Graz University of Technology
Institute for Computer Graphics and Vision

Master's Thesis

Content Creation for Augmented Reality on Mobile Devices

Stefan Mooslechner

Supervisor:
Dipl.-Ing. Dr. techn. Dieter Schmalstieg

Advisors:
Dipl.-Mediensys.wiss Tobias Langlotz
Dipl.-Mediensys.wiss Stefanie Zollmann

Graz, August 2010

Abstract

Nowadays, AR (Augmented Reality) become more and more attractive for different areas of application. Especially the increasing number of smartphones with huge displays, built-in cameras and fast wireless connections extends this area. The most applications in this case deal with pre-assembled content, and the user is a simple consumer. Indeed it is possible to create their own content, but in most cases, a special knowledge about different software solutions is necessary. So, the user has to invest time in learning about these applications. The possibility to create and share AR content in a smart and easy way would widen up the number of users for this field of application.

In this work, we present a prototype to create AR content directly on mobile devices. The user can create new 3D-objects as well as 2D-drawings. We provide different possibilities to color and texture the objects. The building and manipulation of the scenes are done directly at the location where they will be shown. So, a fast and exact adjustment to the environment is possible. We deliver an easy way to build virtual models out of the real environment or to generate totally new objects. Additionally, we use an existing infrastructure to distribute the content to a huge number of users.

This could be a further step to reach more acceptance for AR applications by end users.

Zusammenfassung

AR (Augmented Reality) wird heutzutage immer attraktiver für verschiedenste Anwendungsgebiete. Vor allem die steigende Verbreitung von Smartphones mit großem Display, einer eingebauten Kamera und schnellen Funknetzwerkverbindungen erweitern dieses Einsatzgebiet. Die meisten Anwendungen verwenden allerdings vorgefertigte Inhalte, womit der Benutzer zum reinen Konsumenten wird. Natürlich besteht die Möglichkeit, eigene Inhalte zu erzeugen, aber dazu ist in der Regel ein gewisses Spezialwissen über die benötigte Software notwendig. Somit muss der Benutzer erst Zeit investieren, um diese Anwendungen zu erlernen. Eine einfache Möglichkeit, AR-Inhalte zu erzeugen und mit anderen zu teilen, würde die Anzahl der Nutzer in diesem Bereich wesentlich erhöhen.

In der vorliegenden Arbeit präsentieren wir einen Prototypen, der es ermöglicht, AR-Inhalte direkt auf mobilen Geräten zu erzeugen. Es können sowohl 3D- als auch 2D-Objekte erzeugt werden. Diese können mit unterschiedlichen Farben oder Texturen versehen werden. Die Erzeugung und auch die Manipulation der Objekte werden direkt an dem Ort durchgeführt, an dem sie in weiterer Folge auch angezeigt werden. Dies ermöglicht eine schnelle und genaue Anpassung an die Umgebung. Wir bieten zum einen eine einfache Möglichkeit, virtuelle Objekte direkt aus der realen Umgebung zu erzeugen, aber auch die Möglichkeit, vollkommen neue Objekte zu schaffen. Zusätzlich nutzen wir eine vorhandene Infrastruktur, um die neuen Inhalte mit einer großen Anzahl von Nutzern zu teilen.

Dies könnte ein weiterer Schritt sein, um eine höhere Akzeptanz von AR-Anwendungen beim Endverbraucher zu erreichen.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Stefan Mooslechner

Danksagung

Nachdem eine Arbeit wie diese ohne die Hilfe verschiedenster Menschen nicht möglich wäre, ist es an der Zeit sich bei einigen zu bedanken die Anteil am Gelingen dieses Projektes hatten.

Zu allererst geht mein Dank natürlich an meine Betreuer Steffi und Tobias die auch die Grundidee für diese Arbeit hatten. Sie standen stets mit Rat und Tat zur Seite, wenn dies notwendig war. Sie haben durch viele Diskussionen sehr viel zur Weiterentwicklung dieser Arbeit beigetragen.

Weiters geht mein Dank auch an Prof. Dieter Schmalstieg und an das gesamte Team des ICG. Es war, wie auch bei allen meinen vorangegangenen Projekten, eine Freude an diesem Institut zu arbeiten. Bei Unklarheiten oder Problemen fand ich immer eine offene Tür, auch wenn die betreffenden Personen in keinem direktem Zusammenhang mit meinem Projekt standen.

Danke auch an meine Freunde Claus und Ferdinand, mit denen ich einige aufschlussreiche Diskussionen über meine Arbeit führte. Durch sie konnte ich auch einige zähe Stunden überstehen in denen ich mich geistig im Kreis gedreht habe.

Und zu guter letzt möchte ich mich natürlich auch bei meiner Freundin Karin bedanken die meine Launen ertragen musste, wenn gewissen Dinge nicht auf Anhieb so funktioniert haben, wie ich es mir vorgestellt hatte.

Ein großes Dankeschön an alle.

Contents

1 Introduction **1**

2 Related Work **3**

3 System Overview **6**
3.1 Tracking 8
3.2 Content Creation 9
3.3 Content Management and Distribution 10
3.4 Graphical User Interface 10

4 Tracking Module **13**
4.1 Localization 15
4.2 Tracking 16
4.3 File Server 18

5 Content Creation **19**
5.1 Freeze Mode 19
5.1.1 Implementation details 20
5.2 Scene Graph Representation 20
5.3 3D-Content 21
5.3.1 SG-Representation of 3D-Objects 22
5.3.2 3D-Object generation out of a 2D-Screen 24
5.3.3 Implementation Details 24
5.3.4 Cube 24
5.3.5 Cylinder 25
5.3.6 Polygon 25
5.4 2D-Content 26
5.4.1 SG-Representation of 2D-Objects 26
5.4.2 2D-Objects 28
5.5 Texturing 29
5.5.1 Texture Manager 29

5.5.2 Predefined Textures 30
5.5.3 User-Defined Textures 30
5.5.4 FasTex 31
5.5.5 Implementation Details 32
5.6 Object Manipulation 33
5.6.1 Object Selection 34
5.6.2 Implementation Details 34
5.6.3 Moving 35
5.6.4 Rotating and Scaling 36
5.6.5 Color and Texture Selection 36

6 Content Management and Distribution 37
6.1 Object Explorer 37
6.1.1 Implementation Details 38
6.2 Storage 40
6.2.1 IO-Manager 40

7 Graphical User Interface 43
7.1 Structure of the GUI 43
7.1.1 Color-GUI 44
7.1.2 Texture-GUI 44
7.1.3 State Pattern and Implementation Details 46
7.1.4 Status Line 49

8 Conclusion 51

9 Future Work 53

Bibliography 55

List of Figures

3.1 Overview of the client-server communications 8
3.2 UML-diagram of the system and the main components . . . 12

4.1 Overview of the client-server communication during the pose estimation 15
4.2 Overview of the SIFT and Ferns processing pipeline described by [WRM+08]) 17
4.3 Identification of NFT-Target-Features 18

5.1 Predefined textures 30
5.2 Mapping examples of user-defined textures 31
5.3 Mapping of user-defined textures 32
5.4 Cube during rotation manipulation around the Y-axis with shown CSS 35

6.1 The *Object Explorer* and its special GUI 38
6.2 Rotation of an object displayed by the *Object Explorer* 39

7.1 Design of the GUI and the Status Line 43
7.2 Overview of all possible GUI-states 45
7.3 Displayed Color-GUI with differently colored objects 46
7.4 Design of the Color-GUI and the Texture-GUI 47
7.5 The State Pattern described by [GHJV95] (p. 339) 48
7.6 Available colors for the GUI-Buttons 48
7.7 UML-diagram of all involved classes of the GUI 50

Chapter 1

Introduction

Today, content creation for AR applications is largely limited to applications running on a desktop computer, and most of these applications are used by a small number of professional modelers. All actual known AR applications (e.g. Wikitude) deliver pre-assembled and unchangeable content to the user. To reach a wider acceptance and distribution of AR applications, it will be necessary to provide the possibility to create and share AR content with a huger number of users. At the change from Web 1.0 to Web 2.0, the role of the user changed from a pure consumer to a creator of content. In the same way, the user should be enabled to create and share his own ideas in so-called AR 2.0. Enabling the user an active participation will result in more popular and more often usage of different AR applications.

Not only an easy way of creation of content is necessary to reach user acceptance, but it is also essential to enable sharing with a huge number of other users. Analog to Web 2.0, it is also necessary to create and provide a distribution infrastructure to reach those users.

Nowadays, low-priced mobile devices with high-resolution cameras, (relative) large displays, and a growing calculation power become available more frequently. Additionally, they are equipped with different prospects of wireless communication methods like Wi-Fi or mobile high-speed internet connections. So, we focus the developing on such low-cost devices to take advantage of the fast growing number of them.

In this following work, we present a first prototype to enable AR content creation and sharing on mobile devices for end users. In the current version, the user is tracked in a large indoor working volume. We use a combination of on-line pose estimation and real-time tracking. By analyzing a picture of the current environment, we estimate the actual position

and viewing direction of the user. Based on these data, we load the dataset for real-time natural feature tracking. This is the starting point for all further processes of our project.

Due to the limitations of input devices on the target equipment, we provide an interface to do all inputs and activities using the touch-screen. Therefore we designed a very flat GUI-structure to keep the interface well-arranged and simple. We enable the user creation of their own 3D-objects as well as replications of real objects. 3D-objects are created by defining a base area and extruding it in a next step. Also, annotations or 2D-drawings can be added to the scene. Furthermore, we deliver methods to manipulate the built objects and to color or texture them in a simple way, and we provide predefined textures as well as the possibility to create and use textures from the environment.

In addition we show possible methods to store and reuse single objects or complex scenes directly on the device or on a remote server. The storage on a server is the base to enable the sharing of the content with other users. The other users can view the scene, but they can also manipulate or extend it.

The current application can be a further component to reach more acceptance and a wider spreading of AR applications especially on the mobile sector. The user is involved in the building of content and enabled to share his creations with other users. Exactly these achievements were one of the main bases of the success for Web 2.0, and they could play the same role in AR 2.0.

Chapter 2

Related Work

In this chapter, we show different developments focusing on or touching the field of application we address in our work. We can split up the related work into three main fields: on the one hand, the displaying of AR on mobile devices and on the other hand, the creation of AR content out of the current environment. The third point is the distribution of content to multiple users using a network. In the most cases of the first point, the creation of the content was done on a desktop computer designed by more or less complex programs. The creation of content from the environment is also generally done on desktop computers. So, the usage of a standard PC or a system with a comparable calculation power is necessary for all applications presented up to now.

One of the very first projects targeting AR on mobile devices is MARS (Mobile Augmented Reality System). The first researches on MARS started in 1996, and the refinement of the project is still in process. In one of the early versions of MARS (shown in [HFH+01]), the system was used to display pre-generated content (mainly in an urban environment). The used hardware consists of a laptop computer placed in a backpack, a GPS receiver, a see-through HMD (Head Mounted Display) and a wireless input device. This was one of the first setups to enable mobile AR with the exception of the *Touringmachine*, which provides a theoretical approach.

In a further application [WCVH08], the system was extended by a laser range finder mounted on top of the HMD. This setup was used to annotate objects in the environment. The annotation could be placed on an object just by looking at the desired place. It also offered the possibility to place a label in a correct edge on the surface of an object. The user just had to look along the surface, and the system stores the different distances to the surface and calculates and displays the label in a correct edge on the

object. With this setup, it was also possible to cut off objects out of the environment. The object was segmented by looking along the border of it. The object built in this way was just a 2D-object, but this was one of the first steps to generate content on the fly using a mobile system.

