FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Augmented Reality Video

Situated Video Compositions in Panorama-based Augmented Reality Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik

eingereicht von

Mathäus Zingerle
Matrikelnummer 0525931

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Privatdoz. Mag.rer.nat. Dr.techn. Hannes Kaufmann
Mitwirkung: Dr. techn. Gerhard Reitmayr
Dipl.-Mediensys. wiss. Tobias Langlotz

Wien, 03.09.2012

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Mathäus Zingerle
Lindengasse 42/2/16, 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

First of all I want to thank my parents Walter and Ernestine Zingerle for their ongoing support over all the years.

I want to thank my advisors Hannes Kaufmann and Gerhard Reitmayr for making this work possible, especially for the smooth cooperation between TU Wien and TU Graz. Special thanks go to Tobias Langlotz for helping me out with all the technical details and answering my questions at any time of the day.

Further, I want to thank all my friends who participated in this work, namely Stefan Eberharter, Marco Holzner, Andreas Humer, Stephan Storn, Philipp Schuster, Mario Wirnsberger, Laura Benett and Philipp Mittag, either by helping me with all the video material, taking part in the user study or providing their mobile phones.

I also want to thank all the people who took some time to proofread parts of this work, namely my brother Patrick Zingerle, Brandon Gebka and especially my sister Jacqueline Zingerle and Travis Adams.

Finally I want to thank my friends once again for distracting me from time to time, which kept my motivation going.

Abstract

The rapid development of mobile devices such as smart phones has led to new possibilities in the context of Mobile Augmented Reality (AR). While there exists a broad range of AR applications providing static content, such as textual annotations, there is still a lack of supporting dynamic content, such as video, in the field of Mobile AR. In this work a novel approach to record and replay video content composited in-situ with a live view of the real environment, with respect to the user's view onto the scene, is presented. The proposed technique works in real-time on currently available mobile phones, and uses a panorama-based tracker to create visually seamless and spatially registered overlays of video content, hence giving end users the chance to re-experience past events at a different point of time. To achieve this, a temporal foreground-background segmentation of video footage is applied and it is shown how the segmented information can be precisely registered in real-time in the camera view of a mobile phone. Furthermore, the user interface and video post effects implemented in a first prototype within a skateboard training application are presented. To evaluate the proposed system, a user study was conducted. The results are given at the end of this work along with an outlook on possible future work.

Kurzfassung

Die rasante Entwicklung von mobilen Geräten wie Smartphones hat zu neuen Möglichkeiten im Bereich von Mobile Augmented Reality (AR) geführt. Obwohl es bereits eine Vielzahl an AR Anwendungen gibt, die die Integration von statischem Inhalt (z.B. Text) erlauben, besteht noch immer ein Mangel an mobilen AR Anwendungen, welche die Integration von dynamischen Inhalten (z.B. Video) ermöglichen. In dieser Arbeit wird ein neuartiger Ansatz präsentiert, welcher es erlaubt an einem Interessenspunkt aufgenommenes Videomaterial an derselben örtlichen Stelle wiederzugeben, so dass die Live-Ansicht einer realen Umgebung mit diesem Videomaterial überlagert wird, unter Berücksichtigung des aktuellen Blickwinkels des Benutzers auf die betrachtete Stelle. Das vorgestellte System benützt einen Tracker, welcher auf einem Umgebungspanorama basiert, um eine räumliche Registrierung von Overlays (erzeugt aus dem Videomaterial) zu ermöglichen. Diese Overlays können somit durch das Tracking innerhalb dieser Umgebung nahtlos in die Live-Ansicht einer Anwendung eingebettet werden. Dadurch wird Endbenutzern erlaubt vergangene Geschehnisse zu einem anderen Zeitpunkt wiederzuerleben. Um dies zu erreichen, wird zuerst eine Segmentierung von Vordergrund und Hintergrund in dem aufgenommenen Videomaterial durchgeführt. Basierend darauf wird veranschaulicht, wie diese extrahierte Information präzise und in Echtzeit in der Live-Ansicht von aktuell verfügbaren Mobiltelefonen integriert werden kann. Außerdem wird ein erster Prototyp vorgestellt (als Teil einer Skateboard-Trainingsanwendung), inklusive Beschreibung des User Interfaces und implementierter Post-Effekte. Abschließend werden die Resultate einer im Zusammenhang mit dieser Arbeit durchgeführten Benutzerstudie vorgestellt und eine Aussicht auf mögliche zukünftige Verbesserungen in Bezug auf das vorgestellte System gegeben.

Contents

Part I

Theoretical Foundations

CHAPTER 1

Introduction

1.1 Problem Definition

The availability of inexpensive mobile video recorders and the integration of high quality video recording capabilities into smartphones have tremendously increased the amount of videos being created and shared online. With more than 70 hours of video uploaded every minute to *YouTube*[1] and more than 3 billion hours of video viewed each month[2], new ways to search, browse and experience video content are highly relevant.

In addition, AR has become a new player in the mobile application landscape. It takes its shape primarily in the form of so called mobile AR browsers that augment the physical environment with digital assets associated with geographical locations or real objects. These assets usually range from textual annotations over 2-dimensional (2D) images to complex 3-dimensional (3D) graphics. Most of the current generation AR browsers use sensor-based tracking to register these assets that are usually tagged with Global Positioning System (GPS) coordinates and then based on these coordinates integrated into the users view. Even though it is known that the accuracy of the positioning data delivered by GPS usually lies within a few meters it still allows an ubiquitous augmentation of the environment. Moreover, most smartphones nowadays are equipped with GPS-sensors (along with other sensors such as an accelerometer or compass).

In this work, it is investigated how to offer a new immersive user experience in a mobile context through compositing the user's view of the real world with prerecorded video content.

[1]http://www.youtube.com
[2]http://www.youtube.com/t/press_statistics

Similar to [29], this work is interested in extracting the salient information from the video (e.g. moving person or objects) and offering possibilities to spatially navigate the video (by rotating a mobile device such as a phone) mixed with the view of the real world. Contrary to the work in [29], this work focuses on mobile platforms in outdoor environments and also aims to provide simple ways to record/capture and further process the required video content with only minimal user input. Furthermore, the system which is about to be presented relies on fewer restrictions during the recording, as rotational camera movements are supported and the system does not rely on a green screen type of technology for recording the video augmentations.

Hence, in this work an interactive outdoor AR technique is presented, offering accurate spatial registration between recorded video content (e.g. person, motorized vehicles) and the real world with a seamless visual integration of the previously extracted object of desire contained in the recorded video material; integrated into the live camera view of the user's mobile device. The system allows one to replay video sequences to interactively re-enact a past event for a broad range of possible application scenarios covering sports, history, cultural heritage or education. A variety of user tools to control video playback and to apply video effects are proposed, thereby delivering the first prototype of what could be a real-time AR video montage tool for mobile platforms (see Fig. 9.1).

The proposed system shall operate in three steps. The first step is the shooting of the video, including uploading/transferring the video to a remote or local working station (e.g. desktop PC) for further processing. In a second step, the object of interest in the video frames shall be extracted, and later augmented in place. This preprocessing task can be performed on the aforementioned working station. A segmentation algorithm shall be applied which only requires minimal user input, such as outlining the object of interest in the first frame of the video. Additionally, the background information of the video shall be extracted and assembled into a panoramic representation of the background, which shall later be used for the precise registration of the video content within the application environment. The final step is the "replay" mode. This mode shall be enabled once a mobile user moves close to the position where a video sequence was shot. Assuming the mobile device is equipped with all resources necessary for the replaying of the video the user shall be able to explore the past event in the outdoor environment by augmenting the video into the user's view using best-practice computer vision algorithms.

1.2 Contribution

The proposed system contributes to the field of Augmented Reality by demonstrating how to seamlessly integrate video content into outdoor AR applications and allowing end users to participate in the content creation processes. Thus, several subfields shall be highlighted:

- Creation of suitable video source material for Augmented Reality applications.
- Segmentation of dynamic objects in dynamic video material.
- Mapping of panoramic images with respect to the background information.
- Real-time tracking in outdoor environments without sensor-based tracking.
- Seamless integration of video augmentations into the live view of AR-capable devices, with respect to the current camera pose.
- Application of video effects in real time without pre-rendering the content.

Based on this thesis, and the implemented system, a paper which is about to be published [26] was written together with members of the *Christian Doppler Laboratory for Handheld Augmented Reality*, proving the relevance of the presented work.

CHAPTER 2

Literature And Related Work

The purpose of this chapter is to give some theoretical background about Augmented Reality and its requirements in general, as well as the transition to Mobile Augmented Reality. This includes the comparison of requirements and possible techniques for implementing AR systems targeting outdoor environments which one necessarily encounters in the field of Mobile AR. Moreover, a look at state-of-the-art technologies in the field of Mobile AR and comparable video applications is given at the end of the chapter.

2.1 History of Augmented Reality

Although Augmented Reality was not a wide known term before the 1990's the first Augmented Reality system was actually installed already back in 1968 by Sutherland and described in [44]. Due to the very limited computational power at that time the "head mounted three-dimensional display" was a giant machine and yet only capable of drawing a few simple line graphics onto the display.
It was not until 1982 that the first laptop computer, the Grid Compass 1100, was released. It was the first laptop with the "clamshell" design as we know it today. With a display of 320 x 240 pixels and only a few hundred kilobytes of internal memory it was still extremely powerful for that time. Its portability was limited though due to its weight of 5 kg.

Ten years later the first smartphone was introduced by IBM[1] and carrier Bellsouth. The device did not contain a camera, yet worked as a phone, pager, e-mail client, etc. In 1993 the

[1]http://www.ibm.com

Global Positioning System (GPS), which is widely used today in a variety of devices such as car navigation systems and mobile phones, was released for public use. In the same year Fitzmaurice introduced "Chamaeleon" [15]. It was one of the first prototypes of a mobile AR system. The idea was to use a portable computer to access and manipulate situated 3D information spaces throughout our environment, such that the computer's display acts as an "information lens" near physical objects. The device was aware of its physical position and orientation relative to a map, such as a geographical map. It was able to provide information about cities dependent on the user's gestures and movements. In [15] further possible application scenarios were described, e.g. a computer-augmented library. The idea was that books and shelves emit navigational and semantic information to access an electronic library. Another idea was to remotely access an office, by using 360 degrees panoramic images. With the proposed idea the office would have been accessed by a portable device from home, such that the remote view could be augmented with graphical and audio annotations ("graphical post-its").

In the mid 1990's the term *Augmented Reality* was manifested and the distinction between *Augmented* and *Virtual Reality* was pointed out [11], [33], [7]. One of the widely accepted definitions for Augmented Reality was introduced in 1997 by Azuma in [7]. The definition states that "*AR allows the user to see the real world, with virtual objects superimposed upon or composited with the real world. Therefore, AR supplements reality, rather than completely replacing it. Ideally, it would appear to the user that the virtual and real objects coexisted in the same space*" [7]. Hence, "*maintaining accurate registration between real and computer generated objects is one of the most critical requirements for creating an augmented reality*" [32]. This means when the viewpoint onto the scene of interest is moved the rendered computer graphics need to somehow remain aligned accurately with the 3-D locations and orientations of real objects, i.e. the real-world view of the user [32], [18]. To achieve such alignment an accurate tracking (or measuring) of the real world viewing pose is necessary, because only an accurate tracked viewing pose allows for correctly projecting computer graphics into the real-world view of any AR device [32], [18].

With improving computational power in desktop computers and refined tracking mechanisms (e.g. 2D matrix markers, see [37]) desktop AR Systems were improving steadily, yet mobile AR systems were still practically not available. In 1997 the *Touring Machine*, the first Mobile Augmented Reality System (MARS) was presented as a prototype [14] which was further refined and explored as described in [20]. The system allowed users to access and manage information spatially registered with the real world in indoor and outdoor environments. The user's view was augmented by using a see-through head-worn display, while all necessary hardware was integrated into a backpack which the user had to carry around. Although being fully mobile, the system was not of practical use to end users as the combined weight of the system was just under 40 pounds.

The late 1990's saw the integration of today's basic features of mobile AR into handheld devices, such as cameras and GPS sensors. In the early 2000's mobile AR systems were still developed either as a combination of e.g. head-worn-displays and devices such as Personal Digital Assistants (PDAs) or indeed running on mobile devices, yet depended on a desktop workstation where the computational expensive tasks were outsourced to, for example see [21], [24] or [45]. In 2003 the first system running autonomously on a PDA (i.e. all tasks were carried out on the PDA) was presented as an indoor AR guidance system, whereas in 2006 one of the first systems using a model-based tracking approach (in contrast to e.g. GPS localization) for outdoor AR applications on handheld devices was described in [36].

According to [48] in the following years mobile AR applications primarily (yet not only) improved due to refined algorithms and approaches rather than making use of improved hardware. Although smart phone hardware might not be as sophisticated as desktop computer hardware it still improved a lot in the last few years which actually led to the development of mobile AR applications which are useful to the end user. One could say that most AR applications before that have been developed in terms of scientific surveys, to find out what is feasible and might be of use to end users, yet have been restricted in development due to limited hardware/bandwith resources.

Recent efforts have been made regarding the refinement of tracking and localization mechanisms in wide-area environments (i.e. outdoor) [34], [6] or indeed on how to continually improve the definition of adequate use cases for AR [31]. Another interesting aspect is the fusion of all available device-integrated sensor information to achieve the best possible outcome for tracking and localization [41].

Although the quality of AR applications is increasing steadily there is still a lack of "real" content [17]. Currently available AR browsers are mainly based on the concept of using geo-referenced images, textual annotations, audio stickies, etc. In addition, these annotations can then lead the user to a website or similar for further information. Integrating more sophisticated content like 3D graphics remains a challenging task so far, due to the complex preprocessing which is necessary to create 3D models. Mobile AR browsers like Argon[2] build on the idea of

[2]http://argon.gatech.edu

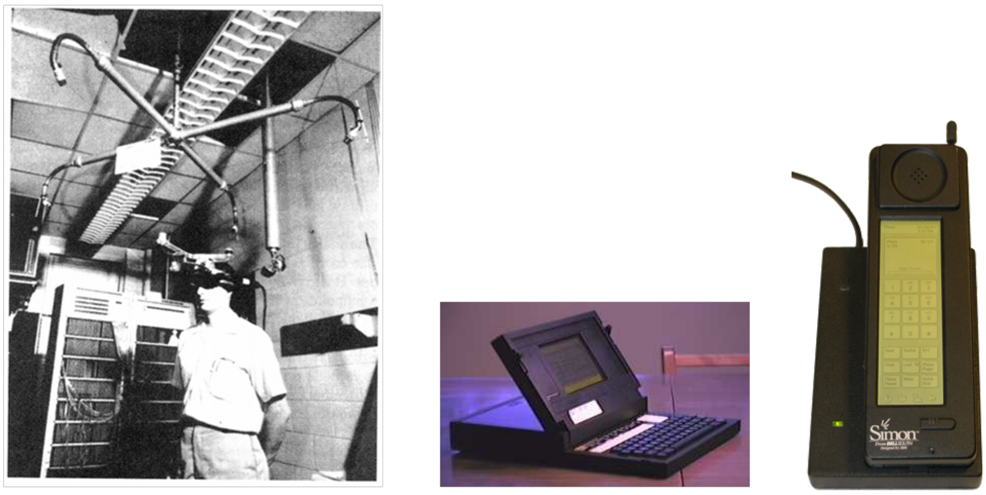


Figure 2.1: (Left) The "head mounted three-dimensional display" - the first Augmented Reality system. Taken from [44]. (Middle) The follow-up model (the Grid Compass 1101) to the first "clamshell" laptop (the Grid Compass 1100). Taken from [5]. (Right) The first "smart phone" known as *IBM Simon Personal Computer*.

letting users create and experience augmented content, yet are limited to static information or it requires a lot of knowledge how to generate the content. This is where the proposed situated video compositing system comes into play, which aims to interactively integrate dynamic and real life video content into an outdoor AR application. Similar to the concept of Argon or Worldboard[3], users could share self-created content; while offering users a novel approach to experience the augmented content in an immersive way.

2.2 Mobile Augmented Reality Video Applications

Current user interfaces of online video tools mostly replicate the existing photo interfaces. Features such as geo-tagging or browsing geo-referenced content in virtual globe applications such as *Google Earth*[4] (or other map-based applications) have been mainly reproduced for video content.

More recently, efforts have been made to explore further the spatio-temporal aspect of videos. Applications such as *Photo Tourism* [43] have inspired work such as [8], allowing end-users to experience multi-viewpoint events recorded by multiple cameras. The system presented in [8]

[3]http://www.teco.uni-karlsruhe.de/hcscw/sub/115.Kirkley/115.kirkley.html

[4]http://earth.google.com

allows a smooth transition between camera viewpoints and offers a flexible way to browse and create video montages captured from multiple perspectives.

However, these systems limit themselves to produce and explore video content on desktop user interfaces (e.g. web, virtual globe) out of the real context. Augmented Reality technology can overcome this issue, providing a way to place (geo-referenced) video content on a live, spatially registered view of the real world. For example, in [19] investigated situated documentaries and showed how to incorporate video information into a wearable AR system to realize complex narratives in an outdoor environment. Recent commercial AR browsers such as Layar[5] or Wikitude[6] are now integrating this feature, supporting video files or image sequences but with limited spatial registration due to the fact that the video is always screen aligned and registered using GPS and other sensors or indeed 2D markers.

Augmented Video has also been explored for publishing media. RedBull[7] for example, presented an AR application that augmented pages of their *Red Bulletin magazine* with video material using Natural Feature Tracking (NFT). The application was running within a webpage as an *Adobe Flash*[8] application, detecting features on a magazine page and playing the video content spatially overlaid on top of that page.

As these projects generally present the video on a 2D billboard type of representation, other works have been exploring how to provide more seamless mixing between video content and a live video view. In [28] within the *Three Angry Men*[9] project the authors investigated the use of video information as an element for exploiting narratives in Augmented Reality. A system was proposed where a user wearing a Head Mounted Display (HMD) was able to see overlaid video actors virtually seated while discussing around a real table. The augmented video actors were prerecorded and foreground-background segmentation was applied to guarantee a seamless integration into the environment, created with the desktop authoring tool presented in [29] and [30].

Whereas the work in [28] used static camera recording of actors, the 3D Live system [35] extended this concept to 3D video. A cylindrical multi-camera capture system was used, allowing capture and real-time replay of a 3D model of a person using a *shape-from-silhouette* approach.

[5]http://www.layar.com
[6]http://www.wikitude.com
[7]http://www.redbull.com
[8]http://get.adobe.com/flashplayer
[9]http://www.cc.gatech.edu/projects/ael/projects/ThreeAngryMen.html

The system was supporting remote viewing, by transmitting the 3D model via a network and displaying the generated 3D video onto an AR setup at a remote location as part of a teleconference system.

While the mentioned applications were proposed for indoor scenarios, Farrago[10] an application for mobile phones, proposed video mixing with 3D graphical content for outdoor environments. This tool records videos that may be edited afterwards by manually adjusting the position of the virtual 3D object overlays on the video image, yet requires the usage of 2D markers or face tracking. Once the video is re-rendered with the overlay, it can be shared with other users.

The video compositing system presented in this work makes use of a panorama-based tracking mechanism to estimate the user's position and keep track of the user's movements. In contrast to the above mentioned related work there is no need for 2D markers or creation of complex 3D models. Additionally, it overcomes problems of other available tracking mechanisms. For example, using only GPS would not be of sufficient accuracy, due to the occurring jitter in urban environments. With Simultaneous Localization and Mapping (SLAM) technologies the usage of markers or tracking targets is obsolete and tracking of, for example, faces or robot movements [16] works fine in indoor environments. Moreover, tracking a device's position in a previously unknown environment can be achieved by SLAM. In [22] the authors present a mobile implementation of a system based on SLAM. However, the applicability of SLAM in outdoor environments targeting a system like the one presented has not been shown so far.

One of the big advantages of the tracking and mapping approach presented in this work, compared to other available tracking mechanisms, is that it similarly to SLAM works in previously unknown environments AND in contrast to SLAM also in outdoor environments; without additional markers and complex preprocessing. One can start building a panorama on-the-fly and the system continuously tracks the user's position. Which is why the proposed system relies on the *Panorama Mapping and Tracking* algorithm as pointed out later.

[10]http://farragoapp.com/

CHAPTER 3

Technological Foundations

Developing a system like the one presented in this work requires a systematic approach to fulfill every single subtask involved in arriving at the final outcome. Throughout the whole work, conceptual decisions had to be made so that single components could be developed as independently as possible, while ensuring that all parts combined play together in the end. The following chapter justifies the employment of software systems, function libraries and additional tools which were required to conduct the presented work. Moreover, a short overview about what needs to be taken into consideration when developing for *iOS*[1] is given at the end of the chapter.

3.1 Software

To realize this work, two different coding platforms, so called *Integrated Development Environment*(s) (IDEs), were set up. The need for two separate environments arises from the fact that on the one hand the *Studierstube ES* framework has primarily been developed under *Microsoft's*[2] *Windows*[3] operating system and the corresponding IDE called *Visual Studio*[4]. On the other hand the final application is targeted to be deployed to devices such as the *iPhone*[5] or *iPad*[6] running

[1]http://www.apple.com/ios
[2]http://www.microsoft.com
[3]http://windows.microsoft.com
[4]http://www.microsoft.com/visualstudio
[5]http://www.apple.com/iphone
[6]http://www.apple.com/ipad

Apple's[7] *iOS* mobile operating system.

Both the *Foreground Segmentation* and the extension of the *Panorama Mapping and Tracking* algorithm - a component of the *Studierstube ES* framework - were developed using the *Visual Studio* IDE. The extension is a requirement to accomplish the creation of the *reference panorama* (see Fig.5.12) as explained in Section 5.3.2.
As noted above and as can later be observed, he main target platform of the final ARVideo application prototype is the *iOS* mobile operating system. Which is why the final application needs to be built under *MacOS*[8] with its IDE known as *Xcode*[9].

In the following segment more details about the development environment are revealed along with decisions about which additional libraries were chosen for the *Foreground Segmentation* step and the rendering of the augmentation overlays in the context of the *Online Video Replay*. To close this chapter details about the necessary camera calibration tool are given.

3.1.1 Visual Studio

The primary IDE which was worked with throughout this project was *Microsoft*'s *Visual Studio*. The main reason for this decision was that one of the most important components of this work, the *Studierstube ES* framework, has been developed using this very same IDE with *C++*[10] being the core development language. As noted above, one subtask of this work was to extend the *Panorama Mapping and Tracking* technique - a component of the *Studierstube ES* framework, see Fig. 3.3 - such that it gains the ability to generate panoramic images omitting unwanted image details such as foreground objects, as explained in Section 5.3.2. Moreover, as described below in Section 3.1.3, *Visual Studio* simplifies integrating additional function libraries such as *OpenCV*.

In the context of this work a cross-platform development environment was set up together with the *XCode* IDE. Thus the same code-base was used to target the *Windows* desktop platform as well as the mobile *iOS* platform. This was realized by platform dependent macros throughout the code to distinguish between said platforms wherever this was necessary (i.e. including platform dependent libraries such as *OpenGL ES*[11]). A big advantage of this cross-platform approach is to gain the best features out of both platforms or even detect leaks in the application which

[7]http://www.apple.com
[8]http://www.apple.com/osx
[9]https://developer.apple.com/xcode
[10]http://www.cplusplus.com/reference
[11]http://www.khronos.org/opengles

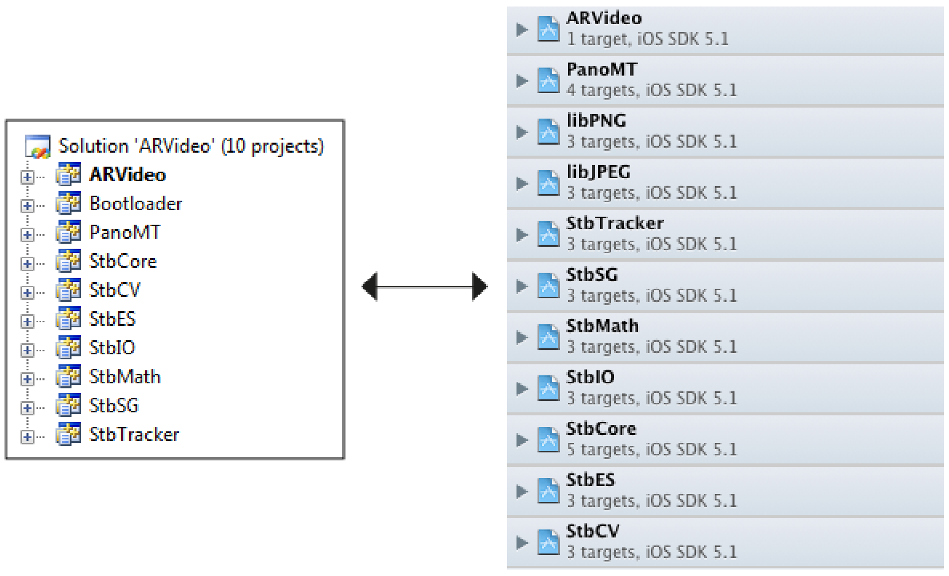


Figure 3.1: Setting up a cross-platform environment. (Left) Excerpt of the Visual Studio environment (Windows Desktop). (Right) Excerpt of the Xcode environment (iOS).

wouldn't have been discovered otherwise. This meant that the main development of the *Online Video Processing* was carried out in the *Visual Studio* IDE targeting a *Windows* executable. Once a certain feature seemed to be working on the desktop it was tried out on the mobile, i.e. *iOS* platform. This double testing might seem cumbersome, yet it really simplifies development and testing in a lot of cases, especially since testing on the mobile device can be much more time-consuming than testing on the desktop platform.

Further, note that with the current described environment, the *Offline Video Processing* is only supported in the desktop environment, as at the moment carrying out this step on the mobile platform is not feasible, yet could be supported in future versions of the *Situated ARVideo Compositing* tool as noted in Chapter 11.

