Augmented Visualization

Master's Thesis

Gábor Sörös

Supervisors:

Peter Rautek, PhD
Institute of Computer Graphics and Algorithms
Vienna University of Technology

Péter Baranyi, DSc
Computer and Automation Research Institute
Hungarian Academy of Sciences

Submitted to the
Department of Telecommunications and Media Informatics
Faculty of Electrical Engineering and Informatics
Budapest University of Technology and Economics

December 10, 2010

Hallgatói nyilatkozat

Alulírott Sörös Gábor, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül saját magam készítettem, és csak a megadott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Tudomásul veszem, hogy az elkészült diplomatervben található eredményeket a Budapesti Műszaki és Gazdaságtudományi Egyetem, a feladatot kiíró egyetemi intézmény saját céljaira felhasználhatja.

Hozzájárulok, hogy jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulensek neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik.

A diplomaterv a Bécsi Műszaki Egyetem Számítógépes Grafika Tanszékén töltött nyolc hónap alatt készült 

Bécs, 2010. november 29.

Sörös Gábor

"Visibile facimvs qvod ceteri non possvnt"

to my grandmothers

Abstract

Contemporary visualization systems often make use of large monitors or projection screens to display complex information. Even very sophisticated visualization systems, that offer a wide variety of interaction possibilities and exhibit complex user interfaces, do usually not make use of additional advanced input and output devices. The interaction is typically limited to the computer mouse and a keyboard. One of the reasons for the lack of advanced interaction devices is the high cost of special hardware.

This thesis introduces the idea of *Augmented Visualization*. The aim of the project is to develop a novel interaction solution for projection walls as well as PC monitors using cheap hardware such as mobile phones or tablets. Several features of mobile devices will be exploited to improve the interaction experience. The main technical challenge of the project is to implement a solution for markerless visual tracking of the changing visualized scene. In the proposed setup, this also requires real-time wireless video streaming between the mobile device and the PC. The real-time tracking of the visualized scene will allow to estimate the six-degrees-of-freedom pose of the mobile device. The calculated position and orientation information can be used for advanced interaction metaphors like *magic lenses*. Moreover, for a group of experts who are analyzing the data in front of the same screen, we can provide a personal augmented view of the visualized scene, for each user on his/her personal device.

The thesis discusses the design questions and the implementation steps of an *Augmented Visualization System*, describes the prototype setup and presents the experimental results.

Keywords: Handheld Augmented Reality, Natural Feature Tracking, Interactive Visualization, Visualization Systems, Human-Computer Interaction

Kivonat

Napjaink vizualizációs rendszerei nagyképernyős információmegjelenítést alkalmaznak, amellyel lehetővé teszik komplex adathalmazok áttekintését is. Ám legtöbbször még a modern rendszerek sem tartalmaznak a hagyományos egéren és billentyűzeten túlmutató be- és kiviteli megoldásokat. Ennek egyik oka a speciális eszközök rendkívül magas ára.

Jelen diplomaterv a *kiterjesztett vizualizáció* koncepcióját mutatja be. A projekt célja egy újfajta ember-gép interakció kifejlesztése nagyméretű megjelenítőt használó alkalmazásokhoz. A felhasználó oldalán könnyen hozzáférhető mobil eszközöket (mobiltelefon, tablet PC) alkalmazunk, amelyek számos kedvező tulajdonsággal rendelkeznek a be- és kiviteli lehetőségek gazdagítására. A koncepció fő technikai kihívása a felhasználói eszköz helyzetének becslése a megjelenítőhöz képest egy tisztán gépi látáson alapuló megoldással. A bemutatott architektúrában ez valós idejű vezetéknélküli videoátvitelt igényel a mobil eszköz és a számítógép között. A mobil eszköz pozíciójának és orientációjának ismeretében akár olyan különleges felhasználói élmény is megvalósítható, mint például a *mágikus nagyító*. Többfelhasználós alkalmazás esetén minden egyes résztvevő a tárgyalt jelenet saját igényei szerint definiált kiterjesztett nézetét láthatja a mobil eszközén.

A diplomaterv megvizsgálja a *kiterjesztett vizualizációs rendszer* tervezési és megvalósítási kérdéseit, bemutatja az elkészült prototípust és értékeli a kísérleti eredményeket.

Kulcsszavak: mobil kiterjesztett valóság, natural feature tracking, vizualizációs rendszer, interaktív vizualizáció, ember-gép interfész

Kurzfassung

Moderne Visualisierungssysteme werden oftmals verwendet um komplexe Informationen auf großen Bildschirmen oder Projektionsflächen darzustellen. Allerdings werden sogar bei sehr komplexen Visualisierungssystemen, mit einer Vielzahl an Interaktionsmöglichkeiten, kaum zusätzliche fortgeschrittene Ein- und Ausgabegeräte verwendet. Die Interaktionsmöglichkeiten beschränken sich normalerweise auf Computermaus und Tastatur. Ein Grund dafür sind die hohen Kosten für spezialisierte Hardware.

Diese Diplomarbeit stellt die neue Idee von *Augmented Visualization* (dt. Erweiterte Visualisierung) vor. Das Ziel dieser Arbeit ist die Entwicklung eines interaktiven Ein- und Ausgabegerätes für Bildschirme oder Projektionswände, unter Verwendung von günstig erhältlicher Hardware, wie zum Beispiel Mobiltelefonen oder Tablet PCs. Um die Interaktionsmöglichkeiten zu erweitern werden verschiedene Funktionen des mobilen Gerätes verwendet. Dabei stellt das optische Tracking der sich ändernden visualisierten Szene, die größte technische Herausfordeung dar. In der vorliegenden Implementierung wird dafür unter anderem die kabellose Übertragung von Videodaten zwischen dem mobilen Gerät und dem PC in Echtzeit benötigt. Das Tracken der visualisierten Szene ermöglicht die Abschätzung der Lage des mobilen Gerätes im Raum mit sechs Freiheitsgraden (6-DOF Tracking). Die errechnete Position und Orientierung des mobilen Gerätes wird verwendet um Interaktionsmöglichkeiten wie *Magic Lenses* umzusetzen. Des weiteren kann für mehrere Experten, die die visualisierte Szene analysieren, ein zusätzliche personalisierte erweiterte Visualisierung darstellen.

Die Diplomarbeit geht auf das Design und die Implementierung eines *Augmented Visualization Systems* ein, und beschreibt den implementierten Prototypen und die experimentellen Resultate.

Schlüsselwörter: Handheld Augmented Reality, Natural Feature Tracking, Interactive Visualisierung, Visualisierungssysteme, Mensch-Maschine Interaktion

Contents

1. Introduction

Between March and November 2010, I have had the chance to study and work at the Institute of Computer Graphics and Algorithms at Vienna University of Technology having access to cutting edge Augmented Reality hardware and software tools. This master's thesis is the outcome of my research stay in Vienna.

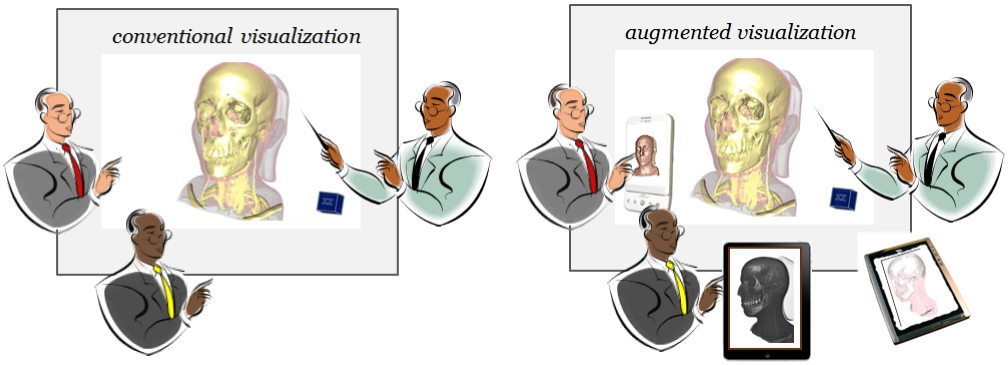
1.1 Motivation

Visualization is the process of transforming data into a visual form, enabling users to observe it. The display enables scientists and engineers to perceive features visually, which are hidden in the data, but are needed for data exploration and analysis [1]. In scientific visualization, usually a group of individuals or experts are analyzing a scene on a large monitor or projection screen. During the discussion, they may have different interests regarding their fields of specialization – for example in the medical domain a surgeon, a cardiologist and a neurologist – may want to investigate different aspects of the underlying data. Furthermore, the experts may want to interact with the visualized scene. We propose the use of their personal mobile devices for this purpose.

1.2 Idea

We aim to develop a novel interaction solution for large projection walls and PC monitors. As interaction device we will use cheap, widely available and well accepted mobile phones or tablet PCs. Our idea is that – besides the common large-screen visualization – users can see their own augmented renderings on their mobile clients according to their interests (see Fig. 1.1(a) and Fig. 1.1(b)). The users can also interact with the common visualized scene, by pressing buttons on the touch screen and by moving the device.

In our concept, the users need to face their camera-equipped handheld devices



(a) Conventional Visualization

(b) Augmented Visualization with personal overlays at the mobile user devices

Fig. 1.1: The concept of Augmented Visualization

to the screen, to get an enhanced view of the discussed content. The position of the mobile device is estimated relatively to the common screen by state-of-the-art computer vision approaches, comparing correspondences of the rendered and the captured images. The novelty in our idea is to track the common display's dynamic content (the changing visualized scene itself). We perform so called natural feature tracking on the video input of the user devices without the need for any static markers. In the proposed architecture this also requires real-time video streaming from and to the mobile client. Fig. 1.2 shows an illustration of the idea. To the best of our knowledge, no similar system exists for this scenario. The estimated 6DoF pose information and other input events (e.g., button or touch-screen events) are transmitted back to the rendering system, which creates a personal augmented view of the scene for each user. The personal augmented image is presented on top of the camera image at the user's side. We named the approach *Augmented Visualization*, where the term augmented refers to Augmented Reality (AR), a technique that overlays virtual objects and information on the real world to enhance human visual perception.

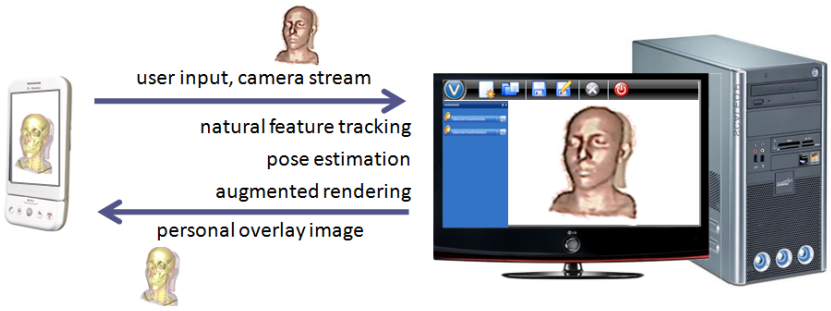


Fig. 1.2: Illustration of the Augmented Visualization System

1.3 Goals

The focus of the presented thesis is to design and implement an *Augmented Visualization System* which is capable of extending the common visualized scene with additional personal information. In this respect the specific goals are:

- Creating a reference image of the visualized scene
- Grabbing the camera image of the user's mobile device
- Transferring both the reference images and the grabbed video images to the processing unit
- Building a database of the visual features of the actual reference image
- Extracting the visual features of the camera image in real time
- Comparing and matching the extracted features with the feature database
- Estimating the camera position relative to the reference image plane
- Allowing the user to interact with the scene, based on the position and orientation information
- Rendering a personal augmented visualization or an overlay image and presenting it to the user

Although this thesis focuses on the implementation questions of a system with one screen and one mobile user, the proposed architecture was also extended to support multiple users.

1.4 Overview of the Thesis

This thesis is divided into the following chapters: Chapter 2 gives a short overview of recent related projects in augmented reality and natural feature tracking. Chapter 3 deals with the design questions of the Augmented Visualization System and introduces the selected main components. Chapter 4 presents the theoretical background of visual tracking and explains an algorithm for tracking natural features in detail. Chapter 5 is devoted to technical standards used in this project for mobile video streaming. Chapter 6 describes implementation issues and solutions to technical challenges. It further presents the tests of the individual components. Chapter 7 shows the results of our implementation, analyzes the limits of the proposed technique and indicates some other future applications, while Chapter 8 summarizes the content of the thesis and its contributions with respect to the stated goals.

2. Related Work

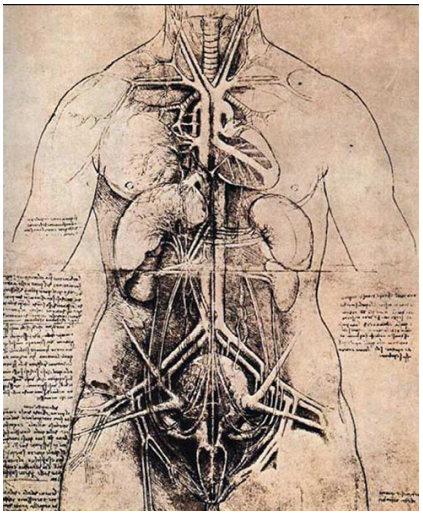
2.1 Augmented Reality: A Bridge Between Pervasive Computing and Visualization

The common technology for many modern applications like systems for architectural planning, driving assistance, medical visualization, interactive tourist guidance and sports broadcasting replays is Augmented Reality (AR). This chapter gives a short overview of recent related projects.

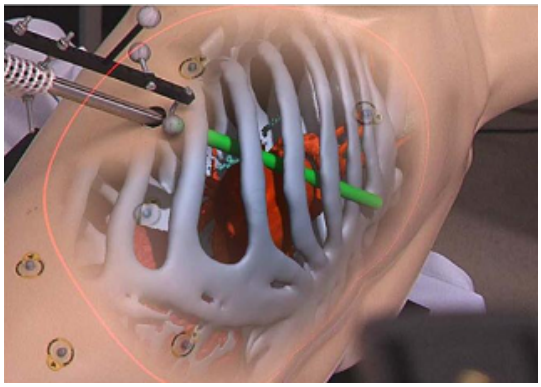
As stated in Azuma's definition [2] (1997), an augmented reality system fulfills three criteria: a) it combines real and virtual worlds, b) it is interactive in real-time and c) it is registered in space. Several projects focus on medical domain applications. Bichlmeier et al [4] (2007) present an augmented surgical training system to provide 'X-Ray Vision' for the doctor. The 3D body data is scanned by CT or MRI, and virtual organs are presented as overlays on the camera image. To fulfil Azuma's third criterion, real-world objects need to be tracked in a common coordinate system, to correctly adjust the position and orientation of the virtual objects. In the mentioned paper infra-red retro-reflective markers are attached to the patient's body and are used to register the virtual and the real body.

Kalkofen and Schmalstieg [3] (2007) introduce a *magic lens* metaphor to provide interactive visual access to occluded objects. The lens is used to discover hidden structures, which are drawn distinctively from to the objects laying outside the field of the lens. We also apply this magic lens metaphor in one of our application conceptions. In Fig. 2.1 four examples of the magic lens metaphor are shown.

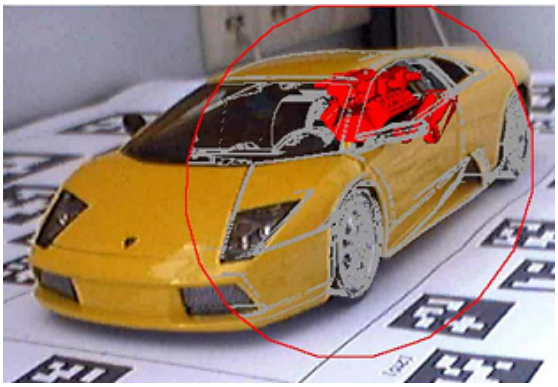
From the pervasive computing perspective, augmented reality is thought to be an enticing universal user interface for all the intelligent devices in our neighborhood. Slay and Thomas [6] (2006) describe a universal interaction controller, a user interface framework and device, designed to support interactions in ubiquitous computing environments, and the in-situ visualization of ambient information in environments equipped with multiple heterogeneous displays. In their scenario, mobile devices are connected with public displays and take the role of a controller to interact with the content shown on the displays. The mobile phone can also act as a clipboard,



(a) Leonardo da Vinci (16th century)



(b) Christoph Bichlmeier (2007)



(c) Dennis Kalkofen (2007)

(d) Gábor Sörös (concept, 2010)

Fig. 2.1: The magic lens metaphor

to temporarily store information, and transfer it between different displays. In an earlier work, Slay et al. [5] (2001) apply artificial markers to interact with a virtual scene in a very intuitive way, by showing markers as commands. They use ARToolKit[1] for fiducial marker detection, which is a predecessor of Studierstube ES [46], an augmented reality framework used in this thesis.

Ballagas et al. [7] (2005) developed a mouse-click function for large screens using ghosted fiducial markers. They appear on a regular grid on the screen, at the time the user clicks with the mobile phone. The recorded image is used to localize the position of the click. A significant drawback of this method is the use of additional 2D barcodes to determine the position of the camera. Further, this method is only meant to use the mobile phone like a computer mouse in order to drag and drop elements on the screen (see Fig. 2.2(a)). During the last years, the recognition of visual markers with mobile phones has become a widespread technology for interaction with objects from the real world. The information gathered through mobile image processing serves as physical hyperlinks to access actual object-related information thus bridging the real world and associated digital information. The breakthrough to avoid artificial markers has already been achieved.

In 2007, Boring et al. [8] presented the Shoot & Copy technique for recognizing icons on advertisement displays (Fig. 2.2(b)). The user can simply take a picture of the information of interest, and the system then retrieves the actual data represented on the screen, such as a stock quote, news text, or a piece of music. The technique does not require visual codes that interfere with the shown content. The captured image region is analyzed on a server (the display's host computer), which compares the image to its own screen content using computer vision and identifies the captured region. A reference to the corresponding data is sent back to the mobile phone. Once the user has time to view the information in more detail, the system allows retrieving the actual data from this reference. Of course only previously stored and analyzed advertisements can be recognized this way. The whole screen or display is used as rich input into the image recognition process. The technology for communicating information via visual patterns blends in with the media that display them. This is the first approach to use the screen content itself as the marker.

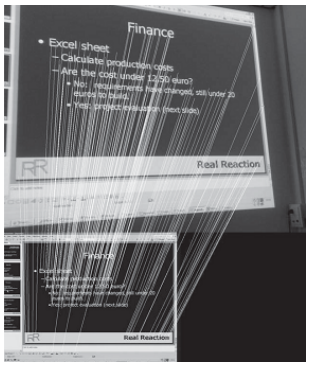
Quack et al. [9] (2008) present a slide-tagging application for smart meeting rooms. The users have the possibility to use their camera phones to click on slides or sections of slides, that are being presented on a projection screen. The user takes a photo of the presented slide using the mobile phone. The phone sends a query to the processing server, where scale-invariant local features are extracted from the

[1] ARToolKit, `http://www.hitl.washington.edu/artoolkit`

photo. Then for each feature a nearest-neighbor search in the reference database of the presentation's slides is executed. The resulting potential matches are verified using projective geometry constraints (Fig. 2.2(c)). This way the actual slide of the presentation can unambiguously be determined. The user gets the information present on the slide to record it for his notes or to add tags.