The *SitePack* system [NKG04] consists of a tablet-PC with a stylus as input device. An additional video camera and a GPS completes the hardware. The system is used to show pre-generated content and targets to architectural usage. The main field of application is the displaying of models of buildings if they are planned or the construction is in progress. The hardware setup is very close to our setup (touch-screen, camera), but a tablet-PC will never be a mass product like a smart phone.

The *Tinmith* project [Pie06] uses a similar hardware equipment like the MARS project. Amongst other things, it enables the generation of 3D-wireframe objects from the environment. The user can specify planes by looking along the faces of a physical object, and the system generates the object from these informations. So, for example it is possible to build a 'copy' of a house by walking around it and looking along the walls of it.

The idea of content generation presented in *OutlineAR* [BMC08] works similar to *Tinmith*. The main difference is the equipment and the input device. *OutlineAR* uses a special input device. It consists of two buttons and a wheel (as known from a computer mouse) in combination with a camera. With this configuration, it is possible to define vertex points, which are the corners of the new built wireframe model. All calculations are done on a standard PC where the input device is connected to. The system offers an easy way to create wireframe boxes from the environment, but it is limited by the specific input device and the necessary calculation power of the PC.

Of course, most of these applications are running on a mobile device, but the 'mobile device' in this case is a laptop computer placed in a backpack or a relatively huge and heavy tablet-PC. Additionally, they need a combination with a HMD, a GPS receiver, a special input device, or other additional equipment. So from our point of view comparing these components with a mobile device is hard to do.

The creation of 3D-models is another gist of our work. There are various desktop-based systems for content creation of 3D-objects, which are also usable for creating AR content. The best known are *Maya, Blender,* or *3ds Max*. DART [MGDB04], for example, is one of the best known toolkits addressing especially AR applications. It implemented on top of the *Macromedia Director* and is targeted to semiprofessional and professional users. It delivers a script language to extend the given feature-set and de-

livers a powerful tool to build AR content. But, aforesaid, the user has to bring some experience, or he needs some training time to get familiar with one of these extensive programs.

Another field is the distribution of the content to a huge number of users. [UTO+04] describes the distribution of AR annotations using a wireless network and a client-server model. The goal of the application is to offer an interactive, AR-based, indoor guideline. If the connection to the server is established, the client sends his position data to server and receives the AR dataset corresponding to the actual position. In this system, the sharing of data is done, but regarding the dataflow, it is a one-way connection from the server to the client. There is no possibility for the user to change the information on the server. (Which is absolutely correct for this field of application.)

There are numerous applications to generated 3D-content for AR applications and also mobile solutions to use and show these contents on (more or less) mobile devices. The global storing on a server and the distribution of some content over a network are also nothing new. But a combination of all of these things and additionally the extension by a simple way of generating 3D-content should be the big advantage of our application.

The main motivation for our work is formulated precisely in [WDH09]:

> *However, being able to create or edit annotations online is also very powerful. There are many new applications that become possible with online annotation and editing, and others that become much easier to build. Such applications would in turn enable many more people to create content. Locating the content correctly also becomes a much more easier process, since it is then possible to directly see the intended location.*

Chapter 3

System Overview

In the following chapter, we give an overview of the complete system and its main components. A more detailed view onto the underlying concepts is described in the specific chapters.

Our main goal of this application is the development of a prototype for AR content creation and distribution. The system should be independent from a working environment restricted by the use of a desktop computer or by the need of very special knowledge of the user. Another goal is to enable the creation of AR content at the same place the content will be shown later. This has the advantage that the content can be optimally fit into the environment.

So, the methodical approach of the current prototype is as follows. The user is tracked in an indoor working volume. For the tracking, we use a combination of two methods. In the first step, we use a remote pose estimation calculated on a server. In the second step, the application loads the tracking data for natural feature tracking (NFT) to the device. So, if the user would like to generate some new AR content, he takes a picture of the environment out of his current position. Now, the application communicates with the first server to estimate the pose. The server sends the result of the calculation back to the application. With this information, the corresponding tracking dataset is loaded from the second server. From now on, the real-time tracking is enable and this is also the initial point for all further actions.

If there is an existing AR scene at this NFT target, the scene is displayed now and the user can watch, manipulate, or extend it. If the scene is virgin, the user can create new content to share it with other users. Because of the limited number of input methods for mobile devices, we fully support the usage of a touch-screen. We built a logical graphical user interface (GUI)

providing all necessary functionalities for the usage of the application. The user can create new 3D-objects and color or texture them. He is able to create his own textures as well as he can use predefined textures from the application. He can do annotations or other 2D-drawings, of course using different colors. Furthermore he can store single objects for his device to take them with him for a later use at another place.

If the creation and manipulation process is completed, the scene is stored again on a server, and is available for other users using the application at the same location.

To enable all these functionalities in one single system and to keep the system maintainable and extendable, we split it up into four main components:

- Tracking
- Content Creation
- Content Distribution
- Graphical User Interface

The advantage of spreading up the system is the possibility to exchange single parts of the system without changing the other components. E.g. it is possible to substitute the currently used tracking module by a module using GPS-tracking without having any changes at the other components.

As shown above, we use the communication with different servers for several times and due to several reasons. On the one hand, it dislocates the cost-intensive calculation for the pose estimation from the mobile device to a much more powerful server. On the other hand, we use a server to store the generated scenes. So, we can easily provide the content to a huge number of multiple users. Figure 3.1 shows the sequences of the communication between the mobile device and different used servers.

The following sections contain a description of the main components and shows an overview of their range of application. At the end of the chapter, the UML-Diagram in Figure 3.2 illustrates the main components and the according C++ classes.

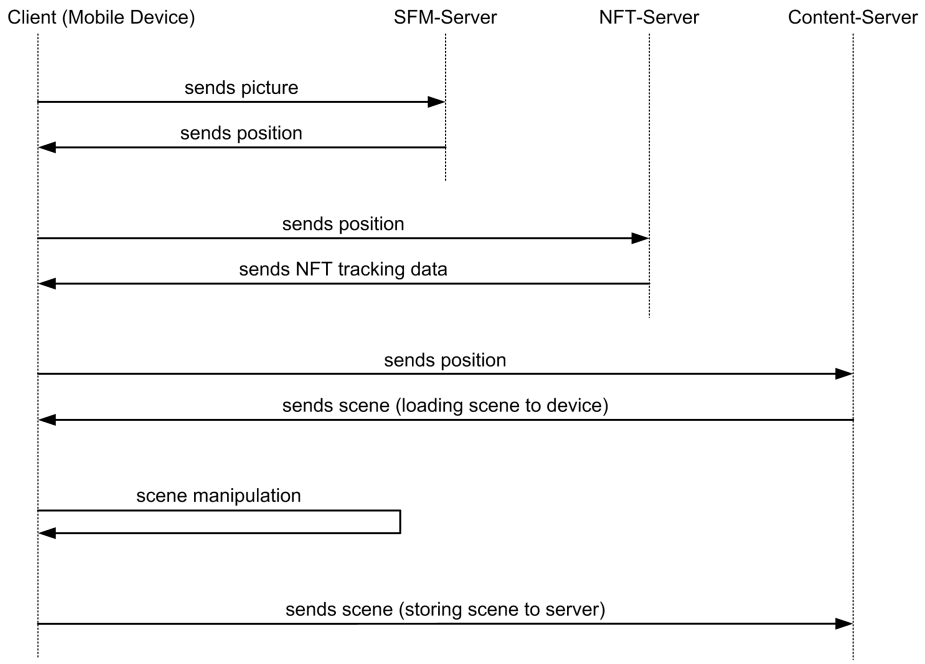

Figure 3.1: Overview of the client-server communications

3.1 Tracking

The function of the Tracking Module is the estimation of the actual position of the user and the real-time updating of this information. The current implementation uses a *Sparse Feature Model* (SFM) for the location of the current position in the environment and *Natural Feature Tracking* (NFT) for tracking. This combination works in a preregistered (mainly indoor) environment.

For the first step, the application takes a picture of the current environment. This picture is converted into a gray scale picture to reduce the amount of traffic between the application and the server. Now, the picture is sent to the SFM-server to do the analysis. The calculation of the position and the viewing direction is done on the server because this part of the pose estimation needs the most calculation power. After the calculations, the server transfers the result back to the application. Based on these data, in the second step, the corresponding NFT tracking data is loaded from a different server. Now, we are able to do real-time tracking for the current location. This is also the basement for all further actions of the application.

If there is an existing scene for that location, it is loaded and displayed, and the user can extend and manipulate it. Otherwise, the user can build up a new scene for this location.

If the user changes his working volume by moving out of sight of the current NFT-target into the sight of another one, the pose has to be estimated again by repeating the steps above.

3.2 Content Creation

Combined with the *Content Management* and *Distribution Module*, the *Content Creation Module* is the core of our application. The creation of 2D- and 3D-content is provided in a simple but powerful way. Also coloring and texturing of the objects can be done in an easy way. Furthermore, we provide a so-called *Freeze Mode* to simplify the content generation since it is really difficult to annotate or draw objects exactly while keeping the position of the device stable in relation to the NFT target. So, if the *Freeze Mode* is activated, the current view is frozen and independent from any motion of the device, but all the other functionalities of the application are unharmed. So, every action of the application can be done in *Freeze Mode* or in the normal mode.

To create a new 3D-object, the user just has to draw the base area and extrude it. We offer a circle to create a cylinder, a quad to create a box, and a polygon to create a polygonal object. To build multiple kinds of objects, the objects can be colored in different ways. It is also possible to texture the objects with predefined textures. To widen up the flexibility of usage, the user can additionally create own textures of his environment. So, the rebuilding of real objects is possible. To do this in a more easy way, a newly created 3D-object can be textured directly with the background corresponding to the place it was created. With this *FasTex* method, the replication of objects from the real world can be done quite fast.

For annotations or sketch drawings, it is possible to create 2D-objects using freehand drawings, lines, circles, and boxes. Of course, these drawings can be done in different colors and different line widths to offer a wide area of usage. While generating a 2D-object, the application switches automatically into the *Freeze Mode* to enable the user a comfortable handling.

After the creation of any objects (both 2D and 3D), the user can assign some manipulation methods to them. To change the position and arrangement of objects, they can be manipulated by the well-known moving and rotating operations. The size of an object can be manipulated by scalings

along the X-, Y-, or Z-axis.

3.3 Content Management and Distribution

The Content Management and Distribution Module is the other core component of our system. On the one hand, it provides an *Object Explorer* for the reuse of objects. On the other hand, it enables a global storage of the scene and all of its components (e.g. textures) on a server.

With the *Object Explorer*, we allow the user the flexible reuse of formerly created objects. It is possible to reuse an object in the same scene where it was created as well as to use it in a completely different environment. The *Object Explorer* shows previews of all stored objects. The user can add and remove objects to the *Object Explorer* or add objects from the explorer to the current scene. Since the objects are stored local to the device, it is possible to create a collection of objects for one's usage. In further versions, it is also imaginable to create a global *Object Explorer*, placed on a server, to enable yet another opportunity to share single objects with other users.

The storage of the scene is done on a public reachable server. So, the scenes are available for everyone using our application. Additionally to the scene graph of the scene, the used textures are stored as png-files. To speed up the loading and saving procedure, the data are packed into a zip-file before the upload to the server is started. For testing reasons with our prototype, it is also possible to load and store a scene directly on the device. Hence, the single modules of the application can be tested without the SFM pose estimation respectively without having a connection to the content server.

3.4 Graphical User Interface

The design of the Graphical User Interface (GUI) is a very important point in terms of the acceptance of a program by the users. The GUI should allow a simple handling of the functions of the program. So, we tried to build a intuitive menu to guide the user through all possible features of the application. We kept the structure of the menu as flat as possible to avoid a huge number of submenus. Additionally, there is an optical information about the current program state by the use of different colors for

the menu buttons.