3.1.2 XCode

The final application *Prototype* presented in Section 9 is running on *Apple*'s *iPhone* and *iPad* devices, which is why it was required to set up an environment under *MacOS* using the *XCode* IDE to target the *iOS* platform; besides the *Visual Studio* IDE as explained above in Section 3.1.1. Usually coding for *iOS* means to work with *The Objective-C Programming Language*[12] (short:

[12]http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC

Objective-C) developed by *Apple*. Due to the nature of both *C++* and *Objective-C* being extensions of the standardized *ANSI C* programming language and *XCode*'s built-in compilers for both, one can build an application targeting *iOS* making use of both programming languages. In the context of this work this was realized by creating an *iOS* "application stub" which is used to "load" the real application written in *C++* with the help of the *Studierstube ES* framework. Note that it is still possible to mix both *C++* and *Objective-C* in one class file, such that not only the "application stub" may contain *Objective-C*, but also "normal" classes wherever an explicit distinction of code is necessary. As noted above in Section *3.1.1* testing and the therefore required compiling of a code base like the one used in this project can be very time-consuming. Especially since *Xcode* and *iOS* devices like the *iPhone* seem to perform a lot of caching of files (e.g. image files), which often makes it necessary to completely clean and rebuild (i.e. compile) all relevant framework parts, in order to achieve the expected behavior/changes in the code/files. Which is why as much work as possible was carried out under the *Visual Studio* environment, as mentioned above as well.

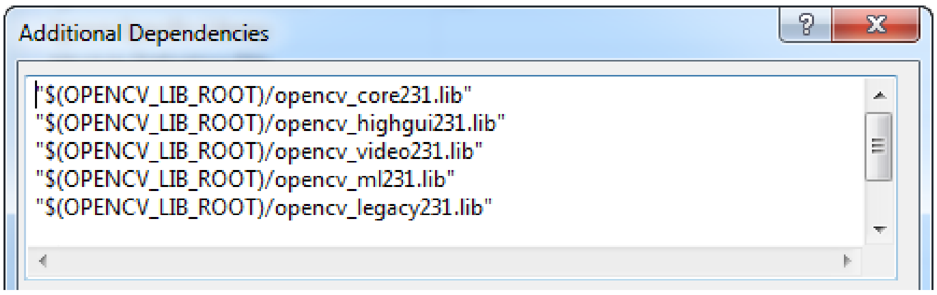
3.1.3 OpenCV

The *OpenCV* library[13] has been chosen for the implementation of the *Foreground Segmentation* step. The reasons for this choice are that it provides a good amount of best-practice computer vision (CV) algorithms, meaning it offers "*a wide range of programming functions for real time computer vision*" [1]. Additionally, it is available as an open-source library, as the name implies, which makes the payment of license fees obsolete. Moreover, it allows the integration of its available libraries into the IDE of choice, namely *Visual Studio*. As explained in Section 5.3.1 the *Foreground Segmentation* step makes use of a combination of CV-algorithms; or in more detail *OpenCV*-implementations of those algorithms; to achieve the task of separating the foreground object and the background information in the recorded video material. Fig. 3.2 indicates how to add dependencies of the *OpenCV* libraries to the *Visual Studio* project properties, so that the desired functions can be called within the project. Which *OpenCV* algorithms are being used is described in detail in Section 5.3.

3.1.4 Studierstube ES

As pointed out earlier, the *Studierstube ES* framework along with its underlying components symbolizes one of the fundamental modules in the realization of this project. *ES* stands for *Embedded Systems* and according to the development group the *Studierstube ES* framework represents "*a radical new way of doing Augmented Reality (AR) on handheld devices*" [4]. It

[13]http://opencv.willowgarage.com/wiki



```
"$(OPENCV_LIB_ROOT)/opencv_core231.lib"
"$(OPENCV_LIB_ROOT)/opencv_highgui231.lib"
"$(OPENCV_LIB_ROOT)/opencv_video231.lib"
"$(OPENCV_LIB_ROOT)/opencv_ml231.lib"
"$(OPENCV_LIB_ROOT)/opencv_legacy231.lib"
```

Figure 3.2: Integration of OpenCV libraries into Visual Studio.

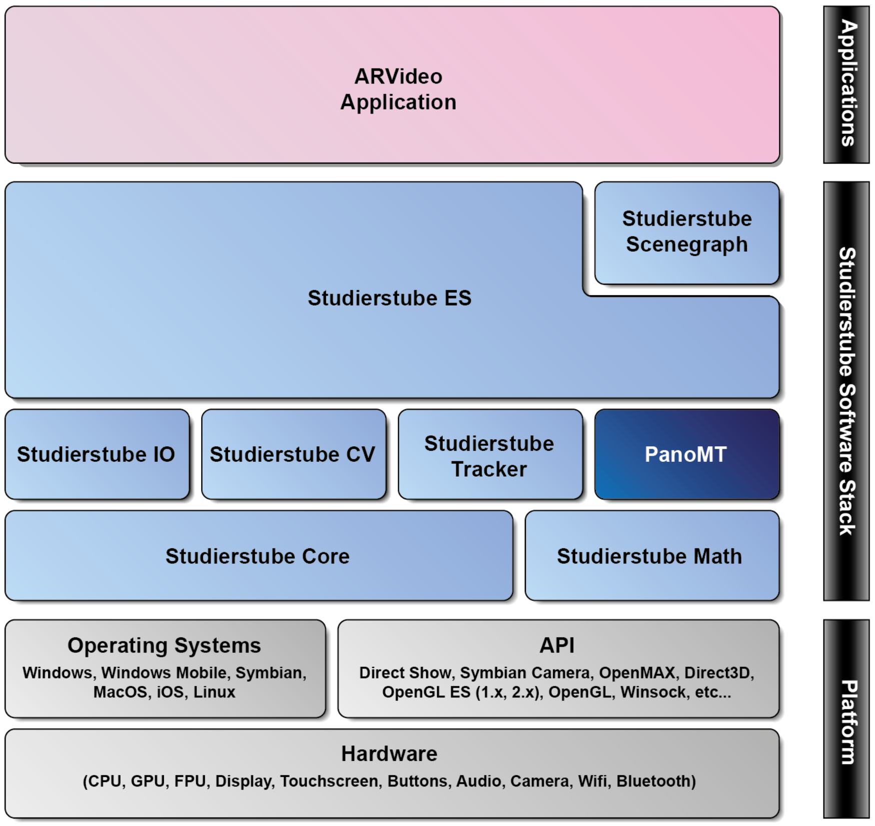

ARVideo Application
Studierstube ES
Studierstube Scenegraph
Studierstube IO
Studierstube CV
Studierstube Tracker
PanoMT
Studierstube Core
Studierstube Math
Operating Systems
Windows, Windows Mobile, Symbian, MacOS, iOS, Linux
API
Direct Show, Symbian Camera, OpenMAX, Direct3D, OpenGL ES (1.x, 2.x), OpenGL, Winsock, etc...
Hardware
(CPU, GPU, FPU, Display, Touchscreen, Buttons, Audio, Camera, Wifi, Bluetooth)
Applications
Studierstube Software Stack
Platform

Figure 3.3: Studierstube ES Overview in the context of this work, adapted from [4].

is further stated that it was written *especially* for supporting applications running on handheld devices (such as PDAs, mobile phones, etc.). Fig. 3.3 illustrates the framework's structure along with all its components and interfaces.

As explained in Chapters 7 and 8 the final *ARVideo Application* stands for a *Studierstube ES Application* sitting on top of the framework as it is illustrated in Fig. 3.3. This means the application is based on said framework and makes use of the single layers and components as depicted in Fig. 3.3. It can be observed that the framework supports a wide range of operating systems (OS) and integration of numerous state-of-the-art Application Programming Interfaces (APIs). As a consequence the framework needs to be set up, i.e. (pre)compiled, accordingly for every different OS. Integrating different APIs and framework features is made possible by providing configuration files, such as eXtensible Markup Language (XML)[14] or header files. Note that the integration of APIs and features is checked at compile- and run-time, respectively, in order to avoid unpredictable behavior. Furthermore, the integration of certain APIs is exclusive, such that, for example, only one renderer (e.g. OpenGL ES, Direct3D) can be compiled and used with the framework.

As can be observed from the overview image in Fig. 3.3 the *Studierstube ES* (*StbES*) component resembles the central and most important component of the framework. This is due to the fact that it contains the basic classes required to compile a *Studierstube ES Application* and while doing so it consolidates all other (pre)compiled components, like *Studierstube Core* and other components, to finally merge or link, respectively, all data which is necessary to execute the application. In the following the remaining components of the *Studierstube Software Stack* are described briefly.

3.1.4.1 Studierstube Core

The *Studierstube Core* (*StbCore*) component defines, as the name suggests, a set of core classes which act as interfaces to the underlying hardware and the OS in use. It is responsible, for example, for setting up the windowing system or allocating memory and additionally contains template classes for common types like *Array* or *Pixel*.

3.1.4.2 Studierstube Scenegraph

The *Studierstube Scenegraph* (*StbSG*) component contains a rich set of classes which can be used to build a scene graph. A scene graph is a hierarchical tree-like structure widely used in CV which usually contains a root node and child nodes, wherein a child node can have child nodes itself, making it the parent node of these child nodes, and so on. Properties or manipulation of parent nodes directly influence the behavior of child nodes, such that, for example, removing a parent node has the consequence of removing all child nodes which belong to the according

[14]http://www.w3.org/XML

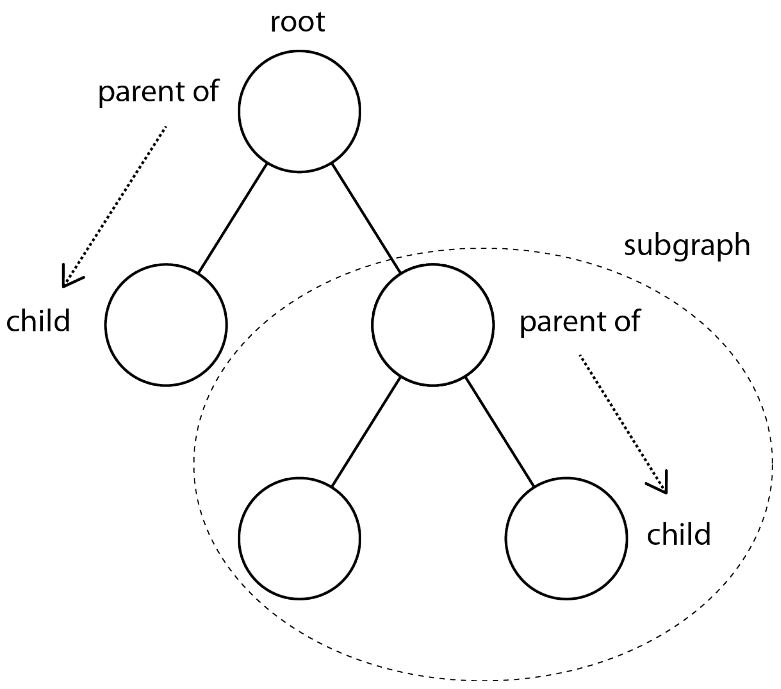


Figure 3.4: Simple scene graph.

parent node. Or in other words, the whole subgraph would be removed. Fig. 3.4 demonstrates the graphical representation of a simple scene graph. By creating and traversing such a scene graph it is possible to control, for example, the graphical appearance of digitally rendered content in a system like the one presented. Hence, the built scene graph within a *Studierstube ES Application* is consulted by the rendering mechanism, in order to control the application's behavior and visual appearance, as for example, shown in Section 5.4.2 and further explained in Chapter 8 by making use of scene graph nodes like *SgTransform* or *SgTexture*. The framework simplifies creating such scene graphs by translating regular XML files and its contained elements to the corresponding scene graph nodes, yet it is still feasible to manipulate the scene graph, i.e. insert/remove nodes, programmatically.

3.1.4.3 Studierstube Math

As the name suggests, the *Studierstube Math* (*StbMath*) component contains classes for mathematical operations. These operations range from simple integer computations (e.g. power of two operations) to more complex computations like creation and transformation of rotation matrices.

3.1.4.4 Studierstube IO

Essentially, the *Studierstube IO* (*StbIO*) component is responsible for reading and writing files from/to the filesystem, which includes textfiles, images, videos and even compressed files, like *zip* files.

3.1.4.5 Studierstube CV

The *Studierstube CV* (*StbCV*) component contains implementations of frequently used/needed CV algorithms, for example, Normalized Cross Correlation (NCC), feature point detection/corner detection (e.g. FAST, Harris Corner), pose estimation, image filtering (e.g. blurring) or image transformation (e.g. warping).

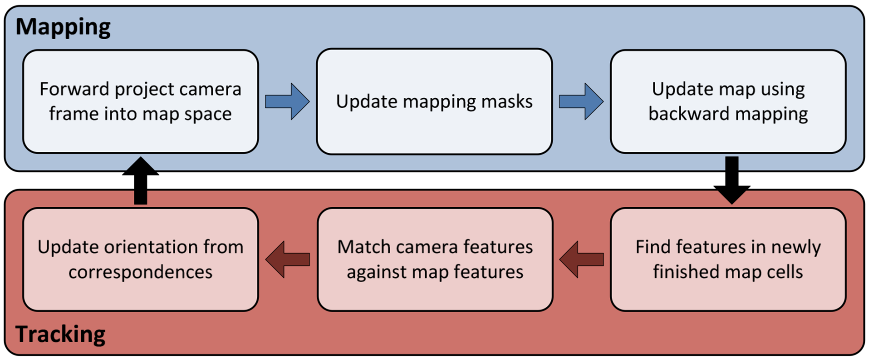
3.1.4.6 Studierstube Tracker

The *Studierstube Tracker* (*StbTracker*) component merely works for marker-based tracking, yet it comes with valuable features such as determining the camera projection matrix (based on the intrinsic camera parameters, which are explained in Section 3.1.7), which consequently helps in projecting camera frames into the panoramic map created with the framework.

3.1.4.7 Panorama Mapping and Tracking

The *Panorama Mapping and Tracking* (*PanoMT*) algorithm; which was first presented in [46]; represents a fundamental basis of this work. As the name of the thesis suggests and as explained throughout it the final application heavily relies on a panoramic map to continuously track and update the user's position within the AR environment.

In Chapter 5 it is explained in detail how the *Panorama Mapping and Tracking* algorithm comes into play; firstly in the *Offline Video Processing* step to create the background reference panorama and secondly in the *Online Video Processing* step to fulfill the registration and keep track of the current camera pose. In order to actually achieve said features, the original functionality of the *Panorama Mapping and Tracking* component is not sufficient as it is not able to process alpha channel information to, for example, omit foreground objects while the panorama is being created (see Fig. 5.11). Furthermore, the localization feature ("registration") does not support matching two panoramic maps against each other, such that the displacement of features is correctly determined and hence the absolute displacement between the two panoramic maps can be determined. Due to the reason that the original implementation does not contain solutions to the two mentioned problems, the *Panorama Mapping and Tracking* algorithm as presented



Mapping
Forward project camera frame into map space
Update mapping masks
Update map using backward mapping
Update orientation from correspondences
Match camera features against map features
Find features in newly finished map cells
Tracking

Figure 3.5: High-level overview of the mapping and tracking pipeline. Taken from [46].

in [46] was extended in the context of this work to provide said solutions. In the following the original functionality of the algorithm is outlined shortly based on [46]; with emphasis on details which are important for the extension of the algorithm, which is given in Chapter 7.

The *Panorama Mapping and Tracking* algorithm represents a novel method for the *"real-time creation and tracking of panoramic maps on mobile phones"* [46]. It uses natural features for tracking and mapping and allows for 3-degree-of-freedom (3-DOF) tracking in outdoor scenarios, while retaining robust and efficient tracking results. Usually the panoramic map is created in real-time from the live camera stream by mapping the first frame completely into the map and extending it by only those areas which are not covered yet in the map for successive frames. Fig. 3.5 illustrates the basic mapping and tracking pipeline. It can be observed that the system operates in a cyclical process, such that the tracking depends on an existing map to estimate the orientation, yet the mapping depends on a previously determined orientation to update the map. Hence, at start-up a known orientation with a sufficient number of natural features must be used to initialize the map.

1. Panoramic Mapping.

As pointed out in [46] a cylindrical map is chosen to create a map of the environment. The main reason for choosing a cylindrical representation is that *"it can be trivially unwrapped to a single texture with a single discontinuity on the left and right borders"* [46].

- **Organization of the map -** The map is organized into a grid of 32x8 cells (see Fig. 3.6) which simplifies processing the unfinished map, as every cell can have either of two states: finished (completely filled) or unfinished (empty or partially filled) and is processed ac-

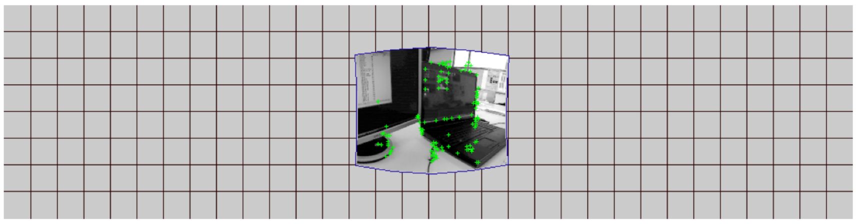


Figure 3.6: Grid of cells compositing the map, after the first frame has been projected. The green dots mark keypoints used for tracking. Taken from [46].

cordingly when new data comes in. Keypoints (green dots in Fig. 3.6) are extracted only for cells which are finished.

- **Projecting from camera into map space -** As mentioned in Chapter 10, only rotational movements are assumed while creating a panorama, which is an acceptable constraint as pointed out in [46] citing [12]. This means a "fixed" camera position is assumed which still leaves 3-DOF regarding rotational movements while projecting camera frames onto the mapping cylinder. To project a camera frame into map space single pixels are processed similarly to the overlay pixel positions as explained in Section 5.4.2. By forward mapping the camera frame into map space the current camera image's area on the cylinder shall be estimated. First, a single pixel's device coordinate is transformed into an ideal coordinate by multiplying it with the inverse of the camera matrix (and removing radial distortion using an intern function). The resulting 2D coordinate is unprojected into a 3D ray by introducing a depth coordinate. Rotating from map space into object space is done by applying the inverse of the current camera rotation. To get the pixel's 3D position on the mapping surface the ray is intersected with the cylinder, which then is converted into the final 2D position. For further details, see [46].

- **Filling the map with pixels -** Due to the chance that using *forward mapping* to fill the map could cause holes or overdrawing of pixels, *backward mapping* is used, which basically works in the reverse way as the above mentioned forward mapping, such that starting from a known 3D position on the cylinder (e.g. within the determined area of the current camera image) the device coordinate and its corresponding pixel color are determined. For further details again refer to [46].

- **Speeding up the mapping process -** Due to the high number of pixels which would have to be processed for every camera image (e.g. >75000 pixels for a 320x240 pixels sized image) a sophisticated approach to drastically reduce the amount of pixels was introduced.

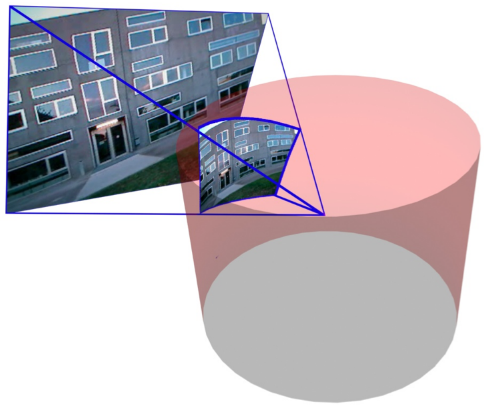


Figure 3.7: Projection of the camera image onto the cylindrical map. Taken from [46].

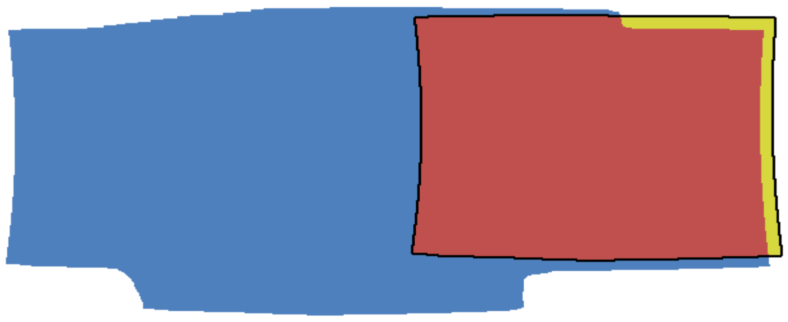


Figure 3.8: Mapping of new pixels. (Blue) Pixels which have been mapped so far. (Black border) Outline of the current camera image. (Red) Intersection of already mapped pixels and the current camera image. (Yellow) Pixels which still need to be mapped. Taken from [46].

It works by mapping the first frame of the panorama completely into the map, while only mapping previously unmapped portions for consecutive frames. This is realized by quickly filtering out those pixels which have been mapped before, by using zero or more spans per row which define which pixels of the row are mapped and which are not. A span encodes only the left and right coordinate of a continuous characteristic, such as mapped pixels, which is why processing said spans is highly efficient. Filtering out already mapped pixels is done by comparing "finished" and "yet to map" spans by a boolean operation, which yields only those pixels which have not been mapped yet. Consequently, only a small portion of pixels have to be fully processed and mapped, which is illustrated in Fig. 3.8.

2. Panoramic Tracking. As depicted in Fig. 3.5, and noted above, the mapping process depends on an estimate of the current camera orientation. How this is accomplished is described briefly in the following:

- **Keypoint extraction and tracking + orientation update/relocalization -** As soon as a grid cell is finished the FAST (Features from Accelerated Segment Test) corner detector algorithm [38], [39] (contained in *StbCV*) is applied to it, finding keypoints as illustrated in Fig. 3.6. By using certain thresholds within the algorithm it is assured that, for every cell, enough keypoints are identified. Organizing the identified keypoints cell-wise simplifies matching these points while tracking. Tracked keypoints are the basis for estimating the current camera orientation (based on the initial orientation and using a motion model with constant velocity) while mapping the current camera image. In order to match keypoints from the current camera image against their counterpart in the map, NCC (also in *StbCV*) is used. Similarly to tracking keypoints from one frame to the other a relocalization feature was implemented, in case tracking failed/couldn't be updated correctly, which also uses NCC. For further details once again see [46].

- **Initialization from an existing map (registration) -** In order to accomplish a registration feature like the one described in Section 5.4.1 it is necessary to load an existing map into memory and then guess the current camera orientation with respect to the loaded map. In this case this is achieved by extracting features from both the live camera image and the loaded map and determining pairs of features to match, in order to speculate on the orientation, wherein one pair of matching features resembles a"hypothesis" of the estimated orientation which then needs to be supported by other pairs of matching features. If the amount of features which support the hypothesis is satisfactory, the calculated orientation is further refined by using a non-linear refinement process, as described in [46].

Chapter 7 outlines how the above presented *Panorama Mapping and Tracking* system was extended to suit the requirements of this work. It is shown how certain areas which should not be mapped within the current frame can be skipped while mapping new pixels; by adapting the above mentioned span-approach. Furthermore, it is shown how the registration process can be realized making use of two panoramic images instead of only one panoramic image and single live camera images.

3.1.5 OpenGL ES

OpenGL ES is an API for supporting graphics rendering in embedded systems - "*including consoles, phones, appliances and vehicles*" [2]. It is a well-defined subset of desktop *OpenGL*[15], "*creating a flexible and powerful low-level interface between software and graphics acceleration*" [2].
In the context of the presented work *OpenGL ES* is in charge of rendering all graphical content on the display, including the augmentation overlays, see Page 46 and Section 5.4.2, respectively. As apparent from Fig. 3.3 *OpenGL ES* is integrated into the *Studierstube ES* framework, which simplifies its use through an application based on the said framework. Due to the cross-platform environment, described on Pages 14 and 15, distinguishing between desktop and mobile compatible versions (1.1 and 2.0, respectively) of *OpenGL ES* was required. Making use of the *Studierstube ES* framework, one must distinguish which version to integrate into the application before compiling the whole application. This can be accomplished by simply activating/deactivating versions in a configuration file.

3.1.6 Qt Framework

In order to simplify the usage of the *Foreground Segmentation* tool, presented in Section 5.3.1, an application user interface (UI) was developed using the *Qt* framwork. According to the official website, *Qt is a cross-platform application and UI framework with APIs for C++ programming*" [3]. *Qt* was originally developed by the norwegian company *Trolltech*, which was acquired by Nokia[16] in January 2008[17]. The reasons for choosing the *Qt* framework are that it is open-source[18] and it can be integrated into the IDE of choice, namely *Visual Studio* (see Section 3.1.1), which was used for developing the mentioned *Foreground Segmentation* tool. The integration of *Qt* into *Visual Studio* is achieved by installing the *Qt Visual Studio Add-in*[19]. Once the plugin is installed *Qt* functionality (e.g. the interface designer) can be chosen from the context menu, as illustrated in Fig. 3.9. Note that it is still required to include the corresponding *Qt dll*s (e.g. *QtCore.dll* or *QtGui.dll*), either by directly packing them into the created executable, or by shipping the *dll*s along with the application.