(a) Point & Shoot [7] (2005)

(b) Shoot & Copy [8] (2007)



(c) Slide recognition [9] (2008)

Fig. 2.2: Application ideas in pervasive computing

We combine the hidden data exploration and the controller perspectives in one technique, and create an extension for an existing visualization system. Our application scenarios are mainly from the medical domain, but we do not deal with intraoperative AR, where the target is the patient's body itself. We apply the magic lens metaphor, where the handheld device corresponds to the lens. Moreover, we neither want to disturb the projected scene nor the displaying screen with any artificial markers, so we aim to apply a markerless tracking solution instead.

2.2 Natural Feature Tracking

Natural feature tracking is ideal for many visual tracking situations, because (unlike all other known techniques) no additional technical equipment is needed in the working area. Natural objects are detected and collected to build a knowledge

database of the environment. The most successful methods rely on wide baseline matching techniques, that are based on sparse features such as scale invariant interest points and local descriptors. The basic idea behind these methods is to compute the position of a query image with respect to a database of registered reference images, planar surfaces or 3D models. The drawback of these methods is their high computational cost.

In 1999, Neumann and You [11] first proposed an AR pose tracking system using a-priori unknown natural features: points and patches of a target image. Their closed-loop combination of feature selection, motion estimation and evaluation achieved robust 2D tracking.

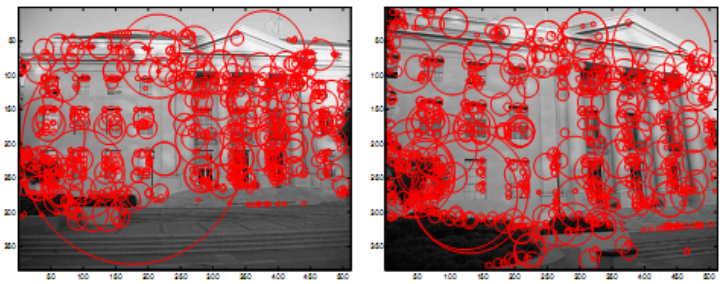
Robertson and Cippola [12] (2004) present an image-based system for urban navigation. The user can take a query photo and the system compares it with previously stored and geotagged images of a street. From correspondences between the query image and the collection of rectified facades, the user's position can be estimated. To cope with large changes of viewpoint, image features are characterized in a way that is invariant to image rotation and scale. This is referred to as wide-baseline matching.

A significant advance was the integration of point features with edge-based trackers, which has been demonstrated for instance by Rosten & Drummond [24] (2005). The tracking of feature points on images has the advantage that the surrounding patch as descriptor can be followed from one frame to another. The edge-based trackers detect whether a prior estimate of the edge is present in the image or not. The disadvantage of using descriptors for interest points is that they are often of limited invariance to aspect and lighting changes. Edges on the other hand are invariant to pose and illumination changes [13].

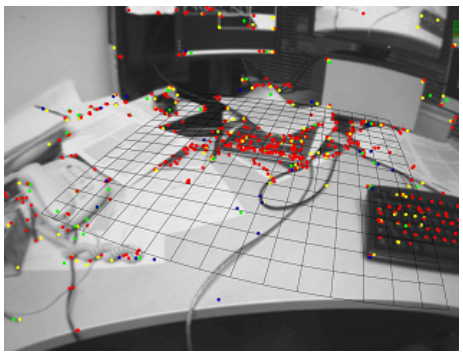
Natural features bring inexpensive augmented reality to unaltered environments. In 2006, Reitmayr and Drummond [14] presented an outdoor AR system with a stand-alone ultra-mobile PC, which was able to localize itself by detecting edges of buildings, comparing with a previously stored, textured 3D model of the street. Zhang and Kosecka [15] (2006) presented a system for image based localization in urban environments. Given a database of views of street scenes, tagged by GPS locations, the system computes the location (longitude and latitude) of a novel query view.

A method called Simultaneous Localization and Mapping (SLAM) came from the robotics community. A SLAM system attempts to add previously unknown scene elements to its initial map. These newly added images provide registration even

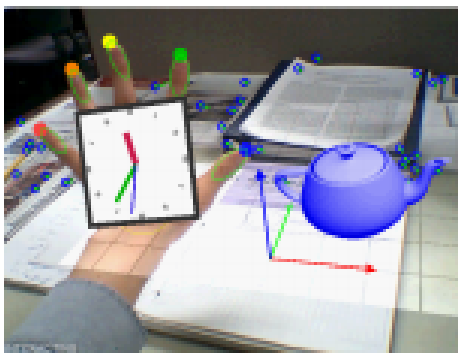
when the original map is out of sensing range[2]. Georg Klein's Parallel Tracking and Mapping (PTAM) [16] (2007) supersedes the method of SLAM in terms of complete absence of an initialization target. PTAM separates the motion tracking and the map building processes of the environment into parallel threads. Thus, it speeds up tracking on every frame significantly while concentrating only on the most useful keyframes in mapping. Klein later also ported the algorithm to iOS (the operating system of the iPhone).



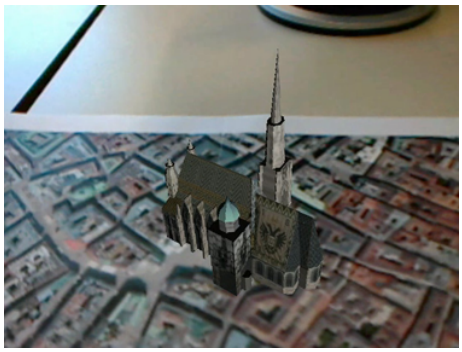
(a) Image-Based Localization [15] (2006)



(b) Parallel Tracking and Mapping [16] (2007)



(c) Hybrid Features [17] (2008)



(d) AR using Studierstube NFT Tracker [23] (2010)

Fig. 2.3: Examples of natural feature tracking approaches

[2] This is called extensible tracking in the AR community

Lee and Höllerer [17] (2008) present a hybrid feature tracking method that uses the bare hand's fingertips for initial camera pose estimation and combines natural features with optical flow tracking. The system runs in real-time on modern mobile phones.

Our idea was inspired by recent works of Wagner and Schmalstieg [18] [19] [21] [22] [23], who developed a multi-platform handheld augmented reality framework called Studierstube ES, which is also capable of real-time natural feature tracking.

3. Design

3.1 Application Scenarios

In our prototype implementation we extend an existing volume visualization system. It processes volumetric data sets and renders 2D views of the content. Such a volume data set is for instance the outcome of a CT/MRI scan, where the tissue density of the body is sampled on a 3D rectilinear lattice. In volume rendering, transfer functions are used to define the color and opacity values of volume elements (voxels) with different density. Thus, the transfer function determines what is shown and how it is colored on the rendered image[1].

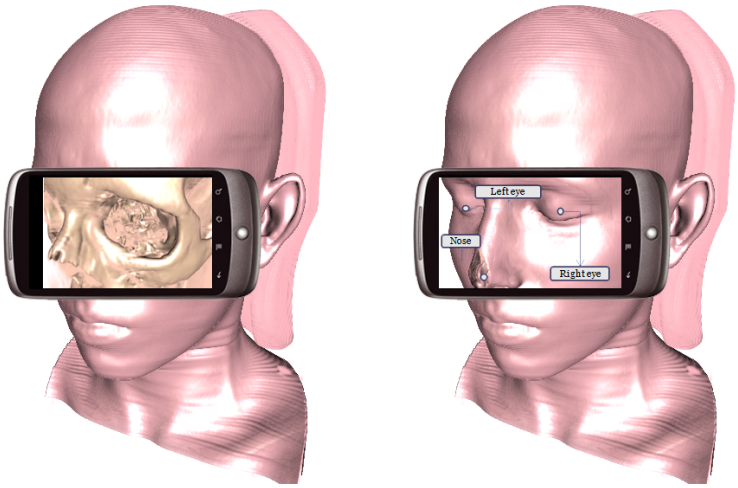
We have defined different interaction scenarios from which two have been realized in the scope of this thesis project. Fig. 3.1(a) shows the concept of a magic lens for our system as described earlier. For this application the position and orientation of the client device must be estimated very precisely. The pose matrix calculated by the system is combined with the modelview matrix of the original scene during rendering (plus zooming, transformation, etc.). It appears to the user as if the mobile could see through the skin. The overlay displays the same dataset just rendered using a different transfer function to see the bones inside the body. If the viewpoint of the original model changes, the overlay should change accordingly. The ideal solution would be to use a front camera of the handheld device to also track the user's eyes, and rotate the camera on the backside in order to give a real see-through feeling.

A second application idea is to modify some parameter of the rendering depending on the position or orientation of the camera, for instance interpolating between two or more transfer functions while the user walks from the left to the right of the screen.

If the volume data is accompanied by segmentation information[2] then direct scene annotation becomes possible. We cast rays from the known user position into the volume and label the scene at intersections with objects (see Fig. 3.1(b)).

[1] In computer graphics applications, the advantage of a volumetric model compared to a triangle mesh is that it also contains all the hidden inner information.

[2] e.g., in a medical dataset this means that we know which voxel belongs to which organ of the patient



(a) X-ray vision with magic lens (b) Direct scene annotations

Fig. 3.1: Application concepts

In a similar use case there is no overlay at all, the user can see simply the camera preview on the screen. If the screen is touched, a ray is cast into the volume and the user can interact (select, brush, etc.) with the data at a specific location.

The following section gives an overview on the basic considerations of designing such an augmented visualization system, and then the applied components are described in detail.

3.2 Overview Considerations

Visualization (e.g., volume rendering) is a computationally demanding task. Therefore the **Rendering Component** needs to run on a special machine (i.e., a high-end PC) to achieve interactive frame rates. The overlay image will be a different view of the same data, so it is advantageous to render it on the same machine. We want to minimize the dependencies between the rendering part and the other parts of the system to make the visualization software easily interchangeable.

The heart of the application is the reliable tracking of the user devices, because their position determines the overlays to be rendered and their motion is translated to interaction with the scene. We need to track the client devices with six degrees of freedom (three translation parameters x, y, z and three orientation parameters *yaw*, *pitch*, *roll*) to know each client's viewpoint. The tracking system is required to capture the trajectory and deliver the current object pose in real-time.

To keep the system modular, we define the **Tracker Component** separated

from the other parts, which will deal with the computer vision algorithms: feature detection, feature description, database building, feature matching, and pose estimation. The input to the Tracker component are the target and the camera image. The output is a 4×4 pose matrix. The typical resolution of a projected reference image is about 1024×768 pixels (which corresponds to thousands of features), and a typical captured camera image is about 320×240 (which corresponds to hundreds of features). As we want to compare these two continuously changing image streams (the interactively rendered scene and the captured video) in a wireless infrastructure, it turns out to be beneficial to run the Tracker not on the mobile device but on a PC. The video images are much smaller than the reference images and therefore data transfer is reduced. Another option to reduce bandwidth needs is to extract the features of the captured image on the mobile device, and to transfer only the descriptors over the wireless network. We have postponed this option for later experiments. In conclusion, we decided for a PC-side Tracker and real-time streaming of the camera images from the mobile device to the PC. With this approach it is also possible to exploit existing tracking libraries for PCs and to avoid any dependency on a specific mobile device model.

The **Mobile Component** will be treated as the third major part, with all the user interaction features encapsulated. For data transfer in such an indoor usecase IEEE 802.11b/g WiFi connection with up to 54Mbit/s data bandwidth is the most suitable[3]. Today's high-end mobiles are already equipped with WiFi b/g antennas, but not with 802.11n, which would increase the available bandwidth by a factor of two. One serious drawback of our approach is that because of the wireless video streaming the number of clients is limited by the connection bandwidth.

Fig. 3.2 shows the main system components of the Augmented Visualization System (AVS) and indicates their functions in time order: 1.) A scene is rendered and presented on the common screen and simultaneously sent to the Tracker for feature extraction; 2.) The Mobile device captures the common content and 3.) continuously streams to the Tracker; 4.) the Tracker component estimates a relative pose from the two images and sends it back to the Mobile device and to the Rendering component to 5.) enhance user interaction and to 6.) render the personal overlay. The three major building blocks are described more detailed in the following.

[3] among the available EDGE, HSPA, Bluetooth and WiFi radio connections

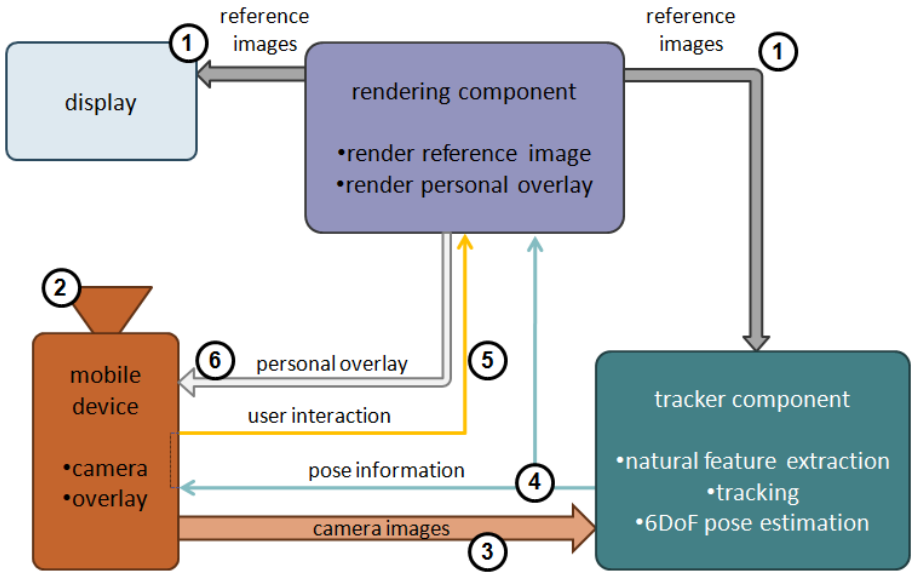


Fig. 3.2: Architecture of the augmented visualization system

3.3 The Rendering Component

The content of the common screen is delivered from a readily available visualization software. At the present, reference images originate from a volume rendering software named VolumeShop [39]. It has been developed at the Institute of Computer Graphics, Vienna University of Technology for several years. Fig. 3.3 shows the basic user interface of VolumeShop. Note the transfer-function editor in the top-right corner. Below the editor, the parameters of the plugin can be manually adjusted. Different types of plugins are implemented, mainly for visualization research purposes. All parameters are stored in an XML file, which describes the whole visualization session. The parameters are potential targets of our investigations to be changed according to the movements of the mobile device. The properties of different plugins and framework elements can be linked together in VolumeShop. Changing one parameter in a plugin also causes a refresh of all linked parameters. We extended this software with additional plugins to enable remote user input from the mobile device.

The JPEG-encoded reference (target) images are streamed from VolumeShop into the Tracker through wired TCP/IP connection. The JPEG-encoding capability is already implemented in the Qt-based[4] plugin framework. To each viewport belongs a TCP server on the computer running VolumeShop, and clients that are interested in the content change can connect to this server. Text messages can also

[4] `http://qt.nokia.com`

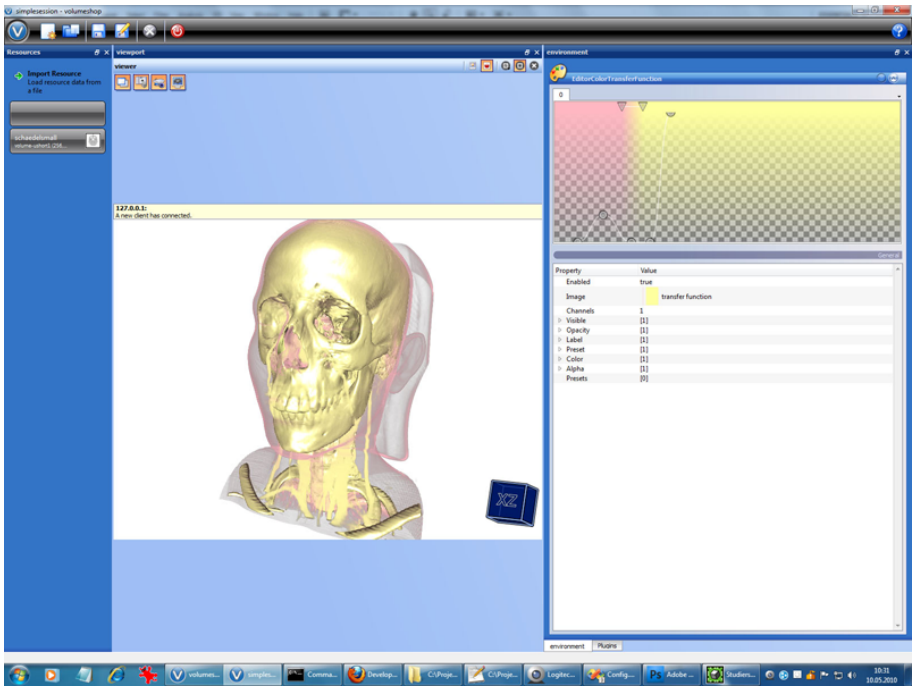


Fig. 3.3: User interface of the VolumeShop rendering framework

be injected the other way around. VolumeShop streams only when a new image is rendered. The same interface can be used to get the overlay renderings produced in a secondary viewport.

To inject interaction events into the rendering software, an additional plugin is needed. As all the rendering properties are stored in XML format, and events occur in an asynchronous manner, it is a straightforward idea to implement a plugin that opens up a UDP port and parses remote XML commands. The XML format shall support different data types including integer, float and matrix. To produce the overlay for the user, a secondary viewport needs to be set up with own parameters and rendering properties. These parameters, for instance the modelview matrix, or a transfer function value can be linked to the remote interactor plugin's properties. This way the user can remote control the properties and parameters of the visualization system.

3.4 The Tracker Component

The Studierstube ES tracking framework has been selected as the Tracker component of the Augmented Visualization System. In 2003, Wagner et al. [18] ported the ARToolKit augmented reality library to Windows CE and thus created the first self-contained AR application on an off-the-shelf embedded device.

This port later evolved into the ARToolKitPlus library, and in the recent years into Studierstube ES[5] at the Christian Doppler Laboratory for Handheld Augmented Reality, Graz, Austria. It is a multi-purpose augmented reality framework for both PC and mobile use. The software was recently bought by the company Qualcomm and is at the moment publicly not available[6]. Based on a cooperation agreement with the Christian Doppler Laboratory, I have had access to the tracking library in binary form for the PC platform. I have also spent several weeks in Graz to make modifications and to write missing functionality for our project.

To get myself familiar with the framework and for testing a marker-based puzzle application was implemented, which can be seen in Fig. 3.4. The left image shows typical markers that are tracked by the software. The right image is augmented with virtual puzzle pieces on top of the detected markers. My later work focused only on natural feature tracking, however, the framework contains rich functionality as for instance windowing support, camera access, tracking, rendering, audio, network, fixed-point mathematics, etc. Studierstube ES is extendable by user applications that utilize its building blocks. The biggest advantage of the framework is that it enables writing portable applications which run both on the PC and several mobile platforms. User applications are written in C++ and the framework can be parameterized by XML-files. For more on Studierstube ES, see [46].