We also provide a *Status Line* to inform the user about the current state of the program or actual informations according to his operations. This *Status Line* is always visible for the user to offer permanent information about user-driven actions (e.g. 'loading scene - please stand by...') or additional functionalities available at the current program state (e.g. 'change transparency [of the selected object] by moving the stylus'). It is also useful to inform the user if some functionality fails (e.g. 'server not reachable').

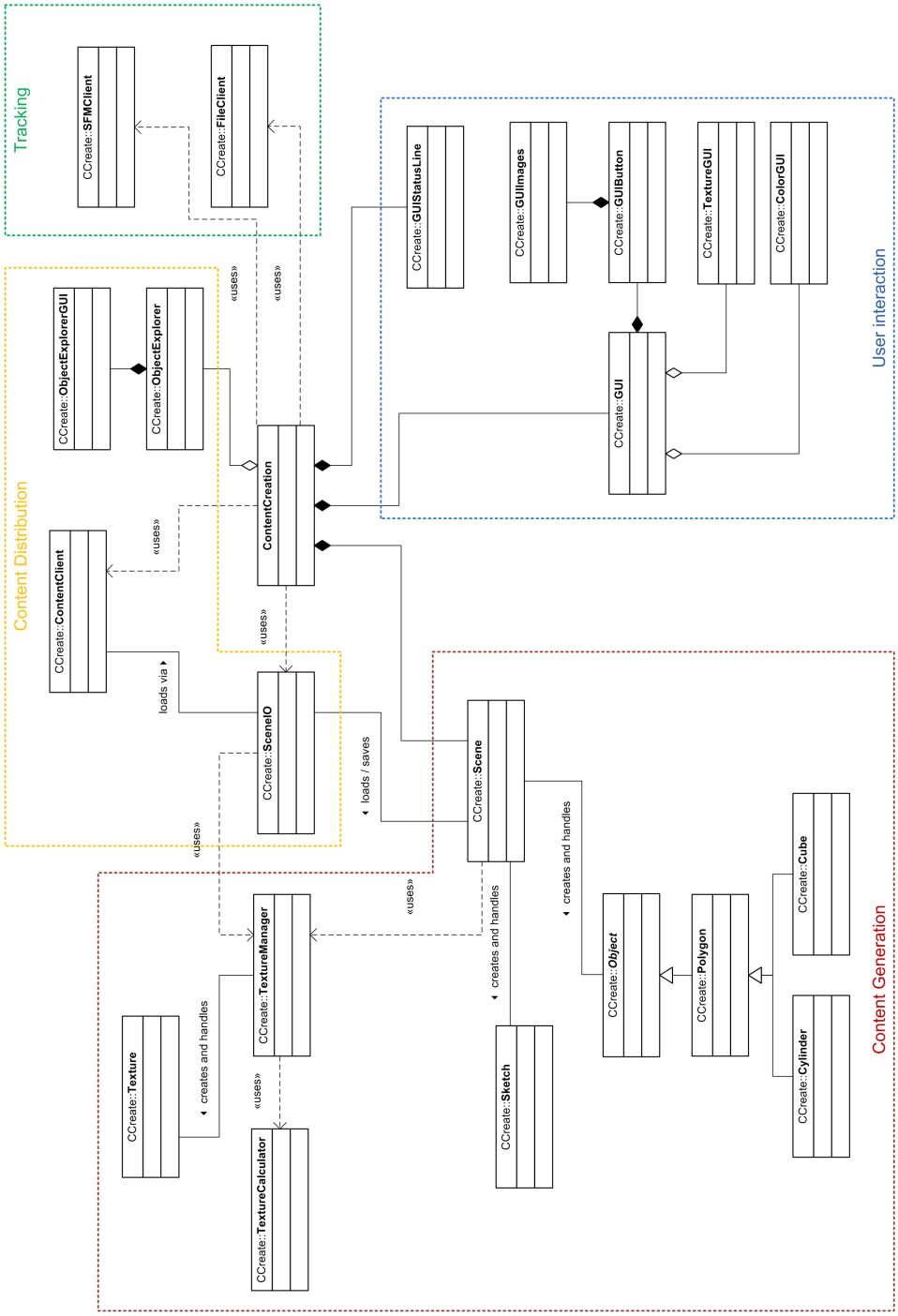

Figure 3.2: UML-diagram of the system and the main components

Chapter 4

Tracking Module

In the following chapters, we describe the tracking method and the used client-server infrastructure.

The pose estimation is based on two steps. We use a combination of a localization module using a *Sparse Feature Model* (SFM) and a real-time tracking module using *Natural Feature Tracking* (NFT). In Fig. 4.1 we show the two steps of the client-server communication.

In the first step, the user takes a picture of his environment from his current position. This picture is converted into a gray scale picture. This transformation is done to reduce the amount of traffic between the application and the SFM server.

The request for the the server must be a string in the form:

```
0xCCCC 0xLLLL PPPP[0..n]\0
```

where:

0xCCCC	is the number of the command for the server as hexadecimal number
0xLLLL	is the length of the payload as hexadecimal number
PPPP	is the payload as a *NULL-terminated* string

The server can handle different commands like: `getNumberOfImages()`, `getImage()`, `setCameraCalibration()` and of course `getImagePose()`.

For a request, the gray scale values have to be normalized between 0 and 255. Due to the *Studierstube* luminescence format delivers the data in the correct format, and no further conversion has to be done. Additionally to

the gray scale values, the request for a pose calculation also has to include the size of the picture. A typical pose request to the SFM server looks like this (where the command number 5 stands for `getImagePose()`):

```
0x00000005 0x0003C04B 320 240 1 110 132 108 86 ...
                                        ... 75 68 211 193\0
```

Normally, the sending of the request and the waiting for the response would block our application, and the user cannot determine whether the application is working and waiting for the response or if it is crashed. Thus, we start the communication with the server (the sending itself and the callback waiting for the answer) in a new, separate thread. With the result that the update of the screen is still done and we can inform the user via the status line about the actual progress of the communication. The answer of the server comprises (amongst others) a 3x4 matrix containing the information about the position where the picture is taken. An example for the response of the SFM server:

```
echo calculateImagePosition called
Reading image: 320 240  1
...
VRCamera { intrinsic 3 3 [ 322.83 0 165.482 0 322.83 125.151
  0 0 1 ] orientation 3 4 [ 0.969698 0.0277322 -0.242728
  1.27363 -0.0065753 0.996139 0.0875427 -0.00546303 0.244219
   -0.083294 0.966136 -1.02639 ] }
...
finished calculateImagePosition
TcpPeer:: handled
```

Now, the first step is finished, and we know where the user is located in our environment.

Based on this knowledge, the application starts the download of the NFT dataset. Therefore we build up a connection to the *File Server* described in section 4.3 at the end of this chapter. The server contains the different available NFT datasets for our registered environment. The knowledge which dataset belongs to which region in the environment is currently stored in the application itself. But it is also imaginable to put these information either on the SFM server or on the *File Server*. As shown with the communication with the SFM server, the communication with the *File Server* runs also in a separated thread. If the download to the device is completed, the dataset is loaded into the application. After a reinitializa-

tion of the tracking part, the application switches into a real-time mode using natural feature tracking for pose estimation. Now, the initialization for the work with our application is done, and all further actions are based on this state.

If the user changes his location by moving out of sight of the actual NFT target, the pose estimation has to be redone by starting again with step one.

So, the range of application of the current setup is limited by the predefined SMF-Model and the predefined NFT-targets. Because of the modular design of the application, however, we are able to extend the range of application easily by changing the tracking module (cp. Fig. 3.2).

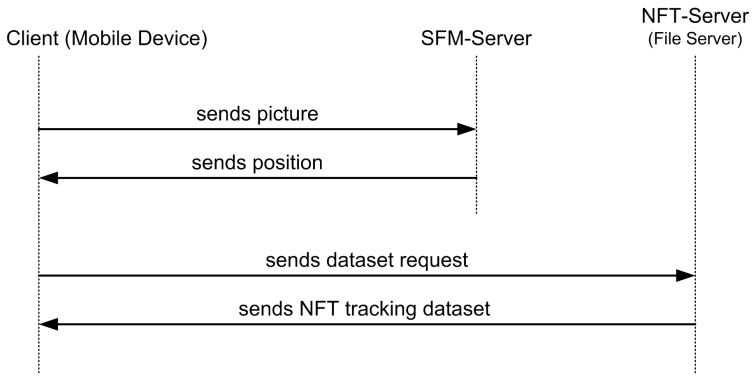

Figure 4.1: Overview of the client-server communication during the pose estimation

4.1 Localization

The prime localization of the user is based on a sparse 3D-point reconstruction. The system calculates a full 6 DOF pose estimation from a single picture taken by the built-in camera of the device. The calculation is done by using the texture information of the environment. In principle, we can see the preregistered environment as a single large 3D-tracking target.

The registration has do be done manually by taking pictures of the environment. Furthermore the usable features of the pictures are extracted and obtained with a structure from motion system. These features are related to each other to build up a single global coordinate system. Therefore SIFT features are build up from every image and a vocabulary tree is used for the matching of related pictures.

For the localization process itself, at first, the features of the taken picture have to be extracted and described. For the feature extraction, a scale space search is used to find the keypoints and get a size estimation in a single step. The next step is feature matching and the removal of outliers. After the first pose estimation, a refinement using a Gauss-Newton iteration is done. The pose is refined until the number of inliers is stable.

The work presented in [AWK+09] is the next evolution step to the system we used in this work. As the underlying concept is equal and the biggest part of the work is the same we have in use, a more detailed knowledge can be found there.

4.2 Tracking

The computation of the tracking is done directly on the mobile device. Due to the limited resources in terms of computation power, memory size and access speed, a special approach for the tracking algorithm is needed, particularly since the tracking should be done in real time. The approach we use in our application is a combination of a modified SIFT (scale invariant feature tracking) descriptor and a modified Ferns classifier.

The original SIFT is known as a powerful but computationally intensive descriptor for features. In the modified version, we use the FAST corner detector instead of the original Difference-of-Gaussians (DoG) to detect the features out of the tracking target. Since FAST is not invariant to scaling of features, the feature database contains different scales, comparable to the well known MIP-Maps. During the tracking process itself, the algorithm keep 'good' and 'bad' features. Since processing time can be saved by also keeping the knowledge of 'bad' features from one frame to the next one.

During the creation process of the descriptor, we also keep the low capacity of the target platform in mind. So, we use a descriptor with 3x3 subregions and 4 gradient bins. The result is a reduction for the feature vector from the commonly used 128 dimensions (4x4 regions and 8 bins) to a 36 dimensional vector. To reduce the influence of noise, a Gaussian filter is used to blur the source frame. During the descriptor matching, we also use a different method in comparison to the original implementation. We use several Spill Trees, combined to a Spill Forest, to reduce the searching time. The removing of the produced outliers is done in three steps. In the first step disoriented features are removed. The second step is based on a simple geometric test. Since the most outliers are removed by the first two tests, the third test is more complex. We calculate expected homogra-

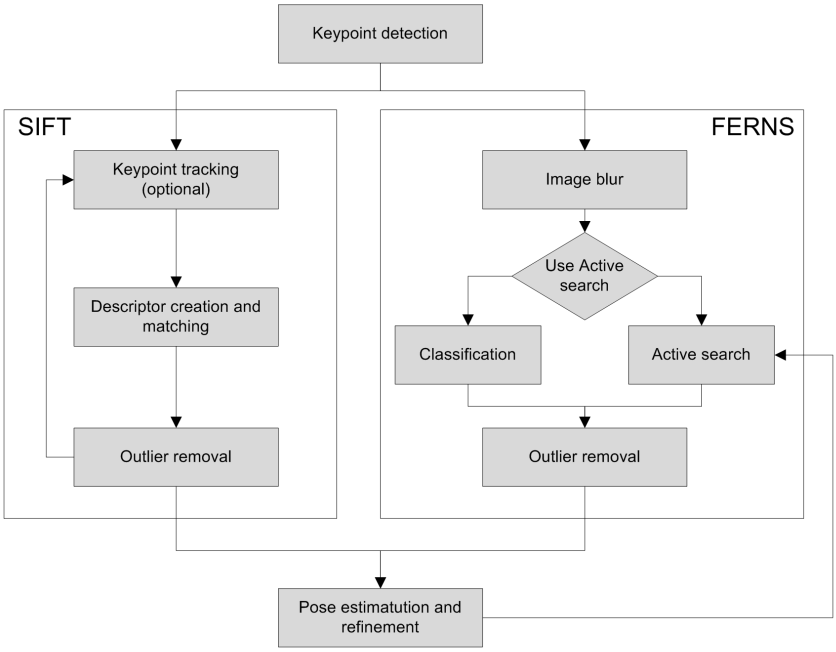

Figure 4.2: Overview of the SIFT and Ferns processing pipeline described by [WRM+08])

phies and check them for the smallest number of outliers.