[15]http://www.opengl.org
[16]http://www.nokia.com
[17]http://qt.nokia.com/about/news/archive/press.2008-01-28.4605718236
[18]http://qt.nokia.com/about/news/lgpl-license-option-added-to-qt
[19]http://qt.nokia.com/downloads/visual-studio-add-in

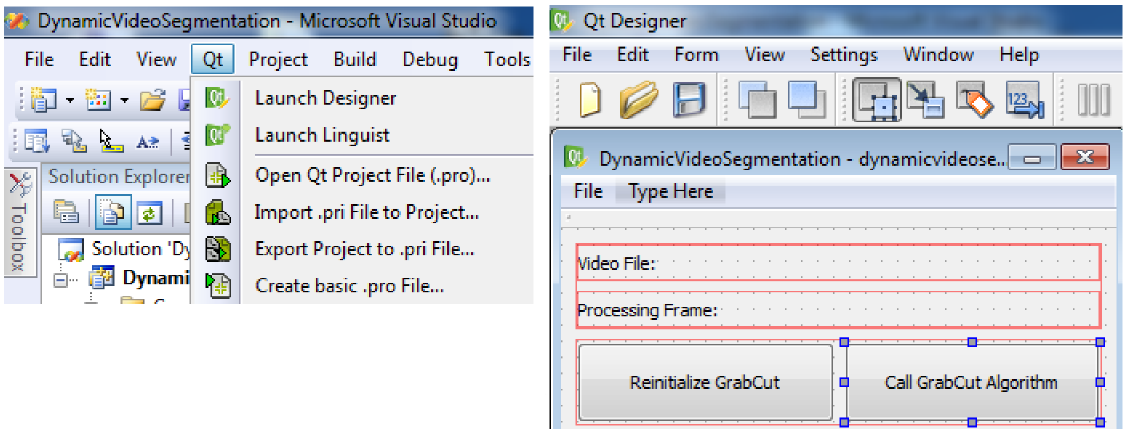

DynamicVideoSegmentation - Microsoft Visual Studio
File Edit View Qt Project Build Debug Tools
Launch Designer
Launch Linguist
Open Qt Project File (.pro)...
Import .pri File to Project...
Export Project to .pri File...
Create basic .pro File...
Solution Explorer
Toolbox
Qt Designer
File Edit Form View Settings Window Help
DynamicVideoSegmentation - dynamicvideose...
File Type Here
Video File:
Processing Frame:
Reinitialize GrabCut
Call GrabCut Algorithm

Figure 3.9: (Left) Added Qt functionality in Visual Studio. (Right) Excerpt of the Foreground Segmentation User Interface in Qt Designer.

3.1.7 Camera Calibration

The camera calibration is a task of crucial importance for the whole *Situated ARVideo Compositing* tool to work properly. Calibrating the camera means to determine the intrinsic camera parameters of the source/target device which are used to record the video scene (see Section 5.2.1) and/or to replay the augmented video material (see Section 5.4.2), respectively. Hence, the camera calibration can be seen as a preliminary task which needs to be carried out before any of the further processing in Chapter 5 may take place. Note that usually it is sufficient to perform the camera calibration task once for a certain type of device (e.g. *iPhone 3GS*) and whenever a device of the same type is being used a corresponding camera calibration file with previously determined parameters may be made use of.

The correctly determined camera parameters need to be considered amongst other things when the *reference panorama* is being created out of the background information taken from the video source material, as explained in detail in Section 5.3.2. Likewise, without a proper camera calibration updating the augmentation overlays while replaying the video content (see Section 5.4.2) would not succeed as the updated overlay position is computed with help of the camera parameters.

As usual in CV to compute the intrinsic camera parameters of a device it is necessary to make use of a calibration pattern (chessboard-alike pattern, as shown in Fig. 3.10). This pattern is shot from different angles and distances with the camera which should be calibrated, such that an adequate amount of images can be used for the calibration process. The amount of pictures which should be taken depends on the tool used for the calibration, yet normally the more images are available the better (i.e. more accurate) the computed results.

In the context of this work the *GML Camera Calibration Toolbox*[20] was utilized to achieve the goal of computing the intrinsic camera parameters. Fig. 3.10 illustrates the toolbox user interface and Fig. 3.11 shows an exemplary calibration result. The parameters which are of interest are the *focal length*, the *principle point* and the *distortion* values. All these values are finally written to the camera calibration file which will later be used by the *Studierstube ES* framework. As can be observed from Fig. 3.11 the focal length and principle point values are used to build the camera matrix. In a more general way this means the following camera matrix can be derived from the calibration results, with f_x,f_y standing for the focal length values in horizontal and vertical direction and c_x, c_y standing for the principle point values in horizontal and vertical direction, respectively:

$$cam = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

The hereby derived camera matrix cam is not only responsible for mapping pixels from the real 3D world onto the 2D image plane and therefore into the panorama created in 5.3.2. It also helps in calculating the overlay's position while the augmentations, which are the basis for the ARVideo rendering, are being updated (see Section 5.4.2).

3.2 Hardware

As mentioned throughout this work *iOS* represents the target platform for the final ARVideo application. Therefore, in order to actually develop the application several *iOS* devices were used for testing:

- iPhone 3GS
- iPhone 4S
- iPad 2

[20]graphics.cs.msu.ru/en/science/research/calibration/cpp

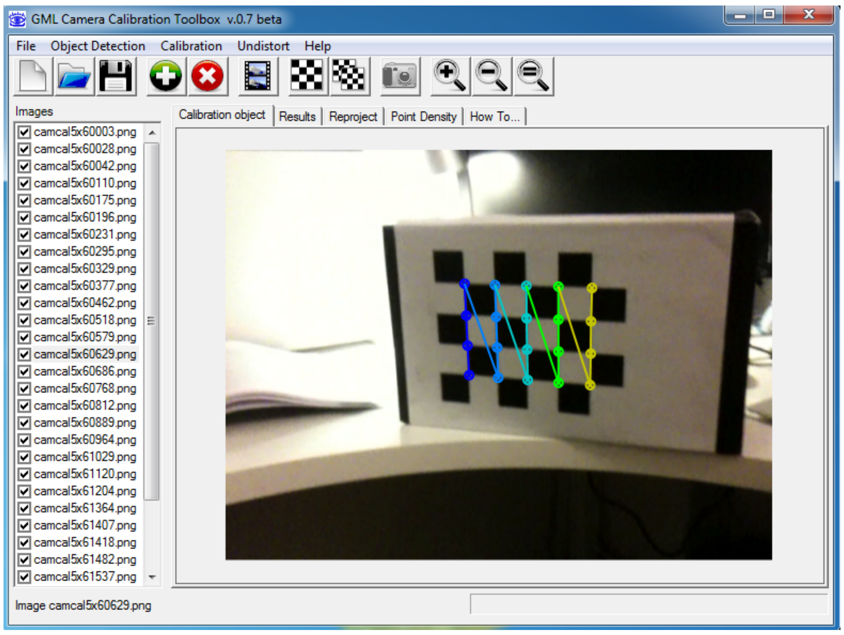
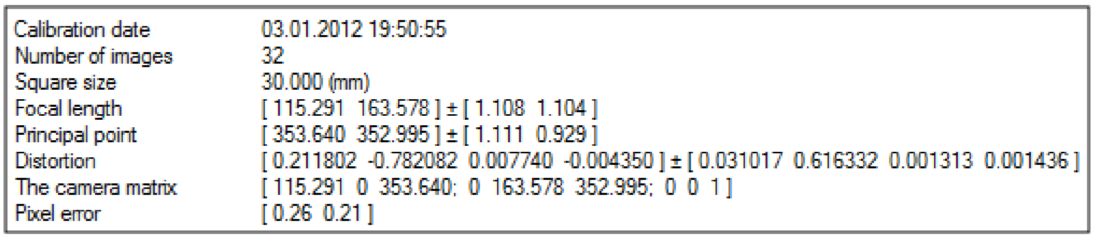


Figure 3.10: Camera calibration tool used to determine the intrinsic camera parameters.



Calibration date	03.01.2012 19:50:55
Number of images	32
Square size	30.000 (mm)
Focal length	[115.291 163.578] ± [1.108 1.104]
Principal point	[353.640 352.995] ± [1.111 0.929]
Distortion	[0.211802 -0.782082 0.007740 -0.004350] ± [0.031017 0.616332 0.001313 0.001436]
The camera matrix	[115.291 0 353.640; 0 163.578 352.995; 0 0 1]
Pixel error	[0.26 0.21]

Figure 3.11: Camera calibration results.

The *iPhone 3GS* served as the main development device and was used along with the *iPad 2* in the conducted *User Study*, presented on Page 93. The *iPhone 4S* was not available throughout the whole project and was therefore primarily used for comparing features such as the frame rate of the running application, visual appearance of the overlays, etc. More details about the differences between said devices are given in Chapter 11.

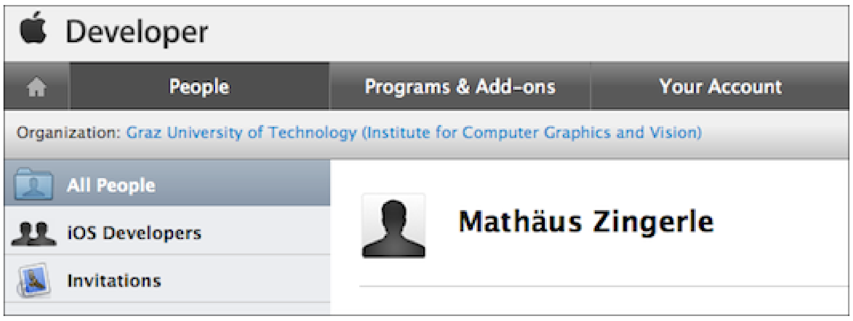


Figure 3.12: Apple Developer Account.

3.3 Miscellaneous

3.3.1 Apple Developer Account

Due to the *Apple* presets it is necessary to be part of the *Apple Developer Program*[21] to be actually able to deploy *iOS* applications to devices such as the *iPhone* or the *iPad*. Registering for a developer account includes paying an annual license fee. There are several "options" to become a member, including registering within/as an organization or becoming a member of a *Developer University Program*, whereas the latter was the case for this project. Fig. 3.12 shows the developer profile which was registered in the context of this work.

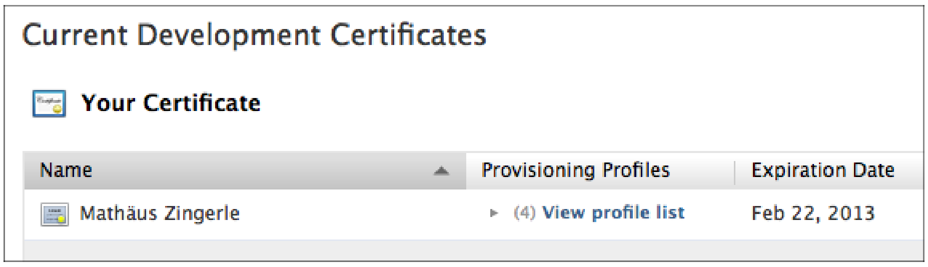
3.3.2 iOS Provisioning Portal

The *iOS Provisioning Portal* is the central location to manage one's developed *iOS* applications. All information about apps, registered devices and other necessary information can be found/maintained there. Access to the portal is granted along with a valid *Apple Developer Account*. Below a short description about the usage/requirements regarding the *iOS Provisioning Portal* is given, in order to be able to test and run one's developed apps on a real device.

3.3.2.1 Development Certificate

Development Certificates are used by Apple to "identify" a developer, such that for the creation of a certain *Provisioning Profile* a valid *Development Certificate* must be chosen. Fig. 3.13 illustrates an overview of the created *Development Certificate* which was used throughout this work. The created *Development Certificate* needs to be downloaded from the *iOS Provisioning Portal* and afterwards imported to *XCode* such that the IDE is able to check for the validity of the imported certificate at the time of compilation.

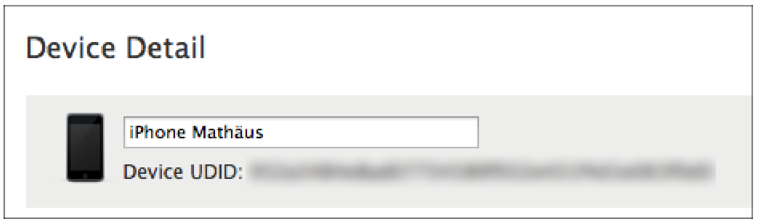
[21] https://developer.apple.com/


Current Development Certificates

Your Certificate

Name	Provisioning Profiles	Expiration Date
Mathäus Zingerle	(4) View profile list	Feb 22, 2013

Figure 3.13: Development Certificate.


Device Detail

iPhone Mathäus

Device UDID: [redacted]

Figure 3.14: Registered Device.

3.3.2.2 Device

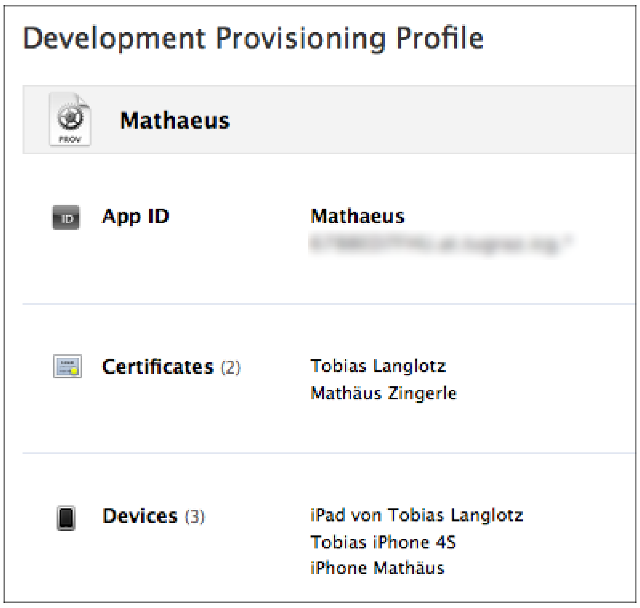
In order to deploy *iOS* apps (even for test purposes) to Apple devices it is required to register said devices in the *iOS Provisioning Portal*. This is done by specifying the unique device ID, named *UDID*. See Fig. 3.14 for one of the devices which was registered for development in the context of this work. The need for registering a device limits the deployment of apps to only those devices which are connected to an *Apple Developer Account* as the one mentioned above.

3.3.2.3 App ID

Apple's way of identifying an *iOS* app is accomplished by the utilization of *App ID*s. These are unique IDs which can be freely chosen, whereas usually the app name is taken with a prefix such as the company name. Later this *App ID* must be referred to in the app build settings, otherwise compilation/deployment of the app will fail.

3.3.2.4 Provisioning Profile

Once the *Development Certificate* has been created, the registration of the *Device* and the specification of the *App ID* are completed it is necessary to generate a *Provisioning Profile*. Provisioning Profiles contain all the aforementioned information and are used to verify an application upon installing/deploying it. This means that in order to distribute the app for testing



Development Provisioning Profile

Mathaeus

App ID	**Mathaeus** [redacted]
Certificates (2)	Tobias Langlotz Mathäus Zingerle
Devices (3)	iPad von Tobias Langlotz Tobias iPhone 4S iPhone Mathäus

Figure 3.15: Provisioning Profile.

purposes, i.e. install your application on a device, it must be preceded by the installation of the according *Provisioning Profile*, otherwise installation will abort/fail. Note that one *Provisioning Profile* can be used to target only one *App ID*, whereas it can target several *Development Certificate*s and *Device*s. More information about developing for *iOS* can be found at the *Apple DevCenter*[22].

[22]https://developer.apple.com/devcenter/ios

Part II

Design + Implementation

CHAPTER 4

Concept - Overview Of The System

With all theoretical and technological preconditions clarified in the previous chapters the elaborated *Situated ARVideo Compositing System* shall now be introduced. The system's main components, arranged into activities, are presented with the help of an Unified Modelling Language[1] (UML) activity diagram, whereas in Chapter 5 the functional design of the system will be presented by describing all parts in detail, while explaining how these parts work together. Chapters 6 to 8 contain the single components' detailed implementation along with UML class diagrams. Additionally, Chapter 9 introduces a prototype of what could be one of the first "sensor-less" outdoor AR video browsers.

4.1 Activity Diagram

Fig. 4.1 depicts the system's main components, arranged into activities and subactivities. The whole process from start to end is seen as the main activity *Situated ARVideo Compositing* which contains three subactivities in order to achieve the desired result of replaying past events in an outdoor environment on a mobile device. As can be observed these three activities are:

1. Content Creation: The recording of an outdoor video scene and the transferring of the video are part of this activity. The outcome shall be the video source material which represents the input for the next activity.

2. Offline Video Processing: To prepare all resources for replaying the original video scene the *Offline Video Processing* comes into play. It takes as input the video material recorded

[1] http://www.uml.org

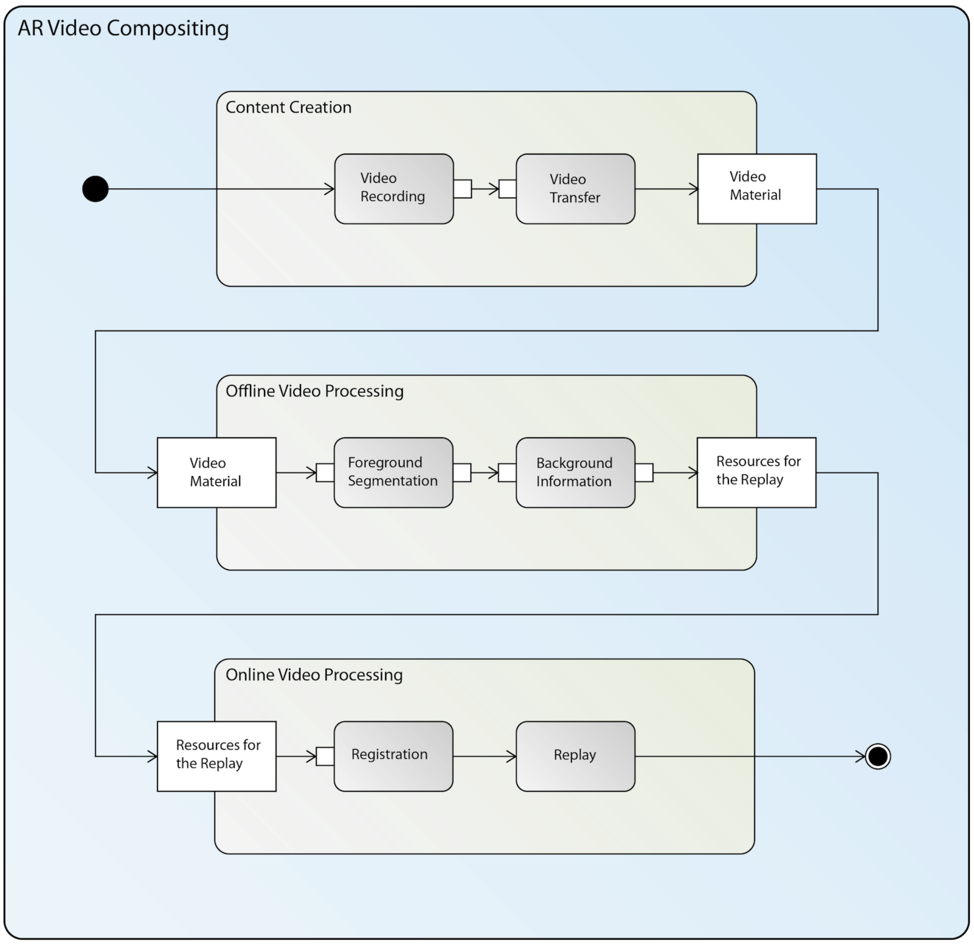


Figure 4.1: Activity Diagram of the Situated ARVideo Compositing System.

in the activity *Content Creation* and processes it by applying the *Foreground Segmentation*. Then, based on the outcome of this subactivity the processing of the *Background Information* is done. Combining all output of the *Offline Video Processing* leads to the *Online Video Processing.*

3. Online Video Processing: The final (sub)activity, which relies on the resources prepared in the above mentioned (sub)activities, is to carry out the registration in the outdoor context and, assuming this was successful, embed and replay the video augmentations in the live camera view of the user's device.

In the following chapters all activities are explained in detail regarding their functional design and the corresponding implementations.

CHAPTER 5

Situated ARVideo Compositing

The purpose of this chapter is to explain the presented system in such a way that each section contains the functional design of the activities and their sub-activities given in Fig. 4.1 in order to help understanding the proposed algorithm.

5.1 Terminology

To actually be able to understand all details and to prevent misunderstandings a few terms and their meanings within this work shall be given in the following listing:

- smartphone, mobile device, mobile, target device: These words are used interchangeably and resemble the device the final application runs on, such as the *iPhone*.
- video frame, video image, frame: These words are used interchangeably and resemble a single image of a previously recorded video, which usually contains about 25 - 30 frames per second.
- camera frame, camera image, frame: These words are used interchangeably and resemble a single live image which gets forwarded by the camera of the mobile device to the application. Note that when only the word *frame* is used it should be clear out of the context if it stands for a *video* or *camera* frame.
- program, application, executable: Usually these words stand for an executable piece of software, which either represents only a part or component of a bigger system or indeed stands for the final executable outcome.

- system: Throughout this work the word *system* basically denotes the combination of software and other tasks which are fulfilled to accomplish the final outcome.
- algorithm, technique: These words stand for the theoretical foundations of a concrete piece of software or might even be used instead of e.g. application.
- method, function: These words are used interchangeably and usually resemble a number of programming statements which can be called within an application.
- library, function library, DLLs, API: As common in programming these terms usually provide interfaces to or concrete implementations of functions which can then be used within the own application.
- vector: Stands for a programming structure which usually contains a list of items of the same type.
- alpha mask, alpha channel, alpha image: These words are used interchangeably and resemble a monochrome image which usually is considered to handle transparency in images, such that the color value of a single pixel encodes the opacity of this pixel in the final image, i.e. when the alpha channel is combined with a standard RGB image.
- skateboard trick: Common and general name for a “jump” performed by a skateboarder.

5.2 Content Creation

5.2.1 Video Recording

The source material which the proposed ARVideo technique relies on is, of course, some video material. This video material may be captured by making use of standard devices such as a smartphone or a digital camera.

In the context of this work the depicted scenario is similar to this one: a person/object is moving in a public area and somebody else captures the scene from a static point of view with a camera within a distance of a few meters. This scene could involve sportspersons performing stunts, cars or other artificial objects or indeed just pedestrians walking by. See Fig. 5.1 for an exemplary situation. Generally, a (translational) movement of the object of interest (i.e. the sportsperson) is assumed, such that usually the recording will not be static. This means that the person recording the video has to rotate the recording device to fully capture the scene, i.e. the object of interest stays within the video frame. Therefore, the created content will usually contain two dynamic aspects:

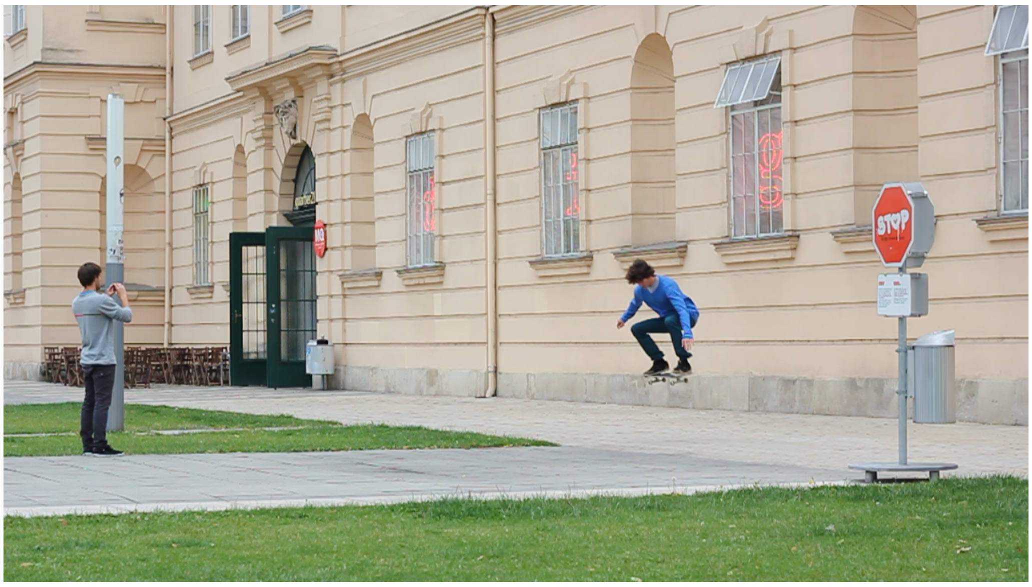


Figure 5.1: Person on the left shoots a video with a smartphone of the person on the right performing a skateboard trick.

1. the moving object (i.e. the sportsperson, car, ...)

2. and the rotational movement of the camera itself.

In Fig. 5.2 a few sample frames are shown which are taken from the video that was captured in the depicted scenario in Fig. 5.1.

5.2.2 Video Transfer

Once the recording of the video has been finished it needs to be transferred to a personal computer for further processing. In the current implementation transferring the video is only realized in an offline manner, i.e. plug in a cable and transfer manually. In the development of the proposed system this was seen as sufficient, yet it shall be noted that in future versions this step may be substituted by uploading the recorded video to a web server for further processing instead of transferring it to a desktop PC or similar. Assuming the transfer of the source video was successful the next step in the workflow of the system - the *Offline Video Processing* - may be executed.