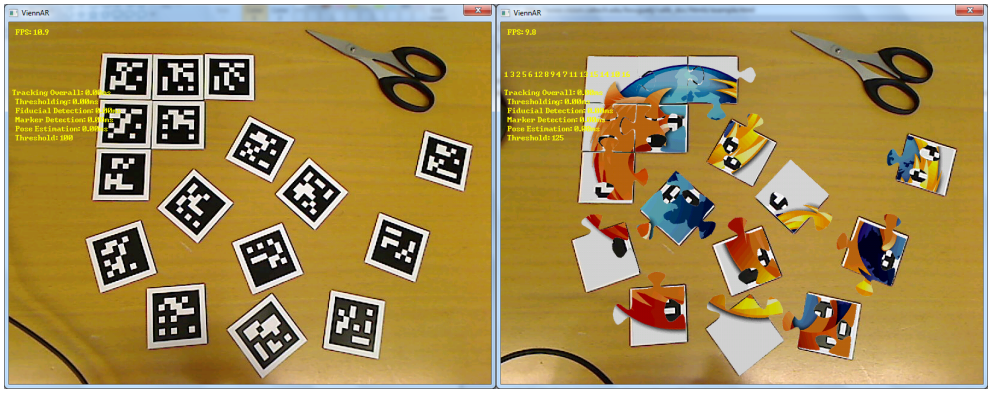


Fig. 3.4: The AR puzzle game for testing purposes

The actual component that performs tracking is embedded in the framework and is called StbTracker. It is a complex computer vision library and has several features for both marker-based and markerless augmented reality applications. Among others

[5] Its name Studierstube comes from the German word for the study room in which Goethe's famous character Faust tries to acquire knowledge and enlightenment. Actually, Studierstube was the project name of former virtual reality efforts of the group and the ES refers to the newly targeted embedded platforms.

[6] Qualcomm has renamed the product to QCAR

it performs feature detection, feature matching, pose estimation, and noise filtering. StbTracker has been designed to support PCs as well as mobile phones. Hence, its memory requirements are very low and processing is very fast. The theoretical background of the applied methods and algorithms behind natural feature tracking are described in section 4.1. Although the StbTracker can perform feature tracking on several types of mobile phones[7], we decided to stream the small camera images from the mobile device to the StbTracker running on a Windows PC.

Our task was to create an application that runs in the Studierstube ES framework and exploits its natural feature tracking capabilities. The Studierstube ES was designed to detect a static target texture on a single video input of a real camera (under Windows, this requires access to a webcam through a DirectShow filter chain). We had to find a solution to inject the mobile camera images through the wireless network as if they were originating from a real device. Further, the Tracker component also needed to be modified to allow the interactive change of our dynamic target texture and recalculation of the target feature database at runtime.

3.5 The Mobile Component

As interaction device, we have selected the Google Nexus One mobile phone with the Android 2.2 operating system because of the broad set of features (IEEE 802.11 b/g and Bluetooth connections, touch-screen, Qualcomm Snapdragon 1GHz CPU, 512MB DRAM, 5MegaPixel autofocus camera, hardware support for H.263 video and JPEG image encoding, etc.) and because of the openness of the operating system. Android is one of the first open-source mobile application platforms[8], developed by Google and the Open Handset Alliance. It includes a Linux-based operating system, a rich user interface, end-user applications, code libraries, application frameworks, multimedia support and much more. While components of the underlying operating system are written in C or C++, user applications are built for Android mainly using the Java programming language and run on the Dalvik Virtual Machine. An emulator for Windows, Linux and MacOS and an extensive open-source SDK from Google are also available. One of the most powerful features of the Android platform is that there is no difference between the built-in applications shipped on the device and applications created with the SDK. This means that powerful applications can be written to tap into the resources available on the device [40].

[7] The Android platform was not featured by the time of the implementation
[8] Android is *not* a hardware platform

The decision to use Studierstube ES on the PC and not directly on the mobile device brings up a new problem. The mobile camera images need to be streamed to the Tracker component on the host PC. Studierstube ES uses a modified version of DSVideoLib[9] to get input images from a webcam attached to the PC. Our first attempt was to simulate the phone stream as a virtual webcam device. Unfortunately, this is very cumbersome in Windows through a chain of DirectShow filters and so we had to reject this approach.

After several discussions with the developers of Studiertsube ES we closed an agreement that they implement a new network video sink (TCP/IP, RTP, RTSP, etc.) in the tracker if we solve the streaming task from the mobile to the PC, including the compressing and decompressing steps of the video. The tracker needs high quality images with a resolution of 320×240 pixels for robust pose estimation. With $8bit$ uncompressed gray values such an image takes about $320 \times 240 \times 1Byte = 75kB$. With $25fps$ this means almost $2MB/s$ transfer bandwidth need. Without compression even the WiFi access is too slow for multiple users. There are several possibilities, two of them are especially attractive with the chosen Nexus One phone: to use the built-in H.263 video encoder and stream the video per RTP/RTSP or to use the built-in JPEG encoder and send individual frames. We decided to neglect software video encoding (e.g., by using the open-source x264) because we expected high processing costs and thus significant delay on the selected phone model.

In our mobile phone application we grab and compress the camera images, stream them to the Tracker component either as continuous video or separate frames. Further, we intercept user input (touch screen or accelerometer events) and send them to the visualization system. Finally, the client must be able to receive and show the VolumeShop-generated overlay imagery. The following chapters present the theoretical background of visual tracking and the technical standards used in this project for mobile video streaming.

[9] DSVideoLib, `http://sourceforge.net/projects/dsvideolib/`

4. Tracking

Six main technologies for tracking do exist: inertial, acoustic, magnetic, mechanical, optical, and radio (GPS) typically for outdoor use. These technologies can be compared in accuracy, latency, robustness, size of tracked area, number of tracked objects, need of line of sight, price, wired/wireless, size/weight of sensors/emitters, and many other parameters. For a good survey of tracking technologies, refer to [10].

This project aims to put aside the high priced special tracking devices and focuses only on optical tracking with off-the-shelf mobile phones or tablets. In terms of visual tracking two different scenarios are possible:

Stationary Observer - "Outside-In Tracking" Outside-in systems have the sensors mounted at fixed locations in the scene. There is a known fixed relationship between scene- and camera-coordinate system. Objects to be tracked are marked with passive or active markers. The number of markers needed on each object depends on the number of degrees of freedom each object is to be tracked. Additional markers can be used to increase redundancy, to improve the pose estimation and to overcome occlusion problems. For our setup this would require one or more cameras to be fixed at the monitor or projection screen. The video stream of these cameras could be used to detect and track the mobile device. However, the mobile device is small and it is going to be very often occluded by the user's hands. Moreover, we do not want to attach any markers on it. Since we do not want to make intrusive changes to the visualization system's setup either, and also want to omit complicated calibration procedures, this approach is not suitable for our system.

Moving Observer - "Inside-Out Tracking" In case of inside-out systems sensors are attached directly to the object which is to be tracked. This means that the camera itself is the object or it is attached to the object. Camera coordinates have to be recovered and the relation between scene-coordinates and camera-coordinates has to be calculated. The camera observes the scene from inside, and the scene is equipped with markers or previously known reference targets. These two methods can also be combined (see for instance the mentioned paper [4]).

A built-in camera is nowadays available on most mobile devices, making them suitable for computer-vision approaches. However, the quality of computer-vision-based tracking is strongly influenced by the camera and image sensor characteristics such as frame size, update rate, color depth or lens distortion, which tend to be rather poor on handheld devices. Combination with other sensors such as a gyroscope, accelerometer, compass or GPS can enhance the capabilities of inside-out tracking, but the implementation of a hybrid tracking approach is out of the scope of this thesis.

Efficient and robust techniques for tracking fiducial markers do exist. The use of artificial black&white patches or retro-reflective markers is very invasive and might lead to lower user acceptance. However, with contemporary advances in technology it is possible to track an arbitrary (previously known) texture, if it contains enough distinctive features. In the computer vision literature this is referred to as natural feature tracking. Since this technology allows real-time tracking and 6DoF pose estimation without any intrusive changes to the visualization system, we chose to implement this technique.

4.1 The Vision-Based Tracking Algorithm

Sophisticated computer vision algorithms make it possible to perform pure visual tracking by using simple camera images. In our system the goal is to track the actually rendered image. Therefore, we restrict our investigations to planar targets. The search for discrete point correspondences between two images consists of three main steps: a) *interest point detection*, b) *feature vector generation* and c) *matching*. First, distinctive interest points (e.g. corners) of the reference image are extracted, then the patches around the keypoints are described as feature vectors and a database of features is built. Then, similarly, features of the captured image are extracted and described, and these feature vectors are compared to the database elements. If one finds at least three correspondences, the transformation between the two images can be calculated, so a pose estimation of the camera relative to the projection wall becomes possible. The here mentioned computer vision methods are widely applied for object detection, recognition, aerial image registration and panorama stitching as well.

4.1.1 Interest Point Detection

Corner detection is the first step of the computer vision pipeline in many tasks such as tracking, object recognition, localization and image matching. A large number

of corner detectors exist in the literature (a good survey can be found in [25]), among them the FAST (Features from Accelerated Segment Test) corner detector is excellent for speed-critical applications, and is also used in Studierstube ES Tracker. A test is performed for a feature at a pixel P by examining a circle of 16 pixels (Bresenham circle of radius 3) surrounding P.

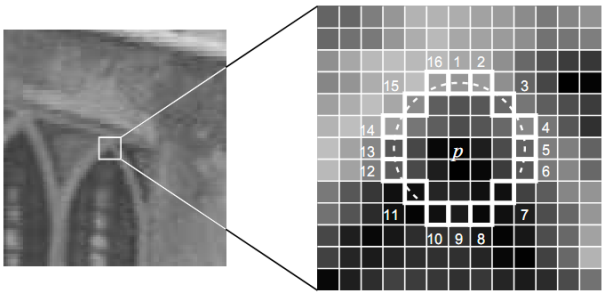


Fig. 4.1: FAST corner detection: 12 point segment test in a patch (image from [25])

An interest point (also called keypoint) is detected at pixel P (see Fig. 4.1) if the intensities of at least 12 contiguous pixels are all above or all below the intensity of P by a given threshold [24] [25]. The highlighted squares in Fig. 4.1 are the pixels used in the corner detection. The pixel at P is the center of a candidate corner. The segment length has been chosen to be twelve because it admits a high-speed test which can be used to exclude a very large number of non-corners: the test examines only the four pixels at 1, 5, 9 and 13 (the four compass directions). If P is an interest point then at least three of these must all be brighter than $I_P + t$ or darker than $I_P - t$, where I_P is the intensity of P and t is the given threshold. If neither of these two cases is true, P is not classified as a corner. The full segment test criterion can then be applied to the remaining candidates by examining all pixels in the circle. The arc indicated by the dashed line in Fig. 4.1 passes through 12 contiguous pixels which are brighter than P by more than the threshold. Typically, about thousand interest points are detected in an image of resolution 640×480, however, this number depends on the image content and the applied threshold.

4.1.2 Feature Descriptors

After extracting features (interest points) of an image, they need to be represented unambiguously in an affine-invariant way. This usually involves scale selection, rotation correction and intensity normalization. Two feature descriptors are used in Studierstube ES, the PhonySIFT and the PhonySURF [23].

PhonySIFT

The PhonySIFT descriptor is a heavily modified version of the SIFT (Scale Invariant Feature Transform), which was invented by Lowe [26] in 2004. Although SIFT refers to the whole chain of detection and description of features, often only the description is meant by this expression. To speed up calculations, the original method of interest point detection (very time-consuming calculations with Difference-of-Gaussians in the scale space) has been replaced by the above mentioned FAST corner detector. Since this approach does not estimate the feature's scale anymore, the resulting descriptor is not scale-invariant anymore. To reintroduce scale estimation, the descriptor database is built for a number of scaled versions of the original image. Then, on each level, around each interest point, a 15×15 pixels wide patch is observed. The patch is divided into 9 subregions (5×5 pixels each), the subregions are blurred with a Gaussian Kernel to gain more robustness against affine transformations and slightly incorrect interest point positions. To estimate the main orientation of the patch, a gradient histogram is used. For all image pixels the gradient direction and magnitude are calculated. The direction is quantized to [0..35] and histograms over the sub-regions are built using 36 bins of possible directions. The gradient magnitude is weighted using a distance measure and is added to the respective bin. In Fig. 4.2, an example for four patches (4×4 pixels each) is shown. The four descriptors are constructed using the above mentioned method. The resulting histogram is searched for peaks. If more than three peaks are found, the feature is discarded. For each detected orientation peak, the patch is rotated using sub-pixel accuracy so that all features become orientation-invariant.

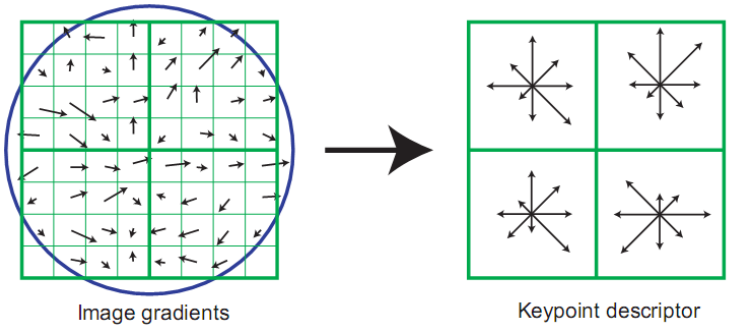


Fig. 4.2: Creating a SIFT descriptor (image from [26])

After assigning an image location, a scale and an orientation to each interest point, descriptor vectors are created: Gradient magnitudes and orientations are estimated again and weighted by the distance from the patch center as well as from the sub-region center. Histograms are created but in this case only 4 orientation

bins are used. The weighted magnitudes are then written into the histogram bins for every sub-region [19]. The descriptor is formed from a vector containing the values of all the orientation histogram entries, corresponding to the length of the summed gradient vectors in each of the 4 bins. There are 9 histograms corresponding to the 9 sub-regions of the patch, so the final descriptor vector has $9 \times 4 = 36$ dimensions. At the end, the descriptor vector is normalized to unit length to compensate illumination changes.

PhonySURF

SURF (Speeded Up Robust Features) [27] is a scale- and rotation-invariant interest point detector and descriptor (however, only its descriptor method is referred here). The original SURF detects interest points using a Hessian matrix-based measure on integral images (also called Summed Area Tables) convolved with second-order derivatives of a Gaussian. In Studierstube Tracker, the interest point detection is replaced by the mentioned FAST detector. The PhonySURF descriptor characterizes the distribution of the intensity content within the interest point neighborhood, similar to the gradient information extracted by PhonySIFT, but it builds on the distribution of first order Haar-wavelet responses in x and y directions rather than the gradient. It uses a window of 32×32 pixels, divided into 8×8-sized subwindows. The descriptor consists of the histogram values of the wavelet-responses in the sixteen subwindows and results therefore in a 128-dimension vector.

4.1.3 Matching

Matching the features of the captured image with the database is done with a brute force approach in our case. Testing every feature against every other feature leads to a computational complexity of $O(n^2)$. As the database is not too large (approx. thousand reference vectors) and the database changes frequently, the brute force approach is not slower or even outperforms the use of any search structure (e.g. SpillTrees, which are used by Studierstube ES in other applications [21]). To compare the feature vectors, Euclidean distance is used.

4.1.4 Outlier Removal and Pose Estimation

Although the presented descriptors are highly distinctive, they still produce outliers in the matching step and those must be removed before doing pose estimation. The outlier removal in Studierstube ES works in three steps. The first step uses the

feature orientations: The relative orientations of all matched features are corrected to absolute rotation using the feature orientations stored in the database. Since the tracker is limited to planar targets, all features should have a similar orientation after this point. The main orientation of the target can be estimated with a histogram and all features that do not support the hypothesis can be filtered out.

The second step uses simple geometric tests. All features are sorted by their matching confidence, starting with the most confident features. Then, lines are fit on selected two features, and all the other features are tested to lie on the same side of the line in camera as well as in object space. Up to 30 lines are tested this way. Features that fail any of the line tests are removed.

The third step removes final outliers using homographies in a RANSAC (Random Sample Consensus) fashion. Since a homography[1] must be computed for the initial pose anyway, this step introduces no overhead at all. In our case a projection transform of a plane into another plane has to be determined. If the focal length of the camera is known, the pose from three matched features can be calculated to up to four pairs of solutions (a solution of the generalized 3-point pose problem can be found in [28]). From four point correspondences the homography is unambiguously determined. The features that passed the previous two tests obviously have a correct orientation and coarsely lie at the right spot in the camera image. The main problem with homographies is that features must not be co-linear and their convex hull should cover a large region of the camera image. To find good candidates, the main direction of the point cloud is estimated using perpendicular regression. Then features are selected that lie at the extremal points in both directions of the line as well as furthest perpendicular to the line. For higher robustness, two candidates are selected in all directions, and these eight points are combined into sets of four (one from each direction) and used to calculate the $2^4 = 16$ possible homographies from object into camera space. All homographies are then used to test the projection of the features. The homography with the smallest number of outliers and all respective inliers are finally selected for pose refinement. At the end, Gauss-Newton refinement is applied to minimize the reprojection error of the point correspondences [19].

4.1.5 Patch Tracking

The frame-to-frame correspondences of the video can be used to speed up the calculations, because features that can be tracked reliably on the subsequent images do not need to be matched against the feature database again. At the same time patch

[1] the terms homography, collineation and projective transformation are synonyms for a transformation that maps lines to lines

tracking also improves the overall robustness since features that passed all outlier tests are forwarded with highest confidence values into the next frame, which improves the outlier removal step. To track features, patches of a size of 8×8 pixels are extracted and blurred with a 3×3 Gaussian kernel. The blurring makes them more robust against any kind of affine transformation and slightly incorrect feature coordinates. The sum of absolute difference is used to estimate patch similarity in the neighborhood. An average difference of up to 8% (empirically determined) per pixel is allowed to treat the patch as correctly matched. Features are only tracked in a search radius of 25 pixels [19]. The described method is able to track the affine-warped patches under extreme tilts close to 90 degrees and even under extreme lightning changes and reflections.

A major obstacle for AR tracking on mobile phones comes from the limited processing capabilities. Typical clock rates are between 200 and 600MHz (and reach up to 1GHz in today's high-end devices), on a single core without floating point unit. It is interesting to notice that over the last five years the available computational resources have improved by only about 100%. The reason for this slow increase in computing power is the low increase in battery power. Battery power is the main constraint for mobile phone design and it has improved by only 10% per year.

If the target image (marker or NFT pattern) is previously known, a precalculated database with an optimized search structure can be stored on the mobile device, and thus the tracking can gain significant speedup. But for our changing target a previously optimized database is not applicable, we need to rebuild the feature database on the fly.

A comprehensive overview of recent projects on tracking for handheld augmented reality can be found in [20]. A common approach to overcome the limitations of mobile devices is to outsource the computation-intensive tracking to a server. The target image analysis is a demanding process and therefore can run only on a PC in real-time. Assuming that the target (the visualized scene) does not change rapidly, transferring the processed target image to the mobile and calculate the position on the mobile device is a good idea. However, in case of rapidly changing target images, this setup would scale very badly. As our target is dynamic, either the actual targets, or the camera images need to be transferred on the wireless channel in real time. We chose for the latter approach.

5. Mobile Video Streaming

In an increasing number of applications, video is transmitted to and from portable wireless devices such as phones, laptops or surveillance cameras. The present third-generation, the approaching fourth-generation systems and the widespread IEEE 802.11 wifi standard enable video streaming in personal communications. However, there are two major limitations in any wireless system: the hostile radio environment, including noise, time-varying channels, electromagnetic interference, and the dependence of mobile devices on battery with limited energy supply. Such limitations are especially disadvantageous for video transmission because of the high bit rate and high energy consumption rate in both encoding and transmitting video bitstreams. Computational energy consumption is especially a concern for video transmission, because motion estimation and compensation, forward and inverse DCT transformations, quantization, and other components in a video encoder, all require a significant number of calculations [29].

To get an impression about the achievable delay and quality, we tested a few commercial mobile streaming applications listed in Table 5.1 but none of them was sufficient for our goals because of very high delay or architectural compatibility issues. At the end, we implemented both the H.263- and the JPEG-based streaming using the Android APIs[1] and the hardware encoders of the Nexus One phone. A very short review of the applied standards is therefore presented here.

