentials of Reusable Object-Oriented Design. *ECOOP '94 Proceedings of the 8th European Conference on Object-Oriented Programming* (pp. 150-162). London: Springer-Verlag.

Sarkar, S., Kak, A. C., & Nagaraja, N. S. (2005). Metrics for Analyzing Module Interactions in Large Software Systems. *The 12th Asia-Pacific Software Engineering Conference (APSEC'05)* (pp. 264-271). IEEE Computer Society.

Schmalstieg, D., & Reitmayr, G. (2005, December). OpenTracker: A flexible software design for three-dimensional interaction. *Virtual Reality, 9*(1), 79-92.

Schmalstieg, D., Fuhrmann, A., Hesina, G., Szalavari, Z., Encarnacao, L. M., Gervautz, M., & Purgathofer, W. (2002). The studierstube augmented reality project. *Presence: Teleoperators and Virtual Environments*, 33-54.

Siltanen, S. (2012). *Theory and applications.* Espoo: VTT.

Silverman, B. G. (1992, June). Human-Computer Collaboration. *Human-Computer Interaction, 7*(2), 165-196.

Singhal, S., & Zyda, M. (1999). *Networked virtual environments: design and implementation.* Newyork, NY: ACM Press/Addison-Wesley Publishing Co.

Sutherland, I. E. (1968). A head-mounted three dimensional display. *AFIPS '68 (Fall, part I) Proceedings of the December 9-11, 1968, fall joint computer conference, part I* (pp. 757-764). New York, NY: ACM.

Terveen, L. G. (1995, April). An Overview of Human-Computer Collaboration. *Knowledge-Based Systems, 8(2)*, 67-81. Elsevier.

Wagner, D. (2007). Experiences with Handheld Augmented Reality. *ISMAR '07 Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality* (pp. 1-13). Washington, DC: IEEE Computer Society.

Wagner, D., & Schmalstieg, D. (2007). Muddleware for Prototyping Mixed Reality Multiuser Games. *Virtual Reality Conference, 2007. VR '07. IEEE* (pp. 235-238). Charlotte, NC: IEEE.

Wagner, D., & Schmalstieg, D. (2009). History and Future of Tracking for Mobile Phone Augmented Reality. *Ubiquitous Virtual Reality, 2009. ISUVR '09. International Symposium on* (pp. 7-10). Gwangju: IEEE.

Yang, R. (2011). The study and improvement of Augmented reality based on feature matching. *Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on* (pp. 586-589). Beijing: IEEE.

Ziegler, E. (2009). Real-time markerless tracking of objects on mobile devices. *B.Sc. Thesis*. Koblenz: University of Koblenz.