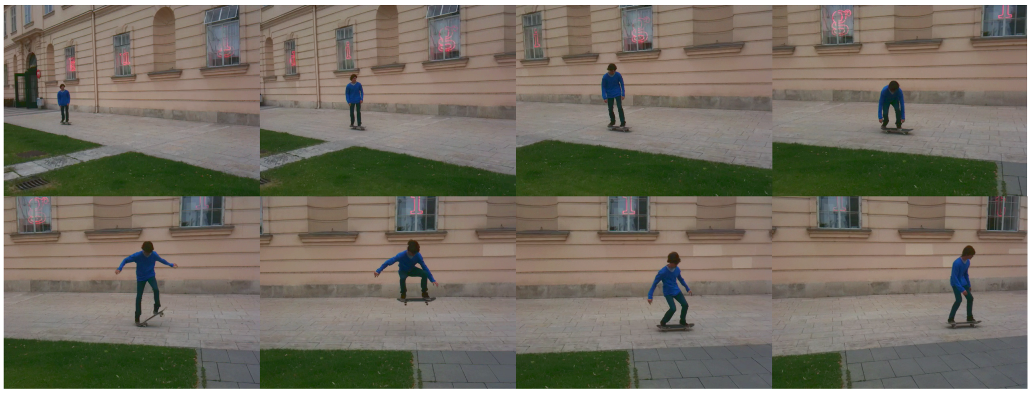


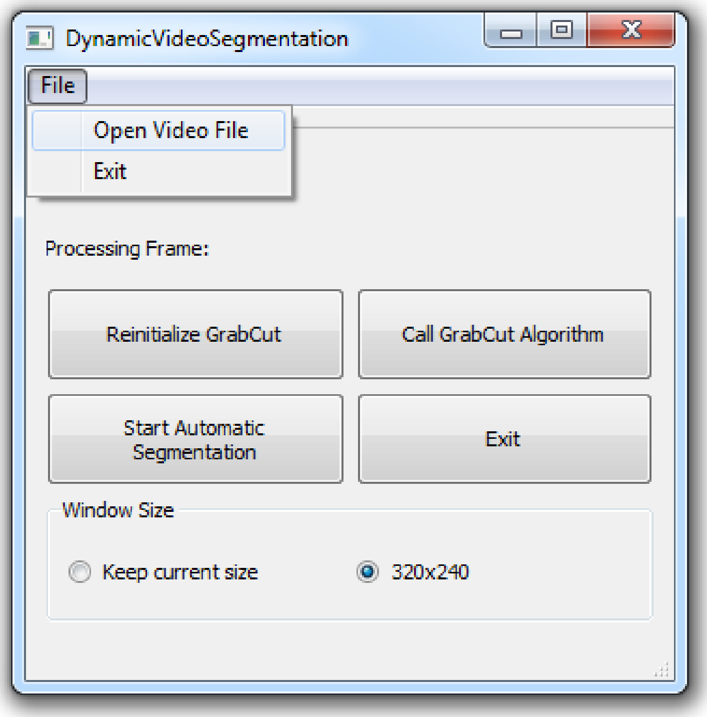
Figure 5.2: Sample frames illustrating a dynamic video scene - recorded with a smartphone.

5.3 Offline Video Processing

Once the video shot in step 5.2 has been transferred to a personal computer the *Offline Video Processing* can be applied to the source material. The focus of this step lies on the extraction of all relevant information which is needed to be able to correctly replay the composited video within the desired context. The main challenge here is to separate the object of interest (foreground) in every video frame from the remaining information such as the background or other moving objects that are not of interest. As can be seen in Fig. 5.2 suitable source material for the ARVideo application contains a moving object (e.g. a skateboarder). Furthermore, it is noticeable that the camera itself is being rotated from left to right during the recording. Due to the highly dynamic aspect of the source material the segmentation of the foreground object requires a sophisticated approach to correctly separate it from the remaining information. The reason why the segmentation is necessary is that the segmented foreground object will later be used as the augmentation overlay in the live camera view. In addition, the background information is not just discarded since it is essential for the creation of the *reference panorama*, which is used to register (i.e. localize) the user in the new context while utilizing the ARVideo application. To fulfill the depicted segmentation task a standalone *Windows* program was developed combining best-practice CV algorithms. In this case this was accomplished using the *OpenCV* library. The reasons for choosing *OpenCV* were given in Section 3.1.3. In the following the segmentation process is described in detail.

5.3.1 Foreground Segmentation

To begin the segmentation process the Windows executable is started up and a video file is opened, see Fig. 5.3. The program works in a way that it processes one frame after another,



Window Size: Keep current size [✓ 320x240]

Figure 5.3: Foreground Segmentation Main Window - opening a video file.

whereas usually only in the first frame user interaction is required. To simplify the user interaction with the video frames an option to open the video file in its full resolution is given. By default, incoming video frames will be resized to 320x240 pixels to minimize the program execution time. Note that this also happens to be the resolution the current *Panorama Matching and Tracking System* works best with.

5.3.1.1 Manual initialization

Once the video file has been opened the user is asked to initialize the foreground-background segmentation of the first frame. First a bounding rectangle is drawn by the user to tell the program in which area of the frame the object of interest can be found at in the beginning. Or in other words, the bounding rectangle defines a - according to the OpenCV documentation - *"region of interest(ROI) containing a segmented object. The pixels outside of the ROI are marked as obvious background"* [1].
Apart from that, it is necessary to roughly sketch the foreground object and mark parts of the background. By doing so the segmentation algorithm (GrabCut) assumes that these pixels be-

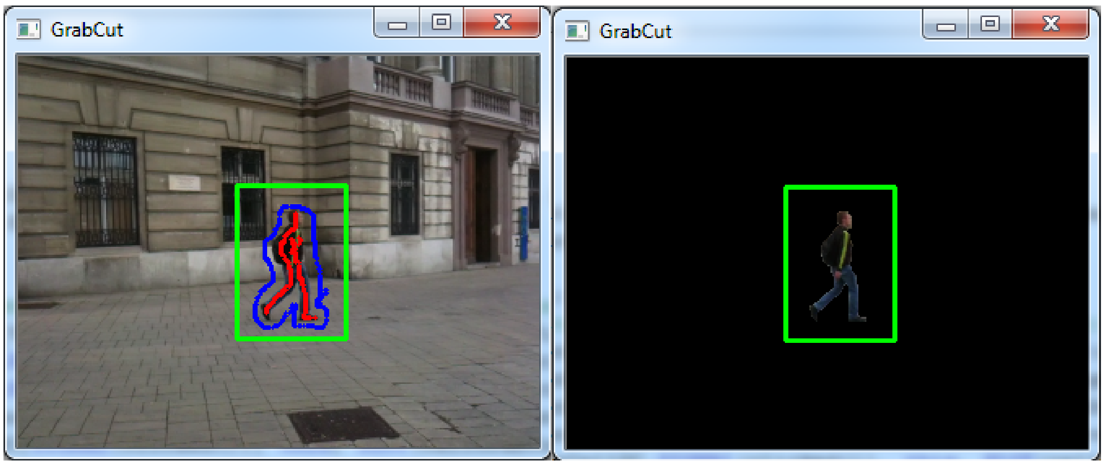


Figure 5.4: (Left) Manual initialization of the segmentation step. User sketches the foreground object (red) and outlines the background (blue). (Right) GrabCut Result. Applying the GrabCut Algorithm yields the segmented foreground object for the initial frame.

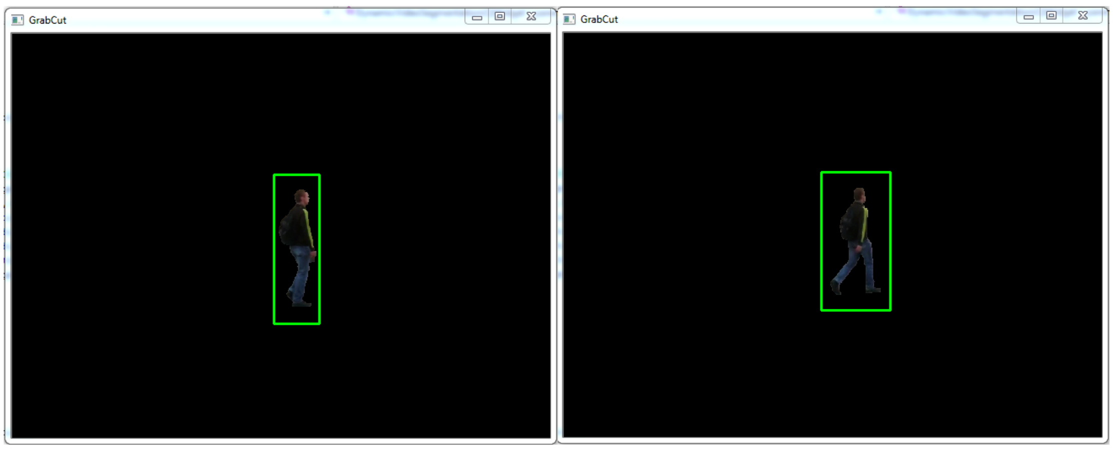


Figure 5.5: Results for calling the GrabCut algorithm on subsequent video frames. Tracking the segment allows segmentation of subsequent frames even in case the appearance changes.

long to the foreground and background, respectively. Based on this assumption the algorithm is able to classify all remaining neighboring pixels - those which are not marked in the beginning - as (probable) foreground or background pixels. Hence, the algorithm is able to completely separate the foreground from the background. The variance of the GrabCut Algorithm used here is described in more detail in Section *GrabCut call*. Fig. 5.4 exemplifies the manual initialization of the GrabCut algorithm and the segmentation result for the first frame of the video.

After the manual call of the GrabCut algorithm the program continues to automatically segment the rest of the video frames. It is not possible to just simply take the previously segmented object and set it as input for consecutive calls. As pointed out earlier the source material is likely to be very dynamic in terms of appearance of the object of interest and secondly the camera movement. Which is why additional processing is required to automatically segment the object of interest in all remaining frames, here labeled as *Pre-GrabCut-processing* and *Post-GrabCut-processing*. As soon as the initial GrabCut call has finished the user can trigger the automatic segmentation by clicking the button *Start Automatic Segmentation*, see Fig. 5.3. When this button is pressed the program enters a loop state which processes all remaining frames in the same way such that first the *Pre-GrabCut-processing* is applied, then the GrabCut algorithm is called again and ultimately the *Post-GrabCut-processing* is carried out, which means that the program performs these three steps in a loop until the end of the video has been reached:

1. Pre-GrabCut-processing
2. GrabCut call
3. Post-GrabCut-processing

5.3.1.2 Pre-GrabCut-processing

The purpose of the *Pre-GrabCut-processing* is to estimate an adequate approximation of the object's position in the current frame and consequentially provide foreground and background pixels (the input) to the GrabCut algorithm; which eventually will be performed on the current frame. To achieve this several subtasks need to be fulfilled, which are combined so that they depend on each others' results. The basic idea is to calculate the optical flow of a sparse feature set from the previous frame to the current one and also to find object contours in the current frame in order to be able to match the results of both algorithms to get an estimation of input pixels for the GrabCut algorithm. To begin, a feature tracking algorithm is ran to track the foreground and background pixels from the previously segmented frame to the current (not yet segmented) frame. The feature tracker in use here is OpenCV's implementation of the *Lukas-Kanade Feature Tracker* [9], whereas any other good feature tracker could be used instead (e.g. the *Farneback* algorithm [13] was also reviewed in context of this work).

Determining the object contours in the currently processed frame works by applying OpenCV's *Canny Edge Detector* algorithm and *findContours* algorithm on the detected edges - hence finding a limited number of object contours. The tracked features and the found contours on their own are not enough to segment the desired object in the frame, yet it is possible to use the acquired information to set the input for the subsequent GrabCut execution. It was observed that the GrabCut algorithm is prone to (false positive) errors if the input pixels (distinguishing foreground and background) are not accurate enough; therefore, it is insufficient to just pass the tracked features to it. Although this may yield satisfactory results in some cases, it still leads to a high number of segmentation errors in most cases. Thus, it is crucial to limit the amount of pixels to those which are very likely to be classified as foreground and background, respectively.

The limitation to only suitable input pixels is accomplished by matching the output of the two previous steps, such that the tracked features are matched pixel-wise against the object contours, consequently limiting the number of pixels which are used as input for the GrabCut algorithm. As mentioned the pixels are matched and therefore limited in order to get rid of outliers, which may have been wrongly tracked by the feature tracker. Furthermore, an assumption is made that only pixels which are part of the same contour may either belong to the foreground or background. Which is why the matching of the tracked features and the found contours also discards features which do not belong to the same contour, hence further limiting the amount of pixels which are passed as input to the GrabCut algorithm.

5.3.1.3 GrabCut call

In order to automatically segment the desired foreground object in the current frame the segmentation approach makes use of the GrabCut algorithm. As described above, the algorithm expects, as input, a bounded area where the object of interest can be found in (a so called region of interest), and additionally it takes a set of classified pixels (fackground) to segment the object. The classified pixels help refine the hard segmentation, although it might also work sufficiently well without providing foreground/background pixels as long as the region of interest is given like discussed below.

The *GrabCut* algorithm itself is based on the *Graph Cut* image segmentation algorithm and "*addresses the problem of efficient, interactive extraction of a foreground object in a complex environment whose background cannot be trivially subtracted*" [40]. The *Graph Cut* was developed combining both texture (color) information and edge (contrast) information unlike classical image segmentation tools which only used either of the mentioned characteristics.
The *Graph Cut* segmentation is basically done by minimizing an energy function, such that the minimization is reduced to solving a minimum cut problem [10], [23]. With the *GrabCut* three enhancements were introduced compared to the original *Graph Cut* technique:

1. Gaussian Mixture Model (GMM): The first enhancement was the utilization of GMMs instead of only monochrome images, as with GMMs it is feasible to process RGB color information. To do so, an additional vector was introduced which assigns a unique GMM color component to each pixel, either from the background or the foreground. Consequently the original energy function is extended by said vector. See [40] for further details.

2. Iterative Estimation: By iteratively minimizing the energy function the GMM color parameters are refined by assuming the result of the initial call to be the input for further calls.

3. Incomplete Labelling: Due to the above mentioned refinement it is feasible to run the algorithm without specifying foreground and background pixels explicitly.

The enhancements listed above were mainly introduced to put a light load on the user, while maintaining satisfactory results. In [40] it is concluded that experiments showed that the *GrabCut* algorithm performs at almost the same level of segmentation accuracy as the original *Graph Cut* algorithm while requiring significantly fewer user interactions. Hence, the *GrabCut* algorithm was seen as the perfect base algorithm for the automatic segmentation approach presented in this work, because it can be assumed that it still yields satisfactory results by specifying only a small set or indeed no input pixels at all. As remarked in Section 5.3.1.2 the presented automatic segmentation approach tries to provide a meaningful guess of classified input (foreground/background). Yet it is still possible that due to the nature of tracking inaccuracies and pixel limitation the labelled pixels are reduced to a small set of pixels which are passed to the *GrabCut* call and as pointed out the result will still be of sufficient quality in most cases.

Note that the proposed segmentation algorithm was developed in terms of a "side product" within this thesis such that the focus lies on the video compositing, for which the segmentation results are used as input. Proper segmentation of dynamic video content itself is a difficult task and could fill a thesis like this easily; which is why the development of the segmentation feature was carried out in a way such that its results were just acceptably good to use as input for the *Online Video Processing*. This and the highly dynamic aspect of the video source material, as explained in Section 5.2, are the reasons why the automatic segmentation algorithm may yield imprecise results in certain cases, especially when the background is very similar to the foreground with respect to the color and/or structure of surfaces. See Fig. 5.6 for an example, where the segmentation detects parts of the background like it would belong to the pedestrian's right

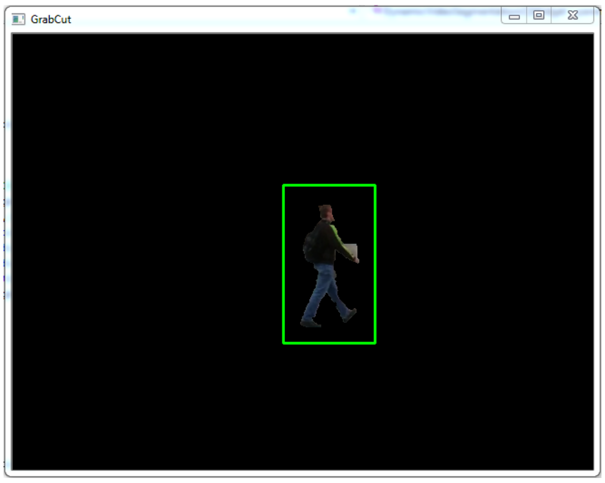


Figure 5.6: Segmentation inaccuracies.

arm. Due to these possible segmentation inaccuracies a feature was introduced which makes it possible to reinitialize the segmentation at a certain frame number such that the user may specify foreground and background pixels again (by clicking the button *Reinitialize Grabcut*, as depicted in Fig. 5.3), like it was done initially in the first frame, as described in Section 5.3.1.1.

5.3.1.4 Post-GrabCut-processing

The aim of the *Post-GrabCut-processing* is to convert the outcome of the GrabCut algorithm to something useful which can be used as the overlay for the *Online Video Replay* and to save the information which will later be needed in the creation of the *reference panorama*. Additionally, the identification of the input for the optical flow computations happening in the *Pre-GrabCut-processing* step - executed on the subsequent video frame - takes place here. Below a short description about each of the mentioned steps is given.

1. Augmentation Overlays.

Converting the outcome of the GrabCut algorithm to the overlay used in the *Online Video Replay* is done by applying the following subtasks:

- Compute a binary alpha mask distinguishing between foreground and background:

  The GrabCut algorithm classifies the output in four different groups of vectors. The probably easiest way to obtain a mask which only distinguishes between foreground and back-

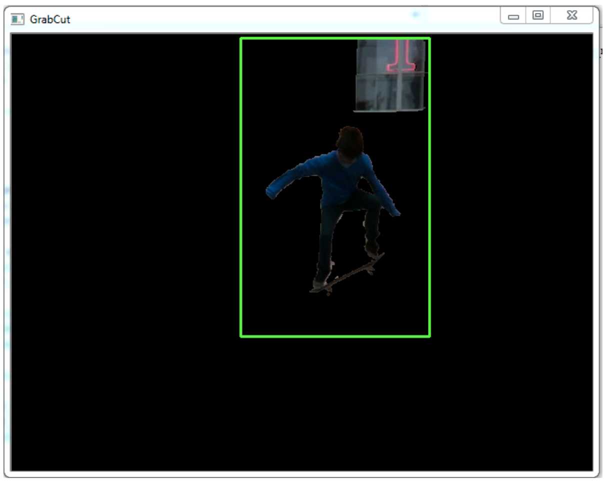


Figure 5.7: Two objects detected as foreground by the GrabCut algorithm.

ground information is to check the value of the first bit with a bitwise-AND-operation on the four different pixel group values (internal values from 0 to 4, see below). This means all foreground pixels (*possible* and *obvious*) will be 1. In [25] it is explained that "*This is possible because these constants are defined as values 1 and 3, while the other two are defined as 0 and 2.*" Therefore, checking the first bit with a bitwise-AND-operation transforms all background pixels to 0 of course.

- Apply thresholding to get an alpha channel mask:

  In order to get an alpha channel encoding transparency values between 0 (fully transparent) and 255 (fully opaque) a simple thresholding function is applied on the result of the previous step. This results in an alpha channel image containing white pixels for the extracted foreground object(s) and black pixels for the background. It is possible that the GrabCut algorithm identifies more than one foreground object in the video frame. However, the current implementation of the automatic segmentation approach presented here is only interested in one foreground object (i.e. the skateboarder). Compare Fig. 5.7 for an illustration where two objects were detected as foreground by the algorithm. Therefore, a way on how to only keep the real object of interest in the final alpha mask, needs to be found. Due to the limitation of the area which is considered by the GrabCut algorithm, namely the region of interest, it is assumed that the largest foreground object *is* the object of desire.

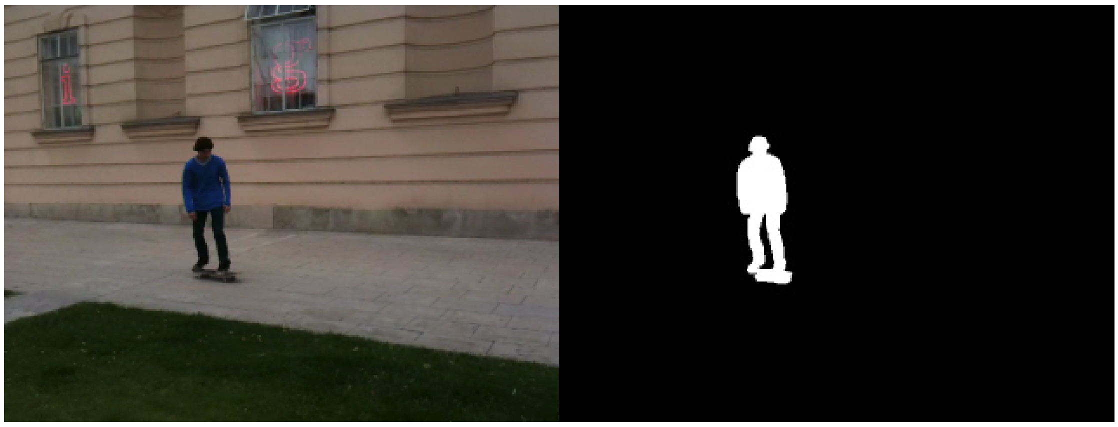


Figure 5.8: Video frame (left) and final alpha mask (right). The white contour resembles the extracted foreground object.

- Compute the largest connected component (LCC):

  To actually only keep the largest foreground object and discard all other foreground pixels, which may have been identified as such, the program computes the outlines of all foreground objects (i.e. contours) and compares their sizes. The function in use returns only the largest contour, often depicted as the largest connected component in computer science.

- Apply a dilation function to smoothen the borders:

  The extracted foreground object might look like it has been cut out rather sharply, which is why a dilation function is applied to smoothen the contour borders. This helps to embed the overlay more seamlessly into the final view in the *Online Video Replay*. See Fig. 5.8 on the right for an exemplary final alpha channel image, where only the largest connected component with dilated borders is visible.

- Write the output files (size-optimized RGBA image + text file containing the offset information per RGBA image):

  The alpha channel mask generated in the previous step forms the basis for the final step of the offline overlay creation. Now one could simply generate an RGBA image combining the normal video frame and the corresponding alpha channel and render the overlay fullscreen in the *Online Video Replay* step. This would further simplify updating the video scene as it is being played back, because the overlay's position wouldn't have to be updated. Using this approach there would be one big drawback though. As it was observed throughout this work in most cases the extracted foreground object resembles only a fraction of the size of the full video frame. Consequently passing the full video frame to the render queue would result in slower rendering of the overlay and therefore slower playback of the live video scene in total. To reduce the data overhead the smallest suitable fraction - containing the extracted foreground object - is saved, along with its offset within the full frame. The offset is needed to correctly update the overlay's position while it is being rendered into the live view.

To create the final overlay the bounding rectangle around it is computed and to simplify the rendering of the overlay the width and height of the rectangle are extended in each case, such that the new width and height are obtained by computing the smallest power of 2 which is bigger than the old width and height, respectively. The reduction from the full size overlay to the size-optimized overlay is presented in Fig. 5.10.

Now once the size of the overlay has been determined it needs to be written to an RGBA image. This is done by defining a ROI in the normal video frame and the alpha mask, each with the position and size of the size-optimized overlay. To obtain the final RGBA image, the ROIs created just before are combined into a new image by adding its color channels, such that the first three components of the new image resemble the RGB components of the first ROI and the fourth component resembles the binary alpha channel contained in the second ROI. The final image will therefore hold four components with a cumulative bit-depth of 32 bit (24 bit RGB image + 8 bit binary alpha channel). Fig. 5.9 illustrates the creation of the final overlay image. To complete the overlay creation step the position and offset per overlay is written to a standard text file in order to access the saved information while rendering the overlay in the *Online Video Replay*.

2. Saving frame information.

Saving the information for the creation of the *reference panorama* means to store the regular video frame along with the full size alpha mask. How to use these images to generate all the information about the background which is needed is described in detail in 5.3.2.

3. Determining the input for the subsequent optical flow computations.

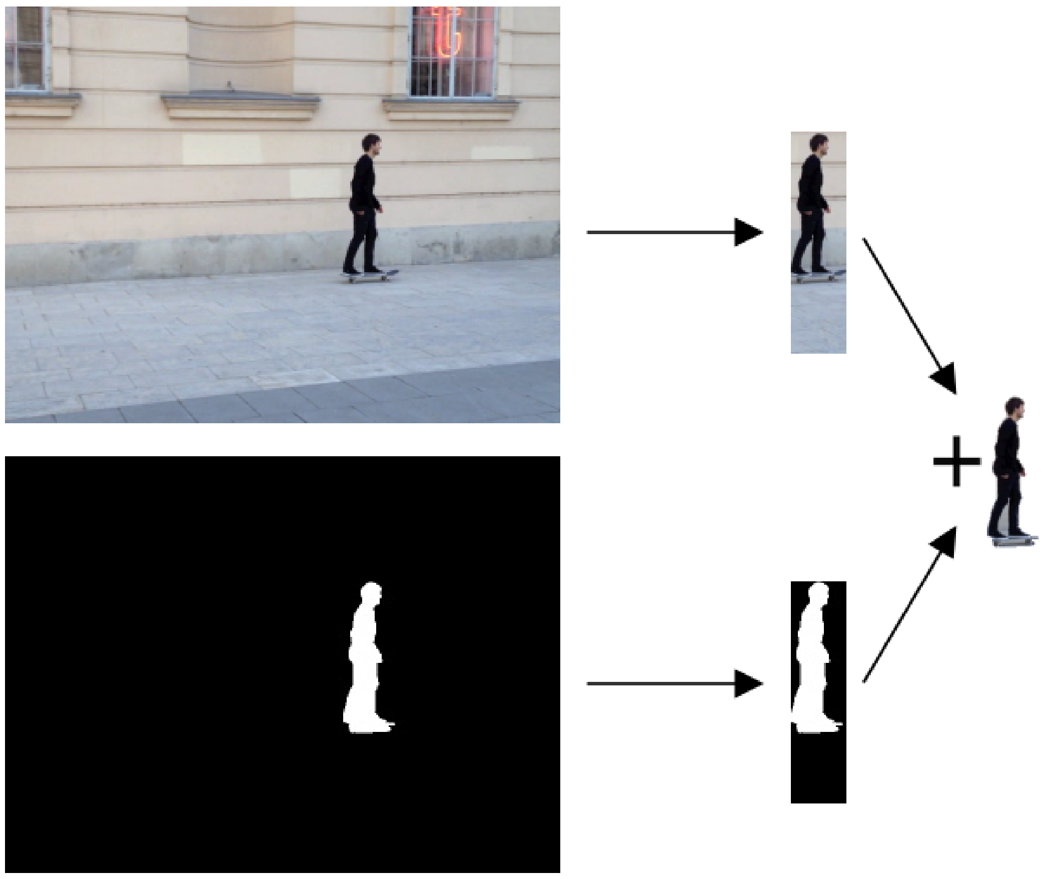


Figure 5.9: Extracting a region of interest in the normal video frame (RGB) and the alpha mask (A) and the combination to the final RGBA overlay.