5.1 The H.263 Video Compression Standard

After testing the mentioned streaming applications we decided to start with H.263 video transfer. The H.263 is a widely-used standard in video conferencing. It was developed by the ITU-T Video Coding Experts Group (VCEG) as an evolutionary improvement based on experience from H.261, a previous ITU-T standard, and the MPEG-1 and MPEG-2 standards. It was further enhanced in projects known as H.263v2 (also known as H.263+ or H.263-1998), MPEG-4 Part 2 and H.263v3 (also known as H.263++ or H.263-2000). MPEG-4 Part 2 is H.263 compatible in the sense

[1] Application Programming Interface

Tab. 5.1: Mobile streaming applications on the market

Product	**Comments**
USBWebCam `http://www.placaware.com`	USB connection between the phone and the PC
VirtualCamera `http://www.soundmorning.com`	streams pictures and movies from files into webcam-based programs
WebcamSimulator `http://www.webcamsimulator.com`	streams pictures and movies from files into webcam-based programs
KnockingLive `http://knockinglive.com`	direct mobile to mobile live video and pics; iPhone or Android; unacceptable delay
LiveCast `http://www.livecast.com`	record and view live streams; too much delay because of centralized architecture
MobiolaWebcam `http://www.mobiola.com`	iPhone, BlackBerry, Windows Mobile or Symbian virtual webcam on the PC; expensive
Movino `http://www.movino.org`	S60 and J2ME clients, open-source, JPEG, MPEG, only for Macintosh users
Qik `http://www.qik.com`	record and share video gallery on a server; too much delay because of centralized architecture
Ustream `http://www.ustream.tv`	interactive broadcast platform not limited to mobile users; clients available for iPhone, Android and S60
Video Remote Computer Camera `http://www.senoctar.com/vrcc`	open source, Symbian, RTP/RTSP protocols, MJPEG compression
SECuRET_LiveStream `http://dooblou.blogspot.com`	Apache webserver on the phone and streams the JPEG-encoded preview images to connected clients over the HTTP protocol.

that a basic H.263 bitstream is correctly decoded by an MPEG-4 Video decoder. The unified H.263 specification document from 2005 and still in force can be found in the ITU-T Recommendation [30]. H.263 video can be legally encoded and decoded with the free LGPL-licensed *libavcodec* library (part of the FFmpeg project) which is used by applications such as *ffdshow*, *VLC Media Player* and *MPlayer*.

Android hides the video compression details from the developers behind the built-in *MediaRecorder API*. The recorder deals with all the camera access and encoding, and writes the encoded video into a 3GP container file. We stream the unaltered video from the mobile phone to the PC, where the mentioned open-source libraries are available for decoding. The description of the encoding and decoding algorithms is out of the scope of this thesis. Curious readers are referred to [31]. The only thing we need to know about H.263 encoding in this project is that there are specific data blocks inside the encoded video stream and their borders are marked by so called start codes. There are different types of start codes (Picture, GOB, Slice, EOS, and EOSBS), however, all of them begin with 16 zero-valued bits and thus they can be easily recognized in a byte stream.

5.2 The 3GP Container Format

The 3GP multimedia container format is defined by the Third Generation Partnership Project (3GPP) for third generation multimedia services. The 3GP file format follows the ISO Base Media file format, in which each file consists of Boxes[2]. In general, a 3GP file contains the File Type Box *(ftyp)*, the Movie Box *(moov)*, and the Media Data Box *(mdat)*. The *ftyp* describes the type and properties of the 3GP file itself and contains a pointer to the metadata inside the file. The *moov* and the *mdat*, serving as containers, include a number of children boxes for each media. All boxes start with a header which indicates both box type and size. In the following, only those boxes are mentioned that are useful for the purposes of this project. An overview is depicted in Fig. 5.1.

The Movie Box *(moov)* contains one or more Track Boxes *(trak)*, which include information about each track. A Track Box contains – among others – the Track Header Box *(tkhd)*, the Media Header Box *(mdhd)*, and the Media Information Box *(minf)*.

The Track Header Box specifies the characteristics of a single track, where a track can be video, audio or other data. Exactly one Track Header Box is present for a track. It contains information about the track, such as the spatial layout (width

[2] also called Atoms

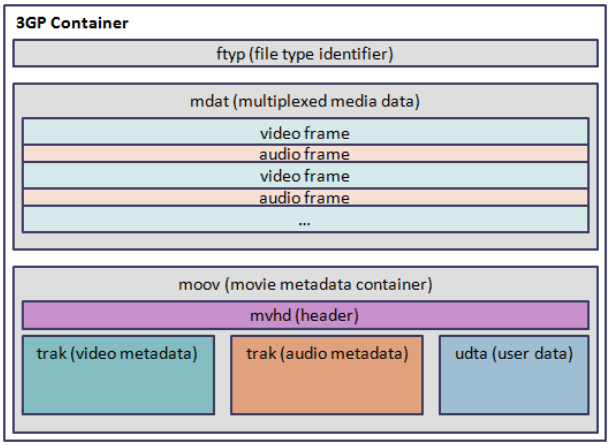


Fig. 5.1: Structure of a 3GP container

and height), the video transformation matrix, and the layer number. Since these pieces of information are essential and static (i.e., constant) for the duration of the session, in a streaming application they must be sent prior to the transmission of the media content. In a stored clip the *moov* box can also be at the end of the file.

The Media Header Box contains the "timescale" or number of time units that pass in one second (cycles per second or Hertz). The Media Information Box includes the Sample Table Box *(stbl)*, which contains all the time and data indexing of the media samples in a track. Using this box, it is possible to locate samples in time and to determine their type, size, container, and offset into that container. Inside the Sample Table Box, we can find the Sample Description Box (*stsd*, for finding sample descriptions), the Decoding Time to Sample Box (*stts*, for finding sample duration), the Sample Size Box *(stsz)*, and the Sample to Chunk Box (*stsc*, for finding the sample description index).

Finally, the Media Data Box *(mdat)* contains the media data itself. Video and audio are interleaved. 3GP files can contain MPEG-4 Part2, H.263 or H.264-coded video streams and several audio formats. The full specification can be found under [32].

5.3 The Real-Time Transport Protocol with H.263 Video Payload

The Real-time Transport Protocol (RTP) is a standardized packet format over IP networks defined in RFC 1889 and RFC 3550 [34]. RFC 3550 mostly is identical to RFC 1889 which it obsoletes. There are no changes in the packet formats on the wire, only changes to the rules and algorithms governing how the protocol is

used. RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services.

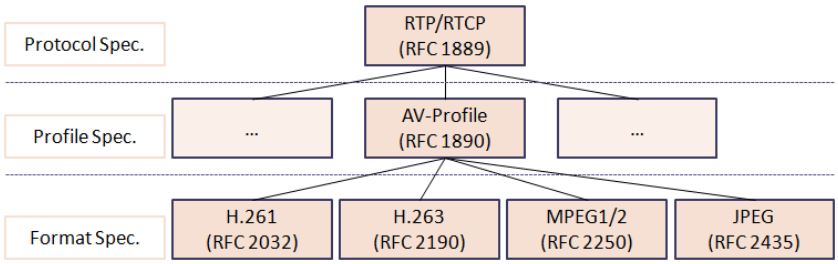


Fig. 5.2: Family of RTP payload formats

It extends UDP practically only in a sense that it adds Sequence Numbers and Time Stamps. The timestamp is used to place the incoming audio and video packets in the correct timing order (playout delay compensation). The sequence number is mainly used to detect losses and change of order. Sequence numbers increase by one for each RTP packet transmitted, timestamps increase by the time covered by a packet. It is expressly possible to define new applications. This leads to different profile specifications as depicted in Fig. 5.2. The so called Payload Format Specifications define how the different payload types must be packed for transmission over RTP. The transport of H.263 video streams over RTP is also standardized by the IETF[3] in RFC 2190 (original H.263), RFC 2429 (H.263-1998) and RFC 4629 (H.263-2000) [36]. When transmitting H.263 video streams over the Internet, the output of the encoder can be packetized directly. All the bits resulting from the bitstream including the fixed length codes and variable length codes will be included in the packet, with the only exception being that when the payload of a packet begins with a start code, the first two (all-zero) bytes of the start code are removed and replaced by setting an indicator bit in the payload header. Fig. 5.3 depicts the structure of an RTP packet header.

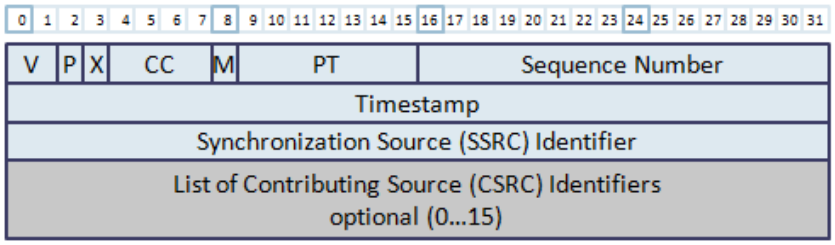


Fig. 5.3: RTP header structure

[3] Internet Engineering Task Force

- Version (V): 2 bits
  This field identifies the version of RTP. The version defined by RFC 1889 and RFC 3550 is two (2).

- Padding (P): 1 bit
  If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload. The last octet of the padding contains a count of how many padding octets should be ignored, including itself. Padding may be needed by some encryption algorithms with fixed block sizes or for carrying several RTP packets in a lower-layer protocol data unit.

- Extension (X): 1 bit
  If the extension bit is set, the fixed header must be followed by exactly one header extension. Since this is not used in our application, the X-bit is always set to 0.

- Marker (M): 1 bit
  The interpretation of the marker is defined by a profile. According to RFC 4629, the marker bit of the RTP header is set to 1 when the current packet carries the end of current frame and is 0 otherwise.

- Payload Type (PT): 7 bits
  This field identifies the format of the RTP payload and determines its interpretation by the application. A set of default mappings for audio and video is specified in RFC 3551 [35]. We can use the RTP/AVP profile with H263-1998 payload, or a so called dynamic payload format. The range 96–127 is left as dynamic payload type and applications can use it as they want. We tested our streaming application with the VideoLAN Client (VLC) which uses dynamic payload type 96 when streaming and defines the content during the RTSP session establishment in an *rtpmap* field.

- Sequence Number: 16 bits
  The sequence number increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence. The initial value of the sequence number should be random (unpredictable) to make known-plaintext attacks on encryption more difficult.

- Timestamp: 32 bits
  The timestamp encodes the sampling instance of the first video frame data

contained in the RTP data packet. The RTP timestamp shall be the same on successive packets if a video frame occupies more than one packet. If temporal scalability is used (if B-frames[4] are present), the timestamp may not be monotonically increasing in the RTP stream. For an H.263 video stream, the RTP timestamp is based on a 90 kHz clock. Since both the H.263+ data and the RTP header contain time information, it is required that those timing information run synchronously. That is, both the RTP timestamp and the temporal reference (TR in the picture header of H.263) should carry the same relative timing information. Any H.263 picture clock frequency can be expressed as $1800000/(cd * cf)$ source pictures per second, in which cd is an integer from 1 to 127 and cf is either 1000 or 1001. Using the 90 kHz clock of the RTP timestamp, the time increment between each coded H.263 picture should therefore be a integer multiple of $(cd * cf)/20$. This will always be an integer for any reasonable picture clock frequency (for example, it is 3003 for 29.97 Hz NTSC, 3600 for 25 Hz PAL, 3750 for 24 Hz film, and 1500, 1250 and 1200 for the computer display update rates of 60, 72 and 75 Hz, respectively) [36]. Several consecutive RTP packets will have equal timestamps if they are (logically) generated at once, e.g., belong to the same video frame. Consecutive RTP packets may contain timestamps that are not monotonic if the data is not transmitted in the order it was sampled, as in the case of interpolated video frames (but the sequence numbers of the packets as transmitted will still be monotonic.)

- SSRC: 32 bits
  Synchronization Source Identifier. This identifier should be chosen randomly, with the intent that no two synchronization sources within the same RTP session will have the same SSRC identifier.

- CSRC Count (CC): 4 bits
  The CSRC count contains the number of CSRC identifiers that follow the fixed header. CC is set to zero as CSRCs are not used in our application.

- CSRC: (0..15) × 32 bits
  List of Contributing Source Identifiers. There are no contributing sources in our application. Thus, the whole RTP header takes 12 bytes in our case.

[4] In advanced video coding standards there are three types of frames depending the prediction. I-frames (intra-coded) are not predicted, P-frames are predicted from previous frames and B-frames (bidirectional) are computed from both previous and subsequent frames. In some standards also I,P,B-slices or I,P,B-blocks are distinguished.

A section of an H.263-compressed bitstream is carried as a payload within each RTP packet. In each packet, the RTP header is followed by an H.263 payload header, which is followed by a number of bytes of a standard H.263 compressed bitstream. The size of the H.263 payload header is variable depending on the payload involved. The layout of the RTP H.263 video packet is shown in Fig. 5.4.

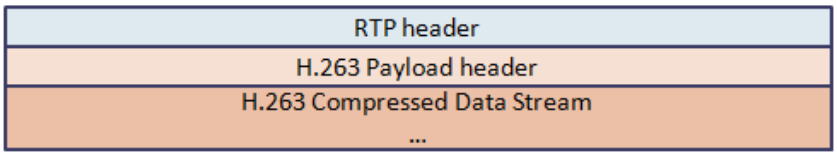


Fig. 5.4: RTP packet containing H.263 data

All H.263 start codes begin with 16 zero-valued bits. If a start code is byte aligned and it occurs at the beginning of a packet, these two bytes shall be removed from the H.263 compressed data stream in the packetization process and shall instead be represented by setting a bit (the P bit) in the payload header. The internal H.263 payload structure is not important for our application, because we do not need to deal with the video decompression. The decompression is automatically done by the receiving player application. Our task is to seamlessly transport the H.263 data stream over the wireless network.

RTP is usually used in conjunction with the RTP Control Protocol (RTCP) which is also defined in RFC 3550. While RTP carries the media streams (e.g., audio and video), RTCP is used to monitor transmission statistics and quality of service (QoS) and aids synchronization of multiple streams. In the research prototype we do not deal with RTCP messages, but a packetizer needs to be implemented on the phone that fills the output of the encoder into RTP packets according to RFC 4629 [36].

5.4 The Real-Time Streaming Protocol

The Real Time Streaming Protocol[5] (RTSP) is a client-server application-level protocol for controlling the delivery of data with real-time properties. It establishes and controls either a single or several time-synchronized streams of continuous media, such as audio and video. In other words, RTSP acts as a network remote control for multimedia servers, with a syntax very similar to the HTTP protocol (simple text transfer). Both the client and the server have an internal state machine. Fig. 5.5

[5] Various RTSP implementations can be found under `http://www.cs.columbia.edu/~hgs/rtsp/implementations.html`

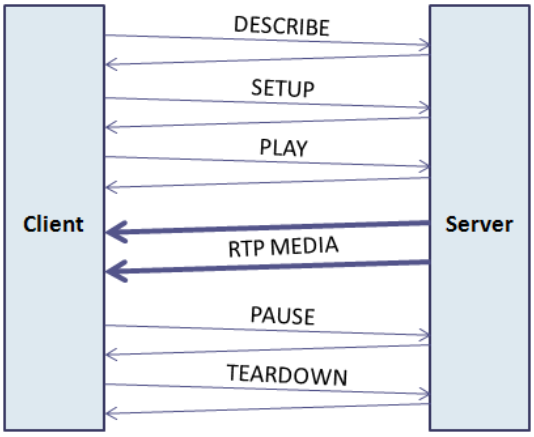


Fig. 5.5: RTSP communication between the Client and the Server

depicts a simple conversation between a client and a server. Each conversation consists of ASCII-coded REQUESTs and RESPONSEs. For example, a SETUP request and response look like this:

```
C->S:
SETUP rtsp://example.com/foo/bar/baz.rm RTSP/1.0
CSeq: 302
Transport: RTP/AVP;unicast;client_port=4588-4589

S->C:
RTSP/1.0 200 OK
CSeq: 302
Date: 23 Jan 1997 15:35:06 GMT
Session: 47112344
Transport: RTP/AVP;unicast;client_port=4588-4589;server_port=6256-6257
```

Here, the client requests a media stream from the address `rtsp://example.com/foo/bar/baz.rm` through RTP transfer and waits for it on port 4588 (and RTCP 4589). The sequence number (CSeq) is incremented after each request-response pair. The session ID uniquely identifies the actual session. The server tells the client its own chosen ports 6256-6257 for this RTP/RTCP communication. RTSP initiates data transfer over transport protocols such as UDP, multicast UDP, TCP, and RTP. RTSP resources are identified by a unique URL starting by *rtsp://* for TCP/IP or *rtspu://* for UDP/IP connection type. Sources of data can include both live data feeds and stored clips [33]. In our project, the RTSP protocol will be used to control our video streaming server on the phone.

6. Implementation and Component Tests

The research prototype of the Augmented Visualization System has been built at the Institute of Computer Graphics and Algorithms. Fig. 6.1 reviews the three main parts and their operation steps. This chapter presents the implementation in detail in the order of the modules they were added to the system. Each component description is followed by a short test of the functionality.

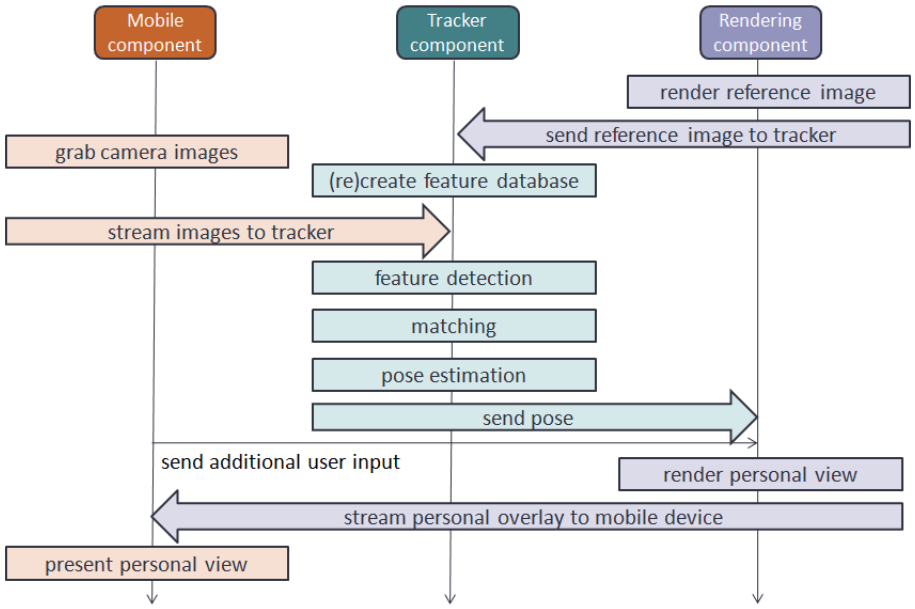


Fig. 6.1: The three main components and their operation steps in the Augmented Visualization System

6.1 The Tracker Component

The Tracker component is the Studierstube ES augmented reality framework running our custom application. For a first prototype, a webcam was used as video input. Later, when the wireless streaming between the mobile component and the PC was working, the Studierstube ES developers in Graz extended the framework by a network video input using the GStreamer[1] video streaming library. The

[1] GStreamer, `http://www.gstreamer.net`

GStreamer uses a filter chain to decode network streams and inject the video frames into the Tracker. The reference images arrive directly from the Rendering component through TCP/IP connection.