The Ferns classifier is also modified for the use on mobile devices. In contrast to the original algorithm, also the FAST corner detector is in use instead of the Laplacian operator. The classification is implemented straightforward as shown by the original authors. Like the SIFT algorithm, Ferns also uses a Gaussian blur filter to lower the influence of noise in the frames. The feature classification does not need any information about the current camera pose. But the knowledge about it can be used to reduce the calculation period for the next frame. In addition, the number of outliers is reduced by using this experience.

[WRM+08] shows much more detailed description and additional performance and robustness tests.

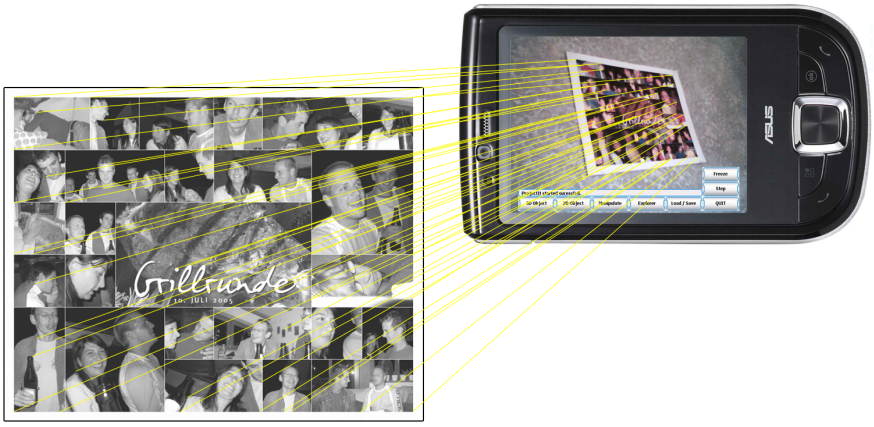

Figure 4.3: Identification of NFT-Target-Features

4.3 File Server

The *File Server* is more or less a by-product of our application. It is in use for testing purpose to provide the download of different files from a server. The *File Server* is a simple socket server application. It provides the possibility to load files from a server by sending the filename. The server works for binary files as well as for text files. After connecting to the server, the client just sends the path and the name of the target file. The server opens the file and starts the downstream to the client if the file exists. If the file is not available, the server sends an error message to the client.

The implementation of the file server is as simple as possible. So, it does not deliver an extended error handling, e.g. if there are connection problems. This was done because the file server is not the main goal of this work, but it was necessary to test the functionality.

The *File Server* is in use to provide the download of the NFT dataset for the mobile device.

Chapter 5

Content Creation

In this chapter, we show the functionality of the *Freeze Mode*, details of 2D- and 3D-content creation and give an insight into the internals of the several objects. We also describe the details of the different texturing methods and the possibilities of manipulations of the generated objects.

5.1 Freeze Mode

With the *Freeze Mode*, we handle one of the main problems we found during the first testing sessions on the mobile device. It is really difficult to write a text or do accurate drawings in 2D with the stylus on the small display. In fact it is much easier to create 3D-objects in that way but it is a real challenge to do one of these things and keep the device stable in relation to the NFT target because it is nearly impossible to keep the device still and focused on the target and simultaneously generate some useful content with the stylus on the device. So, we decided to give the user the possibility to take a snapshot of the environment and use this snapshot as base for any creations.

While the *Freeze Mode* is active, the background of the display is not updated. So, if the user is in his desired position, he activates the *Freeze Mode* by clicking on the corresponding button on the screen. Now, it is possible to move around the mobile device without losing the focus of interest. And so, the user can hold the device in a comfortable position to make his inputs. Of course, it is also possible to create objects without using the *Freeze Mode*, but it is much more comfortable if the mode is used. If the user starts to create a 2D-object, the *Freeze Mode* is automatically activated for a better usability.

5.1.1 Implementation details

The *Freeze Mode* is available at any time by clicking the corresponding button on the display. It is one of the two 'static' functions of the GUI and can be set at any time or rather any program state. To give the user a feedback if the *Freeze Mode* is active, a colored margin is shown on the display.

If the *Freeze Mode* is activated, there were some internal changes to be to keep the scene correct and editable. At first, of course, the update of the display has to be stopped. Normally, the *VideoFrame* is directly copied to the *BackBuffer* by a *memcopy* operation. In case of an active *Freeze Mode*, this operation will not be done.

Furthermore the scene graph has to be adapted. The camera still delivers the tracking data of the NFT target, and of course, the scene is geared to the real tracking target and not to the one shown on the frozen display. To keep the scene stable to the displayed view, the *SgTransfromSeparator*, which is responsible to handle the tracking data, is replaced by a new one. The values of the new separator are just the values of the original separator at the activation time. No update will be done to these values even if the tracker delivers new position data.

Additionally the projection matrix of the NFT target is stored. This matrix is used to calculate all transfers between the 2D-display space and the 3D-scene space. These calculations are done by projecting and unprojecting a specific point of the display onto the XY-plane of the target.

5.2 Scene Graph Representation

In this section, we specify the structure of the scene graph for the whole scene. The scene graph fragments for single 3D- and 2D-objects are described in the chapters 5.3.1 and 5.4.1.

The scene graph contains the entry point for the NFT tracking information and the *ObjectExplorer*, a light positioned 'in the camera' and a root node for the user-generated content.

An example of a virgin scene without any objects:

```
1  <Scene>
2    <MatrixCamera projMatrix="REF StbTrackerNFT2.projMatrix"
                name="MatrixCamera"/>
3    <DirectionalLight direction="0 0 -1" />
```

```
4    <TransformSeparator name="ObjectExplorerRoot" />
5    <TransformSeparator name="ProjectItPoster"
               active="REF Poster.visible">
6      <MatrixTransform matrix="REF Poster.matrix"
                 name="MatrixTransform"/>
7      <TransformSeparator name="ProjectItObjects" active="true" />
8    </TransformSeparator>
9  </Scene>
```

In line three, the directional light is placed in the position of the camera (as shown in line two). The direction of the light is the same as the camera viewing direction to reach a light effect independent from camera motions.

Line four is the root node for the *Object Explorer*. If the *Object Explorer* is active, it is mounted here and the following node is set inactive.

Line five is the separator for the tracking data of the current target.

Line six delivers the tracking data to the scene graph.

Line seven is the root node for any user-generated content. Any object of the scene is a child node of this node. If a scene is saved, only the children of this node are stored. If a scene is loaded, the loaded scene graph is placed as a child of this node.

5.3 3D-Content

The creation of any 3D-object is based on the generation of a 2D-base area. By extruding this base area, the flat 2D-object becomes a 3D-object. In the current version, we are able to generate three different 3D-objects:

- Cube
- Cylinder
- Polygon

The base area of any of these objects is aligned to the XY-plane of the NFT-target and the extrusion is done along the Z-axis. So, after the creation, any 3D-object is axially aligned to the coordinate axis of the corresponding NFT target. Of course, the alignment can be changed using the rotation function from the *Manipulation* menu.

During the creation process of any object, a dummy object is created and displayed. These dummy objects enable an easier creation because there a some unknown parameters during the creation process. For instance, we do not know if the user draws the polygon clockwise (CW) or counterclockwise (CCW). A similar problem arises while drawing a quad. The quad is built from the user by drawing the diagonal. There is a difference if the diagonal is drawn from top left to bottom right or from bottom left to top right. Because of this, the first approach to take a simple, one-sided polygon as dummy object was just partly successful. Of course, the specific character of the dummy object depends on the final object type, but generally, it is a double-sided, filled polygon. Another difference between the dummy object and the real object is that the origin of the real object is in the geometrical center of the object, and the vertices are arranged symmetrically around it. The translation relating to the NFT target is done by the *SgTransform* node. While the positions of the vertices of the dummy object is just relating to the NFT target. Calculations for such adjustments can only be done if all the entire knowledge from the dummy object is available.

A more detailed description about the differences and the building process can be found in the specific sections.

5.3.1 SG-Representation of 3D-Objects

The scene graph representation of all different 3D-objects is very similar. It does not make a difference whatever we take a cylinder, a box or a polygon. At the end of the day, every object can be seen as a polygon with a varying number of coat elements. The sample code shows the scene graph representation of a polygon, but it is representative for any of our 3D-objects. An example of a 3D-object including the position, rotation and scaling information is as follows:

```
1  <SgTransformSeparator>
2    <SgTransform translation="X Y Z" name="Translate" />
3    <SgTransform rotation="W X Y Z" name="Rotate" />
4    <SgTransform scaleFactor="X Y Z" name="Scale" />
5    <SgDrawingStyle blending="true"
               blendFuncSrc="BLEND_FUNC_ONE_MINUS_SRC_ALPHA" />
6    <SgMaterial diffuse="1 1 1 0" />
7    <Node name="ProjectItObjectPOLYGON">
8      <SgTextureSeparator>
9        <SgTexture file="" name="1" />
```

```
10        <SgGeometryVertices>
11          <Vec3Array type="coordinates" size="5">
12            X Y Z; X Y Z; X Y Z; X Y Z; X Y Z;
13          </Vec3Array>
14        </SgGeometryVertices>
15        <SgGeometryTexCoords>
16          <Vec2Array type="tex-coords" size="5">
17            S T; S T; S T; S T; S T;
18          </Vec2Array>
19        </SgGeometryTexCoords>
20        <SgGeometryNormals>
21          <Vec3Array type="normals" size="5">
22            X Y Z; X Y Z; X Y Z; X Y Z; X Y Z;
23          </Vec3Array>
24        </SgGeometryNormals>
25        <SgGeometry hasNormals="true"
                      hasTexCoords="true" hasColors="true">
26          <IntArray type="indices" size="9">
27            0 1 2 0 2 3 0 3 4
28          </IntArray>
29        </SgGeometry>
30            :
31            :
32            :
33            :
34      </SgTextureSeparator>
35    </Node>
36  </SgTransformSeparator>
```

The lines two to four keep the transformation information about the object.

Line six contains the color information about the object.

Line seven is the root node for all geometrical information about the object.

Line nine contains the information about the texture. If a 'name' is specified, the texture is stored under this name. Since the *Studierstube* scene graph cannot deal with stored textures just represented by the file name in the SG node, any texture is loaded by our *Texture Manager*.

The lines 10 to 29 contain the vertex coordinates, the texture coordinates,

the normal vectors, and the indices of the vertices for the top surface.