This means to retrieve vectors containing pixel positions, which can be set as input for the feature tracker. Basically executing the GrabCut algorithm delivers such vectors containing pixel positions which reflect the segmented object and the background information, respectively. By looking at these different output vectors in detail it is feasible to further process the classified pixel positions in any meaningful way. As just mentioned the different output vectors can be grouped into foreground and background pixels, yet it is still possible to further split up those groups; namely *possible* and *obvious* foreground and background pixels, respectively. This leaves us with four different vectors of classified pixel positions [1]:

- GC_BGD (internal value 0) defines an obvious background pixel.
- GC_FGD (internal value 1) defines an obvious foreground (object) pixel.
- GC_PR_BGD (internal value 2) defines a possible background pixel.

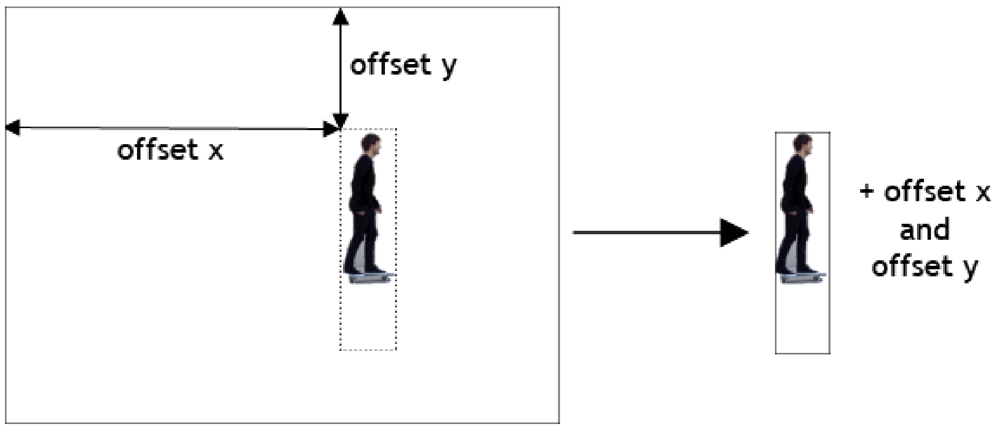


Figure 5.10: Reducing the size of the overlay to minimize the data flow. (Left) Overlay in the size of the full video frame (320x240 pixels). The dashed line outlines the optimized size of the overlay and the arrows indicate the offset in horizontal and vertical direction. (Right) Instead of storing the full size overlay only the size-optimized overlay (in this case 32x128 pixels) along with its offset information is stored.

- GC_PR_BGD (internal value 3) defines a possible foreground pixel.

The reason why this further classification is important here, is that the behavior of the automatic segmentation approach, which is presented, heavily varies depending on whether *possible* or *obvious* pixel vectors are passed as input to the subsequent feature tracker calls. It was observed that the feature tracker works best computing the optical flow for *obvious foreground pixels* and *possible background pixels*. Note that two instances of the feature tracking algorithm are actually required to be able to compute the optical flow for the foreground and background features separately.

5.3.2 Background Information

After the foreground segmentation has been applied to each frame the background information is being processed. Due to the possibility that a user can rotate the camera while recording the video - as pointed out in Section 5.2 - the recorded frames hold different portions of the scene's background. Furthermore, the foreground object also occludes parts of the background, reducing the amount of visual features that are later available for vision-based registration. Therefore, the aim in this step is to reconstruct as much background information as possible to simplify the registration process which is performed in the *Online Video Processing* step. Hence, it is necessary to not only take into account the background information from one video frame but from all frames and integrate them into one panoramic image.

The panoramic image is created using a modified version of the *Panorama Mapping and Tracking* approach presented in [46]. The original implementation uses features in the incoming video frames to register the frames and stitch them into a panoramic image. In [46] it was also demonstrated how to track the camera motion R_S of the recording camera while constructing the panoramic map. Similarly, the described system in this work assumes the camera movements are only of rotational nature. Note that for stitching together a panoramic image the *Panorama Mapping and Tracking* technique relies on a camera calibration file which needs to contain the intrinsic camera parameters determined in Section 3.1.7.

The reason an adapted version of the *Panorama Mapping and Tracking* algorithm is being used, is that it is essential to only map background pixels into the panoramic image, because mapping foreground objects into the map would result in a distorted panoramic image or the algorithm might even suspend while trying to create the panoramic image. To accomplish the altered behavior it is not sufficient to feed the tracking and mapping algorithm with the normal video frames, such that each new frame gets mapped into the panoramic image, as these frames contain both background AND foreground information. Hence, auxiliary information needs to be provided to the algorithm, to point out which areas of the frame (i.e. the foreground object) should be omitted while the panorama is being created. This extra information is made available by using a binary alpha image which outlines the contour of the previously segmented foreground object. As mentioned in the *Foreground Segmentation* step this alpha image has been created for each video frame. So instead of just mapping the whole video frame into the panoramic image, the foreground object is omitted in the mapping process and consequently only background pixels are mapped into the panoramic image for each frame which is being processed. This yields of course missing information in the panoramic image in the beginning. These resulting holes can be closed by continuous mapping of the video frames as the foreground object moves within the camera frames revealing occluded background information in later frames, as shown in Fig. 5.11 and explained in detail in Section 7.1. Note that the areas containing missing information in the beginning do not start to detect visual features which are later matched in the registration process (here denoted by green dots) before the missing information is filled up. This prevents detection of features which should not be classified as such (because they would belong to the foreground).

Once the information holes are closed, the remaining frames are mapped into the panorama in the usual way, i.e. without considering the information encoded in the alpha channel image. Fig. 5.12 demonstrates a final panoramic image (reference panorama). As already mentioned the *Panorama Mapping and Tracking* approach presented in [46] allows to determine the camera

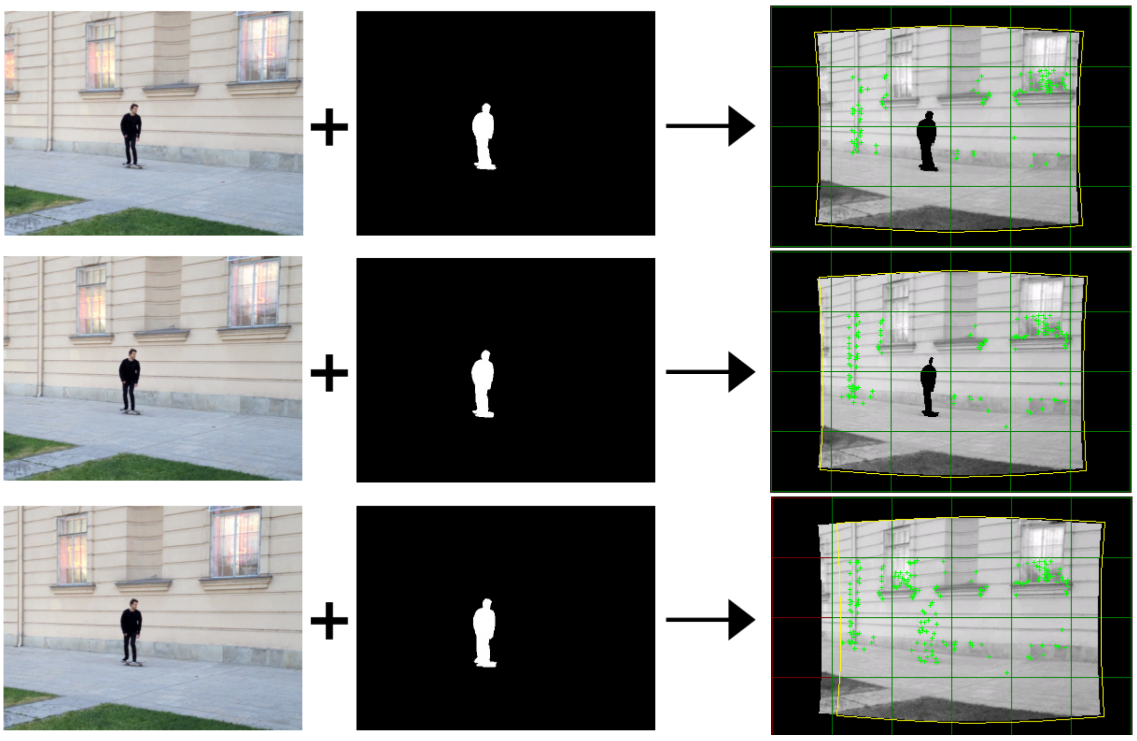


Figure 5.11: Omitting foreground objects which are encoded in the alpha channel (middle) while mapping video frames (left) into the panoramic image (right). Missing background information (black holes at the right) is closed over time as the foreground object moves through the frame.

motion R_S at a certain frame position while the panorama is being created. In the *Online Video Processing* the camera rotation at a certain frame position is of crucial importance. To later access the rotation at a designated frame number, all camera motions are stored in a textfile along with the reference panorama image.

5.4 Online Video Processing

The *Online Video Processing* is the final part of the presented system where everything comes together such that the online replay of the created overlay in a suitable context is realized on a mobile device (i.e. a smart phone). In order to prepare the replay, several steps need to be considered. At first, all material which plays a role in the *Online Video Processing* step must be transferred to the destination device. The current implementation only supports transferring the sources directly onto the device (e.g. copy all files into the application's resources folder). An overview of all obligatory resources is given in Fig. 5.13.

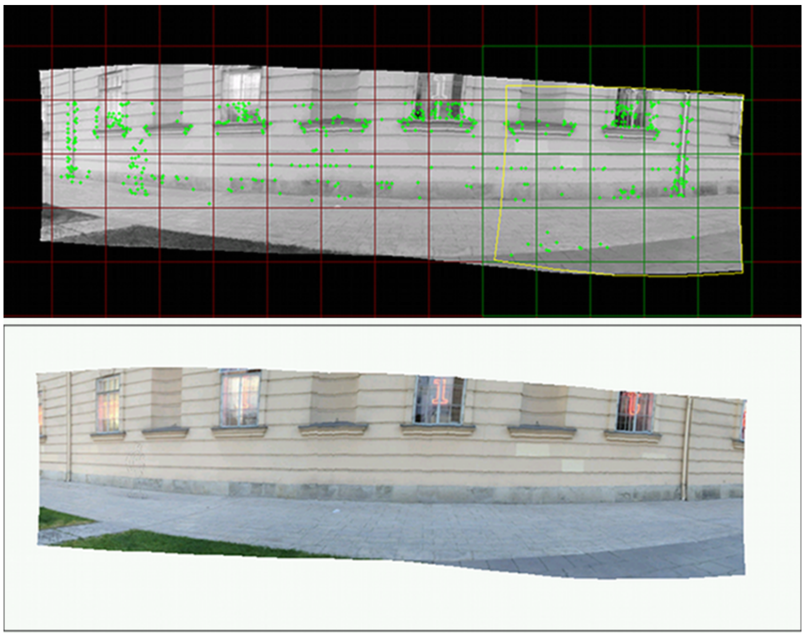


Figure 5.12: Finished panoramic image created using only the available background information. (Above) Internal representation including detected visual features. (Below) Panoramic image which is being stored.

As pointed out earlier the purpose of the presented system is to replay augmented video material in a live camera view such that past events, which took place at a certain location, can be replicated with respect to the user's view onto the scene. This implies the need to actually return to the location where the event took place (i.e. where the original video material was recorded); because only then the currently implemented system is able to correctly replay the augmentations. See Chapter 11 for possible other use cases of the system, where it is not necessarily the case that the content is replayed in the same context the source material was recorded.

After the identification of the location where the online replay shall happen, a registration process must be carried out. The registration is inevitable for the system to operate correctly. Due to the registration of the user's movement within the scene the according information can be passed to the system, such that the augmented material's position is updated in real time and therefore blends in correctly while the content is being replayed. In the following the registration process is described in detail as well as the replay (aim of the application) once the registration was fulfilled successfully.

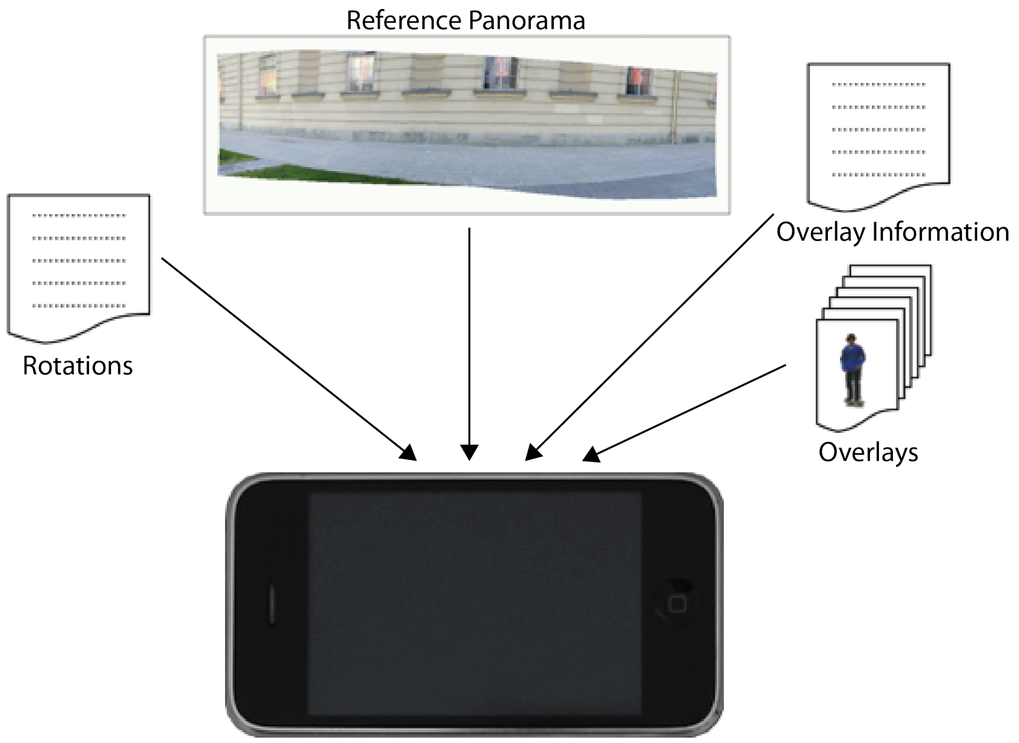


Figure 5.13: Resources which need to be transferred to the target device for the online replay of the video scene.

5.4.1 Registration

Assuming the target device is equipped with all source material that is mandatory for the proper replay and the user resides at the location where the original event took place the registration process can be started. For this purpose the system also relies on the *Panorama Mapping and Tracking* technique presented in [46]. In other words once again a panoramic image is built from the live camera feed and moreover also the camera motion R_T is tracked for every incoming frame, yet this time there is no need for considering alpha channel images to distinguish between foreground and background objects.

The use of the panorama-based tracking allows for a high precision of the registration and the tracking, as it does not rely on noisy sensor values. As mentioned earlier this comes with the drawback of supporting only rotational movements. However, most users only perform rotational movements while using outdoor AR applications [17] making this constraint acceptable in most scenarios.

To start the registration process the ARVideo application is started up. The application immediately begins to build a new panorama from the live camera feed. Furthermore, as the application has been started up the *reference panorama* (see Fig. 5.13) and all other resources are loaded into memory to be accessible from there on.

While building the new panorama of the environment the application tries to match the loaded panorama holding the background pixels against the newly built panorama. The matching is performed using a point feature technique (in this case PhonySIFT, see [47]). As soon as the overlapping area - the area holding image information that is contained in both panoramas - is big enough, the matching using PhonySIFT should succeed and provide the transformation T_{ST} describing the relative motion between the camera used to record the video (the source camera s) and the camera where the video information should be registered in (the target camera t). By assuming that the user of the system is roughly at the same position where the video was recorded it is acceptable to constrain the transformation T_{ST} to be purely rotational. The therefore determined transformation is illustrated in Fig. **??**.

Supposing the registration was successful the user is able to replay the augmentation overlays and therefore experience the recorded event in his own way on his target device. How the correct replaying works is described in the following.

5.4.2 Online Video Replay

For the Online Video Replay to function properly a few preconditions must be taken into account:

- User resides at the vicinity where the original video scene was recorded.
- Successful registration of the live camera view with respect to the reference panorama.
- Available resources for the replaying (overlays + overlay information per frame, rotation information per frame).
- Successful tracking and updating of the newly built panorama within the live context.

Provided that all the preconditions above are fulfilled the replaying of the video content can be started. Considering the transformation T_{ST} it is feasible to transform each pixel from the source panorama into the target panorama and vice versa. This allows the replaying of the video information by overlaying the current environment (live camera view) with the object of interest from the previously recorded video frame. To accomplish the desired behavior firstly the live camera frames are loaded into the system such that the actual transformation R_T is determined. Secondly the overlays need to be rendered into the frame with the correct transformation, i.e. the live camera frame is correctly augmented with the video content. Every loaded overlay is processed such that applying the combination of the transformation R_S (the orientation of the source camera computed in the offline video processing step), the transformation T_{ST} (the transformation between the source and the target camera gained from the *Registration*) and the transformation R_T (the orientation of the target camera computed using the panorama-based tracking) augments the live camera view with the video content. In more detail this means to apply the following final transformation R_O on every rendered overlay:

$$R_O = R_T * T_{ST} * {R_S}^{-1} \tag{5.1}$$

Due to the fact that the proposed system utilizes size-optimzed overlays (see Section 5.3.1.4) it needs to be taken into account to also update the overlay's offset within the video frame which was identified while processing the segmented foreground object (see Section 5.3.1.4). Due to the *OpenGL ES* rendering mechanism it is not effective to simply add the determined offset (in pixels) per overlay to the overlay's position within the frame. As a consequence the rendering must be preceded by transforming the pixel offset into the according internal offset dependent on the used target camera. For example, to translate pixel coordinates p_x, p_y into internal screen coordinates x, y using the camera matrix cam (see Section 3.1.7) and a constant screen depth of $z = 1$, the computation given in Equation (5.2) has to be made.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = cam^{-1} * \begin{bmatrix} p_x \\ p_y \\ z \end{bmatrix} \tag{5.2}$$

Note that using the panorama-based tracker an update of the transformation R_T for each frame is obtained in real time.

This allows rotating the target camera completely independently...

... (relative to the orientation of the source camera) while replaying the video scene. Moreover, the updated transformation R_T is essential for maintaining the precise registration of the video in the current view. The freedom of the view onto the scene creates an immersive user experience. Because of the dynamics of the recorded video material (see Section 5.3) the user is challenged to rotate the target device to fully capture the replayed video scene. Hence, if the user holds still

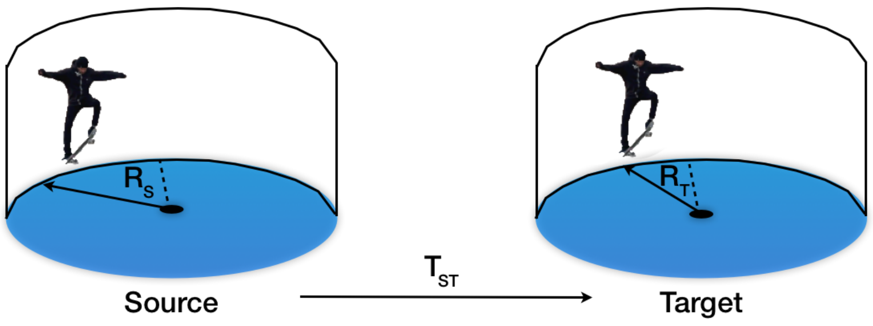


Figure 5.14: Illustration of the occurring transformation between the source camera and the target camera used for replaying the augmented video.

and does not move the target device at all, the replayed video will get out of view, just as other people or objects moving through the scene. In other words the presented system allows...

... replaying past events from an own point of view with the impression of these events just happening now.

CHAPTER 6

Foreground Segmentation (Implementation)

The foreground segmentation task as described in Section 5.3.1 is accomplished by a stand-alone program which was written in the *C++* programming language making use of *OpenCV* libraries and the *Qt Framework*. Which is why an own chapter is denoted to the description of the implementation of the foreground segmentation. The description starts by giving an overview with the help of the application's class diagram and then proceeds by explaining the central class in detail along with its properties (variables) and methods.

Note that for the sake of readability the description of variables and methods belonging to classes is kept simple and in a general way, such that methods only contain its basic signature without parameters which get passed along with the method call and furthermore variable and method return types are given in a general and not language-specific way (e.g. *integer* instead of the *C++*-specific type *int*).

6.1 Class Diagram

The *Class Diagram* illustrated in Fig. 6.1 gives an overview of the *DynamicVideoSegmentation* application and its contained classes. As can be observed the classes are arranged by color to distinguish between classes belonging to the *Qt Framework* (green), classes provided by the *OpenCV* libraries (blue) and classes which make up the *DynamicVideoSegmentation* application (light red) and which were developed in the course of this work.

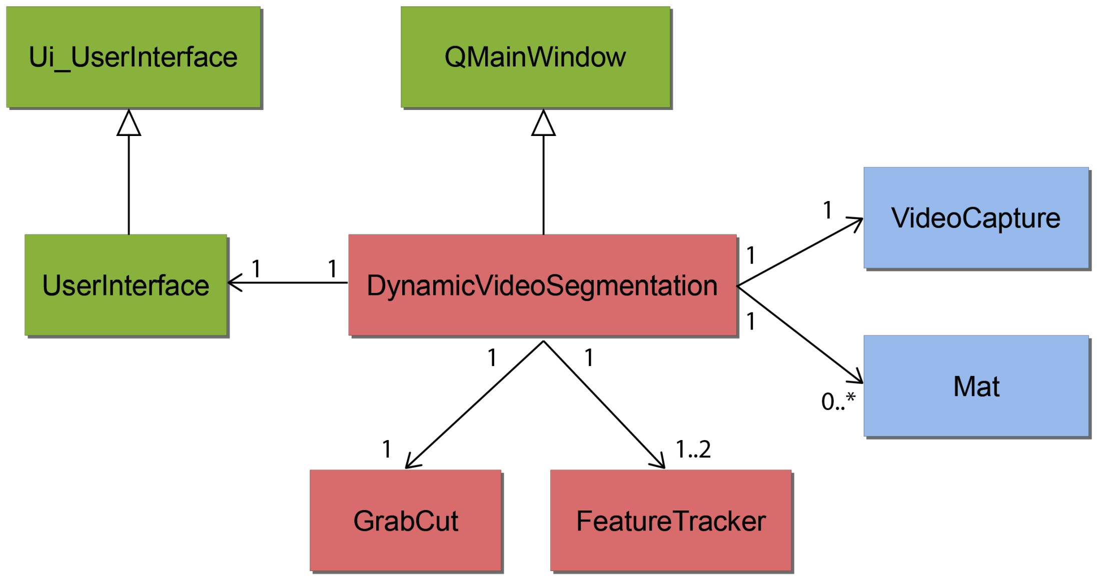


Figure 6.1: Foreground Segmentation Class Diagram.

The *UserInterface* class gets generated automatically by the *Qt Visual Studio Add-in* after creating the user interface manually in the *Qt Designer*, which comes with the *Qt Framework* and the said add-in. The *UserInterface* class derives from the class *Ui_UserInterface*, which is a metaclass that gets generated automatically as well.

The *FeatureTracker Class* has been developed based on an example class taken from [25], whereas the *GrabCut* class was taken from an official *OpenCV* source[1]. Both classes have in common that they contained the minimal functionality of the according algorithms which was then extended and adapted to suit the requirements of this work. Hence, these classes won't be described in detail as the original example files already contain all vital information. However, the *DynamicVideoSegmentation Class* will be explained in full detail as it brings together all of the other classes.

6.2 DynamicVideoSegmentation Class

The *DynamicVideoSegmentation* Class symbolizes the main application entry point of the *Foreground Segmentation* algorithm. As can be observed from Fig. 6.1 the *DynamicVideoSegmenta-*

[1]https://code.ros.org/trac/opencv/browser/trunk/opencv/samples/c/grabcut.cpp?rev=2326

tion class inherits the class *QMainWindow*. This enables the application to be controlled through the user interface, which is shown in Fig. 5.3.

Basically the application flow can be abstracted such that the *DynamicVideoSegmentation* application is created and initialized with all necessary data, then the user opens a video and applies the manual *GrabCut* segmentation and then the program stays in a loop where the automatic segmentation is carried out on the remaining input frames. The detailed processing was already described in Section 5.3.1. In the following the *DynamicVideoSegmentation* class is described in more detail regarding its attributes (i.e. member variables) and methods. Additionally, *Qt*-specific methods which are made use of to handle user interaction within the class (so called *signals* and *slots*) are explained as well.

Member variables

- *ui* : *UserInterface*. Automatically generated instance of the user interface created with the *Qt Visual Studio Add-in* as explained in Section 3.1.6.
- *fileName* : *String*. Variable which holds the filename of the input video.
- *textureInformation* : *OutputFileStream*. FileStream which is used to write the overlay information (position + offset) per segmented foreground object into a text file, which later is used as input for the *Online Video Replay* as illustrated in Fig. 5.13.
- *frameNr* : *integer*. Holds the current frame index.
- *frameNrStr* : *Character Array*. Holds the current frame index converted to characters. The continuous index is then appended, for example, to the base filename when writing the output frames.
- *contourX* : *integer*. Encodes the horizontal position of the foreground object's most left/top coordinate.
- *contourY* : *integer*. Encodes the vertical position of the foreground object's most left/top coordinate.
- *contourWidth* : *integer*. Encodes the width of the foreground object, such that the sum of *contourX* + *contourWidth* equals the foreground object's most right position.
- *contourY* : *integer*. Encodes the height of the foreground object, such that the sum of *contourY* + *contourHeight* equals the foreground object's most bottom position.