6.1.1 Reference Image Streaming

To keep the Rendering component only loosely connected to the other components, the Tracker component receives the target images through a TCP/IP socket and sends back various input events in the same way. This allows to easily connect other renderer components in the future. The viewports in VolumeShop are equipped by video recording output, and this includes the feature of streaming the images to a network socket. At this point only small modifications of the software were needed. We adjusted the output resolution of the image stream because the tracker is limited to target images with width and height each being a multiple of 8 pixels[2]. The own packet format consists of a short header containing the width, height and byte-length of the image followed by the JPEG-compressed frame.

6.1.2 The Studierstube ES Application

The Studierstube ES framework is not published and therefore no source code of the tracking application is available. However, the pseudocode of the main thread and the reference receiver thread are listed in the Appendix. The configuration file for Studierstube ES is also listed as an example how the framework can be paramaterized for simple applications. The theoretical background is described in section 4.1. In the augmented visualization application, the Studierstube Natural Feature Tracker v2 component is responsible for all the tracking tasks. The tracker consist of an internal target detector and an internal patch tracker. As the detection step is the much more computationally demanding one, it is done once, then the tracker switches to the patch tracking mode for increased performance. For the first frame, or if the dataset has changed (a new picture arrives from the Rendering component), it repeats the detection step using robust feature matching. For all the following frames, it starts the patch tracker for tracking individual small patches, unless the track is lost or the dataset is changed again.

[2] This 8 is a fundamental constraint in Studierstube ES because the algorithms are highly optimized for devices with limited processing capabilities

6.1.3 Camera Calibration

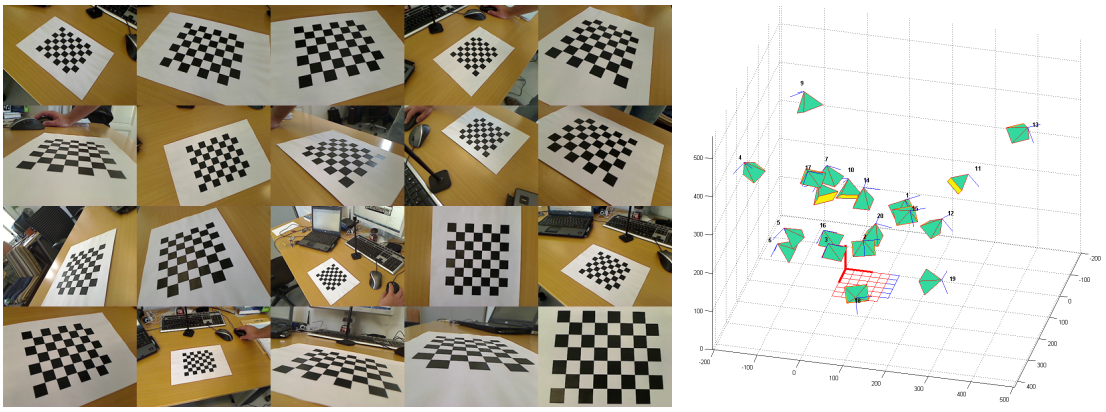
At the first development stages we used commercial webcams, a Logitech QuickCam and a more advanced Logitech C905 to feed video into Studierstube ES. First, a calibration procedure is needed to calculate the radial and tangential distortion parameters in order to undistort the incoming camera image to achieve better target detection. A semi-automated MATLAB toolbox [45] was used for this purpose. Calibration is done by using a set of planar checkerboard images (Fig. 6.2(a)), photographed from different viewpoints (Fig. 6.2(b)). The position and the orientation of the camera are also called the extrinsic parameters of a specific image capture. Corner extraction and non-linear optimization are performed by the toolbox to match the camera image to the calibrated target. The optimization results can be used to determine the focal length, principal point and distortion parameters, the so called intrinsic parameters of the camera. As an example, the intrinsic parameters of the C905 camera are listed here:

```
StbTracker_CamCal_v1
size: 800 600
principle point: 399.27342 300.87471
focal length: 657.53025 659.06986
radial distortion: 0.03456 -0.09578 0.00005 0.00055 0.0
iterations: 10
```

Both the calibration toolbox and the tracker apply the pinhole camera model (see Appendix). The focal length is the distance of the image plane from the projection center, the principal point is the position of the image where the optical axis of the camera intersects it. All distances are measured in pixels. The radial distortion parameters are also taken into account during tracking (however, the distortion was minimal in case of our cameras). Later, the camera of the Nexus One mobile phone has been calibrated as well, using the same procedure.

6.1.4 Tracking Test

In Fig. 6.3 test pictures of my application with the Studierstube Tracker are shown. At this test phase, a calibrated webcam was used. The red cube indicates that the tracker is in focus and the pose estimation is correct. The cube is always drawn on the estimated target plane. Once the target was found using feature matching, the PatchTracker takes over and estimates the camera motion on the fly. When the target is lost or the target changes, a new matching step reinitializes and the tracking recovers in less than one second. Our observation was that a reference



(a) Photos used for camera calibration

(b) World-centered view of camera calibration

Fig. 6.2: Camera calibration images

image contains about thousands of and a captured image about hundreds of interest points.

6.2 The Mobile Component

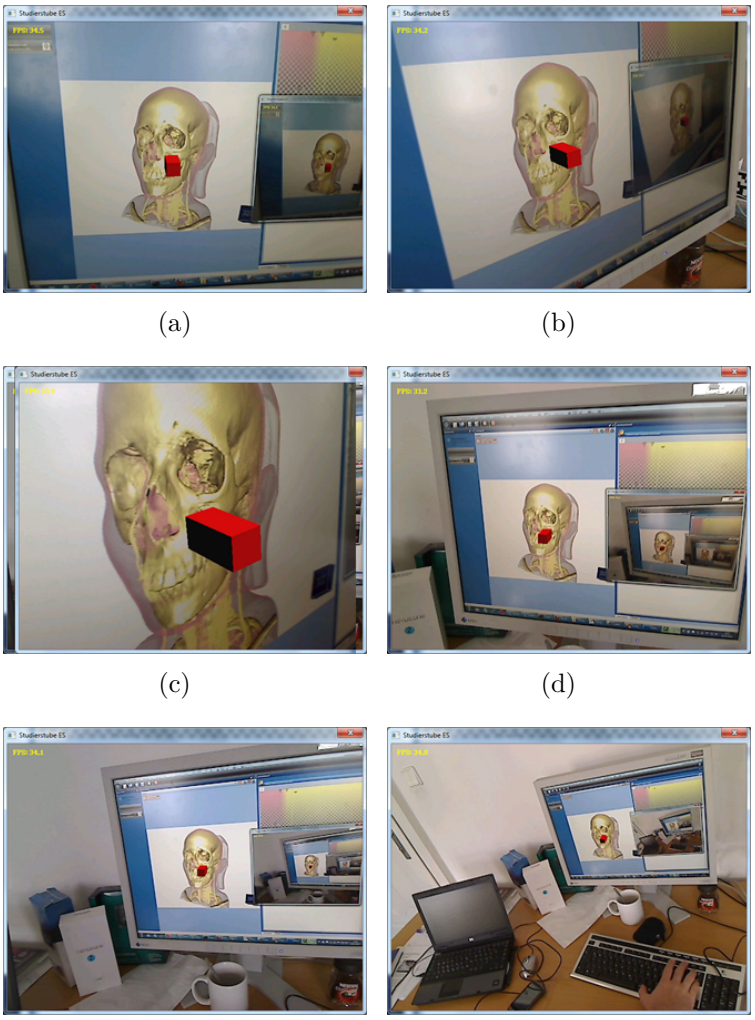
The mobile client of the AVS was developed on an Android phone for this thesis.

6.2.1 Android Development

The Eclipse integrated development environment with the Android SDK was used to create the client application in the Java language. The DDMS (Dalvik Debug Monitor Server) perspective in Eclipse provides comprehensive access to device functionality, such as a console, command line tools, file explorer, debugger, etc. For an extensive material on Android development please refer to [40], [41], [42], [43], [44].

The Android platform strongly follows the modularity paradigm, in which every application consists of modules that can even be reused by other applications at runtime. The four main component types are Activities, Services, Broadcast Receivers and Content Providers.

Activities are the main components of every application. These serve as graphical user interface for the user. All layouts and resources are defined in XML files in a very intuitive way and the activities are responsible for the GUI management. In Android, there is always one foreground application, which typically takes over the whole display except for the status bar. When the user runs an application, Android starts it and brings it to the foreground. From that application, the user



(a) (b)

(c) (d)

(e) (f)

Fig. 6.3: Testing of my application with the Studierstube NFT: The red cube augments the real visualization scene; and indicates that the tracking is working.

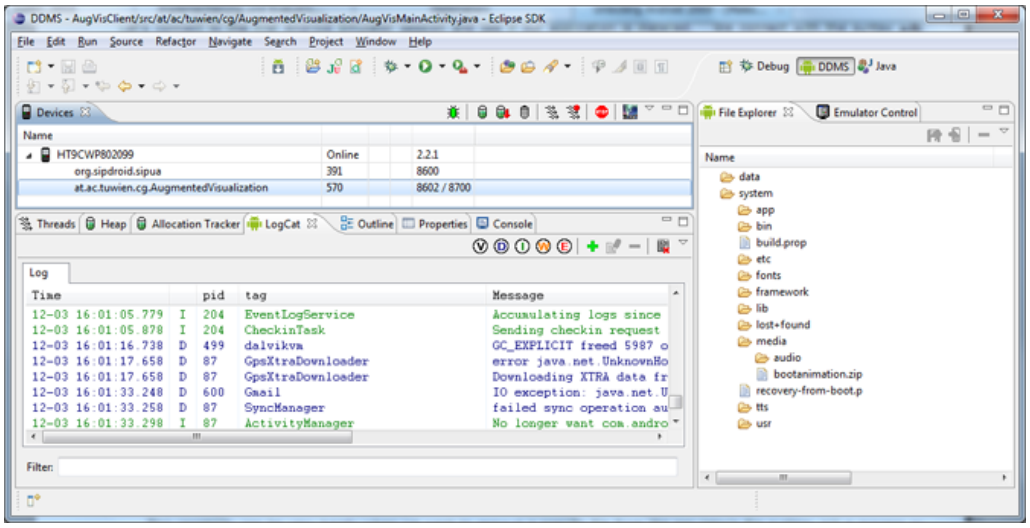


Fig. 6.4: DDMS Perspective in Eclipse for Android development

might invoke another application, or another screen in the same application (in both cases another Activity). All Activities are recorded on the application stack by the system's Activity Manager and the user can navigate back and forth between them. Our application *AugVisClient* has four activities: the *AugVisStartActivity* as the start screen and wireless setup, the *AugVisSettingsActivity* for settings management, the *AugVisHelpActivity* that gives hints on usage and the *AugVisMainActivity* that actually encapsulates all the client functionality for the AVS.

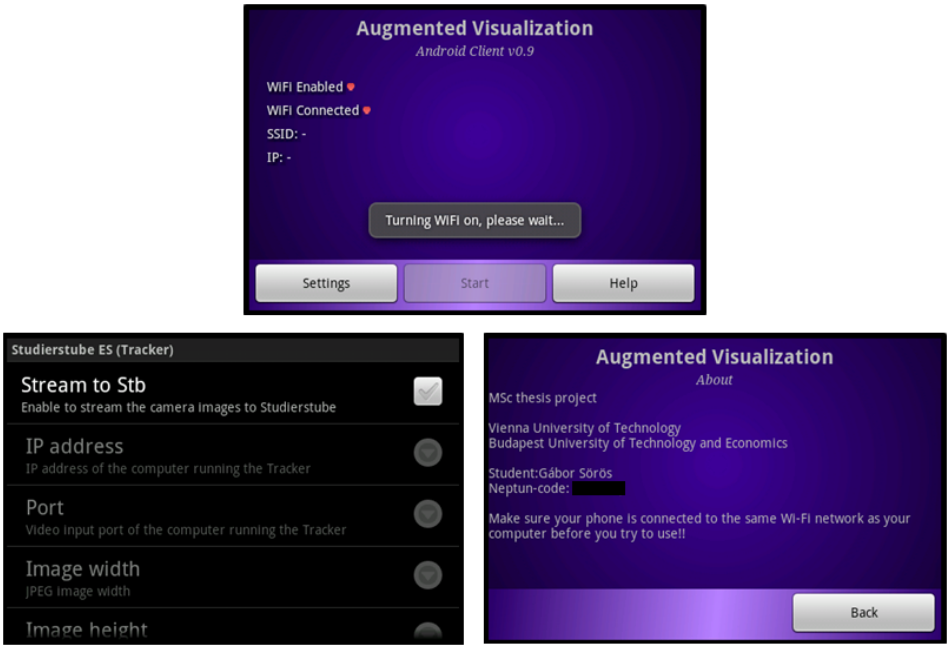


Fig. 6.5: GUI of three of the four AugVisClient Activities

In Android, the Activities have their own life cycle (an internal state machine, see Fig. 6.6) which is independent from the process life cycle they are running in. During its lifetime, each activity of an Android program can be in one of these states. The states are maintained by the operating system and applications are notified at state change through the *on...()* method calls [42]. I describe the functions with respect to my *AugVisMainActivity*.

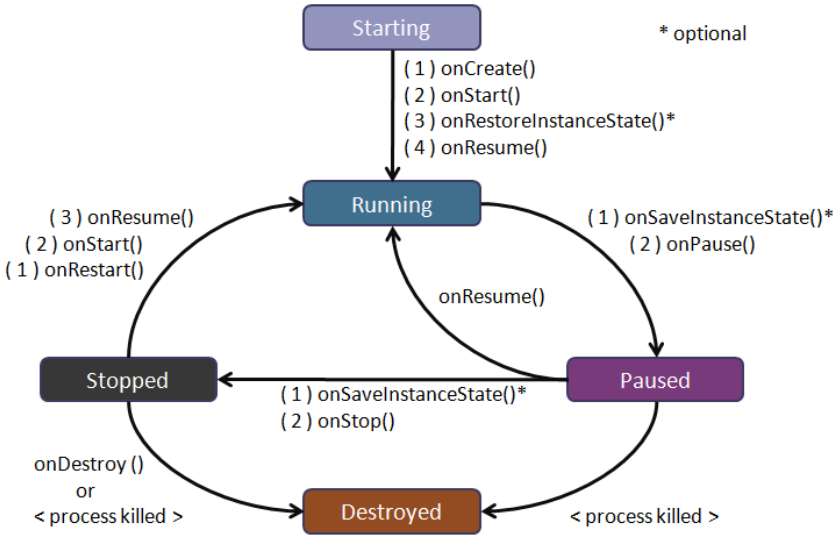


Fig. 6.6: Activity life cycle in Android (figure after [42])

- *onCreate(Bundle):* is called when the activity first starts up. It is usually a good point to perform one-time initialization tasks such as creating the user interface. *onCreate()* takes one parameter that is either *null* or a so called *bundle* containing state information previously saved by the method *onSaveInstanceState()*.

- *onStart():* indicates the activity is about to be displayed to the user. We use this to register the broadcast receiver.

- *onResume():* is called when the activity can start interacting with the user. Time to initialize the camera, spawn networking threads and connect to other AVS components.

- *onPause():* indicates the end of the foreground life time, usually because an other activity has been launched in front of it. A point to save all unsaved data and stop our networking threads.

- *onStop():* is called when the activity is no longer visible to the user and it won't be needed for a while. If memory is tight, the system may simply terminate the process.
- *onRestart():* is called, when the activity is being redisplayed to the user after it was stopped.
- *onDestroy():* is called right before the activity is destroyed. This is the point to release resources.
- *onSaveInstanceState(Bundle):* Android will call this method to allow the activity to save per-instance state, such as a cursor position within a textfield. Usually this does not need to be overridden because the default function is sufficient.
- *onRestoreInstanceState(Bundle):* This is called when the activity is being reinitialized from a state previously saved by the *onSaveInstanceState()* method. The default implementation restores the state of the user interface.

Services do not have a graphical user interface, but rather run in the background for an indefinite period of time. For example, a service might play background music as the user attends to other matters, or it might fetch data over the network or calculate something and provide the result to activities that need it. Services are a powerful tool to run processes from an Activity that live even after the Activity process has ended. Our application does not need any Services.

Broadcast Receivers receive and react to broadcast announcements. Many broadcasts originate in system code, for example, announcements that the time zone has changed, that the battery is low, or that a network connection state changed. Applications can also initiate broadcasts, for example, to let other applications know that some data has been downloaded to the device and is available for them to use. Our application contains a Broadcast Receiver that listens to network state changes and initiates an alarm intent for other components in case of network failure.

Content Providers make a specific set of the application's data available to other applications. The data can be stored in the file system, in an SQLite database, or any other self-defined form. Our application has no Content Providers.

Intents are used to activate Activities, Services, and Broadcast Receivers. They are also used in special cases for asynchronous communication between different components. Intents could be considered as the fifth type of building blocks in an Android application. An intent also holds the content of the message. For activities and services, it names the action being requested and specifies the URI of the data to act on, among other things. For example, it might convey a request for an activity to present an image to the user or let the user edit some text.

The Manifest File is included in each Android application. Applications declare their components in the manifest file that's bundled into the Android package (the *.apk* file), that also holds the application's code, files, and resources. The framework parses the manifest file and gets informed about the anatomy of the respective application. As mentioned earlier, every application runs in its own Linux process. The hardware forbids one process from accessing another process's memory. Furthermore, every application is assigned a specific user ID. In addition, access to certain critical operations is restricted, and the programmer must specifically ask for permission to use them in the manifest file. When the application is installed, the Package Manager either grants or prohibits the permissions based on certificates, and if necessary, user prompts [42]. For our application we needed the following framework features and hence corresponding permissions:

- CAMERA: camera access
- WRITE_EXTERNAL_STORAGE: for testing
- INTERNET: access to the network
- ACCESS_NETWORK_STATE: query connection state
- CHANGE_NETWORK_STATE: force to connect
- ACCESS_WIFI_STATE: query WiFi sate
- CHANGE_WIFI_STATE: force to connect
- WAKE_LOCK: prevent from WiFi sleep and display dimming

6.2.2 H.263 Streaming

We have investigated several available projects how to implement the RTSP/RTP and H.263-based video streaming. Different related projects are listed in Table 6.1. We first experimented with an *ffmpeg* port to Android while binding it to our application through the Native Development Kit, but our compilation efforts failed for the ARM hardware platform. Then we decided to use the built-in Android *MediaRecorder API* which accesses the hardware encoders, however, hides them

Tab. 6.1: Open-source RTSP/RTP streaming libraries on the market

Project	**Comments**
ffmpeg `http://www.ffmpeg.org`	cross-platform solution to record, convert and stream audio and video; contains *libavcodec* as subproject
liveMedia `http://www.live555.com`	Source-code libraries for standards-based RTP/RTCP/RTSP/SIP multimedia streaming, suitable for embedded and/or low-cost streaming applications.
VideoLAN `http://www.videolan.org`	open-source, cross-platform multimedia player and framework; large number of audio and video codecs available; also uses the *liveMedia* library for streaming and *libavcodec* for decoding
Xuggler `http://www.xuggle.com/xuggler`	uncompress, modify, and re-compress any media file (or stream) from Java
FENG `http://lscube.org/feng`	multimedia streaming server also based on *ffmpeg*
GStreamer `http://www.gstreamer.net`	a pipeline-based multimedia framework written in C with the type system based on *GObject*
GStreamer-RTSP `http://www.gstreamer.net`	*GStreamer*-based RTSP server

from the developer. The MediaRecorder is intended to be used for audio & video recording into files. It seamlessly compresses media[3] and stores it on the SD-card of the phone. However, Android is based on Linux, where a network socket – as well as other devices – are equivalent to files and can be accessed through a *FileDescriptor* interface. It is a straightforward idea to give the socket's FileDescriptor to the MediaRecorder as destination. I created a simple TCP connection between the phone and the PC, so that the deflected stream is written into a file on the PC.