Starting at line 30, the same information like in the lines 10 to 29 for the bottom surface and for all coat surfaces follows.

The lines 34 to 36 are just the closing tags of the object.

5.3.2 3D-Object generation out of a 2D-Screen

To create 3D-objects, we first generate 2D-objects in the XY-plane of the NFT target. To draw and build the objects on the correct place, we need to know where the ray defined by the touching point of stylus and the screen meet this plane. Therefore we had to invert the projection from the 3D-scene to the 2D-screen to get the coordinate of the origin of the ray. Thus, the calculation depends on the camera parameters we use a build in *Studierstube* method to calculate the intersection. For this calculation we just need to specify the X and Y coordinate of the desired point on the screen, and the result is a 3D-coordinate in the scene or more precisely, on the XY-plane of the target.

5.3.3 Implementation Details

After generating any ground surface, the new object is selected automatically. This has the advantage for the user that he can perform the extrusion directly in connection with the creation. The extrusion itself is done by adapting the Z values of all vertices of the top surface and of every second vertex of the coat surfaces. Due to the equal structure of the scene graph of all 3D objects (one top surface followed by one bottom surface followed by 2 to n coat surfaces), there is no further knowledge about the object needed. During the extrusion, the Z values of the following vertices are adapted: all of the top surface and every second of the coat surfaces. This ensures a regular growing of the 3D-object. Since the origin of the objects is in the geometrical center of the objects (as described above), the Z value of the translation node has to be adjusted by half the value the extrusion has been done.

5.3.4 Cube

The first step to generate a cube is the creation of a quad. The quad is built up by the user by drawing a diagonal. By clicking on the display, he

specifies the first corner. The quad is created while moving the stylus. If the user releases the stylus, the dummy object is converted into the real 3D-object. In this case, the dummy object is needed because during the drawing process, there is a difference whether the diagonal is drawn from top left to bottom right or from bottom left to top right.

The conversion from the dummy object to the real object is done in a simple way. At first, we check whether the quad was drawn CW or CCW. Depending on this information, we set the order of the vertices. Now, we read the vertex positions out of the dummy object and calculate the size of the quad. With this information, we can create the vertices for the real object. The texture coordinates and the normals for the cube are independent of these information, as they are equal for each cube. At last, the dummy object is removed from the scene graph, and the newly generated object is added to it. Hence the object is ready to extrude to become a real 3D-object.

Since the projection from the display onto the 3D-scene space is done by projecting into the XY-plane of the NFT target, the newly created object is also built aligned to this plane.

5.3.5 Cylinder

The cylinder is crated by generating and extruding a circle. Since there is no real circle available, we built the dummy object of a polygon consisting of 36 triangles. (The number of triangles is changeable within the C++ code.) To draw the ground surface, the user has to click on the display at the center of the new cylinder. By moving the stylus in any direction, the ground circle is created. If the size of the ground surface is in the desired size, the stylus can be released and the real object can be built. In contrast to the creation of the real cube, the CW versus CCW check is not necessary for the cylinder. Instead, the calculation of the texture coordinates and the vertex normals has to be done.

Equal to the cube, the real object takes place in the scene graph instead of the dummy object. Also equally to the cube, the ground surface is built in the XY-plane of the target, and the extrusion is done along the Z axis.

5.3.6 Polygon

By clicking on the display, the user specifies the first corner. By moving and releasing the stylus, the second corner is defined. From now on moving and releasing the stylus generates a new corner. During the movement

of the stylus, a preview of the current surface is displayed. To finish the creation process the user either performs a double click at the last edge or presses the *Finis* button of the GUI. In the actual version of our application, only the creation of convex polygons is allowed. Since there is no test if this specification is met, the behavior of polygon with overlapping edges during the extrusion is undefined.

For the building of the real object, we start again with the CW versus CCW check. After the registration of the vertex coordinates, the texture coordinates and the vertex normals are calculated and stored in the scene graph of the object.

Analog to the other 3D-objects the dummy object, is removed from the scene graph, and the new object is included.

5.4 2D-Content

The creation process of 2D-objects can be split up into two different steps. Similar to the dummy objects during the creation of 3D-objects, we now use a frame buffer until the drawing phase. Any created pixel is stored in this buffer until the user finishes the creation process.

If the user finishes the creation either by leaving the 2D-mode or by clicking the *Finish* button of the GUI, the real object is created. Therefore we calculate the used space of the drawing and start the generation of the texture analog to the method described in section 5.5. The single difference of the new produced texture is the different color mode. Since the 2D-objects have to deal with transparencies, we use the RGBA444 format instead of the common RGB565. After the creation of the texture, all necessary parts for the scene graph representation are generated and assigned to the global scene graph.

5.4.1 SG-Representation of 2D-Objects

As shown above, any 2D-object is in principle a planar rectangle tagged with a texture.
There is an example of a 2D-object including the position, rotation, and scaling informations:

```
1  <SgTransformSeparator>
2    <SgTransform translation="X Y Z" name="Translate" />
3    <SgTransform rotation="W X Y Z" name="Rotate" />
4    <SgTransform scaleFactor="X Y Z" name="Scale" />
```

```
5    <SgDrawingStyle blending="true"
                blendFuncSrc="BLEND_FUNC_ONE_MINUS_SRC_ALPHA"
                blendFuncDst="BLEND_FUNC_SRC_ALPHA" />
6    <SgMaterial diffuse="1 1 1 1" />
7    <Node name="ProjectItObjectSKETCH">
8      <SgTextureSeparator>
9        <SgTexture file="" name="alpha1" />
10       <SgGeometryVertices>
11         <Vec3Array type="coordinates" size="4">
12           X Y Z; X Y Z; X Y Z; X Y Z;
13         </Vec3Array>
14       </SgGeometryVertices>
15       <SgGeometryTexCoords>
16         <Vec2Array type="tex-coords" size="4">
17           0 0; 0 1; 1 0; 1 1;
18         </Vec2Array>
19       </SgGeometryTexCoords>
20       <SgGeometryNormals>
21         <Vec3Array type="normals" size="4">
22           0 0 1; 0 0 1; 0 0 1; 0 0 1;
23         </Vec3Array>
24       </SgGeometryNormals>
25       <SgGeometry hasNormals="true"
                  hasTexCoords="true">
26         <IntArray type="indices" size="6">
27           0 1 2 2 1 3
28         </IntArray>
29       </SgGeometry>
30     </SgTextureSeparator>
31   </Node>
32 </SgTransformSeparator>
```

The lines two to four keep the transformation information about the object.

Line six contains the color information about the object.

Line seven is the root node for the geometrical information about the object.

Line nine contains the information about the texture. The prefix 'alpha' at the filename is an advice for a texture with transparencies.

The lines 10 to 29 contains the vertex coordinates, the texture coordinates, the normal vectors, and the indices of the vertices for the rectangle.

The lines 30 to 32 are just the closing tags of the object.

5.4.2 2D-Objects

In the current version of our application, we provide four different 2D-objects respectively drawing states.

- Freehand
- Line
- Box
- Circle

In the *Freehand* mode, sketch drawing can be done. Every movement of the stylus has a direct effect at the display. If the *Line* mode is selected, the line starts where the display is clicked by the stylus. During the movement of the stylus, a preview of the line is displayed. If the stylus is released, the line in the specified color is drawn. The behavior while drawing a box or a circle is comparable to the drawing of a line. Here, also a preview is displayed until the stylus is released. Now the drawing is displayed unchangeable and in the correct color.

Additionally to the different drawing modes, we enable further scope for design:

- Coloring
- Changing the line width

During the creation process of 2D-objects, the additional color GUI is also shown. The current drawing color can be selected from the user by clicking on the corresponding color button. Of course it is possible to use different colors within a single 2D-object. The line width of the *Freehand* and of the *Line* mode are variable and can be changed by using the *LineWith++* or *LineWith–* buttons of the GUI.

5.5 Texturing

Texturing is the main functionality to allow the user the rebuilding of realistic models of the real environment. It should be easy to create a texture for the user as it should be easy to assign the textures to any kind of object.

There are three different methods to texture a 3D-object in our application. On the one hand, we provide ten different predefined textures, on the other hand, the possibility to generate and store user-defined textures and additionally the so-called *FasTex* function. Using *FasTex* allows the user to texture an object during the creation and enables a very fast and easy tool for texture generation. The administration and the handling of the textures is done by the *Texture Manager*.

5.5.1 Texture Manager

The *Texture Manager* performs three main functions:

- Load and save user-defined textures
- Handle multi-used textures
- Handle predefined textures

In the *Studierstube* frame work, it is not possible to store a texture in a graphic file in a direct way. Of course it is possible to load a picture from a file (e.g. jpg or png) and assign it as texture to an object. But there is no possibility to save a texture. Moreover, it is impossible to get a handle to an actually used texture from the scene graph nor from the object the texture is assigned to. So, the *Texture Manager* administrates the handles of all textures of the current scene.

Since the double usage and double storage of one and the same texture is useless, the *Texture Manager* also keeps the overview of multi-used textures. If a texture is assigned to a further object, the *Texture Manager* just passes a handle to the texture to avoid a multiple memory consumption.

The *Texture Manager* also knows about the details of the predefined textures. So, it loads the textures from the database in case of the first use and provides a handle to them on every further usage. Since the predefined textures are stored locally anyway, the texture itself is not stored anymore if the scene is stored.

5.5.2 Predefined Textures

As shown in Fig. 5.1, there are ten different predefined textures available. We selected ten commonly used surface designs. Since these textures are part of the application, they are not saved as picture if the scene is stored. For these textures, there only exists a reference in the scenegraph, and the *Texture Manager* handles the loading and displaying if they are in use.

To assign one of these textures to a 3D-object, first the object has to be activated. By selecting the menu *MANIPULATE* ⇒ *TEXTURE*, the GUI is extended by the preview of the different textures (as shown in Fig. 7.3 and 7.4). Now the desired texture can be assigned by pressing the corresponding button.

Figure 5.1: Predefined textures

5.5.3 User-Defined Textures

To increase the flexibility of the application, we provide the possibility to create textures from the environment. To create one's own texture, the user hast to select *MANIPULATE* ⇒ *TEXTURE* from the menu. Now, the GUI is extended by the special texture menu as described in section 7.1.2. As shown in Fig. 7.4, this extension contains four user-defined buttons additionally to the predefined texture buttons. After pressing the *CREATE* button, the user has to specify four corners on the touch-screen. During the creation process, the so far selected area is highlighted by a line on the display. The selected area does not have to be rectangular since it would be impossible to create textures from an incline angle. To finish the creation process, either the user has to double-click on the last set corner point or he has to press the *FINISH* button of the menu. The finish button was added to the GUI because during our tests we found out that a double-click on

the mobile device with the stylus is not as easy as a double-click with the mouse on a desktop computer.

If the user selects more or less than four corners, the application shows an error message in the *Status Line* and the creation process is stopped. Now the selected area is mapped onto a 128 x 128 pixel sized texture and assigned to one of the four user-defined buttons. (Details about the mapping algorithm follow in the next section.) A message in the *Status Line* informs the user which button the currently built texture was assigned to. If the user creates more than four textures, the 'oldest' texture is discarded and the new texture is assigned to this button.

In Fig. 5.2, we show two different cut-off areas and the resulting textures.