- *IS_RUNNING* : *boolean*. Flag which is *true* as long as the automatic segmentation is running.
- *MANUAL_INPUT_GC* : *boolean*. Flag which is *true* if the *GrabCut* algorithm should wait for user input.
- *cap* : *VideoCapture*. Instance of the *OpenCV VideoCapture Class*[2]. The *VideoCapture Class* contains convenient methods for opening a video file and reading it frame per frame.
- *featureTracker_FG* : *FeatureTracker*. Instance of the *FeatureTracker Class*, which is responsible for tracking foreground features (remember that the user marks some foreground pixels to initialize the *GrabCut* call and that those pixels get tracked over consecutive frames).
- *featureTracker_BG* : *FeatureTracker*. Instance of the*FeatureTracker Class*, which is responsible for tracking background features.
- *grabCutInput_FG* : *Vector<Point>*. Vector which gets filled by the *DynamicVideoSegmentation Class* with the 2D pixel coordinates which serve as the foreground input pixels for the *GrabCut* algorithm.
- *grabCutInput_BG* : *Vector<Point>*. Vector which gets filled by the *DynamicVideoSegmentation Class* with the 2D pixel coordinates which serve as the background input pixels for the *GrabCut* algorithm.
- *grabCutResult_FG* : *Vector<Point>*. Vector which gets filled by the*GrabCut Class*. Represents the pixels which are detected as foreground by the *GrabCut* algorithm.
- *grabCutResult_PRFG* : *Vector<Point>*. Vector which gets filled by the*GrabCut Class*. Represents the pixels which are detected as *probable* foreground by the *GrabCut* algorithm.
- *grabCutResult_BG* : *Vector<Point>*. Vector which gets filled by the*GrabCut Class*. Represents the pixels which are detected as background by the *GrabCut* algorithm.
- *grabCutResult_PRBG* : *Vector<Point>*. Vector which gets filled by the*GrabCut Class*. Represents the pixels which are detected as *probable* background by the *GrabCut* algorithm.

[2]http://opencv.itseez.com/modules/highgui/doc/reading_and_writing_images_and_video.html

- *featureTrackerResult_FG* : *Vector<Point>*. Vector which gets filled by the*GrabCut Class*. Represents the pixels which are tracked from one frame to the other by the *FeatureTracker* algorithm run on the foreground pixels.

- *featureTrackerResult_BG* : *Vector<Point>*. Vector which gets filled by the*GrabCut Class*. Represents the pixels which are tracked from one frame to the other by the *FeatureTracker* algorithm run on the background pixels.

- *frame* : *Mat*. Instance of the *OpenCV Mat Class*[3]. The *Mat Class* is a basic n-dimensional structure in *OpenCV* and is widely used as input/output in plenty of *OpenCV* algorithms. Therefore, it is perfectly suitable to hold any kind of data, such as simple 2D representations of pixel coordinates, but also more complex data, like for example image data containing 3 or more color channels. The variable *frame* represents the normal video frame which is opened using the before mentioned *VideoCapture Class*.

- *edges* : *Mat*. Input to the *applyEdgeDetection()* method and will therefore contain the detected edges in the *frame*.

- *GCMask* : *Mat*. Contains the *GrabCut* output encoded in the four different values, as explained in Section 5.3.1.4.

- *featureMask* : *Mat*. Contains the tracked features in a 2-dimensional *Mat* structure to match them against the found contours.

- *finalMatchMask* : *Mat*. Contains the matched (features against contours) pixel coordinates.

- *contours* : *Vector< Vector<Point> >*. Structure which is filled by the *OpenCV* method *findContours()*. Is used for different purposes. Firstly to match the tracked features against the contours in the frame and secondly for finding the contours of the final foreground object.

- *scaledFrame* : *boolean*. Flag which is set to *true* if the input frame is scaled to *320 x 240* pixels, as described in Section 5.3.1. Depending on the value writing the output must be adapted to compensate for the smaller/bigger frame sizes.

[3]http://opencv.itseez.com/modules/core/doc/basic_structures.html

Methods

- *DynamicVideoSegmentation(); - DynamicVideoSegmentation* constructor, initializing member variables like the *boolean* fields *IS_RUNNING*. To setup the user interface for the application the variable *ui*'s *setup()* method is called with the current instantiated class as parameter (parameter "this" as in programming very common). Actually this call is automatically generated by the *Qt Framework*. As explained later on in more detail the *Qt Framework* provides a mechanism for handling user interaction within the program, such that certain code parts are executed in case the user clicks, for example, on a button. Due to this mechanism it is necessary to *connect* so called *signals* with their according *slots*. All these connections are established in the constructor.

- *void prepareSegmentation(); - Method* which is called right after the user opens a video file. Prepares the segmentation by creating output directories, opening the *VideoCapture* instance and creating the *textureInformation* output file stream.

- *void preGrabCutProcessing();* - Performs all processing which needs to be carried out before the program executes the *GrabCut* algorithm, as explained in detail in Section 5.3.1.2.

- *void postGrabCutProcessing();* - Performs all processing which needs to be carried out after the program executes the *GrabCut* algorithm, as explained in detail in Section 5.3.1.2.

- *void segment();* - Method which performs the segmentation and the according pre- and post-processing of frames, depending on which state is activated (manual/automatic).

- *void createDir();* - Creates a directory on the filesystem with the name which gets passed along with the method call. This is made use of, for example, to create the output directories.

- *void applyEdgeDetection();* - Applies the *Canny Edge Detector* as mentioned in Section 5.3.1.2 to determine edges distinguishing between different objects in the frame.

- *int getLargestConnectedComponent();* - Returns the index of the largest connected component as already specified in Section 5.3.1.4.

- *void drawFeaturesIntoMask();* - As the name suggests draws features (points) into a mask (e.g. *featureMask*) to further process them in a 2D structure instead of a "flat" structure (vector). Is used to, for example. match features against contours as stated below in the method *matchFeaturesWithContours()*.

- *void drawContoursIntoMask();* - As the name suggests draws contours (vector of points) into a mask to further process them in a 2D structure instead of a "flat" structure (vector). Is used to, for example, to match features against contours as stated below in the method *matchFeaturesWithContours()*.

- *void matchFeaturesWithContours();* - Matches the foreground features against the found contours in the current frame to limit the foreground pixels which get passed to the *GrabCut* algorithm, which was made clear in Section 5.3.1.2.

- *void applyDilation();* - Takes two masks (of type *Mat*) as input and performs the *OpenCV dilation*[4] operation on the first input mask and writes it to the second one. This is done to smoothen the borders of the final object contour.

- *void writeOutput();* - Writes all output files (normal frame, alpha channel, RGBA overlay image, texture information) to the filesystem.

Slots

The *Qt Framework* contains a mechanism for dealing with all kinds of events which can occur while executing a program, such as button clicks or user interface interaction in general, internal state changes, and so on. These events or actions, are bound to so called *signals*, which can be connected to so called *slots*, in order to reasonably react to the occurrence of such an event. Basically *signals* and *slots* are just methods which need to be explicitly defined along with meaningful parameters (Note that most *Qt* classes already come with a meaningful set of *signals* and *slots*). Connecting *signals* and *slots* works in a way that a certain class, for example the *DynamicVideoSegmentation Class*, registers itself by explicitly defining a "callback" from one certain *signal* to a desired *slot*. That way it is possible to execute appropriate code whenever a user clicks on, for example, a user interface button. In the following the *slots* defined in the *DynamicVideoSegmentation Class* are explained in more detail.

- *openFileDialog();* - *Slot* which is called whenever the user clicks on *"File - Open Video File"*, compare Fig. 5.3. As the name suggests it open a file dialog to give the user the possibility to choose a video file which should be opened.

- *activateManualInputGrabCut();* - *Slot* which, in case the automatic segmentation is running, stops it and sets the variable *MANUAL_INPUT_GC* to *true*, which consequently enables the user to reinitialize the segmentation by providing foreground and background pixels, whenever the automatic segmentation seems to be too imprecise.

[4]http://docs.opencv.org/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html

- *callGrabCut();* - *Slot* which is called when the user clicks on the button *Call GrabCut Algorithm*, i.e. to perform the manual *Grabcut* call in the beginning.
- *startAutomaticSegmentation();* - *Slot* which is called when the user clicks on the button *Start Automatic Segmentation*. It sets the boolean variable *MANUAL_INPUT_GC* to *false*, processes the results of the manual *GrabCut* execution by calling *postGrabCutProcessing()* and triggers the method *segment()* which then stays in the loop, described in Section 5.3.1.1, to automatically segment the remaining frames.
- *closeProgram();* - Stops all running processes and exits the program.

CHAPTER 7

Adaptation of PanoMT

The following chapter addresses the extension of the *Panorama Mapping and Tracking* algorithm, presented in Section 3.1.4.7, in terms of newly implemented features. The need for implementing additional functionality arose from the fact, that the original implementation didn't cover all requirements of the presented system as already noted in Section 3.1.4.7 and Section 5.3.2, respectively. As demonstrated in Section 5.3.2, dealing with alpha channels, which encode the information distinguishing between foreground and background objects, is necessary to create an undistorted panoramic map that represents the "reference panorama". The *reference panorama* is consulted while a user registers the target device within the "replay environment", as shown in Section 5.4.1. To actually perform the registration task the live camera panorama is matched against the loaded *reference panorama*, as explained in Section 5.4.1 as well. The original implementation of the *Panorama Mapping and Tracking* technique is not able to match two panoramic maps against each other, yet only the current single camera frame against the loaded panorama. Consequently the implementation had to be extended to support both, the handling of alpha channels and the matching of two panoramic images. Fig. 3.3 illustrates, in contrast to the general overview given in [4], a "customized" overview of the *Studierstube ES* framework with the altered *PanoMT* component (color-coded dark blue) along with the *ARVideo* application sitting on top of the framework. In the following the details about the implemented adaptations are given.

7.1 Handling alpha channels

The idea of how to handle binary alpha images, in order to discard the mapping of foreground pixels contained in a video frame while the panoramic map is created, builds upon the original idea which maps pixels row- and span-wise, respectively. In Section 3.1.4.7 it is explained that a span is a simple structure which encodes the left and right coordinate of a continuous characteristic, like unmapped pixels, and is very efficient for processing. Hence, due to the nature of the original implementation and the spans' efficient processing the adaptation also makes use of said spans. In contrast to the original implementation the adapted version does not process full spans. Full spans basically cover the whole interval of the row-wise image content from left to right, which makes perfectly sense if you don't have to encounter information holes, such as undesired foreground objects which should not get mapped. Consequently a different approach had to be found, in order to cope with rows where the background is "interrupted" by pixels which belong to a foreground object.

Basically, this was achieved by splitting up the full span of a row into several spans, such that the created spans cover exactly those parts of a row that contain background information and therefore should get processed and mapped into the panoramic image. The new approach processes rows of a camera image as follows:

- If the current pixel belongs to the background (black pixels in the alpha channel) either start a new span or append the pixel position to the active span if there is one. If the current pixel belongs to the foreground (white pixels in the alpha image) stop creating the active span and insert it into the list of spans for this row.
- Process the list of spans like the full span in the original implementation, which means to apply the boolean operation to each of the spans in the list. Hence calling the boolean comparison on the list of spans returns several small areas of unmapped pixels.
- Map all areas with unmapped pixels of the current row.

Fig 7.1 illustrates how video frames are being processed with the newly implemented feature in case the frames contain a foreground object (in this case arbitrarily shaped) and the according alpha channel is provided. Note that usually the frame masks would be distorted at the borders (compare Fig. 3.8), yet this has been ignored in the illustration for the sake of simplicity. Further processing of frames closes the information hole over time as this information gets available through successive frames.

7.2 Matching Of Two Panoramic Images

Basically, the matching of two panoramic images works in the same way as the standard matching algorithm described above in Section 3.1.4.7. There are a few things which needed to be con-

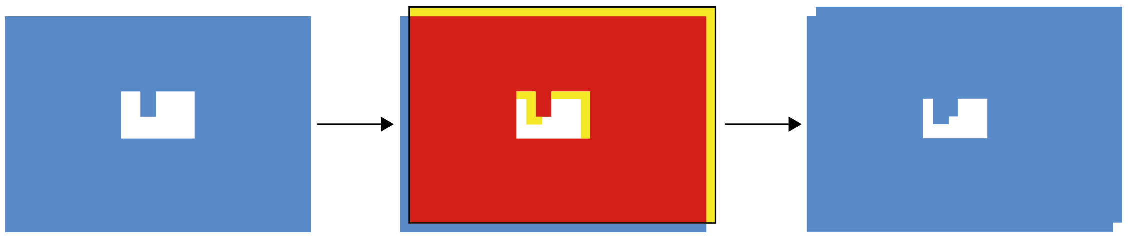


Figure 7.1: Mapping of pixels when foreground objects are considered. (Left) Blue: Mapped pixels in the first frame. White area inside: Pixels which have been omitted, because of the encoded foreground object in the alpha channel. (Middle) Black border: Mask for the second camera frame. Red: Intersection of already mapped pixels and current camera frame. Yellow: Pixels that still need to be mapped (exaggerated for better visibility). (Right) Blue: Mapped pixels after processing two frames. White: Still unmapped pixels.

sidered though, in order to match two panoramic maps against each other instead of a panoramic map and a single camera frame.

First off, the algorithm was limited to use only a few image sizes which could be passed to the algorithm to consequently match the passed image with the panorama. Due to the design of the matching algorithm it is necessary to pass the second panorama as a “simple” image rather than as the intern representation of the corresponding panorama. This means that the full size of the panoramic image was added to the list of allowed resolutions of the image which gets passed. Consequently, the extraction of features in the loaded panorama (which shall be matched against the live panorama) is performed in the same way as for an ordinary camera frame. Since usually a panoramic map is not completely closed and filled out with pixel information it was required to somehow ignore regions in the panoramic image which have not been filled with information. Otherwise these regions, especially borders between mapped pixels and unmapped pixels, would yield wrongly detected features by the FAST corner detector. This is due to the fact that the FAST corner detector would recognize these transitions between mapped and unmapped pixels as “corners” in an image, which most of the time represent good features.

In order to be able to not consider these regions, firstly the FAST corner detector is applied to the panoramic image in the normal way. Secondly, the feature pixel positions of the loaded map are read in. Remember that when a panorama is created it is split up into cells (32x8), whereas only for completely filled cells features are being detected and consequently written out when the panorama is saved. This saved pixel positions are now considered and only if detected features by the FAST corner detector and saved pixel positions match it can be assumed that this is indeed a feature lying within the mapped pixel information. After executing the intermediate

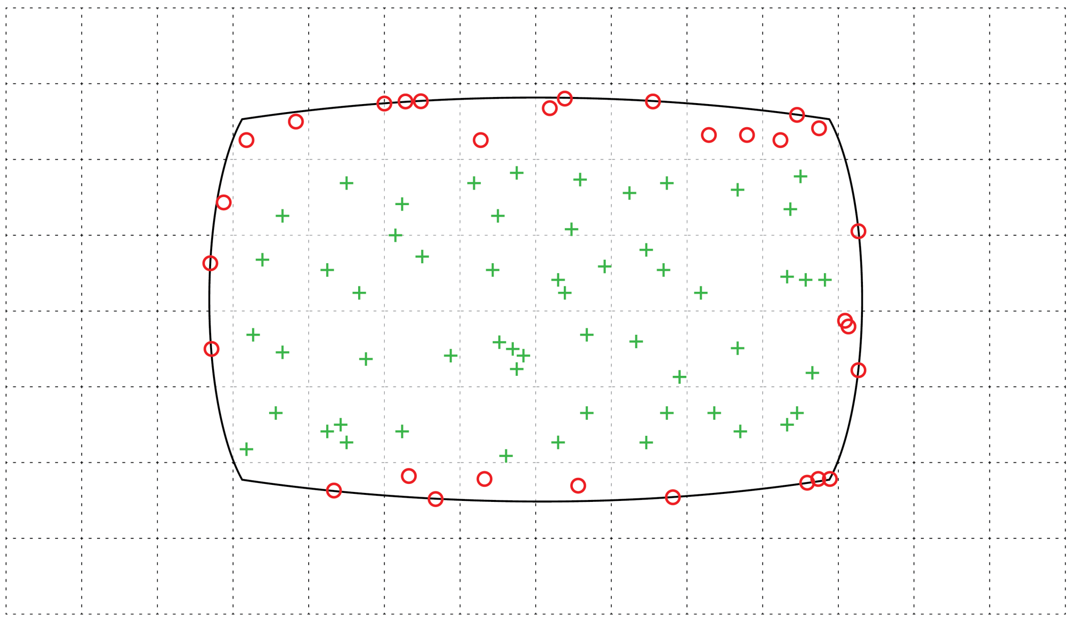


Figure 7.2: Outline of a sample panoramic image and how it could be passed to the matching process, after filtering out features which should not be considered. The green crosses depict features which lie inside complete cells and therefore are considered to be valid. The red circles depict features which are dismissed, in order to prevent wrongly detected features to be matched.

step the matching can process can be resumed in the normal way. In Figure 7.2 the outline of an examplary panoramic image is illustrated, whereas the green little crosses depict features which have been extracted by the FAST corner detector AND have been detected as features at the time the panorama was created originally. These pixels will be further processed in the matching step. The red circles in contrast resemble features which have been detected by the FAST corner detector, yet they lie outside of completely filled cells, which is why they will be dismissed. Note that two possible types of features might be in the range of the dismissed features. Firstly, the ones which lie inside the panoramic image, yet outside of completely filled cells (this means these features would actually belong to the panorama) and secondly, the ones which appear at the border lines between mapped and unmapped pixels. Due to the fact that the second type of features must not be considered in the matching process, all features outside complete cells are dismissed to prevent any wrongly detected features to be matched later on.

CHAPTER 8

Background Information + Online Video Processing (Implementation)

This chapter describes the implementation of firstly the feature which creates the *Background Information* as explained in Section 5.3.2 and secondly the implementation of the *Online Video Processing* as explained in Section 5.4. The reason for combining the description for both arises from the fact that both are based on the *PanoMT* component and hence share a lot of properties and code. Distinguishing between different requirements was achieved by executing different code parts whenever needed, as explained in detail below. The description starts by giving an overview with the help of the application's class diagram and then proceeds by explaining the most important classes in detail along with its properties (variables) and methods.

Note that for the sake of readability the description of variables and methods belonging to classes is kept simple and in a general way, such that methods only contain its basic signature without parameters which get passed along with the method call and furthermore variable and method return types are given in a general and not language-specific way (e.g. *integer* instead of the *C++*-specific type *int*).

8.1 Class Diagram

The *Class Diagram* illustrated in Fig. 8.1 gives an overview of the *ARVideo* application and its contained classes. As can be observed the classes are arranged by color to distinguish between classes belonging to the *Studierstube ES* framework (blue) and classes which make up the

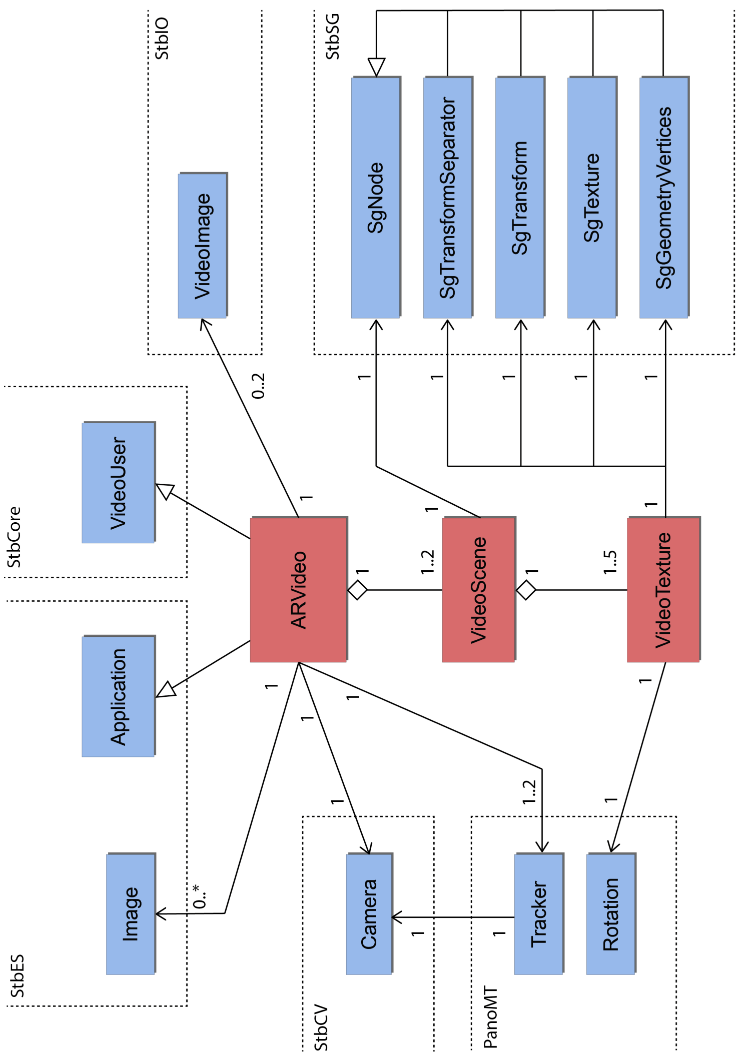


Figure 8.1: ARVideo Class Diagram

ARVideo application (light red, also compare Fig. 3.3) and which were developed in the course of this work. Furthermore, the classes belonging to the framework are outlined with dashed lines to indicate their membership to a single framework component, to further enhance understanding how the single parts play together. In the following the single classes and their connections are described in more detail.

8.2 ARVideo Class

The *ARVideo* class symbolizes the main application entry point. Its implementation consolidates data from all the other available classes and is therefore seen as the most important class to describe in detail. Basically the *ARVideo* class resembles a *Studierstube ES Application*, as illustrated in Fig. 3.3, respectively. As depicted in the *Class Diagram* in Fig. 8.1 *ARVideo* is a subclass of *Application*, such that the *Application* class is inherited by the *ARVideo* class. Further it can be observed that the *ARVideo* class is also a subclass of the *VideoUser* class. As indicated in Fig. 8.1 *Application* belongs to the *StbES* component of the *Studierstube ES* framework and *VideoUser* to the *StbCore* component, respectively. Inheriting *Application* therefore enables the *ARVideo* class to be managed by *StbES* as an executable application making use of all different components of the framework. The reason that also *VideoUser* is inherited by *ARVideo* is that this allows the forwarding of camera frames to the *ARVideo* class, which will be described in more detail below.

Basically the application flow can be abstracted such that the *ARVideo* application is created and initialized with all necessary data and then stays in a loop where the according update and render functions are called. Depending on user input different actions/functions are triggered in between which control the application's behavior.
In the following the *ARVideo* class is described in more detail regarding its attributes (i.e. member variables) and methods. Comparable member variables (e.g. icon images) are listed and explained only once, to not needlessly inflate the description of the implementation.

Further note, as mentioned before, that the *ARVideo* application is developed in such a way that it is possible to use it for both, the creation of the *reference panorama* (see Fig. 5.12) in the desktop mode and the *Online Video Replay* using an *iOS* device. In order to realize this behavior the class distinguishes between certain code parts using the macros *AUTHORING_MODE* and *LIVE_MODE*. With the help of these macros it is feasible to execute different code to compensate for the different requirements of the two modes.

Member variables

- *BTN_SIZE* : *integer*. Styling the size and position of UI icons is done with integer values.
- *camera* : *Camera*. Instance of the *Camera Class* described in Section 8.5.
- *tracker* : *Tracker*. Instance of the *Tracker Class* described in Section 8.6.

- *localize* : *boolean*. Variable which is set to *true* or *false* depending on if the localization in the panoramic image (i.e. the *Registration*, see Section 5.4.1) was successful. As long as it is *false* the current panoramic image gets matched against the loaded reference panorama (as illustrated in Fig. 5.12 and Fig. 5.13, respectively).

- *localizedRotationInPanorama* : *Rotation*. Instance of the *Rotation* class. Its value is set when localizing the current rotation within the current context was successful (i.e. the *Registration*, see Section 5.4.1).

- *isTracking* : *boolean*. Variable which is set to *true* or *false* depending on if the tracking mode is activated.

- *validTracking* : *boolean*. Variable which is set to *true* or *false* depending on if tracking the rotation in the current view is successful or not. Correctly replaying the augmentation overlays (see Section 5.4.2) relies on the valid tracking within the current context.

- *captureIndex* : *integer*. Index to keep track of captured video frames.

- *captureNext* : *boolean*. Used to determine if capturing of frames was successful/shall continue.

- *highRes* : *boolean*. Used to distinguish between different camera resolutions. Is set once in the beginning and depending on its state high or standard resolution icons are used.

- *showControls* : *boolean*. Used to determine if the UI controls shall be displayed or not (see UI description in Section 9.1 for reference).

- *playOn* : *Image*. Instance of the *Image Class*. described in Section 8.7. Displays the *Play* button in its active state, i.e. may be clicked.

- *playOff* : *Image*. Instance of the *Image Class*. Displays the *Play* button in its inactive state, i.e. it is not clickable. Buttons shall be inactive in case when there would not be any sense clicking the button in the current execution state of the application or to prevent wrong/unexpected or even not-desired behavior.

- *playOnDown* : *Image*. Instance of the *Image Class*. Displays the *Play* button in its state while it is being clicked, such that the user gets visual feedback that the button is actually clicked.

- *isPlayDown* : *boolean*. Flag which is *true* as long as the *Play* button is down, i.e. clicked. Is checked to set the desired behavior, i.e. activate playing of the video.