Unfortunately, the output of the recorder is an incomplete 3GP container file (described in section 5.2) which wraps around the pure video data. As the video metadata (e.g., the length) is unknown at the beginning, the recorder first leaves the *ftyp* and *moov* boxes empty, and fills them after the recording ended. There is no problem when writing the output into a file on the SD-card, but since sockets are not seekable, these meta blocks can never be filled with correct information. Instead, our files at the PC contain zeroes at the beginning and both the *ftyp* and the *moov* boxes at the very end. This leads of course to files that cannot be decoded by a client. Moreover, as storing a clip into a file is assumed here, the *moov* box comes by definition at the end, after the actual data, and thus the output of the recorder can not be streamed directly (*moov* should come first to initialize a decoder). A correct and one of our corrupted 3GP files are show in Fig. 6.7. A powerful hex editor[4] and a container file analyzer[5] helped us to reveal the differences. The 'mdat' character sequence signs the beginning of the *mdat* box. The last four bytes of the *ftyp* box should tell the exact length of the following data, which is also false in our case.

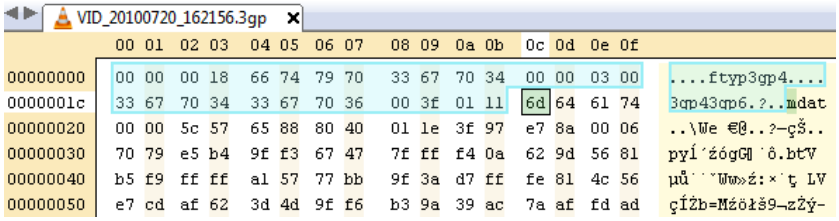
Fortunately, as the H.263 standard is developed for streaming, it contains all the necessary information for decoding. The corrupted 3GP container needs to be peeled off. We know the size of the fixed header the recorder writes (28 bytes *ftyp* box + 4 bytes 'mdat' label) and after that, it just starts writing pure H.263 data. So what we need on the PC side is to cut off the first bytes and then parse the H.263 data ourselves or feed it to a decoder like *ffmpeg* to play/decode the video. However, we found that although the resulting H.263 file can be played by VLC Player, the content of the video looked destroyed. However, in the background one could suspect the correct video. By comparing[6] the received file on the PC and the same content streamed into a local socket looped back on the phone's localhost and written onto the SD-card, we found the origin of the problem. The two files differed

[3] Android-supported video encoders: MPEG4-SP, H.263, H.264, supported audio encoders: AMR-NB, supported container formats: MP4, 3GP

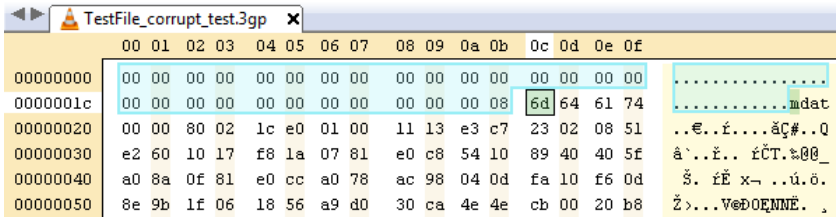
[4] HHD Free Hex Editor Neo, `http://www.hhdsoftware.com/free-hex-editor`

[5] Yamb - Yet Another MP4Box User Interface, `http://yamb.unite-video.com`

[6] WinMerge, `http://www.winmerge.org`



(a)



(b)

Fig. 6.7: The beginning of a normal and our corrupted 3GP file

only in *0x0D 0x0A* vs. *0x0A* bytes. The reason is that different operating systems use different markers to indicate the end of a line. Under Android (a variant of Linux), the end of a line is indicated by a line feed character (\LF, *0x0A*). Under Windows, lines end with a combination of two characters: carriage return (\CR, *0x0D*) followed by a line feed (together abbreviated as \CRLF). The only problem with the corrupted file was that on the PC we opened it in character mode, and every received \LF character was automatically replaced by a \CRLF, which lead to the distorted video. The problem was solved by changing to byte-mode transfer. I consider this problem to be worth mentioning, due to my tremendous efforts (three weeks of reverse engineering and debugging) for finding the solution.

At this point we were able to transfer pure H.263 video streams to the PC over TCP. The available video players usually support a network stream input of standardized RTP packets. The use of RTP in between is desirable, so that we can avoid the manual loading and initialization of the video decoders. These steps are all done automatically in the player softwares. Therefore, the next step is to parcel the pure video stream into payloads of distinct RTP packets.

To avoid the need of any relaying application on the PC, the RTP packets must be constructed already on the phone. This is solved by a trick of looping back the socket on localhost and receiving the MediaRecorder output in a secondary thread of our application (the very same way as I found the solution of the \CRLF-problem). In that second thread RTP packets are assembled from the received stream according to the standard. To control the data flow, an RTSP server is also integrated. The packets are either sent to the connected RTSP clients or deleted depending on the

state of an internal softswitch. The switch is set by the RTSP server thread.

6.2.3 Multi-Threading in Android

Android provides a few possibilities *(Threads, Services)* to handle long-lasting operations in background. If an application does not respond to user input in 5 seconds, the operating system terminates it immediately throwing an ANR (Application Not Responding) error. The difference between a *Thread* and a *Service* is that while the former is always bound to the actual application, the latter overlives its parent and other applications can also use it. For our needs, Threads give enough functionality. In Android, the communication between the spawned threads and the main application is done through so called *Handlers*. A Handler is actually a pointer to the application's own *MessageQueue* and can be given to the Threads as parameter. This way the Threads can enqueue *Messages* or *Runnables* back into the MessageQueue. A Message transfers data between the components, while a Runnable contains runnable code. We use both approaches in the AugVisClient.

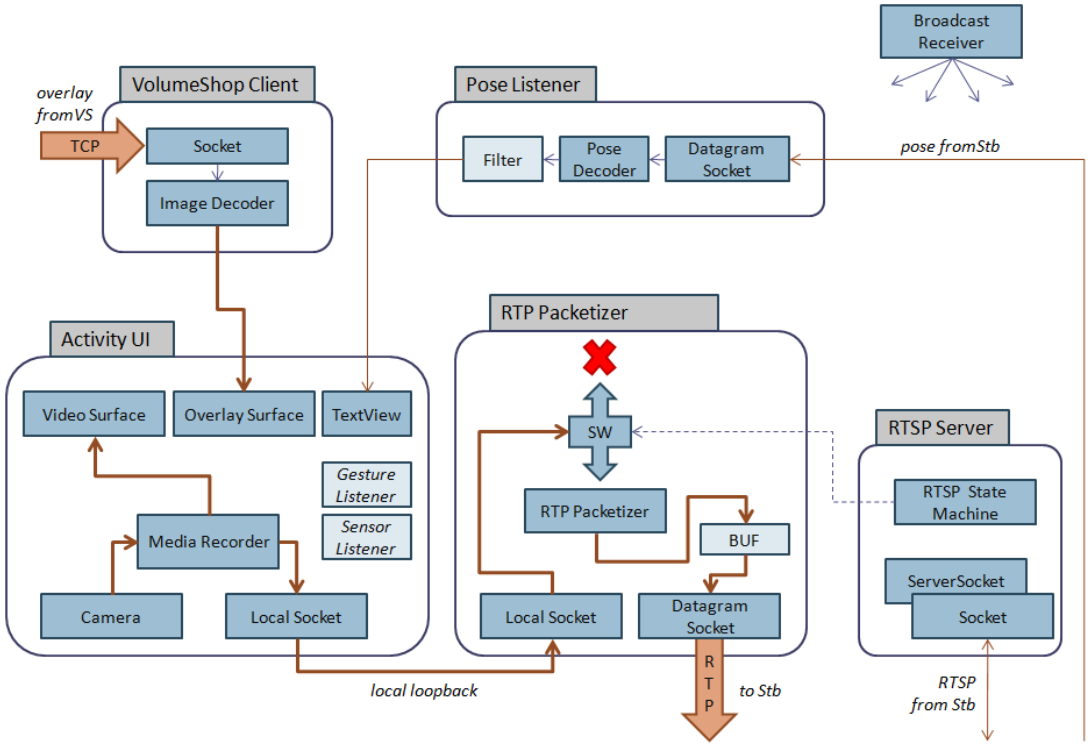


Fig. 6.8: Architecture of the AugVisClient application

The *AugVisMainActivity* will perform extensive network communication in many directions. The networking functions in Java contain blocking statements (e.g. receive timeouts, etc.) and therefore the application must not call them from the UI

thread. I have defined the *AugVisPacketizerThread*, the *AugVisRTSPServerThread*, the *AugVisStbPoseThread*, and the *AugVisVSClientThread* modules as depicted in Fig. 6.8. The images originating from the camera hardware are processed by the MediaRecorder and written into the loopback socket. At the same time, the camera preview is displayed to the user on a GUI surface. The compressed video stream is then read by the RTP packetizer which constructs standard RTP packets with the H.263 video payload. Depending on the state of the internal switch the packets are either deleted or sent to a peer. The remote peer can control the switch using the RTSP protocol. The pose from the Tracker is received within a separate thread and after filtering it is shown to the user in a text box. The overlay images from the Rendering component arrive into a decoder and then they are presented in the main Activity. The user can choose if he/she wants to see the camera preview or the overlays. Note that by extending the Activity with a *GLSurfaceView* (a built-in Android tool to present OpenGL renderings) on top of the camera preview we could achieve traditional augmented reality.

6.2.4 The RTP Packetizer

As mentioned, the packetizer is constantly reading from the loopback socket and is searching for start codes in the H.263 stream. The video data is continuously filled into RTP packets, starting a new packet and setting the Marker bit at every start code. We also applied code pieces (the RTP packet implementation and the standardized packet construction) from an open-source video telephone application named Sipdroid[7].

6.2.5 The RTSP Server

I have implemented a minimal RTSP server and client applet pair in Java and then ported the server onto Android. The OPTIONS, DESCRIBE, SETUP, PLAY, PAUSE, TEARDOWN Requests are handled and thus communication with the VLC Player is possible. It is minimal implementation, because it does not contain any sophisticated message parser and at the moment it is compatible only with VLC. A full RTSP conversation between two VLC clients captured by the Wireshark[8] network analyzer was a guiding instruction during the assembly of my server. The example conversation is listed in the Appendix. The main difference to that is the

[7] `Sipdroid,www.sipdroid.org`
[8] Wireshark, `http://www.wireshark.org`

DESCRIBE Response from our server that contains the description of our video stream in an SDP (Session Description Protocol [37]) format:

```
RTSP/1.0 200 OK;
Server: Test Server
Content-Type: application/sdp
Content-Base: rtsp://<phone IP address>:<phone RTSP port>/
Content-Length: 191
Cache-Control: no-cache
CSeq: <sequence number>

v=0
o=android 000000 000000 IN IP4 <phone IP address>
s=AugmentedVisualization
c=IN IP4 0.0.0.0
t=0 0
a=control:rtsp://<phone IP address>:<phone RTSP port>/
m=video 0 RTP/AVP 96
a=rtpmap:96 H263-1998/90000
```

The *Content-Length* field tells the client the length of the content in bytes (miscalculating the length easily leads to application crashes of the VLC client). The *v* is the version code of the Session Description Protocol, *o* is the session owner field. The owner is named "android", it is reachable through the Internet on an IPv4 address <phone IP address>. The session name *s* is "AugmentedVisualization". The *c* field tells that the destination address can be any IPv4 address. The *t* field's zero values mean that the server is always available. The first *a* field contains the control address for all following mediums. We have only one medium *m*, an RTP/AVP 96 (audio-video profile with payload type 96). As 96 means a not standardized dynamic type, a second *a* field must define in an *rtpmap* that the packets will contain H263-1998-compressed video with clock rate 90000Hz. In the *m* field, the second parameter is a port suggestion for client, and 0 means the server has no preference (i.e., client can choose and use it in its SETUP Request).

The VLC player can successfully connect to the RTSP server and stream the live video from the phone. However, the transfer has a significant delay of about *3* seconds which is by far not applicable for position tracking. The developed components could be applied in the future for other applications, but for the AVS, we found that this solution is unacceptable. Therefore a second approach, the JPEG-compressed frame streaming was implemented.

6.2.6 JPEG Streaming

The GStreamer chain in Studierstube ES is also capable of injecting video that is received in a form of separate JPEG frames. The method is also referred to as Motion JPEG. The drawback of the MJPEG approach is that the temporal coherence between subsequent frames is not exploited for compression and thus the bandwidth need is higher. On the other hand, this method is highly suitable to our goal having minimal delay.

For the sake of simplicity, we replaced the cumbersome *MediaRecorder API* with the easier *Camera API*. The RTSP/RTP communication is also substituted by simple UDP transfer.

The Camera class lets us adjust camera settings, take pictures, and manipulate camera previews. The preview images are accessible through the *PreviewCallback* interface of the *Camera* class. The camera maintains a *PreviewBufferQueue* in which several *PreviewBuffers* are waiting to get filled with a preview image and passed to an object that implements the callback interface. The callback function holds the actual image and the camera pointer as parameter. This new approach (from Android version 2.2) with multiple preview buffers bridges a serious problem of older Android versions. In the former implementations with only a single *PreviewBuffer*, if the callback processing took too much time, the camera flooded the system with subsequent preview frames and this led to bursts in garbage collection and periodic halt of applications. The usage of multiple buffers makes possible to drop the surplus frames immediately resulting in smooth-running garbage collection. The buffer of every processed frame is pushed back to the queue again for reuse.

The object that implements the *PreviewCallback* interface is a modified version of our previous packetizer thread. The callback function copies the preview image into the own memory area of the packetizer, pushes back the applied *PreviewBuffer* to the camera's *PreviewBufferQueue* and returns. We use this method to take every small-resolution preview frame with a high refresh rate instead of taking the full-resolution images. The frames then get converted and compressed in the packetizer and are sent to the Tracker component with inconsiderable delay.

The current camera settings are available as a *Camera.Parameters* object. It can be used to specify the image and preview size, image format, and preview frame rate. We set the parameter values as shown below:

```
Camera.Parameters parameters = camera.getParameters();
parameters.setPreviewSize(< width from Settings >, < height from Settings >);
parameters.setFocusMode(Camera.Parameters.FOCUS_MODE_FIXED);
parameters.setAntibanding(Camera.Parameters.ANTIBANDING_50HZ);
```

```
parameters.setWhiteBalance(Camera.Parameters.WHITE_BALANCE_AUTO);
parameters.setPreviewFormat(ImageFormat.NV21);
parameters.setPreviewFrameRate(15);
camera.setParameters(parameters);
```

The size of the preview image can be set in the Settings Activity. The focus mode is set to fixed, because we calibrated the camera in a specific lens position. To avoid the flickering effect of the artificial lighting on the images we turned on the anti-banding for $50Hz$. The white balance is set to automatic. On the Nexus One, only the uncompressed NV21 preview format with $15fps$ refresh rate is available. The NV21 (also called YUV 4:2:0) image format consists of a plane of 8 bit Y (luminance) samples followed by an interleaved V/U plane containing 8 bit 2x2 subsampled chroma (Chrominance U and Chrominance V) samples[9]. As the Tracker works on gray-scale images, we could just take the Y luminance values from the beginning (the first *width* $\times$ *height* bytes) for further compression. But the internal Android JPEG-compressor does not recognize whether it deals with a gray-scale or a color image and compresses all the channels (R,G,B), resulting in a three times bigger file to transmit. It does the same with preview images that are converted to one-channel gray-scale images using the *EFFECT_MONO* switch. The relationship between the YUV and the RGB color space is given by the following equation:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.146 & -0.288 & 0.434 \\ 0.617 & -0.517 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The camera also requires an Android *SurfaceView* object for rendering of the preview frames. We use the *VideoSurface* that was also used for the *MediaRecorder* (depicted in Fig. 6.8). The *SurfaceView* is a GUI element which is created during the *onCreate()* method of the Activity. However, if it is passed to the *Camera* or the *MediaRecorder*, before it is actually visible on the screen, the application crashes. This is a known issue with Android. The solution is to implement the *SurfaceCreated()* callback function of the *SurfaceView*. This function is called first when the surface is already on the screen. Therefore, in our application this callback starts all the initialization processes of the media elements and the networking threads instead of the usual *onResume()* method of the Activity.

To summarize, here are the tasks performed by our client application:

- Gain control of the preview surface

[9] http://www.fourcc.org/yuv.php#NV21

- Open and setup the camera
- Create multiple preview buffers and push them to the camera
- Implement the PreviewCallback interface in the worker thread
- Copy the preview frame to the worker
- Convert, compress and send the frame

In the final setup, the GStreamer pipeline in the Tracker consists of a UDP sourcefilter, a JPEG decoder, a ColorSpace converter, and a filter that scales the image.

6.2.7 Overlay Presentation and User Interaction

The VolumeShop client *(AugVisVSClient)* thread (Fig. 6.8) has been developed independently from the video transfer. It connects to a VolumeShop viewport and decodes the received JPEG frames through a built-in image decoder class of Android. The images are pushed to the main Activity and shown to the user. The client also intercepts user input such as touch-screen or trackball events and sends back control messages to the Rendering component. This way the user can for instance rotate the scene by finger movements on the mobile device. A screenshot of the video that demonstrates the interaction can be seen in Fig. 6.9.

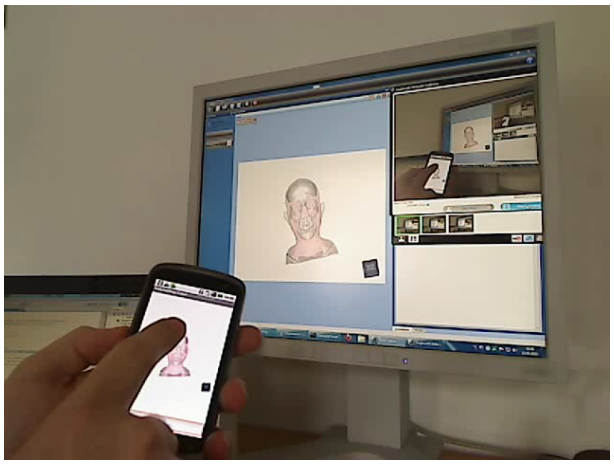


Fig. 6.9: A touch-screen controller for VolumeShop (on Google Nexus One)

6.3 The Rendering Component

The VolumeShop framework has been extended by additional plugins to integrate it into the Augmented Visualization System.