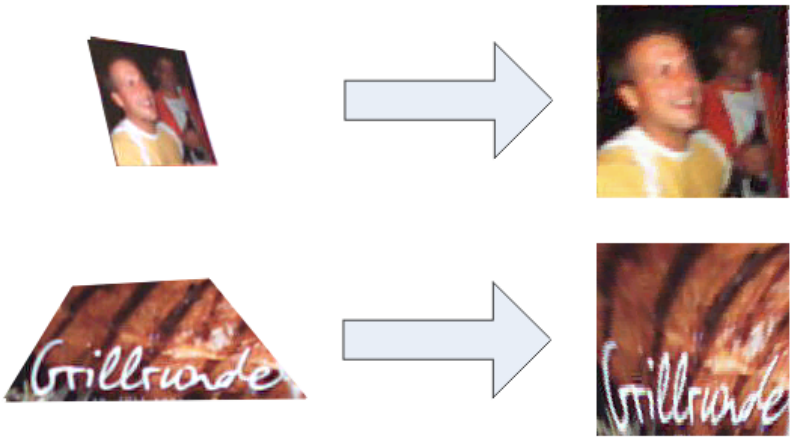

Figure 5.2: Mapping examples of user-defined textures

5.5.4 FasTex

FasTex is a special form of a user-defined texture. It generates a texture from the displayed background and assigns it to the currently active object in one single step. Therefore the user just had to press the *FASTEX* button which is available in any 3D-object menu. In contrast to the 'normal' user-defined texture, a *FasTex*-generated texture is not assigned to one of the user-defined buttons. So, a reuse of the same texture with another object is not possible. The *FasTex* functionality is only available during the creation process of an object. If the object has been manipulated in any way, *FasTex* cannot be used any more.

Of course, any texture built with *FasTex* is handled by the *Texture Manager* in the same way as another user-generated texture.

5.5.5 Implementation Details

The *Texture Manager* is implemented as *Singleton*. (Singleton is a common and often used Design Pattern, so we give a detailed description here. Detailed information is available in [GHJV95] (Page 144ff).) It contains a vector to handle the predefined textures and a vector to handle the user-defined textures. Furthermore it provides the functionality to load and save textures from and to the default file system or from and to zip files. Since it is impossible to store a texture simultaneously when saving a *Studierstube* scene graph, the *Texture Manager* is involved if a scene is loaded or stored by the *IO-Manager*. The filename of the corresponding texture is stored in the 'name' field of the SgTexture node, and if necessary, the *IO-Manager* calls the *Texture Manager* during the reading and writing operations.

By reason that any 2D-drawings are displayed and stored as textures and these textures need an alpha channel, the *Texture Manager* also enables the translations between RGB565 and RGBA4444 color formates. And of course it handles the different 2D-textures.

The calculation of the result color for the user-defined textures is done straight forward as shown in Fig. 5.3 and the formulas 5.1 to 5.4. The vectors A to D point to the four corners of the selected quad.

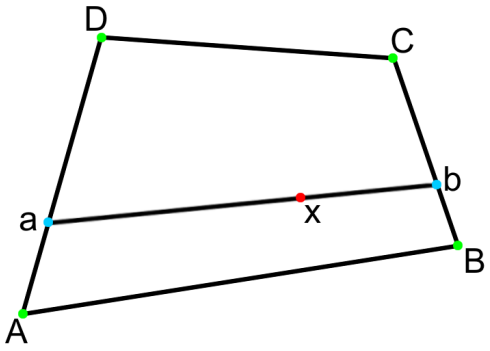

Figure 5.3: Mapping of user-defined textures

The vectors **a** and **b** are calculated as follows:

$$\mathbf{a} = \mathbf{D} + v * (\mathbf{A} - \mathbf{D}) \tag{5.1}$$

$$\mathbf{b} = \mathbf{C} + v * (\mathbf{B} - \mathbf{C}) \tag{5.2}$$

The vector **x** is calculated by using the two vectors **a** and **b**:

$$\mathbf{x} = \mathbf{a} + u * (\mathbf{b} - \mathbf{a}) \tag{5.3}$$

Now, **x** is the base for the bilinear interpolation of the color values of the four neighbor pixels where u and v are the normalized texture coordinates in the range 0..1:

$$\begin{aligned} Texture(u,v) = &(1-u)(1-v) \cdot x(\lfloor u \rfloor, \lfloor v \rfloor) \\ &+ u(1-v) \cdot x(\lfloor u \rfloor + 1, \lfloor v \rfloor) \\ &+ (1-u)v \cdot x(\lfloor u \rfloor, \lfloor v \rfloor + 1) \\ &+ (1-u)(1-v) \cdot x(\lfloor u \rfloor + 1, \lfloor v \rfloor + 1) \end{aligned} \tag{5.4}$$

In Fig. 5.2, we show two example results of this kind of calculation.

The *Texture Manager* also keeps an overview of the used storage names for the single textures. The textures are stored as png files where the file names are sequential numbers. If a scene is loaded, the *Texture Manager* keeps the highest number in use and starts the sequence for new file names one number above.

The *Texture Manager* also differentiates between a texture for a 3D-object and a texture for a 2D-object. Textures for 2D-objects are stored with the prefix *alpha* followed by the sequential number (e.g. alpha7). The main difference to a texture for a 3D-object is the differing color format. For 2D-objects we use not only red, green, and blue as color channel but also an alpha channel to get the transparency.

5.6 Object Manipulation

Object manipulation provides the essential functions to build up interesting and complex scenes out of a small number of simple objects. Therefore it is necessary to arrange the single objects in the scene and to edit the surface of them. We offer three main methods to arrange respectively rearrange single objects in the scene:

- Moving
- Rotating
- Scaling

Additionally to the transformation functions, the look of a scene is influenced by:

- Coloring
- Texturing

With these operations, it is possible to create complex scenes out of simple objects.

5.6.1 Object Selection

To select the different objects of the scene, we provide a very simple method for the user. By clicking on the 'Step' button of the GUI, the user can choose one object after the other. Starting with the last created object, the selection is done object by object backwards until the first object is reached. Of course, this is not the most elegant method to select an object from a 3D-scene, but it fullfills our purpose. In one of the next *Studierstube* iteration there should be a ray-picking function to select single objects from a scene graph. As soon as this functionality is available, it will take just a few changes in the code to implement it. Additionally, the 'Step' button is needless and the GUI would be a bit more cleaner.

5.6.2 Implementation Details

The selection and manipulation of the objects is done by manipulations of the scene graph. Therefore the application contains a vector which holds a pointer to the root node of each object. Furthermore it contains a special manipulation separator node where all changes were performed to. If the user presses the 'Step' button, the application steps backward through this vector. It takes the root node from the scene graph and replaces it with our special manipulation separator node. Additionally, the color of the object is changed to a light green to mark the object as active. Thus, all manipulations (e.g. translations, changing colors or textures) are addressed to the manipulation node reaches the current active object without any accurate knowledge about the object itself. As shown in 5.3.1 and 5.4.1, the general structure of the scene graph of any object is identical. Thus, there is no

difference by either moving a cube, a cylinder, or a 2D-object. If the user selects another object, the current active object is, after resetting the color, placed back again in the scene graph where it had been before.

5.6.3 Moving

To move an object the object, first has to be selected. If the user chooses *MANIPULATE* $\Rightarrow$ *MOVE* from the menu, an additional object is inserted into the scene. A colored coordinate system symbol (CSS) is added to the currently selected object. The colors of the CSS for X, Y, and Z are analog to **R**ed, **G**reen and **B**lue, and the same colors are in use for the GUI buttons to select the axis where the transformation belongs to. For the moving operation, the CSS is aligned by the axis of the corresponding NFT target.

Now the user has to choose the axis to perform his manipulation. If an axis is selected, the label of the button changes from '- X -' to '–>> X <<–' to highlight the selection. Any movement of the stylus is translated into a movement of the object. Therefore the origin point is stored and compared to the actual stylus position. Moving of the stylus to the right or to the top performs a moving of the object in a positive way along the axis and vice versa. The translation is transmitted directly to the corresponding scene graph node of the object (cf. 5.3.1, line two of the scene graph).

Figure 5.4: Cube during rotation manipulation around the Y-axis with shown CSS

5.6.4 Rotating and Scaling

The usage of rotating and scaling manipulations are as far as possible the same as described in the section above. The only real difference is that the CSS for rotating and scaling is not aligned to the axis of the NFT target, but it is aligned at the axis of the object. In Fig. 5.4, we show an object during the rotation with the object-aligned CSS. The operation is performed around the Y-axis as can be seen at the changed label of the button ('–>> Y <<–' instead of '- Y -').

5.6.5 Color and Texture Selection

To change the color of a 3D-object, the object has to be selected first. After choosing *MANIPULATE* $\Rightarrow$ *COLOR* from the menu, the GUI is extended by the color menu as shown in Fig. 7.3. Now the user can select one of the 16 defined colors to assign them to the object. Additionally it is possible to change the transparency of the object by moving the stylus on the touch-screen. Any changes done here are also written directly in the corresponding scene graph node.

The creation and assigning of different textures was described in detail in section 5.5, so we give further explanations there.

Chapter 6

Content Management and Distribution

In the following chapter, we describe details about the possibility of the reuse of objects using the *Object Explorer*. Furthermore we talk about the storage as well as the distribution of the scenes.

6.1 Object Explorer

With the *Object Explorer*, we enable the user a simple way to reuse single objects. The objects can be inserted into the scene where they were created as well as in a completely different environment. So, the user can use the *Object Explorer* taking single AR objects with him. Relating to the distribution of AR models of real existing objects, the *Object Explorer* offers a great benefit.

The main functionality of the *Object Explorer* provides:

- Browse through the *Object Explorer*
- Add an object to the *Object Explorer*
- Remove an object from the *Object Explorer*
- Insert an object into the scene

To show the *Object Explorer*, the user has to choose *Explorer* $\Rightarrow$ *Show* from the GUI. If the *Object Explorer* starts, an additional, special GUI is displayed. In Fig. 6.1, we show the *Object Explorer* including the GUI and the preview of an object. By clicking on one of the two arrow buttons,

Figure 6.1: The *Object Explorer* and its special GUI

the *Object Explorer* shows the next or rather the previously stored object. Since we deal with 3D-objects, the preview of the single object is not displayed statically. In fact, the object is rotating to allow to regard it from different angles and from different views. Fig. 6.2 shows the preview of an object during the rotation.

To add an object to the *Object Explorer*, the user just has to select the object and press the *Add 2 Expl.* button from the GUI. The new object is added at the end of the *Object Explorer*.

An object can be removed from the *Object Explorer* by pressing the *Delete Object* button. If the button is pressed, the current shown object is deleted from the *Object Explorer* and the next object is previewed.

To insert an object from the *Object Explorer* into the current scene, the user just has to press the *Add 2 Scene* button. The object is inserted into the scene graph as if it was a newly created object. It is placed in the origin of the actual NFT target and of course can be manipulated as well as any other 3D-object. Needless to say that our *Texture Manager* takes care of the texture of the object and manages a possible reuse.

6.1.1 Implementation Details

The *Object Explorer* is more or less a self-contained module. The only point of contact with the application is the node in the scene graph. As shown in chapter 5.2 (line four of the scene graph), there is an extra root node for the *Object Explorer*. During the time the *Object Explorer* is active,

Figure 6.2: Rotation of an object displayed by the *Object Explorer*

the regular scene is not shown so as to avoid confusions of the user. All other operations like loading, saving, and displaying of the objects are administrated by themselves. On the file system, there is an own folder only to store files belonging to the *Object Explorer*. In addition to the stored objects and textures, this folder includes a text file (*index.txt*) containing structural informations. The structure of the file is as follows:

```
N F:file1.xml T:texture1.png .... F:fileN.xml T:textureN.png
```

where:

N	is the number of objects
F:xxx	is the filename of the object
T:xxx	is the texture name of the object

A concrete example looks like this:

```
3 F:0.xml T:DEFAULT_TRAFFICSIGN1 F:1.xml T:1.png F:2.xml T:NONE
```

In this case, the *Object Explorer* contains three objects. The first is textured with a default texture, the second one with a user-defined texture, and the third has no texture.