- *isPlaying* : *boolean*. Flag which is *true* as long as the replaying of the video is active. It is further used to update corresponding button states accordingly (i.e. activate *Decrease/Increase Speed* buttons).
- *isPlayButtonActive* : *boolean*. Flag which is *true* as long as the *Play* button is active, i.e. clickable. It is used to render the according UI icons.
- *playSpeed* : *integer*. Variable which indicates at which speed the video is being played back, i.e. regular speed, double speed, and so on. Its value gets regulated by the *Decrease and Increase Speed* buttons.
- *skip* : *boolean*. Flag which is used to realize a decreased playback speed of the video. It is set to true *true* if for a certain frame the overlay should not be rendered, such that it takes for example the double amount of incoming frames to replay the video (which indicates replaying the video at half speed). The value of the *skip* variable is directly bond to the *playSpeed* value, such that *playSpeed* values smaller than 0 have the consequence of setting the value of the *skip* variable to *true*.
- *increaseCounter* : *boolean*. Flag which is used to realize an increased playback speed of the video. It is set to true *true* if for a certain frame the rendered overlay index should be increased, such that it takes less incoming frames to replay the video. This means of course that certain overlay augmentations do not get rendered at all to imitate the faster replaying. The value of the *increaseCounter* variable is directly bond to the *playSpeed* value, such that *playSpeed* values bigger than 0 have the consequence of setting the value of the *skip* variable to *true*.
- *mouseDown* : *boolean*. Flag which is set to *true* if the system registers a mouse down event occurring on the screen/display. The associated pixel coordinates (see below) are used along with the flag's value to determine in which region (i.e. button) the mouse down event occurred such that according actions are carried out (e.g. mouse down event occurred within the *Play* button's pixel coordinates, which would set the before mentioned *boolean* variable *isPlayDown* to *true*.
- *mouseX* : *integer*. X coordinate of the associated mouse move event.
- *mouseY* : *integer*. Y coordinate of the associated mouse move event.
- *overlayIndex* : *integer*. Variable which indicates the index of the rendered overlay. At regular playback speed is increased by 1 for every frame. In case the playback speed is decreased the variable would only be increased by 1 for, e.g. every second frame, (due to

the reason that for every second frame the rendering of the overlay shall be skipped, see above). In case the playback speed is increased the variable would be increased by, e.g. 2 for every incoming video frame to simulate the faster replay of the overlays (bond to the *increaseCounter* variable, see above).

- *videoMainScene* : *SgNode*. *StbSG* node which represents the root node of the videos which shall be replayed. At runtime the according *StbSG* nodes belonging to the different video scenes get attached/detached to this node, such that the corresponding video scene is visible/not visible.
- *panoPreview* : *Image*. Preview of the created panoramic image which is visible on the screen while the user is recording the panorama.
- *frameCol* : *Image*. Colored representation of the current camera frame. Gets passed to the *tracker* instance.
- *frameGray* : *Image*. Gray level representation of the current camera frame. Gets passed to the *tracker* instance.
- *videoOneTextureActive* : *boolean*. Flag which is *true* as long as the first video scene is active. Together with the flag for the second video scene it is checked to determine which effect or layer could be enabled/disabled.
- *videoImage* : *VideoImage*. Instance of the *VideoImage Class* as described in Section 8.8.
- *fileNameFrame* : *String*. Holds the basic filename of frames which shall be read in with help of the *videoImage* variable.

Methods

- *ARVideo();* - *ARVideo* constructor, initializing all necessary member variable values, like the *boolean* fields described just above.
- *void init();* - Method which is called by the *StbES* component after creating the *ARVideo* instance. The framework expects an *init()* method in any *StbES Application*. Usually it shall be called only once and basically its purpose is to load configuration data, images and so on. It starts by registering the current *ARVideo* instance as a *VideoUser* making use of the corresponding *StbES* function call. Then the *Camera Calibration* file (determined in Section 3.1.7) gets loaded by calling the *load()* function on the *camera* instance. With the successfully loaded camera calibration data it is possible to initialize the *tracker* instance by setting its camera to said instance.

After that the *highRes* variable's value is set by determining if the current render target's resolution is bigger than the internal threshold distinguishing between standard and high resolution. Accessing the render target is possible due to the *StbES* component. After that all the images which represent the different icons (buttons which are used for controlling the app and the video playback and so on) are loaded into memory using an internal convenient function provided by the *StbES* component.

Finally in *LIVE_MODE* the configuration/input data for the *videoScene* instances is read in. It starts by reading the scene root node (using *StbSG* and the scene graph XML configuration file) and setting it to the before mentioned variable *videoMainScene*. If reading the scene root node was successful the method proceeds with creating two instances of the *VideoScene Class* (each *VideoScene* represents one of the video layers, see Section 5.4.2), by calling its constructor with the previously created *camera* instance and the base filename (of type String) for the *VideoTexture*s it should contain along with the filename's extension. In order to add/remove the created *VideoScene*s to/from the scene graph and hence to/from the rendering queue it is necessary to set the mentioned *videoMainScene* as the *VideoScene*'s parent node. Additionally, the overlay rotation information and texture information (position + offset) are loaded by calling the according methods on the *VideoScene* instances. Lastly the single *VideoTexture* nodes belonging to a *VideoScene* are read in from the XML configuration file by calling the *VideoScene* method *readVideoNode()* (described in the *VideoScene Class*).

Furthermore, in *AUTHORING_MODE* the *VideoImage* instances for reading in camera frames from the system as a stream (explained in method *vu_newFrame()* below) are created by calling the function *createVideoStreams()* which will be explained later on.

- *void update();* - Basically the entry point in the processing of a new video frame. This method gets called by the *StbES* system before any of the rendering takes place such that this method is used to alter variable values which are needed for the rendering. The *increaseCounter* variable is checked and the *playSpeed* variable is updated accordingly. Afterwards the preprocessing of the *VideoScene*s takes place. If the video mode is inactive, such that none of the two *VideoScene*s is being played at this moment the *VideoScene*s do get removed from the scene by detaching them from the main scene root. This is necessary such that no *VideoTexture* is rendered. If the video mode is on but not playing, i.e. the *Pause* mode is active, nothing needs to be updated. Yet if the video mode is on and the video is playing, the *VideoScene*s need to be updated such that the single *VideoTexture*s are being updated (texture + rotation). This is done by getting the current camera rotation from the *tracker* and updating the *VideoScene*s accordingly. Furthermore, the *VideoScene*s need to know which overlay to render, i.e. what is the current *overlayIndex*. Thereupon the *VideoScene*s *updateVideoScene()* functions are called. If the function returns *false* this means that the last *overlayIndex* was reached and is therefore set to 0 in order to restart the corresponding video.

- *void render3D();* - Method which would render 3D content and is called after the *update()* method. In this case no 3D content is available, hence nothing is done here.
- *void render2D();* - Method which is called by the system after the *render3D()* method. Basically all rendered content is 2D, which is why quite a lot of processing takes place in this method. To begin with the *StbES* system's renderer target is asked for the tracking status, such that if the tracking process is still valid. Which is important, because the rendered video content only makes sense if tracking and updating the position within the context is valid. The *showControls* flag is checked and if *true* the UI controls (buttons, slider control and so on) are drawn onto the screen. Depending on if the video is playing or not either the the slider control or the preview of the current panorama (which is the basis for updating the position) is drawn at the bottom of the screen. Furthermore, the buttons' states (active/inactive) implicate which icons (on/off) are to be rendered. All the images are drawn by calling the according *draw()* function of the *Image Class*, passing the image's position in pixel coordinates (defined manually along with variables like *BTN_SIZE*, whereas the pixel coordinates are increased to the double size if the *highRes* variable is set to *true* in order to compensate for the bigger screen).
- *void vu_newFrame();* - Like the methods above gets called by the system and passes the current camera frame to the *ARVideo Class*. The method contains vital parts which distinguish between the above mentioned *AUTHORING_MODE* and *LIVE_MODE*. Basically in both modes the current camera frame is prepared for passing it to the *tracker* instance and depending on which task should be executed either the *tracker*'s *localize()* or *update()* method is called with the prepared camera frame (although in *AUTHORING_MODE* the *localize* feature is kind of obsolete and not called normally).

The need for distinguishing between the two modes arises from the fact that in the *AUTHORING_MODE* the goal is to create the *reference panorama* out of the previously created video frames and in the *LIVE_MODE* the actual replay of the augmented material takes place. As explained earlier the *Foreground Segmentation* outputs these camera frames along with the corresponding alpha channels. These camera frames now represent the actual frames which are needed to create the *reference panorama*. This means instead of mapping the live camera frames which would be delivered by the system, those stored frames are mapped into the panoramic image using the adapted *Panorama Mapping and Tracking* technique, which handles the alpha channels to omit the foreground object in the frames. In order to pass the frame and its alpha channel to the *tracker* they need to be read in making use of the previously mentioned instance of the *VideoImage Class*. The *VideoImage Class* simplifies reading a sequence of frames as a stream from the filesystem and is provided by the *StbIO* component of the *Studierstube ES* framework. Note that actually two different *VideoImage* instances are needed as on the one hand the regular camera frames need to be read in and on the other hand the corresponding alpha channel images at the same frame index. Once the frames are loaded the *tracker*'s update function is called to map the current frame into the panoramic image. To end the *AUTHORING_MODE*-specific part the *VideoImage* stream indices are increased by one so that in the next call of *void vu_newFrame()* the correct frame is read in. In case there are no more frames in the stream the single rotations of all frames which were mapped into the *reference panorama* are stored. These rotations are kept track off by the *tracker* when the corresponding frames are mapped into the panoramic image and along with the final *reference panorama* will later be used as part of the input for the *Online Video Replay* as depicted in Fig. 5.13.

In case the *LIVE_MODE* is active the processing differs; instead of reading stored frames from the filesystem the real live camera frames, which are directly passed along when the method is called by the *StbES* system, are used. This means the frames are passed to the *tracker* to create a panorama out of the current view on the scene. At first though to begin with, the registration task has to be carried out. Which means that as long as the current view onto the scene could not be localized, the *localize()* method is called in addition to the *tracker*'s *update()* method, as explained in detail in Section 5.4.1.

Only if the registration was successful the method proceeds in further calls in a way that it only updates the current position within the context. To end the *LIVE_MODE*-specific part the values for *increaseCounter* and *skip* are set depending on the corresponding *playSpeed* value, as remarked above.

- *void mouseDown();* - Is also called by the system and delivers the horizontal and vertical pixel coordinates of the mouse down event on the screen. Basically it checks if the mouse down event's position occurred somewhere inside of a button's area (e.g. *Play* button) or similar (e.g. slider control). If so the corresponding boolean flag (e.g. *isPlayDown* is set to *true*).

- *void mouseUp();* - Is triggered by the system as soon as the mouse down event is over. Based on the boolean flags set in the *mouseDown()* method it controls the application's behavior by manipulating even more boolean flags (e.g. if *isPlaying* was *false* before and *isPlayingDown* is *true* the variable *isPlaying* should be set to *true* in this case to start the playing of the video and should be set to *false* if *isPlayingDown* is *true* and *isPlaying* was *true* before, which would mean the video *was* playing until now and shall be paused).

- *void mouseMove();* - Another method called by the system and as the name suggests as long as the mouse is being moved (and pressed). Basically it just stores the actual pixel coordinates in the before mentioned member variables *mouseX* and *mouseY*. These values are used for example to update the slider control's position and therefore help to navigate through the overlay video scene.

- *boolean inButton();* - In order to determine if a mouse down event occurred in one of the UI's control button this method gets called. The method checks if the pixel coordinates of a mouse down event are within the button's outer borders and if so returns *true*.

- *void createVideoStreams();* - As noted in the *init()* method the *createVideoStreams()* method is responsible for creating the *VideoImage* instances which are used to read camera frames from the filesystem as sort of a input stream. The base filenames for the “regular” video frame and the alpha channel frames are given as strings and the *VideoImage* instances are then created and its *init()* method is called. More details about this in Section 8.8.

8.3 VideoScene Class

An instance of the *VideoScene Class* essentially represents a video layer of the *ARVideo* application; as explained in Section 9.2. The implemented *Prototype*, presented in Chapter 9, contains two such layers, which can be played simultaneously or alternately. The class encapsulates all necessary information of a *VideoScene* to simplify adding/removing it from the “main scene” (remember that the presented system uses a scene graph to add/remove the augmentations to/from the rendering queue). In the following the member variables and methods of the class are described in more detail.

Member variables

- *cam_inv* : *Matrix*. Holds the inverse camera matrix (determined through the *Camera Calibration* explained in Section 3.1.7), which is consulted in case an instance of the *VideoTexture Class* needs to be updated to render it at the accurate position. The variable gets set upon creating an instance of the class (see method *init()* in the *ARVideo Class*) by passing the according parameter. *Matrix* is an intern representation (3x3 float) of a matrix in mathematical sense.

- *frames* : *integer*. Holds the number of “frames” the *VideoScene* takes to be played completely. Is determined by setting it to the number of *rotations* which is also the number of overlays.

- *parentScene* : *SgNode*. Represents the root node of the main scene and is set through the according method *setParentScene()*.

- *rotations* : *Vector<Rotation>*. Vector which holds the rotation information per overlay (compare Fig. 5.13). Is filled in the beginning by calling the method *loadRotations()*.

- *texturePositions* : *Vector<Vec4F>*. Vector which holds the necessary information (position + offset) per overlay (compare Fig. 5.13 and Fig. 5.10). Is filled in the beginning by calling the method *loadTexturePositions()*. A single entry is of type *Vec4F* which is a simple intern representation of a *vector* containing 4 float values.

- *textureFileNameBase* : *String*. Contains the base name of the texture files to be loaded. Together with the overlay index and the *textureFileNameEx* the according overlay file is loaded in the method *updateTextureFile()*. Is set in the constructor.

- *textureFileNameExt* : *String*. Contains the extension of the texture files. Is set in the constructor.

- *idxCalc* : *character Array*. Used to transform the integer value of the current overlay index into “String” format, to be able to append it to the base file name.

- *rotCamera* : *Rotation*. Holds the current camera rotation and is used to update the texture rotation in the method *updateTextureRotation()*. Is set and updated for every camera frame through the method *setRotationCamera()*.

- *frameNr* : *integer*. Index of the overlay which should be rendered. Is updated through the method *setFrameNr()*. This is necessary, because as described in the *ARVideo Class* it is possible that the overlay which should be rendered needs to be updated faster than the actual camera frames, in order to realize the faster replay of the video. Based on this index the indices for the textures which make up the *Flash Trail Effect* as described in Section 9.2 are computed.

- *videoTextures* : *VideoTexture Array*. Contains the *VideoTexture* instances of a *VideoScene*. Every *VideoTexture* resembles an overlay which should be rendered if required. The first item in the array acts as the regular overlay if the video is played without any effect, whereas the remaining overlays in the array serve for realizing for example the *Flash Trail Effect*. The need for different instances of the *VideoTexture Class* arises from the fact that every single overlay needs to be updated with the correct rotation and rendered independently from the others.

Methods

- *VideoScene();* - *VideoScene* constructor. The mentioned *textureFileNameBase*, *textureFileNameExt* and the *cam_inv* fields are set to the values which get passed with the constructor call. Furthermore, the *VideoTexture* instances are created and the *videoTextures* array is filled.

- *void readVideoNode();* - Method which is called on a *VideoScene* instance to read all necessary information from the XML configuration file regarding a single *VideoTexture* node.

- *void loadRotations();* - Called once after creating a *VideoScene* instance to read the rotation information per overlay from the corresponding text file.

- *void loadTexturePositions();* - Called once after creating a *VideoScene* instance to read the texture information (position + offset) per overlay from the corresponding text file.

- *Rotation getRotationForFrame();* - Returns the rotation associated with a texture file by providing the according index.

- *Vec4F getTextureDetailsForFrame();* - Returns the texture information associated with a texture file by providing the according index.

- *boolean addTextureNode();* - Tries to add a *VideoTexture* instance to the parent scene, in order to add this *VideoTexture* to the rendering queue. If adding the node was successful returns *true*, otherwise *false*. This is necessary when for example the video should be started and the "main" overlay should be added, or indeed as well when the *Flash Trail Effect* is activated and the remaining overlays should be added to the scene.

- *boolean removeTextureNode();* - Removes a *VideoTexture* instance from the parent scene, in order to remove this *VideoTexture* from the rendering queue; for example when the

Flash Trail Effect is deactivated or the video is stopped at all (no overlay should be visible any more). Returns *true* if successful.

- *boolean checkTextureState();* - Checks the state of a *VideoTexture* instance, i.e. if it is active and hence already added to the main scene, or if it needs to be activated.
- *void updateFadeOutTextures();* - Method which loops through all *VideoTexture* instances which are used for the *Flash Trail Effect)* (that is all but the first one) and depending on its state updates the according texture.
- *void updateFadeOutTexture();* - Actually updates all texture-specific data like the image file which should be loaded onto this *VideoTexture* instance, the associated geometry (determined through its coordinates) and the rotation which should be applied to the texture at this frame depending on the associated own rotation (loaded from the text file, see method *loadRotations()*) and the current value of the field *rotCamera*. The corresponding method calls are described below.
- *void updateTextureGeometryVertices();* - Updates the texture's geometry vertices in a way that the original pixel coordinates of the segmented overlay at the given index are transformed into screen coordinates of the actual render target, as explained in Section 5.4.2.
- *void updateTextureRotation();* - Updates the texture's rotation at the given index, as explained in Section 5.4.2.
- *bool updateTextureFile();* - Loads the overlay file (an image) which is associated with the given index onto the *VideoTexture*. Returns *true* if successful, and *false* if for example the image file was not found.
- *bool updateVideoScene();* - Is usually called once per incoming camera frame in the *ARVideo Class* and is responsible for updating the regular *VideoTexture* of this scene - if active - and furthermore depending on the process of the video (frame number) and activated/deactivated effect sets boolean flags which are passed to the *updateFadeOutTextures();* method in case the remaining textures need to be added/removed or simply updated (image, rotation, geometry). Additionally, if the scene is not active at all, but has been before, the scene is removed completely by calling the method *removeVideoScene()*.
- *void removeVideoScene();* - Simply deactivates all *VideoTexture* instances belonging to this scene, such that in the next rendering step no textures are rendered to the screen.
- *void setParentScene();* - As mentioned sets the value of the field *parentScene*.

- *void setRotationCamera();* - As mentioned sets the value of the field *rotCamera*.
- *void setFrameNr();* - As mentioned sets the value of the field *frameNr*.
- *integer getNumberOfFrames()* - Returns the number of total “frames” this video scene contains.

8.4 VideoTexture Class

The *VideoTexture Class* encapsulates all data which makes up a single overlay. It contains the scene graph nodes which are necessary to process a single overlay independently from the others, such as a texture node which holds the overlay image, a transform node which is responsible for correctly rotating the overlay and a geometry vertices node which serves for accurate positioning of the overlay. The transform separator node makes it possible to independently apply a transformation; such as the one computed in Equation (5.1). Otherwise applying a transformation like the one mentioned would result in applying this transformation on all available content which would result in undesired behavior. In the following the member variables and methods of the class are described in more detail.

Member variables

- *depth* : *float*. Holds the texture’s screen depth (compare value z in Equation (5.2)).
- *active* : *boolean*. Flag which is set to the texture’s state (active/inactive).
- *transformSeparator* : *SgTransformSeparator*. Scene graph node and as illustrated in the *Class Diagram* is a subclass of *SgNode*. As mentioned above functions as the node to separate the applied transformation from the remaining scene.
- *transform* : *SgTransform*. Scene graph node and as illustrated in the *Class Diagram* is a subclass of *SgNode*. Holds the transformation to apply in order to correctly render the overlay.
- *texture* : *SgTexture*. Scene graph node and as illustrated in the *Class Diagram* is a subclass of *SgNode*. Holds the texture which loads the overlay image and which is finally being rendered through *OpenGL ES*.
- *geometryVertices* : *SgGeometryVertices*. Scene graph node and as illustrated in the *Class Diagram* is a subclass of *SgNode*. Contains the geometry vertices which make up the screen coordinates which are computed using Equation (5.2).

Methods

- *VideoTexture();* - Constructor which creates an instance of the class. In addition, initializes the value of the field *active*.
- *virtual VideoTexture();* - *VideoTexture* destructor. In this case nothing needs to be explicitly deleted in case an instance is destroyed.
- *boolean setTransformSeparator();* - Sets the value of the field *transformSeparator*. Returns *true* if the node is not *NULL*.
- *SgTransformSeparator getTransformSeparator();* - Returns the field *transformSeparator*.
- *boolean setTransform();* - Sets the value of the field *transform*. Returns *true* if the node is not *NULL*.
- *SgTransform getTransform();* - Returns the field *transform*.
- *boolean setTexture();* - Sets the value of the field *texture*. Returns *true* if the node is not *NULL*.
- *SgTexture getTexture();* - Returns the field *texture*.
- *boolean setGeometryVertices();* - Sets the value of the field *geometryVertices*. Returns *true* if the node is not *NULL*.
- *SgGeometryVertices getGeometryVertices();* - Returns the field *geometryVertices*.
- *void setActive();* - Sets the value of the field *active*.
- *boolean isActive();* - Returns the boolean value of the field *active*.
- *void setDepth();* - Sets the value of the field *depth*.
- *float getDepth();* - Returns the float value of the field *depth*.

8.5 Camera Class

The *Camera Class* is part of the *StbCV* component and is made use of to work with the camera calibration data; determined in Section 3.1.7. According to its documentation the class stores the intrinsic parameters in fixed- and floating point values. Hence, it is able to project 3D points into the screen and convert between undistorted and distorted 2D coordinates.

8.6 Tracker Class

The *Tracker Class* is part of the *PanoMT* component and plays an important role as it is responsible for tracking the camera motion within the user's context and updating the camera view (mapping pixels into the panorama) as described in the method *vu_newFrame()* of the *ARVideo Class* and in Section 5.4.1.

8.7 Image Class

The *Image Class* is part of the *StbES* component and according to the class documentation represents the base class for images that can be rendered into an *OpenGL (ES)* frame buffer. The implementation contains several convenient functions for pixel/image manipulation such as flipping or scaling the image. Instances of the *Image Class* are widely used throughout the presented system (control buttons, panorama preview and so on).

8.8 VideoImage Class

The *VideoImage Class* serves for reading in image frames from the filesystem as a "stream", as mentioned in Section 8.2. This is needed in case the *ARVideo* application is executed in *AUTHORING_MODE* which was also explained above. The *VideoImage* class provides convenient methods for returning the frame for a given index and its properties like width, height and so on and furthermore also has methods which guarantee that the advancing within the stream is secure, i.e. the next frame which is about to read in and returned really exists.

CHAPTER 9

Prototype

Video augmentations can be used for a wide range of applications covering areas such as entertainment or edutainment if integrated into outdoor AR applications such as browser systems. In the context of this work a prototype of a mobile video editing AR application was developed. Inspired by modern tools proposed in desktop video editing applications, the implemented system focused on some of their major features:

- video layers,
- video playback control and
- video effects.

In the following an overview of the user interface and the post-effects which are currently implemented is given. Additionally, a user study was conducted to get first-hand-user-experience by domain experts in the field of skateboarding and video editing. This is described in detail in Chapter 10.

9.1 User Interface

The interface of the prototype is inspired by the design of graphical user interfaces generally found in video editing tools. Three different groups of functions were created; distributed around the screen that can be accessed with touch screen operations. The control groups of the implemented user interface are (illustrated in Fig. 9.1):

1. the video control group,

2. the video layer group and

3. the video effects group.

The menu options can easily be hidden by touching the screen at any position where no menu option is found or in case the menu options are not pressed for a certain amount of time (e.g. 3 seconds) the menu disappears to give more space to the video, but will be redisplayed as soon as the user touches the screen.

The functions in the video control group consist of the playback control found in most video players:

- play control buttons,
- time slider and
- speed buttons.

With the play/pause button the user can start the video from its initial position and pause at any time. Triggering the stop button results in the video disappearing from the scene at all. When the video is playing the slider at the bottom can be utilized to interactively control the video's progress, i.e. rewind or forward a few frames. The speed buttons' functionality is simply to decrease or increase the speed at which the video is being played.

The video layer group provides access to the different video sequences which might be accessible at a certain location (i.e. public area, park, building, ...), organized in different depth layers. The end-user can control and activate/deactivate these different layers, which can be played independently of each other or simultaneously.

The last group of items, the video effects, trigger real-time video effects that can be applied to the different video sequences. Each effect can be switched on or off and the effects can be combined with each other. More details about the layers and the post effects are given in the next section.

9.2 Post Effects and Layers

Applying visual effects is an important part of video post-productions. Effects are used to highlight actions and create views that are impossible in the real world, such as slow motion or

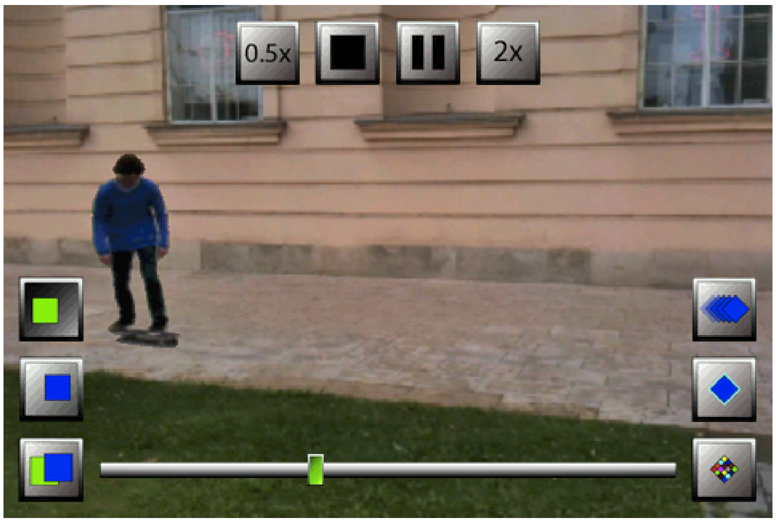


Figure 9.1: Screenshot of the prototype showing a video augmentation together with the implemented interface with the control groups for video playback (top and bottom), video layers (left side) and video effects (right side).

highlighting of elements within the video. Normally these effects are applied to the video material in a rendering step that is carried out in an offline manner [27]. Because of the nature of the presented approach it is possible to perform a wide variety of these video effects in real-time on a mobile device without the need of pre-rendering the video content. In the context of this work space-time visual effects such as multi-exposure effects, open flash and flash-trail effects were explored. Multi-exposure effects simulate the behavior of a multi exposure film where several images are visible at the same time. This behavior can easily be simulated for cameras with a fixed viewpoint by augmenting several frames (or in this case overlays) of the existent video at the same time. This results in having the subject appearing several times within the current view, such as in a multiple exposure image.