6.3.1 The RemoteInteractor Plugin

The RemoteInteractor plugin opens a UDP port (default 6678) and waits for datagrams containing XML-snippets. The XML-snippets are transformed to (and exposed as) properties of the plugin. The plugin accepts only XML that is wrapped in a *<data> ... </data>* element. This data-element may contain an arbitrary number of *<property>* elements. Each property-element must have the attributes *"type"* and *"name"*. The value of the property must be contained in the text of the property element. The supported types are the ones that are supported by VolumeShop. Examples of XML-snippets accepted by this plugin are:

```
<data>
    <property name="myintproperty" type="integer">123</property>
    <property name="myfloatproperty" type="float" >1.234</property>
</data>
```

```
<data>
    <property name="mymatrix" type="matrix">
        ((7.89812e-005;-0.000180452;0.00270937;0);
        (0.00249485;0.00107478;-1.14642e-006;0);
        (-0.00107188;0.00248832;0.000196985;0);
        (0;0;-1.73205;1))
    </property>
</data>
```

6.3.2 The MatrixTransformator Plugin

The Studierstube ES tracker and VolumeShop renderer are using different matrix notations. In the tracker all transformation matrices are multiplied from left[10] and in the renderer all matrices are multiplied from right[11]. Moreover, the dataset axes were rotated relative to the world coordinate system, so before and after geometric transformations we needed to change the coordinate systems. Therefore, a translator plugin was implemented, that takes the pose matrix and constructs an output matrix that is applicable to VolumeShop. Moreover, we found the pose output of the tracker quite noisy, so an averaging filter of the last five values is also applied in this plugin.

6.3.3 Overlay Generation

We set up different visualization sessions in VolumeShop to experiment with the outcomes of the augmented visualization. The common setup for all the sessions

[10] in mathematics, vectors are usually column vectors
[11] in computer graphics, vectors are traditionally row vectors

is to use the *RemoteInteractor* plugin (to receive XML input from the other components), the *MatrixTransformator* plugin (to convert between the tracker and the renderer coordinate systems) and two of the available volume rendering plugins. The first rendering plugin generates imagery for the main display (that is also sent to the Tracker component for analysis) and the second rendering plugin generates the overlay renderings. The properties of the plugins are linked together according to the actual user scenario.

When the remote interactor receives the estimated pose from the Tracker it passes it to the MatrixTransformator plugin. The pose and the modelview matrix of the main rendering are combined according to predefined rules. The content of the second rendering is generated using the transformed matrix and streamed to the mobile device. For instance in the multiuser demo (see chapter 7) two remote interactors and two hidden rendering plugins were present, using two different transfer functions.

6.4 Putting It All Together

We tested the prototype of the Augmented Visualization System with the following setup. The Rendering component was set up on a regular desktop PC, the Tracker component was set up on a commercial laptop and the Mobile component was installed on the Nexus One phone. The computers are connected via twisted pair cables to the LinkSys WRT54GL wireless Access Point. The mobile phone communicates with the other components over WiFi.

Fig. 6.10 depicts the data flow between the individual components during operation. The operation steps are as discussed during the design: 1.) A scene is rendered and presented on the common screen and simultaneously sent to the Tracker for feature extraction; 2.) The Mobile device captures the common content and 3.) continuously streams to the Tracker; 4.) the Tracker component estimates a relative pose from the two images and sends it back to the Mobile device and to the Rendering component to 5.) enhance user interaction and to 6.) render the personal overlay.

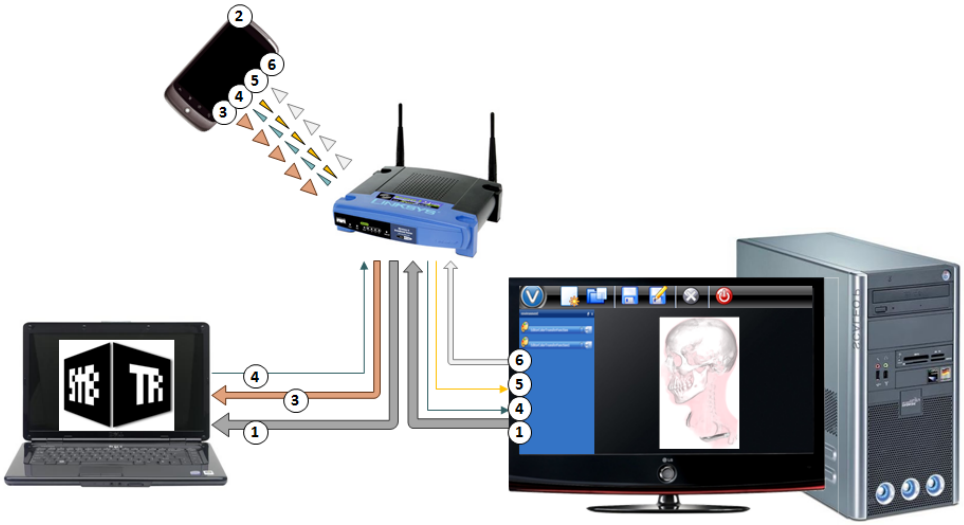


Fig. 6.10: The prototype of the Augmented Visualization System

7. Conclusion and Discussion

7.1 System Tests

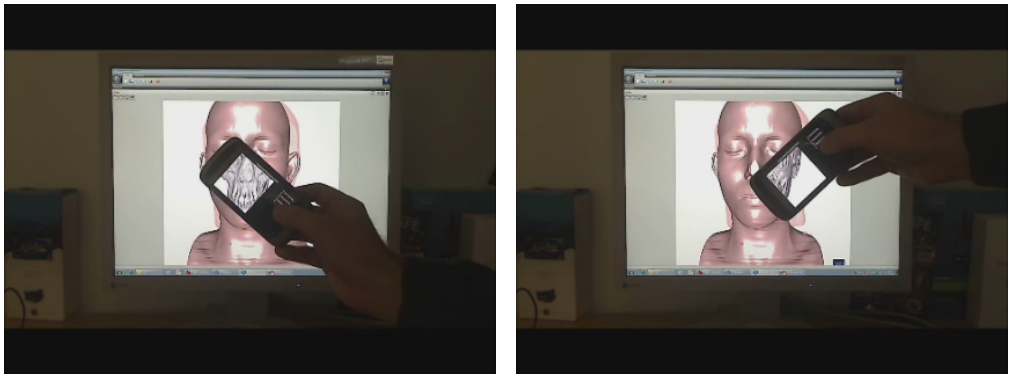
To demonstrate the capabilities of the novel Augmented Visualization System we implemented a couple of use scenarios.

The first scenario presents a magic lens application (Fig. 7.1), where the position and orientation of the mobile device are estimated. From that viewpoint the data set is rendered with a different transfer function showing other hidden features of the data (in this case the bones inside the body). The registration between the two images is not perfect due to the fact that the camera is not exactly in the middle of the phone and the pixel metrics provided by the Tracker are currently not correctly converted into the rendering coordinate system. This is however a remediable inaccuracy. By applying a constant multiplicative distortion of the distance parameter, zooming effects become possible as well.

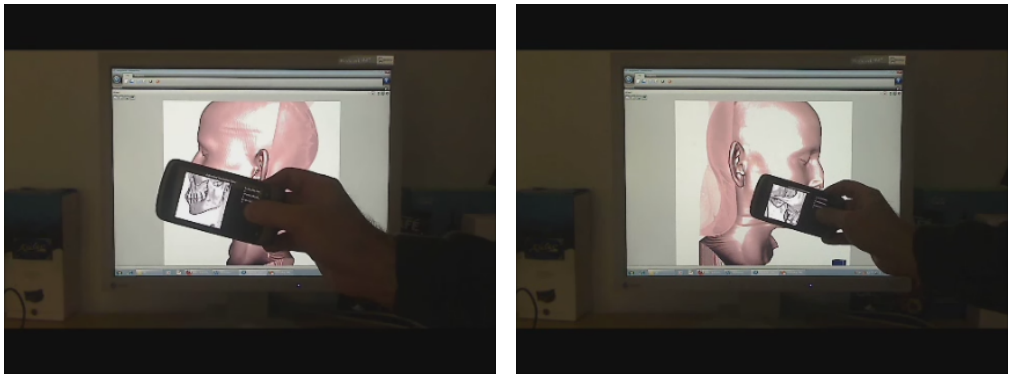
The second use scenario shows the same application with two mobile devices. At this point the multiuser setup requires multiple instances of Studierstube ES running. However, the limitation of exactly one video input in the tracker is supposed to change in the near future. As Fig. 7.2 shows, the distinct clients have different transfer functions. Here the opacity transfer function is the very same, but the colors are different.

The third test scenario demonstrates the interactive change of rendering parameters depending on the mobile device's orientation (Fig. 7.3). The overlay rendering is registered to the position and the transfer function depends on the orientation of the mobile device, so that one can see different tissues by adjusting the orientation.

In all three setups the change of the target image causes less than $500ms$ outage in tracking. The tests with a projection screen instead of the PC display were also successful with only small differences to mention. The matrix transformations have to be recalculated for the longer distances. The tracking works only in the case of distortion-free projection, since the relative distances between feature points need to be the same as in the reference image, to successfully calculate the homography.

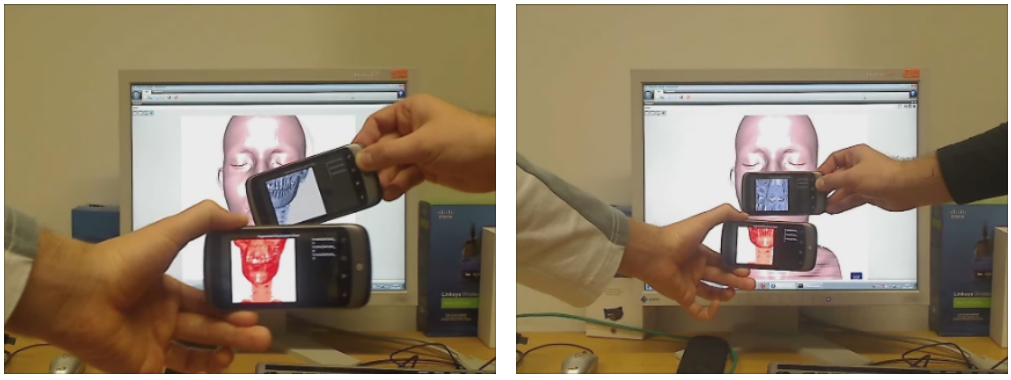


(a) (b)



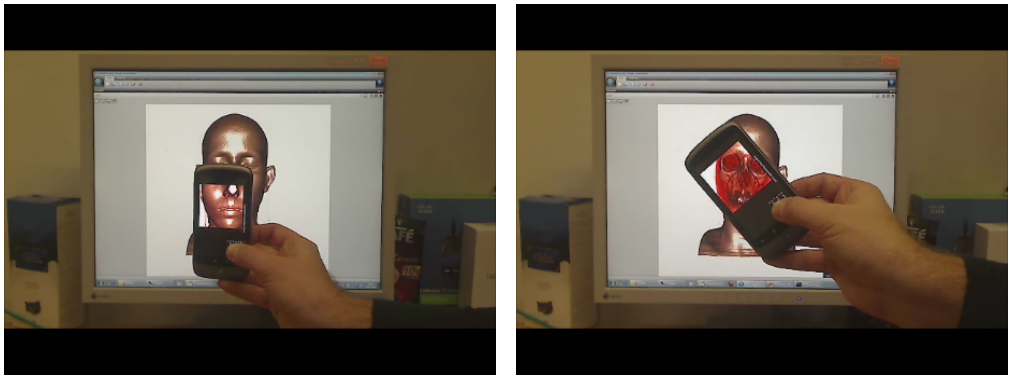
(c) (d)

Fig. 7.1: Snapshots from a system test video (X-Ray vision)

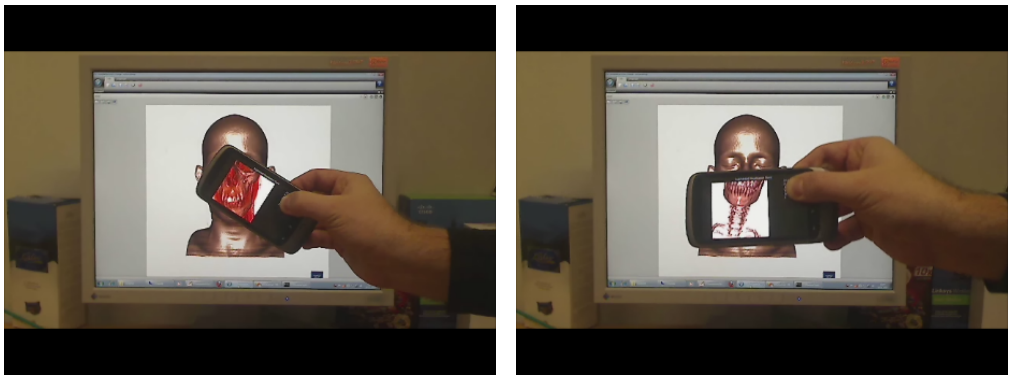


(a) (b)

Fig. 7.2: Snapshots from a system test video (multiple users)



(a) (b)



(c) (d)

Fig. 7.3: Snapshots from a system test video (orientation-dependent transfer function)

7.2 Discussion

In conclusion, we managed to implement a prototype of the Augmented Visualization System that fulfills all the goals we set out. The presented use scenarios are achieved with interactive frame rates and lead to a greatly enhanced user experience. The prototype has been built using the VolumeShop rendering framework, however, our approach can be used together with any visualization software that implements the presented minimal interface. The tracking works smoothly with 15 fps, but the rendering frame rates strongly depend on the complexity of the underlying data set. The pose estimation was also tested with a large projection screen without any decrease in performance. However, a test with a glossy laptop screen with significant reflections gave negative results.

The bottleneck of the system is definitely the radio transmission in both directions. With the current implementation the presented system is not scalable in terms of the number of users. The wireless bandwidth limits the number of videostreams to the server. Once the Android version of Studierstube ES is accessible, the system can be extended for more users with some architectural modifications. Our Stb-application is portable from the PC to the mobile, as it is embedded into the framework. The PC could run an instance of Studierstube ES which receives the rendered images, detects the features and builds the feature database. Then, it sends the database to all mobile clients in a multicast manner. The clients run Studierstube ES as well, but with feature detection on their own camera images. This setup would reduce the required bandwidth drastically, because only the database needs to be transferred and only once to a multicast group instead of the distinct whole video streams in the presented architecture.

Besides better scalability of the system, it would be interesting to extend our approach for multiple target images (e.g., a display wall consisting of several displays). As already mentioned, the accompanying sensors of the mobile device could improve the tracking accuracy for instance with a Kalman-filter. The accelerometer data could be utilized to turn the mobile into a Wiimote-like 3D interaction device.

The mobile user interface could be enhanced by a bundle of features, for instance a context-sensitive GUI, a personal transfer function editor for each participant, etc. In conclusion, our readily available prototype implementation opens a wide variety of research directions.

7.3 Other Applications

This form of augmented reality technology aligns well with the concept of pervasive computing, to provide a rich integrated ambient environment. The modular architecture of our system makes it possible to introduce further application scenarios.

7.3.1 The Virtual Collaboration Arena

As the presented system modules loosely connect to each other, the Rendering component is interchangeable to any other image generator, if it is also capable of processing the underlying data to produce the overlays. Our system is planned to be applied as a user interface component of the Virtual Collaboration Arena [38], a distributed robot control middleware at the Computer and Automation Research Institute of the Hungarian Academy of Sciences. The Augmented Visualization System could be attached to the VirCA system by implementing a thin TCP/IP server as a CyberDevice in the Arena.

7.3.2 An Interactive Board Game

Another prospective application is an interactive board game with a surface display[1]. The players sit around the display, which is actually a living board of the game: a plane for moving the figures on and at the same time telling stories and playing animations. If the players face their handheld devices to the board, they can get specific personal information on their game state, that no one else can see (like secret cards, information from the co-allied players, etc..). Whole series of strategic games can be envisioned this way.

[1] e.g., Microsoft Surface

8. Summary

This thesis project has dealt with the implementation of a novel computer-vision based interaction device for visualization applications. The rendering software was already available at the Vienna University of Technology and the Tracker component was under development at the Graz University of Technology. The contribution was to bring these components together to achieve enhanced user experience with off-the-shelf mobile devices. The mobile component was fully developed in the scope of this thesis. In the proposed architecture, real-time video streaming from the mobile device to the PC is applied. The main activities were as follows:

- I studied computer vision, augmented reality and advanced computer graphics
- I read several cutting-edge papers on these topics
- I studied the basics of visualization systems
- I selected the building blocks and created a plan of the Augmented Visualization System
- I created plugins for the VolumeShop rendering software in C++
- I travelled several times to Graz to consult with the Studierstube ES developers
- I created a Studierstube ES application in C++ that receives reference images and builds a feature database, receives video images and matches them to the reference, then estimates the pose of the camera relative to the reference target and sends the estimated pose matrix to other components of the system for subsequent processing
- I calibrated our cameras using MATLAB
- I designed and implemented the mobile client of the Augmented Visualization System on Android using the Java language and the Android SDK
- I implemented real-time H.263-compressed video transmission and JPEG-compressed frame transmission from the mobile phone to the PC using the RTP and RTSP protocols
- I connected the components into a working system and conducted successful user tests.

Bibliography

[1] N. Gershon - From perception to visualization, *in L. Rosenblum et al. (editors) - Scientific Visualization - Advances and Challenges, pp.129–139., Academic Press, 1994*

[2] R. Azuma - A Survey of Augmented Reality, *in Teleoperators and Virtual Environments, vol.6, no.4, pp.355–385, 1997*

[3] D. Kalkofen, E. Mendez, D. Schmalstieg - Interactive Focus and Context Visualization for Augmented Reality, *in Proc. 6th IEEE and ACM International Symposium on Mixed and Augmented Reality, 13–16 Nov. 2007, Nara, Japan*

[4] C. Bichlmeier, F. Wimmer, S.M Heining, N. Navab - Contextual Anatomic Mimesis Hybrid In-Situ Visualization Method for Improving Multi-Sensory Depth Perception in Medical Augmented Reality, *in Proc. 6th IEEE and ACM International Symposium on Mixed and Augmented Reality, 13–16 Nov. 2007, Nara, Japan*

[5] H. Slay, M. Phillips, R. Vernik and B.H. Thomas - Interaction Modes for Augmented Reality Visualization, *in Proc. Australian Symposium on Information Visualisation, 3–4 Dec. 2001, Sydney, Australia*

[6] H. Slay, B. Thomas - Interaction and Visualization across Multiple Displays in Ubiquitous Computing Envrioments, *in Proc. 4th Int. Conf. on Computer Graphics, Virtual Reality, Visualization and Interaction in Africa, 25–27 Jan. 2006, Cape Town, South Africa*

[7] R. Ballagas, M. Rohs, J. G. Sheridan - Mobile Phones as Pointing Devices, *in Proc. Workshop on Pervasive Mobile Interaction Devices, 11 May 2005, Munich, Germany*

[8] S. Boring, M. Altendorfer, G. Broll, O. Hilliges, A. Butz - Shoot & Copy: Phonecam-based Information Transfer from Public Displays onto Mobile Phones, *in Proc. 4th International Conference on Mobile Technology, Applications and Systems, 10–12 Sep. 2007, Singapore*

[9] T. Quack, H. Bay, L. V. Gool - Object Recognition for the Internet of Things, *in Proc. 1st Int. Conf. on Internet of Things, 26–28 Mar. 2008, Zurich, Switzerland*

[10] B. D. Allen, G. Bishop, G. Welch - Tracking: Beyond 15 Minutes of Thought, *SIGGRAPH 2001 course 11, Annual Conference on Computer Graphics & Interactive Techniques, ACM Press, Addison-Wesley, 12–17 Aug 2001, Los Angeles, USA*

[11] U. Neumann, S. You - Natural Feature Tracking for Augmented-Reality, *in IEEE Transactions on Multimedia, vol.1, no.1, pp.53–64, 1999*

[12] D. Robertson, R. Cipolla - An Image-Based System for Urban Navigation, *in Proc. British Machine Vision Conference, 7–9 Sep. 2004, London, United Kingdom*

[13] G. Klein - Visual Tracking for Augmented Reality, *PhD Thesis, University of Cambridge, 2006, Cambridge, United Kingdom*