During the initialization process, the index file is parsed, and an adequate data structure with the file names is built. Additionally, some needed scene graph nodes are built, for instance the `SgPoseAnimator` for

the rotation of the previewed object. After the initialization, the first object is displayed, and the *Object Explorer* is working.

The *Object Explorer* manages the loading and saving of the objects by itself. It also deals with the file names for the objects and the textures. If the user adds or removes an object, the corresponding files are saved or deleted, and the index file is adapted.

6.2 Storage

Saving and loading of scenes is, of course, very important for an application like ours. Without the possibility of reusing single objects or complex scenes, the creation of content does not really make sense. With this prototype, we provide two different options to store a scene. It can be stored locally on the device for testing purposes and globally on a server for multiple user access.

The loading and saving operations are completely handled by the *IO-Manager*, which also takes care of the communication with the pulic server.

6.2.1 IO-Manager

The *IO-Manger* handles all things related to loading and storing of the scene except the loading and storing of textures. The textures are handled by the *Texture Manager*. The *IO-Manger* is also in charge of the communication with the global server. He deals with the IO operations of files from the file system as well as with IO of single files from a zip-file. The *IO-Manager* also takes care of the location where a scene was loaded from. So there is only a single *Save* button in the GUI, and the *IO-Manager* knows about the correct target of the request.

6.2.1.1 Local Storage

The local storage is more or less just implemented for testing reasons. Otherwise, it would be impossible to work with the application without a connection to the content server. For every NFT-target, the scene is stored in a single xml-file. The textures of the particular scene are stored in a separate folder. In contrast to the global storage, all necessary files are stored separately on the device and not combined in a single zip-file.

6.2.1.2 Global Storage

The global storage is the core of the distribution module. A usage by a huge number of users is only possible if the scene and all according data are stored on a publicly reachable server. In our case, we use the 'imagination' server which is also in use for the *IP-City* project. The server allows the upload and download of files after a login. All communication with the server is done with 'http-post' requests. At the beginning of the communication, a cookie is created to keep the identification of the device. The server can deal with the following commands:

login	requires UserName and Password
UserName	user name for login
Password	password for login
ObjectID	object ID for specific operations
TypeID	type ID to differ the object types (e.g. zip, jpg, txt)
queryObjects	requires a name and returns a list of ObjectIDs
deleteObject	requires ObjectID and deletes an object (is not working correctly during our testing sessions, so the objects are left on the server and have to be removed manually)
getObject	requires ObjectID and downloads the object from the server
uploadObject	requires typeID, a name, and the data of the object
logout	to close the connection

If a scene is downloaded from the server, after the login, the application sends a *queryObjects* request with the according file name. The name depends on the actual NFT target. The server returns either a single object ID or, if there are more than one file with the same name, a list of an object ID. (Due to the reason that the *deleteObject* command does not work, it is most likely to get a list of IDs.) If the server returns a list of IDs, the largest ID is the one to choose because the server assigns the IDs as serial numbers. The next sent request is the *getObject* request to start the download of the file.

The global scenes are stored in so-called piz-files where piz stands for **P**roject**I**t**Z**ip file. (ProjectIt was the working title of the application.) The usage of a zip-file has two advantages. On the on hand, the traffic to the server is reduced, and on the other hand, there is only one single file to handle. Otherwise, the number of files for a scene can vary depending on the number of used textures.

If the download is completed, the file is mounted into the *Studierstube*

file system. *Studierstube* can handle zip files as if they were 'normal' folders on the file system. So, the loading process is handled by the *IO-Manager* using the build-in *Studierstube* functionality.

If all changes to the scene has been done, the scene has to be uploaded to the server again. Before the scene is uploaded to the server, all relating files (scene file and textures) are stored in a piz file again. After the *IO-Manager* has connected to the server, he sends the file via the *uploadObject* request. Now, the new scene is available for any other user working with our prototype.

Chapter 7

Graphical User Interface

In the following chapter, we describe the functionality and the structure of the Graphical User Interface (GUI) and the Status Line. We illustrate the design and explain the implementation details.
The design and structure of the GUI are important for the user acceptance of any software. So, the structure of the GUI is clearly and logically arranged.

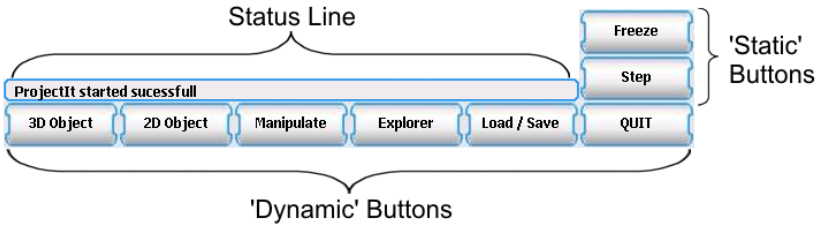

Figure 7.1: Design of the GUI and the Status Line

7.1 Structure of the GUI

The GUI is placed at the bottom of the screen and is built up of the *Status Line*, six 'dynamic' and two 'static' buttons. The 'dynamic' buttons change their functionality, text, and color depending on the current program state. The two 'static' buttons keep the same functionality all over the application. (See Fig. 7.1)

Figure 7.2 shows all menu entries. It also illustrates that there are only

two more layers under the main menu. So, we enable the entire functionality of the program with a very flat menu structure.

The two 'static' buttons are the *Freeze-Button* and the *Step-Through-Button*. The *Freeze-Button* allows the user to freeze the current view. So, the movement of the device has no influence on the displayed content.
With the *Step-Through-Button*, the created objects can be selected one by one. This is necessary for every manipulation (coloring, texturing, moving, ...) to any objects.
These two functions are in use very often. So, we decided to take them out of the 'dynamic' menu to keep the depth of the menu as flat as described above.

A detailed description of all of the menu functions is done in the corresponding specific chapters.

7.1.1 Color-GUI

In addition to the standard GUI, we provide an extension to select different colors. This GUI is only displayed if it is necessary and useful. The *Color-GUI* allows the selection of the surface color of 3D-objects or the selection of the pencil color for 2D-objects. The user can choose from 16 predefined colors.

To keep the work space as wide as possible, the *Color-GUI* is shown at the very right side of the display. Figure 7.4(a) shows the single *Color-GUI*, and 7.3 shows the *Color-GUI* on the display of the mobile device.

7.1.2 Texture-GUI

Analog to the *Color-GUI*, the *Texture-GUI* is also an extension of the default GUI. It provides ten predefined textures and four buttons for user-defined textures. If the user generates own texture from his environment, the texture is assigned to one of the four buttons. Additionally the number of the button is shown in the *Status Line* to inform the user which button the texture is assigned to.

Also analog to the *Color-GUI*, the *Texture-GUI* is placed on the right side of the display to keep the work space for the user wide. Figure 7.4(b) shows the design of the *Texture-GUI*.

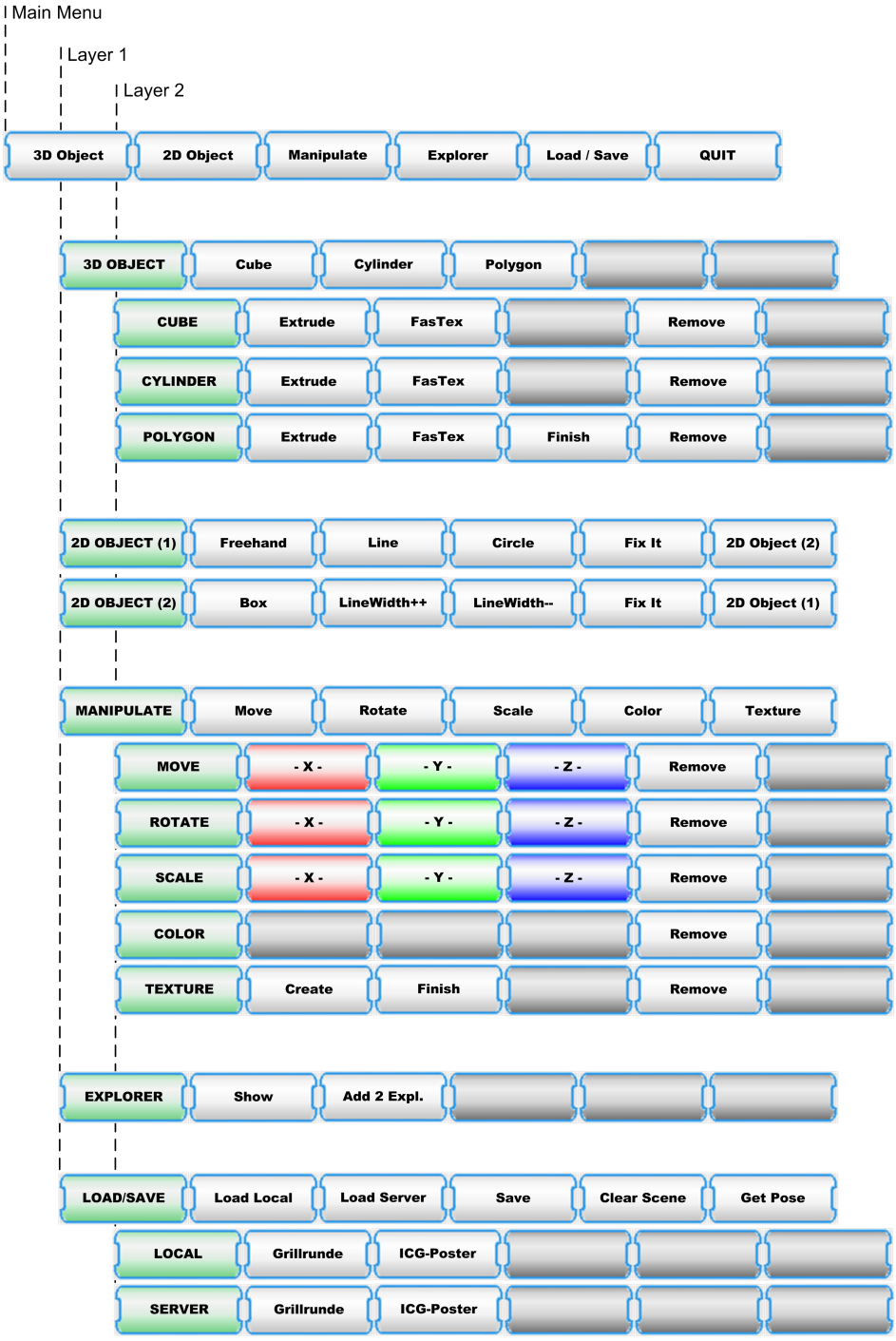

Figure 7.2: Overview of all possible GUI-states

Figure 7.3: Displayed Color-GUI with differently colored objects

7.1.3 State Pattern and Implementation Details

According to [GHJV95], the *State Pattern 'allows an object to alter its behavior when its internal state changes'*. So, every button of the GUI changes its behavior according to the current menu state.

The big advantage of this approach is that the button triggers different actions according to its state. Hence, it is unnecessary to write endless `if - else if` (or rather `switch - case`) statements, which are hard to maintain if any changes are necessary.

7.1.3.1 State Pattern

The state pattern is a clean way to change the behavior of an object during runtime. In Fig. 7.5 we show UML-Diagram of the GoF approach.

We expand the functionality of the *classical* state pattern by a *returnToPreviousState()* method. So, the actual state not only knows about eventual following states but also knows about the primary state. Hence, a step to the menu a layer above the current state is easy to perform. There are two advantages of this functionality. On the one hand, it is not necessary for a child state to know something about its parent state. So, in case of refactoring or other changes in the state hierarchy, there are no changes at the called state. On the other hand, it is possible to access one and the same child state from different calling states, and the method will return to the correct caller state.