An extension of this effect is the Flash trail effect. This effect also allows seeing multiple instances of the same subject yet the visibility depends on the time passed by (see Fig. 9.2). This effect supports a better understanding of the motion in the recorded video. The Flash trail effect was implemented by blending in past frames of the augmented video with increasing amount of transparency, which was altered working with *OpenGL ES* shaders[1]. Thereby the strength of the transparency and the interval between the frames can be freely adjusted.

[1]http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf

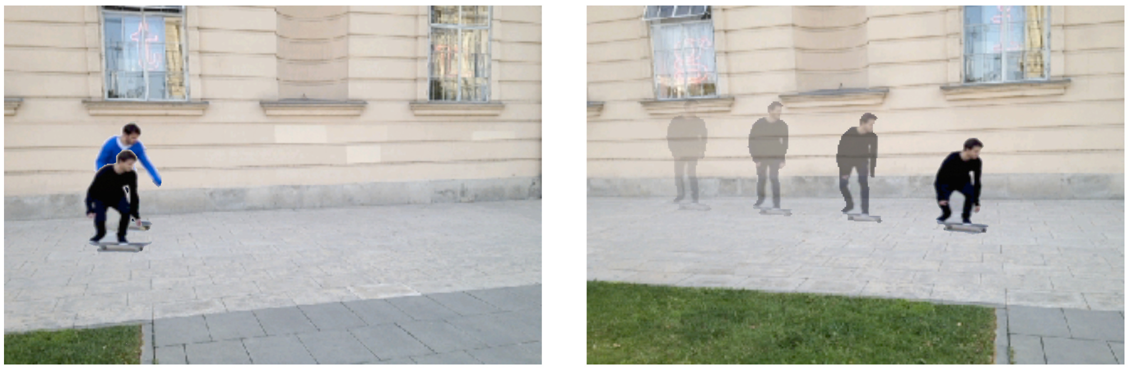


Figure 9.2: Examples of layers and an example for realized post effects as used in the Skateboard Tutor Application; captured from an iPhone 4S. (Left) Playing back two video augmentation layers allows the comparison of the riders' performance. (Right) Flash-trail effects visualize the path and the motion within the video.

The presented prototype allows to play back more than one video at the same time by still allowing a seamless integration into the environment (shown in Fig. 9.2 on the left). This allows it to compare actions that were performed at the same place but at a different point of time by integrating them into one view, thus bridging time constraints. Each video in the system corresponds to a video layer and the user is able to switch between these layers or play them simultaneously.
The applying of video effects or video layers do not require any preprocessing but is carried out on the device while playing back the video. Hence, effects and layers can be combined or switched off/on demand, which supports the immersive user experience.

Part III

Results

CHAPTER 10

User Study

To evaluate the presented prototype a preliminary user study was conducted, such that it was possible to gather first user feedback on the given technique as well as to identify flaws and improvements or to get additional ideas for the applicability of the ARVideo compositing system. In order to evaluate the proposed *Prototype* it was demonstrated to end-users as part of a *Skateboard Tutor Application*. All details about the application and the user study and its results are given below.

10.1 Skateboard Tutor Application

Skateboard videos are a good representative of dynamic real world content naturally evolving in our real world environment. The different ranges of maneuver (tricks) performed with a skateboard are largely bond to the environment and location through natural or artificial obstacles and ramps. Skateboard videos make also use of a variety of camera shooting techniques (perspectives, movement, optics) due to the dynamic of the skateboarder evolving in the real environment. On the other hand, Tutorial/How-To videos represent a large part of YouTube videos [42] today, confirming the potential of this format for skills or competences learning. Skateboard tutorials (>30.000 hits on YouTube) serve therefore as a good application of the presented technique.

The *Skateboard Tutor Application* allows recording skateboard tricks that can be shared with other users for demonstration and learning purposes. The application can be used to overlay the prerecorded video content (extracted skateboarders) in place to replay and experience the tricks and actions performed by another user (or from a previous day) in the correct context

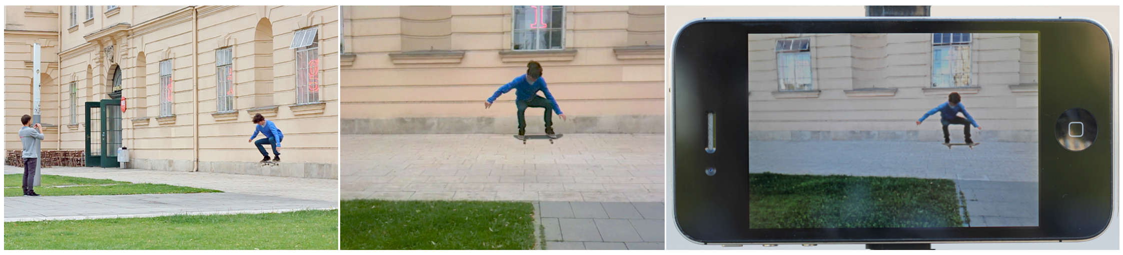


Figure 10.1: Application scenario as used during the user study. (Left) Skateboarder was recorded with a mobile phone while performing his actions. (Middle) Frame of the recorded video sequence. (Right) The same action as augmented within the skateboard tutor application as captured from an iPhone 4S.

(as indicated in Fig. 10.1). It can support the learning process as online skateboard videos, generally recorded with fish-eye lens, can give a distorted perception of the skateboarder in the real environment. The test skateboard videos were recorded with standard smartphones and are processed using the presented approach of situated video compositing for AR. Furthermore, the proposed post effects and layers are included in the test application. The layer approach allows recording skateboard maneuver on the fly, which can later be played back in parallel with other stored maneuver for comparison (e.g. speed, height of jumps). The flash-trail effect can be used to highlight the motion and the path of the rider.

10.2 Scenario And Setting

Producers of skateboard videos are usually also consumers, leveraging the possibility to collect feedback for both, the creation of video augmentations and experiencing said video augmentations. Consequently the evaluation of the *Prototype* within the *Skateboard Tutor Application* was seen as an adequate use case scenario to gain valuable feedback. Thus to conduct the user study skilled skateboarders (domain experts) were invited, where each of them had gained experience in creating skateboard videos or tutorials already and who had published their videos online via popular sharing platforms.

The main objective regarding the user evaluation was to identify the usefulness and applicability of the presented approach as well as the usability of the created prototype. In total there were 5 expert users with >7 years of skateboarding experience (all male, 25-28 years), all of them were involved in producing skateboard videos, some produced videos for marketing. All considered themselves as not overly tech-savvy. Two had minimal knowledge about augmented reality, none had any experience with any kind of AR application beforehand. One participant

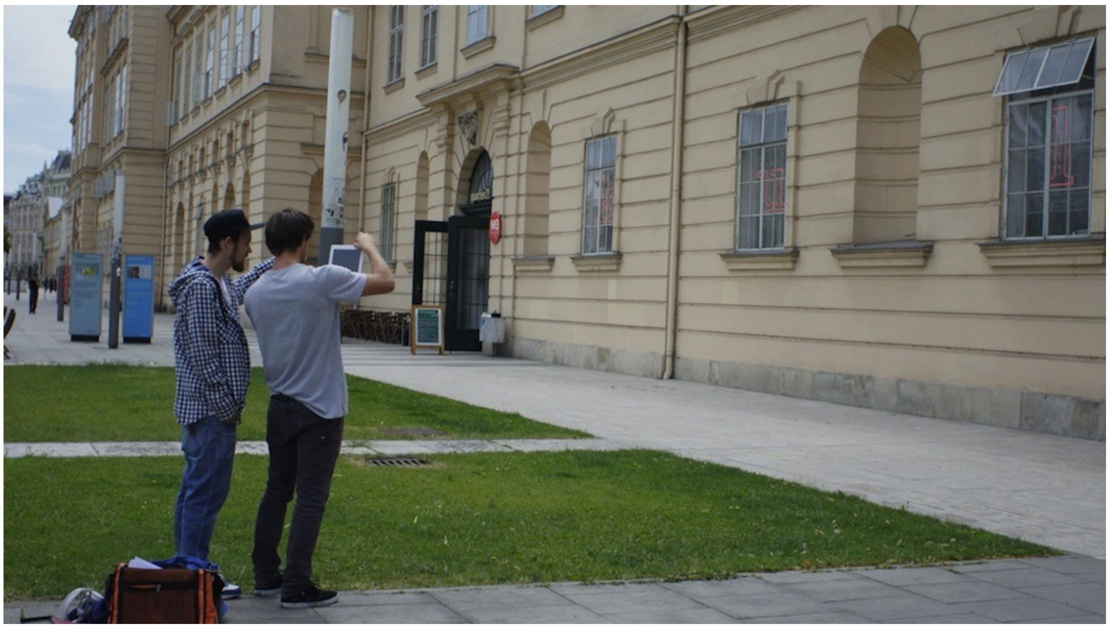


Figure 10.2: Evaluation of the prototype with domain experts using an Apple iPad2.

stated to be not very familiar with the usage of mobile devices such as smartphones as he didn't own one and restricted his usage of mobile phones to place calls or write messages.

10.3 Procedure

All participants had the chance to get hands-on experience with the prototype as it was demonstrated on both an iPhone 3GS and an iPad2 (depicted in Fig. 10.2). After introducing the project, the participants were given a short demonstration of the application, showing the different features. They were able to test the integrated effects as well as to try the video layers by playing two video layers that were augmented at the same time. Two participants were selected to create their own skateboard videos which were later augmented, while all other users only had the chance to experience the augmented videos. After the participants finished trying out the prototype they were asked a series of questions as part of a semi-structured interview.

10.4 User Study Results

In the interview all participants confirmed that the prototype was easy to use. Regarding the comfort factor with the device and interacting with the application, only one user didn't feel really comfortable (who was the participant not experienced with smartphones). The participants were also asked about the social aspect of using the application outdoor in a busy area; whereas all participants replied to be really comfortable on this aspect. All users commented that the presented system was easy to learn and the current interface was also well received.

Three of the five participants said that they really enjoyed the freedom of having control of the camera orientation during playback, as it is not relying on the recording camera orientation (explained in Section 5.4.2). In general the users highlighted positively the possibility of playing several videos/layers at the same time, that are overlaid in parallel. It was described as a really useful mechanism for comparing videos, e.g. comparing their own runs with the tutorial video to detect differences. Furthermore, the appliance of the flash-trail effect also gained positive feedback such that it seemed to be useful for studying ”the line” a rider skates and thus further assisting in the comparison between different videos/actions.

When asked about the general applicability and the usefulness of experiencing *video augmentations in place* the participants commonly saw great potential for the system regarding the usage in other application areas. However, two of them pointed out during the interview that the users have to visit the place, which makes more sense in certain specific cases. Both of them stated that they therefore generally see it more as a gadget as they could not think of other convincing use-cases at that point of time. However, when they were presented with other possible use cases (city guides, parades/events within the city) at the end of the interview they noted that they see also potential in these kinds of applications yet stated the need to experience it for a more reliable answer.

The last part of the interview focused on the visual quality of the technique, in term of spatial and visual integration. Two participants had the feeling that the scene and the rider were 3D and giving a sense of ”authenticity”, one perceived the rider to be 2D but the scene to be 3D, while the remaining two participants stated that it was all overlaid in 2D. They all commented that the movement of the augmented skateboarders within the scene was very realistic. Even after being explicitly asked they could not remember to have seen any drifting between the augmentation and the background. However, when asked about the seamless visual integration, mixed answers were received. The participants stated that sometimes the skateboarder seemed to appear differently to e.g. people walking by, as the augmented overlays were too dark or incorrectly lit in the actual context. Two participants also noticed small segmentation errors (e.g. parts of the skateboard were disappearing in a couple of video frames).

The two participants that generated their own augmentation videos said that the system in total is rather easy to use (without any previous knowledge about how to proceed) and the steps which were required are acceptable for the generated outcome. When asked about constraints in the camera motion during shooting - limited to rotational movements of the camera (see Section 5.4.1) - they said that it is likely to be acceptable in most cases. They explained that a huge majority of the people is making short videos with smartphone devices from a single point of view. One of the participants observed (translated to english): ”The given constraints fit the medium, as I think the majority of the short online videos were shot in this [constrained] way”. From this it follows that the presented system is likely to fulfill all criteria which laypersons usually come across recording skateboarding videos. Finally, during the open questions one participant proposed the possible use of the video compositing system as a ”mobile blue screen”, which would allow users to capture objects and scenes and assemble them together using the layer view.

CHAPTER 11

Discussion/Conclusion

11.1 Discussion

Overall the evaluation of the prototype of the proposed system with the chosen skateboard experts showed that the presented approach has advantages over existing mobile video applications (shooting, video effects, playback). However, the final outcome and the usefulness strongly depend on the use case. Even though all of the user study participants were not tech-savvy they had no problems to learn and handle the prototype application.

A major limitation of the prototype pointed out by the participants was the visual quality of the overlays. Even though the ratings were above average the users complained about the lack of visual coherence: The video augmentation looked different from the current environment. In the user study's case this was mostly caused by cloudy weather conditions during recording time resulting in low contrast actors, while it was mostly sunny during the playback of the video augmentations. This could be treated in future versions of the prototype by implementing an adaptive visual coherence. The basic idea is to compare the background panorama of the video with the current environment to adjust the video augmentation in terms of contrast and color.

Another problem was that the segmentation sometimes was not accurate enough, especially if applied to a well-structured background as required for vision-based registration. However, more sophisticated segmentation algorithms and better algorithms for tracking the segmented objects exist yet these would require more expensive computation or GPU implementations and need to be investigated in the context of this work. Especially since the segmentation as used in the current system has inverse requirements compared to the vision-based registration presented

in this context: A less structured background achieves in general significantly better results in foreground-background segmentation, while it poses a hard problem when used to register the augmentation based on the background information.
Despite these drawbacks, the application showed that augmented video could be an interesting element especially as video content is often easier to create than 3D content, making the demonstrated video compositing approach interesting for many applications.

Professional applications can benefit from video augmentations as realized in this work. Augmented reality-based tourist guides could display more interactive content e.g. by capturing the guide for later replay. Furthermore, authoring such content is less demanding than creating dynamic 3D content. This allows to easily create in-situ narratives similar to the concept of situated documentaries presented by Höllerer et al., 1999. Many augmented reality applications can gain value from the simplicity of creating video augmentations employing the proposed approach, allowing laypersons to create interactive content and share it with friends. This enables the creation of videos of certain events (e.g. parades, street artists etc.) and play-back in place at a different time.

Separating the constraints which occur during shooting and the ones present at replaying the video content can be explored further. Cinematography components such as camera type, camera movement, visual style of the image, location and its content are some examples of elements that can be altered, modified or ”warped” between the record and replay. One could imagine to record a cyclist of the Tour de France with a rolling camera technique and replay the recorded scene fixed in another location. Further, real-time montage with live video, online content and collaborative editing might leverage the full potential of mobile AR.

11.2 Conclusion

In this thesis an approach for in-situ compositing of video content in mobile Augmented Reality was presented. It was shown how to create and process video files for the usage in mobile outdoor AR as well as how to register these video scenes precisely in the user's environment using a panorama-based tracking approach. Even though the approach is constrained to rotational movements of the camera due to the usage of a panoramic representation of the environment, it could be applied to many existing outdoor AR applications, as this motion pattern is common for using AR browsers, as well as for shooting short videos. Possible use cases of video footage in future mobile AR applications were proposed. Furthermore, it has been shown how to interactively experience videos of past events in outdoor AR environments.

To do so an example application ("skateboard tutor") was developed and demonstrated. The prototype of the mentioned example application allows experiencing skateboard tricks and actions recorded beforehand that are then augmented in-place and displayed at interactive frame rates on mobile phones.
Future work should target better segmentation algorithms and an improved visual coherence between the overlay and the augmented environment. Porting the offline processing of the video material to a mobile environment could be another future step.

Bibliography

[1] OpenCV. `http://opencv.willowgarage.com/wiki`.

[2] OpenGL ES. `http://www.khronos.org/opengles`.

[3] Qt framework. `http://qt.nokia.com`.

[4] Studierstube ES. `http://handheldar.icg.tugraz.at/stbes.php`.

[5] The Grid Compass. `http://home.total.net/~hrothgar/museum/Compass/`.

[6] Clemens Arth and Manfred Klopschitz. Real-time self-localization from panoramic images on mobile devices. *IEEE International Symposium on Mixed and Augmented Reality 2011 Science and Technolgy Proceedings*, pages 37–46, October 2011.

[7] Ronald T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments 6*, pages 355–385, 1997.

[8] Luca Ballan, Gabriel J. Brostow, Jens Puwein, and Marc Pollefeys. Unstructured video-based rendering: Interactive Exploration of Casually Captured Videos. In *ACM SIGGRAPH 2010*, volume 29, New York, New York, USA, July 2010. ACM Press.

[9] Jean-Yves Bouguet. Pyramidal Implementation of the Lucas Kanade Feature Tracker, Description of the algorithm, 2000.

[10] Yuri Boykov and Marie-Pierre Jolly. Interactive Graph Cuts for Optimal Boundary & Region Segmentation of Objects in N-D Images. *In Proceedings of Interaction Conference on Computer Vision*, pages 105–112, 2001.

[11] T.P. Caudell and D.W. Mizell. Augmented reality: an application of heads-up display technology to manual manufacturing processes. *In Proceedings of 1992 IEEE Hawaii International Conference on Systems Sciences*, pages 659–669, 1992.

[12] Stephen DiVerdi, Jason Wither, and Tobias Höllerer. Envisor: Online environment map construction for mixed reality. *In Proceedings of IEEE VR 2008 (10th International Conference on Virtual Reality)*, 2008.

[13] Gunnar Farnebäck. Two-Frame Motion Estimation Based on Polynomial Expansion. *In Proceedings of the 13th Scandinavian Conference on Image Analysis*, pages 363–370, 2003.

[14] Steven Feiner, Blair MacIntyre, Tobias Höllerer, and Anthony Webster. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. *Personal Technologies*, 1(4):208–217, December 1997.

[15] George W. Fitzmaurice. Situated Info Spaces. *Communications of the ACM*, 36(7), 1993.

[16] Benjamin R. Fransen, Evan V. Herbst, Anthony Harrison, William Adams, and J. Gregory Trafton. Real-time face and object tracking. *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2483–2488, October 2009.

[17] Jens Grubert, Tobias Langlotz, and Raphael Grasset. Augmented Reality Browser Survey. Technical report, Graz University of Technology, 2012.

[18] T. Guan and C. Wang. Registration Based on Scene Recognition and Natural Features Tracking Techniques for Wide-Area Augmented Reality Systems. *IEEE Transactions on Multimedia*, 11(8):1393–1406, December 2009.

[19] Tobias Höllerer, Steven Feiner, and John Pavlik. Situated Documentaries: Embedding Multimedia Presentations in the Real World. In *Proceedings of the 3rd IEEE International Symposium on Wearable Computers (ISWC '99)*, pages 79–86, October 1999.

[20] Tobias Höllerer, Steven Feiner, Tachio Terauchi, Gus Rashid, and Drexel Hallaway. Exploring MARS: developing indoor and outdoor user interfaces to a mobile augmented reality system. *Computer & Graphics*, 23(August):779–785, 1999.

[21] M. Kalkusch, T. Lidy, N. Knapp, G. Reitmayr, H. Kaufmann, and D. Schmalstieg. Structured visual markers for indoor pathfinding. *The First IEEE International Workshop Agumented Reality Toolkit*.

[22] Georg Klein and David Murray. Parallel Tracking and Mapping on a camera phone. *ISMAR '09 Proceedings of the 2009 8th IEEE International Symposium on Mixed and Augmented Reality*, pages 83–86, October 2009.

[23] Vladimir Kolmogorov and Ramin Zabih. What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(2):147–159, March 2004.

[24] Rob Kooper and Blair MacIntyre. Browsing the Real WWW - Maintaining Awareness of Virtual Information in an AR Information Space. *International Journal of Human-Computer Interaction*, 16(3):425–446, 2003.

[25] Robert Laganière. *OpenCV 2 Computer Vision Application Programming Cookbook*. Packt Publishing, 2011.

[26] Tobias Langlotz, Mathäus Zingerle, Raphael Grasset, Hannes Kaufmann, and Gerhard Reitmayr. AR Record&Replay: Situated Compositing of Video Content in Mobile Augmented Reality. *Accepted for ACM OZCHI, 2012*.

[27] Christian Linz, Christian Lipski, Lorenz Rogge, Christian Theobalt, and Marcus Magnor. Space-time visual effects as a post-production process. In *Proceedings of the 1st international workshop on 3D video processing - 3DVP '10*, New York, New York, USA, October 2010. ACM Press.

[28] Blair MacIntyre, Jay David Bolter, Jeannie Vaughn, Brendan Hannigan, Maribeth Gandy, Emanuel Moreno, Markus Haas, Sin-Hwa Kang, David Krum, and Stephen Voida. Three Angry Men: An Augmented-Reality Experiment In Point-Of-View Drama. *In Proceedings of TIDSE 2003*, pages 24 – 26, 2003.

[29] Blair MacIntyre, Marco Lohse, Jay David Bolter, and Emmanuel Moreno. Ghosts in the Machine : Integrating 2D Video Actors into a 3D AR System Georgia Institute of Technology. In *2nd International Symposium on Mixed Reality*, 2001.

[30] Blair MacIntyre, Marco Lohse, Jay David Bolter, and Emmanuel Moreno. Integrating 2-D video actors into 3-D augmented-reality systems. *Presence: Teleoperators and Virtual Environments*, pages 189–202, 2002.

[31] Alessandro Mulloni, Hartmut Seichter, and Dieter Schmalstieg. Handheld augmented reality indoor navigation with activity-based instructions. *In Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services - MobileHCI '11*, 2011.

[32] U. Neumann and S. You. Natural feature tracking for augmented reality. *IEEE Transactions on Multimedia*, 1(1):53–64, March 1999.

[33] Milgram P. and F. Kishino. Taxonomy of Mixed Reality Visual Displaystle. *IEICE Transactions on Information and Systems*, pages 1321–1329, 1994.

[34] Qi Pan, Clemens Arth, Gerhard Reitmayr, Edward Rosten, and Tom Drummond. Rapid scene reconstruction on mobile phones from panoramic images. *IEEE International Symposium on Mixed and Augmented Reality 2011 Science and Technolgy Proceedings*, pages 55–64, October 2011.

[35] Simon Prince, Adrian David Cheok, Farzam Farbiz, Todd Williamson, Nik Johnson, Mark Billinghurst, and Hirokazu Kato. 3D Live: Real Time Captured Content for Mixed Reality. In *ISMAR '02 Proceedings of the 1st International Symposium on Mixed and Augmented Reality*, September 2002.

[36] Gerhard Reitmayr and Tom Drummond. Going out: robust model-based tracking for outdoor augmented reality. *2006 IEEE/ACM International Symposium on Mixed and Augmented Reality*, pages 109–118, October 2006.

[37] J. Rekimoto. Augmented Reality Using the 2D Matrix Code. *In Proceedings of the Workshop on Interactive Systems and Software*, 1996.

[38] E. Rosten and T. Drummond. Fusing points and lines for high performance tracking. *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, pages 1508–1515, Vol. 2, 2005.

[39] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. *Computer Vision–ECCV 2006*, pages 1–14, 2006.

[40] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. GrabCut: Interactive Foreground Extraction using Iterated Graph Cuts. *ACM Transactions on Graphics*, 23(3):309–314, August 2004.

[41] Gerhard Schall, Daniel Wagner, Gerhard Reitmayr, Elise Taichmann, Manfred Wieser, Dieter Schmalstieg, and Bernhard Hofmann-Wellenhof. Global pose estimation using multi-sensor fusion for outdoor Augmented Reality. *2009 8th IEEE International Symposium on Mixed and Augmented Reality*, pages 153–162, October 2009.

[42] Ankur Satyendrakumar Sharma and Mohamed Elidrisi. Classification of multi-media content (videos on youtube) using tags and focal points. *Unpublished manuscript*, 2008.

[43] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3D. *ACM Transactions on Graphics*, 25(3):835, July 2006.

[44] Ivan E. Sutherland. A head mounted three dimensional display. *In Proceedings of the AFIPS Fall Joint Computer Reference, Washington D.C.*, pages 757–764, 1968.

[45] V. Vlahakis, J. Karigiannis, M. Tsotros, and M. Gounaris. ARCHEOGUIDE: First results of an Augmented Reality, Mobile Computing System in Cultural Heritage Sites. *In Proceedings of Virtual Reality, Archaeology, and Cultural Heritage International Symposium*, pages 131–140, 2001.

[46] Daniel Wagner, Alessandro Mulloni, Tobias Langlotz, and Dieter Schmalstieg. Real-time panoramic mapping and tracking on mobile phones. In *2010 IEEE Virtual Reality Conference (VR)*, pages 211–218, March 2010.

[47] Daniel Wagner, Gerhard Reitmayr, Alessandro Mulloni, Tom Drummond, and Dieter Schmalstieg. Pose tracking from natural features on mobile phones. *2008 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*, pages 125–134, September 2008.

[48] Daniel Wagner and Dieter Schmalstieg. History and Future of Tracking for Mobile Phone Augmented Reality. *2009 International Symposium on Ubiquitous Virtual Reality*, July 2009.