[14] G. Reitmayr, T. W. Drummond - Going out: Robust Model-based Tracking for Outdoor Augmented Reality *in Proc. 5th IEEE and ACM International Symposium on Mixed and Augmented Reality, 22–25 Oct. 2006, Santa Barbara, USA*

[15] W. Zhang, J. Kosecka - Image-Based localization in Urban Environments, *in Proc. Int. 3rd Symposium on 3D Data Processing, Visualization and Transmission, 14–16 Jun. 2006, Chapel Hill, USA*

[16] G. Klein, D. W. Murray - Parallel Tracking and Mapping for Small AR Workspaces, *in Proc. 6th IEEE and ACM International Symposium on Mixed and Augmented Reality, 13–16 Nov. 2007, Nara, Japan*

[17] T. Lee, T. Höllerer - Hybrid Feature Tracking and User Interaction for Markerless Augmented Reality, *in Proc. IEEE Virtual Reality Conference, pp.145–152, 8–12 Mar. 2008, Reno, USA*

[18] D. Wagner - Handheld Augmented Reality, *PhD Thesis, Graz University of Technology, 2007, Graz, Austria*

[19] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, D. Schmalstieg - Pose tracking from natural features on mobile phones, *in Proc. 7th IEEE and ACM*

International Symposium on Mixed and Augmented Reality, 15–18 Sep. 2008, Cambridge, United Kigdom

[20] D. Wagner, D. Schmalstieg - History and Future of Tracking for Mobile Phone Augmented Reality, *Int. Symposium on Ubiquitous Virtual Reality, 8–11 Jul. 2009, Gwangjou, Korea*

[21] D. Wagner, D. Schmalstieg, H. Bischof - Multiple target detection and tracking with guaranteed framerates on mobile phones, *in Proc. 9th IEEE and ACM International Symposium on Mixed and Augmented Reality, 19–22 Oct. 2009, Orlando, USA*

[22] C. Arth, D. Wagner, M. Klopschitz, A. Irschara, D. Schmalstieg - Wide area localization on mobile phones, *in Proc. 9th IEEE and ACM International Symposium on Mixed and Augmented Reality, 19–22 Oct. 2009, Orlando, USA*

[23] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, D. Schmalstieg - Real-Time Detection and Tracking for Augmented Reality on Mobile Phones, *in IEEE Transactions on Visualization and Computer Graphics, vol.16, no.3, pp.355–368, 2010*

[24] E. Rosten, T. Drummond - Fusing points and lines for high performance tracking, *in Proc. IEEE International Conference on Computer Vision, 17–20 Oct. 2005, Beijing, China*

[25] E. Rosten, T. Drummond - Machine learning for high-speed corner detection, *in Proc. European Conference on Computer Vision, 7–13 May 2006, Graz, Austria*

[26] D. G. Lowe - Distinctive Image Features from Scale-Invariant Keypoints, *in International Journal of Computer Vision, vol.60, no.2, pp.91–110, 2004*

[27] H. Bay, A. Ess, T. Tuytelaars, L. V. Gool - SURF: Speeded Up Robust Features, *in Computer Vision and Image Understanding (CVIU), vol.110, no.3, pp.346–359, 2008*

[28] D. Nistér, H. Stewénius - A Minimal Solution to the Generalised 3-Point Pose Problem, *in Journal of Mathematical Imaging and Vision, vol.27, no.1, pp.67–79, 2007*

[29] Al Bovik (editor) - The Essential Guide to Video Processing, *ISBN: 978-0-12-374456-2, Elsevier, 2009*

[30] ITU-T Specification for H.263: Video coding for low bit rate communication *http://www.itu.int/rec/T-REC-H.263/, Jan. 2005*

[31] L. Hanzo, P. Cherriman, J. Streit - Video Compression and Communications - From Basics to H.261, H.263, H.264, MPEG4 for DVB and HSDPA-Style Adapt, *ISBN: 978-0-470-51849-6, Wiley, 2007*

[32] 3GPP TS 26.244 Transparent end-to-end packet switched streaming service (PSS); 3GPP file format (3GP), *http://www.3gpp.org/ftp/Specs/html-info/26244.htm, v9.2.0, Jun. 2010*

[33] H.Schulzrinne: Real-Time Streaming Protocol, *http://www.cs.columbia.edu/~hgs/rtsp/rtsp.html, 01 Oct. 2008*

[34] RFC 3550 Real-Time Transport Protocol, *http://www.ietf.org/rfc/rfc3550.txt*

[35] RFC 3551 RTP Profile for Audio and Video Conferences with Minimal Control, *http://www.ietf.org/rfc/rfc3551.txt*

[36] RTP Payload Format for ITU-T Rec. H.263 Video, *http://www.ietf.org/rfc/rfc4629.txt*

[37] RFC 2327 SDP: Session Description Protocol, *http://www.ietf.org/rfc/rfc2327.txt*

[38] A. Vamos, I. Fulop, B. Resko, and P. Baranyi - Collaboration in Virtual Reality of Intelligent Agents, *Acta Electrotechnica et Informatica, vol.10, no.2, pp.21–27, 2010*

[39] S. Bruckner, M. E. Gröller - VolumeShop: An Interactive System for Direct Volume Illustration, *in Proc. IEEE Visualization, pp.671–678, 23–25 Oct. 2005, Minneapolis, USA*

[40] W. F. Ableson, C. Collins, R. Sen - Unlocking Android - A developer's guide, *Manning Publications, 2008*

[41] A. Becker, M. Pant - Android, Grundlagen und Programmierung, *ISBN: 978-3-89864-574-4, DPunkt Verlag, 2009*

[42] E. Burnette - Hello, Android!, *ISBN: 978-1-934356-17-3, The Pragmatic Bookshelf, 2008*

[43] R. Meier - Professional Android Application Development, *ISBN: 978-0-470-34471-2, Wiley Publishing, 2009*

[44] Android Developer's Guide, *http://developer.android.com*

[45] J. Y. Bouguet: Matlab Camera Calibration Toolbox, *http://www.vision.caltech.edu/bouguetj/calib_doc/index.html*, *9 Jul. 2010*

[46] T. Langlotz: Studierstube ES, *http://studierstube.icg.tu-graz.ac.at/handheld_ar/*, *28 May 2010*

[47] D. Kalkofen: Visual AR, *http://studierstube.icg.tu-graz.ac.at/visual_ar/*, *28 May 2010*

Appendices

RTSP Communication Between VLC Players

RTSP conversation between two VLC clients captured by the Wireshark network analysator.

```
OPTIONS rtsp://192.168.1.101:4444/ RTSP/1.0
CSeq: 2
User-Agent: LibVLC/1.1.4 (LIVE555 Streaming Media v2010.08.22)


RTSP/1.0 200 OK
Server: VLC/1.1.2
Content-Length: 0
Cseq: 2
Public: DESCRIBE,SETUP,TEARDOWN,PLAY,PAUSE,GET_PARAMETER


DESCRIBE rtsp://192.168.1.101:4444/ RTSP/1.0
CSeq: 3
User-Agent: LibVLC/1.1.4 (LIVE555 Streaming Media v2010.08.22)
Accept: application/sdp


RTSP/1.0 200 OK
Server: VLC/1.1.2
Date: Tue, 21 Sep 2010 11:58:49 GMT
Content-Type: application/sdp
Content-Base: rtsp://192.168.1.101:4444/
Content-Length: 319
Cache-Control: no-cache
Cseq: 3

v=0
o=- 15006869690428324266 15006869690428324266 IN IP4 ORESTES
s=Unnamed
i=N/A
c=IN IP4 0.0.0.0
t=0 0
a=tool:vlc 1.1.2
a=recvonly
a=type:broadcast
a=charset:UTF-8
a=control:rtsp://192.168.1.101:4444/
m=video 0 RTP/AVP 96
b=RR:0
a=rtpmap:96 H263-1998/90000
a=control:rtsp://192.168.1.101:4444/trackID=0


SETUP rtsp://192.168.1.101:4444/trackID=0 RTSP/1.0
CSeq: 4
User-Agent: LibVLC/1.1.4 (LIVE555 Streaming Media v2010.08.22)
Transport: RTP/AVP;unicast;client_port=49178-49179
```

```
RTSP/1.0 200 OK
Server: VLC/1.1.2
Date: Tue, 21 Sep 2010 11:58:49 GMT
Transport: RTP/AVP/UDP;unicast;client_port=49178-49179;
           server_port=59750-59751;ssrc=CB90AF8A;mode=play
Session: a6ba8e9aa53f6f6e
Content-Length: 0
Cache-Control: no-cache
Cseq: 4


PLAY rtsp://192.168.1.101:4444/ RTSP/1.0
CSeq: 5
User-Agent: LibVLC/1.1.4 (LIVE555 Streaming Media v2010.08.22)
Session: a6ba8e9aa53f6f6e
Range: npt=0.000-


RTSP/1.0 200 OK
Server: VLC/1.1.2
Date: Tue, 21 Sep 2010 11:58:49 GMT
RTP-Info: url=rtsp://192.168.1.101:4444/trackID=0;seq=59780;rtptime=420790680
Session: a6ba8e9aa53f6f6e
Content-Length: 0
Cache-Control: no-cache
Cseq: 5


GET_PARAMETER rtsp://192.168.1.101:4444/ RTSP/1.0
CSeq: 6
User-Agent: LibVLC/1.1.4 (LIVE555 Streaming Media v2010.08.22)
Session: a6ba8e9aa53f6f6e


RTSP/1.0 200 OK
Server: VLC/1.1.2
Date: Tue, 21 Sep 2010 11:58:49 GMT
Session: a6ba8e9aa53f6f6e
Content-Length: 0
Cache-Control: no-cache
Cseq: 6


TEARDOWN rtsp://192.168.1.101:4444/ RTSP/1.0
CSeq: 7
User-Agent: LibVLC/1.1.4 (LIVE555 Streaming Media v2010.08.22)
Session: a6ba8e9aa53f6f6e


RTSP/1.0 200 OK
Server: VLC/1.1.2
Date: Tue, 21 Sep 2010 11:59:06 GMT
Session: a6ba8e9aa53f6f6e
Content-Length: 0
Cache-Control: no-cache
Cseq: 7
```

The Transformation Pipeline in Computer Graphics

In 3D computer graphics, one approach for depicting a virtual scene is the rendering pipeline based on rasterization. The virtual world is viewed by a camera through a rectangular window. During the image synthesis it has to be calculated how virtual models occlude each other from the camera viewpoint and which object can be seen in a specific pixel of the rendered image. This calculation can be easily performed in the so called viewport space, where projection and occlusion are trivial tasks. To transform virtual models into viewport space, the vertices of the models undergo the transformation pipeline shown in Fig. A.1. The transformations are performed as matrix multiplications in homogenous coordinates. The virtual models are available in their local *modeling* coordinate system. The *model transformation* brings them into a common virtual world coordinate system. The *view transformation* on one hand places the camera to the desired position and orientation in the world and on the other hand transforms the model coordinates as they are seen from the camera. The *projection transformation* projects the vertices of the model onto the image plane. After the homogenous division the coordinates are given in the Cartesian system. At the end, the *viewport transformation* results in pixel coordinates. In OpenGL, the model matrix and the view matrix are handled together as the model-view matrix.

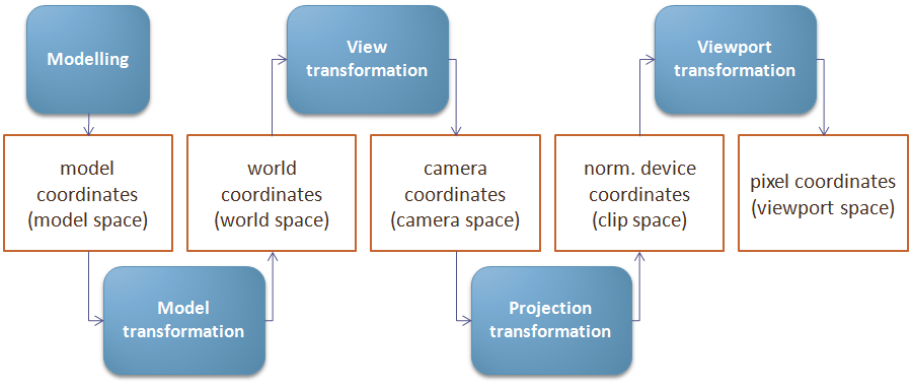


Fig. A.1: The transformation pipeline in computer graphics

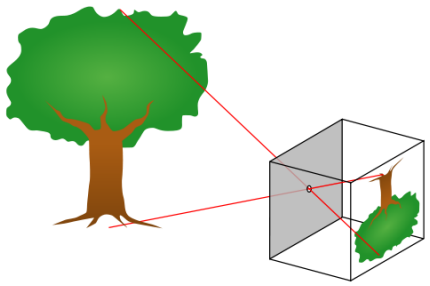


Fig. A.2: The pinhole camera model

The Pinhole Camera Model

A pinhole camera is a simple camera without a lens and with a single small aperture (the pinhole). Light travels from one point along a straight path through the pinhole and falls onto the images plane (see Fig. A.2). The object is projected upside-down on the image plane. Since the process entails a central projection, the images in the pinhole camera are rendered in ideal perspective. The distance of the image plane from the projection center is known as the focal length *(f)* of the camera (Fig. A.3). If f is known, the distances of (at least three) known object points and the distances of their projected correspondences on the image can be used to determine the distance of the camera from the object. Another special characteristic is the infinite depth of field which means objects have equal sharpness no matter if they are very close to the camera or far away. The pinhole camera model is used in OpenGL as well as in Studierstube ES.

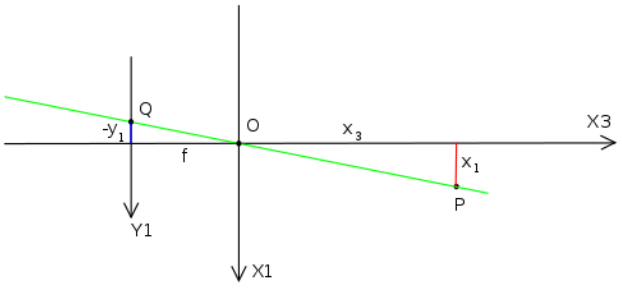


Fig. A.3: Schematic of the projection in the pinhole camera

Calculation of Euler Angles from a Rotation Matrix

Any orientation can be described as the multiplication of three rotations. The angles of those three rotations are called Euler angles (heading, attitude and bank). This conversion uses conventions as described on page: `http://www.euclideanspace.com/maths/geometry/rotations/euler/index.htm`. The coordinate system is right-handed, the positive angles follow the right-hand rule, the order of euler angles is 1.) heading, 2.) attitude and 3.) bank. All angles are in radian and the matrix row-column ordering is:

$$\begin{bmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{bmatrix}$$

```
void calculateEulerFromMatrix(float *heading, float *attitude,
                              float *bank, Matrix * Mtx)
{
    float pi = 3.14159265f;

    if ( Mtx->Get(1,0) > 0.998) { // singularity at north pole
        *heading = atan2(Mtx->Get(0,2),Mtx->Get(2,2));
        *attitude = pi/2;
        *bank = 0;
        return;
    }

    if (Mtx->Get(1,0) < -0.998) { // singularity at south pole
        *heading = atan2(Mtx->Get(0,2),Mtx->Get(2,2));
        *attitude = -pi/2;
        *bank = 0;
        return;
    }

    *heading = atan2(-Mtx->Get(2,0),Mtx->Get(0,0));
    *bank = atan2(-Mtx->Get(1,2),Mtx->Get(1,1));
    *attitude = asin(Mtx->Get(1,0));

    return;
}
```

Pseudocode of the Main Thread in the Custom Studierstube ES Application

Algorithm 1

```
loop
  get input video frame
  show video frame
  if tracker is active then
    if PatchTracker finds target then
      //calculate and send pose
      calculate pose
      send pose (in XML structure)
      render 3D Studierstube AR scene (not used)
    else
      //run feature matching
      detect keypoints on video frame
      create feature descriptors
      match to feature database
      remove outliers and calculate homography
      if target found then
        restart PatchTracker
      else
        continue
      end if
    end if
  end if
  render 2D GUI of Studierstube ES
end loop
```

Pseudocode of the Receiver Thread in the Custom Studierstube ES Application

Algorithm 2

```
loop
  //wait for new reference image
  receive from network socket (blocking)
  convert color JPEG to grayscale RAW
  downsample both the old reference and the new reference
  calculate sum of pixel differences
  if sum ≥ threshold then
    //the new received reference differs from the previous one
    set Tracker inactive
    detect keypoints (for multiple scales)
    create feature descriptors (for multiple scales)
    refresh feature database (delete previous features)
    set reference as new target
    set Tracker active
  end if
end loop
```

Studierstube ES Config XML File

```
<?xml version="1.0" encoding="UTF-8"?> <!-- MyApp config file -->
<StbES>
   <!-- define type and target of logging -->
   <Logging
      file="data/log.txt" draw-fps="true" level="info"
      write-video="false" append="false" />
   <!-- define the tracker application's window size and orientation -->
   <Window
      width="640" height="480" rotation="0" />
   <!-- define the augmented reality render target -->
   <RenderTarget
      type="PixMap" width="320" height="240"
      fitToVideoImg="false" windowScale="true" />
   <!-- select the video input of the tracker -->
   <Video
      type="GStreamer" file="data/dsvl.cam.xml" grayscale="always" sync="full" />
   <!-- choose a scene file to render on the target (traditional AR) -->
   <Scene
      file="data/scene.xml" />
   <!-- define GUI elements -->
   <WidgetManager
      font="data/raster_8x12_256.png" char-width="8" char-height="16" />
   <!-- setup a tracker to use -->
   <Tracker
      active="true" displayInfos="full">
      <!-- select the Natural Feature Tracker v2 -->
      <StbTrackerNFT2
         database="data/dummy" target-path="data"
         cam-file="data/Gabor_Nexus3_320x240_StbTracker"
         render-clip="100 5000" mode="single" frame-rate="30">
         <!-- define a target (only for initialization) -->
         <Target
            name="VS" ref="dummy" />
      </StbTrackerNFT2>
   </Tracker>
   <!-- select an application to load -->
   <Application
      name="MyApp" />
</StbES>
```

List of Figures

List of Tables

Contents of the CD

- PDF version of this document
- Demonstration video of tracking
- Demonstration video of H.263 streaming
- Demonstration video of JPEG streaming
- Demonstration video of user-interaction using the touch-screen
- Demonstration video of the overlay streaming
- Video of the X-Ray vision test
- Video of the test with multiple users
- Video of the test using orientation-dependent transfer functions
- AugVisClient source code for Android

Acknowledgements

Hereby I would like to express my deepest gratitude to my supervisors, Dr. Peter Rautek in Vienna and Dr. Péter Baranyi in Budapest for their valuable suggestions and feedbacks. I also wish to thank Dr. Clemens Arth and Dr. Hartmut Seichter from the Christian Doppler Laboratory for Handheld Augmented Reality for their professional support with Studierstube ES and to Meister for the wonderful and stimulating working atmosphere.

Furthermore I wish to thank my parents and my family, who have always supported me during my studies and provided all the conditions of starting a successful career. I am grateful to my girlfriend for her patience and love.

My work would not have been possible without financial support from the Peregrinatio Foundation and the Institute of Computer Graphics and Algorithms. The equipment was also supported by the HUNOROB project (HU0045, 0045/NA/2006-2/ÖP-9), a grant from Iceland, Liechtenstein and Norway through the EEA Financial Mechanism and the Hungarian National Development Agency.