(a) Color (b) Texture

Figure 7.4: Design of the Color-GUI and the Texture-GUI

7.1.3.2 Implementation Details

In Fig. 7.7, we show the UML-Diagram of the GUI and all belonging classes.

The *GUI*-Class is the core component of the implementation. It controls the buttons and holds an instance of the actual *MenuState*. It is also responsible to show and hide the *ColorGUI* and the *TextureGUI*.

We decide to implement our own *GUIButton*-Class instead using the existing *StudierstubeES::Button* because its current implementation has some disadvantages. One of the biggest disadvantages (especially for our application) is the fact that a mouse click on a button also triggers all other registered *RawInputListeners*. So, an application which uses the mouse input e.g. for drawing has no chance to decide whether the goal of the mouse event is a click on a button or a click on the work space. There is no native possibility, like 'EventHandled' or a similar one, to catch a mouse click at the *StudierstubeES::Button* and stop the event passing further through the *Studierstube* event system. For that reason the *GUI*-Class provides the boolean variable `mouseEventHandled` to distinguish between a click on a button and a click on the work space.

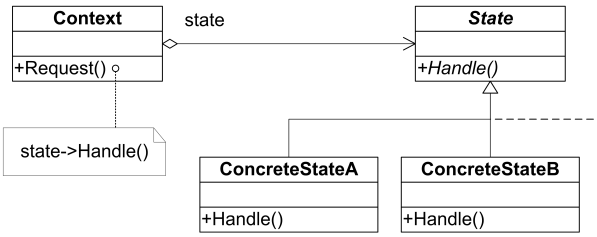

Figure 7.5: The State Pattern described by [GHJV95] (p. 339)

The *GUIButton*-Class provides the concrete GUI-Button. It contains informations about the text of the button, the position where a button is displayed, and whether the button is pressed or not. It also contains a pointer to the *GUIImages*-Class to make the different backgrounds available.

At last, the *GUIImages*-Class provides the variably colored background images for the GUI-Buttons. There are seven different colors available (cf. Fig. 7.6):

- gray as default color
- dark gray for inactive buttons
- dark green for the currently active state
- orange for the currently pressed button
- red, green and blue for manipulations along the X-, Y-, and Z-axis

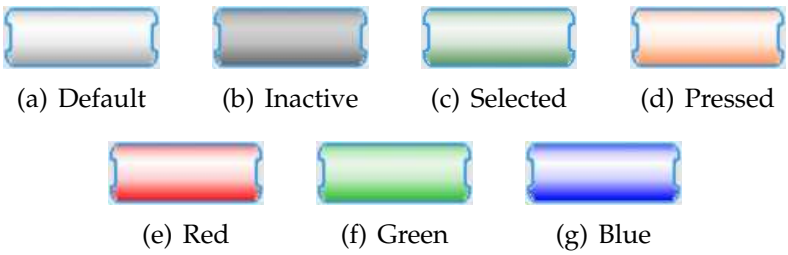

Figure 7.6: Available colors for the GUI-Buttons

The class delivers a pointer to every image. So, every image is represented only once in the memory to save memory capacity.

7.1.4 Status Line

The aim of the *Status Line* is to offer the user information about the current state of the program and additional information about actions he has set.

It is important to give the user a feedback, especially if the processing of a user input takes a longer time, or e.g. depends on the response of a server. Otherwise, it is hard for the user to understand why there is no directly visible reaction to his action.

7.1.4.1 Implementation Details

The *Status Line* is a simple class, implemented as Singleton. So, it is reachable from all over the program.

It supplies three main methods: `addInfo()`, `addError()` and `clearLine()`.

The difference between `addInfo()` and `addError()` is the color of the displayed text. Using `addInfo()` displays a black text and `addError()` displays a red text. So, `addError()` is used if an unexpected behavior occurs to give the user an additional hint.
The result of the clearLine() method is, as expected, an empty *Status Line*.

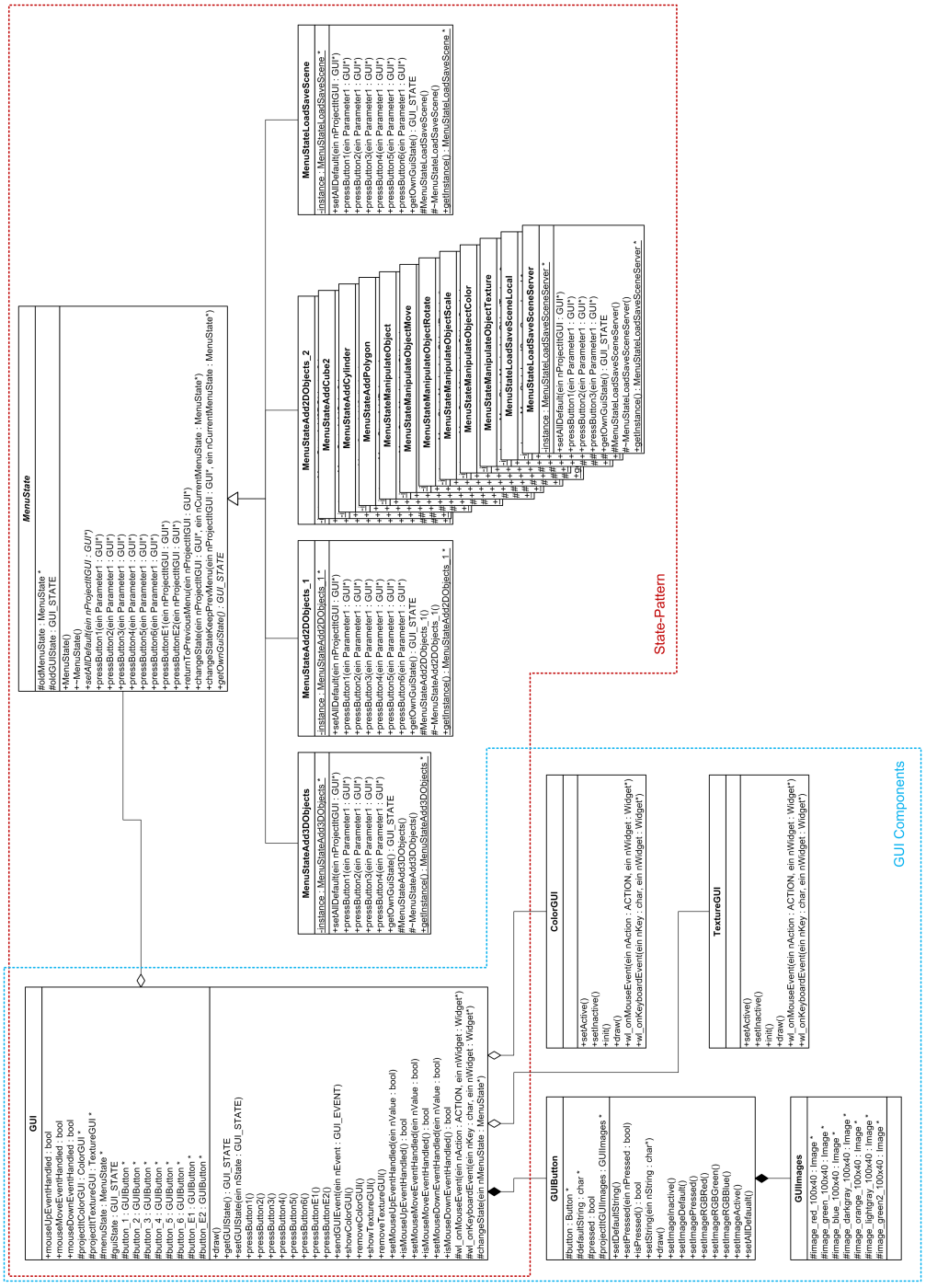

Figure 7.7: UML-diagram of all involved classes of the GUI

Chapter 8

Conclusion

In this work, we presented a prototype to enable the creation and sharing of AR content in a new and easy way. The system runs directly on mobile devices and is independent from the use of desktop computers. It gives the user a kind of freedom to create virtual objects from his current environment and allows the sharing of these objects with others. We take care of the limitations of the target platform and we deliver a simple interface for all kinds of manipulations.

At the initialization phase, the pose estimation of the user has to be done. The localization of the position is calculated on a remote server using a picture of the actual environment. The user just has to press one button if he wants to start working with our program. After a few seconds, when the calculations are done, the NFT dataset is loaded automatically. With this dataset, the online tracking is enabled. If there is already a AR scene available for the actual location, the scene is also loaded and displayed to the user.

Now, the generation and manipulation of the content is possible. Any action to the application can be done using the touch screen (in combination with the stylus) of the mobile device. Compared to a computer mouse with at least two or three buttons, the capabilities of the stylus are limited. But we keep that in mind by building a very flat GUI structure. Any function can be reached via the clearly arranged GUI with a depth of at most three layers.

New objects can be added just by drawing the desired base area of the object and by extruding this area. In this simple way 3D-objects can be built. It is possible to build boxes, cylinders and polygons. To enable more attractive scenes, the objects can be colored in different ways or they can be textured. We provide a number of predefined textures and additionally

the possibility to generate new textures out of the environment. With these textures the replication of virtual objects from the real world can be done real simple. The texture can be generated by defining the four corners of it, or, the object can be textured with the current background by a single click.

Another type of object we offer is a simple 2D-object. The user can add some geometrical objects like circles or rectangles to the scene. It is also possible to use the stylus as a simple pen to do some sketch drawings. The width of the line is changeable just as well as the drawing color. With these 2D-objects, simple annotations can be add to the scene.

Manipulations and rearrangements of the different object can be done by the well-known moving, rotating, and scaling operations. A cross hair with different colors in the main axis supports these functionalities. The colors of the cross hare are the same as the corresponding buttons at the GUI to allow an intuitive usage.

The sharing and distribution of the scenes is automatically done again. If the user has finished the editing of the scene, the upload to the publicly reachable server is started by just one click. The system itself knows about the current location and stores the scene and all the depending information. Now, the new scene is available for any other user located in the same environment.

From my point of view, the presented system could be a first step to enable the user the generation of AR content. To become a widely accepted application, some improvements are necessary. It is possible and easy to create simple objects but it is really hard to generate complex scenes.

We did no user study because it would go beyond the scope of this work, but I think one of the main results would be the complicated handling of the single objects and the difficult object selection. It is imaginable to implement e.g. a *view and select* method where the user is able to select an object just by pointing the view at it.

Another problem coming along with complex scenes is the required loading time on the currently available devices. The improvements of the mobile processors, however, will lower this problem in the next few years.

After the re-engineering of the interface followed by a figured-out user study, this project can be the base of further interesting developments in the direction of mobile AR content generation.

Chapter 9

Future Work

The future work can be split up into two different areas. On the one hand, some improvements of the prototype are possible, for example improvements to enable a better handling for the user. On the other hand, some extensions are imaginable, e.g. extensions to widen up the field of application of the system. And, of course, a very interesting topic will be a user study for our application.

As shown in chapter 5.6.1, the selection of the different objects has to be done by selecting one object after the other until the desired object is reached. This method is, especially for huge scenes, not very comfortable. The implementation of a ray picking algorithm would bring a great benefit for the usability. There are considerations to implement ray picking directly in the *Studierstube* frame work. If that extension is done, the embedding of this technique in our prototype can be done in a short time and would have a big impact on the usability.

Another improvement could be the adaption of the size of user-generated textures. In the actual version, all user-generated textures are mapped to a 128×128 pixel sized texture. This size was found as a good trade-off between all available input sizes and was also taken due to storage reasons. If the original cut-off from the user is very small or very huge, the fixed sized texture may deliver poor results. Here it is imaginable to adapt the size of the texture according to the size of the user cut-off to reach more attractive results.

The next possible improvement could be the possibility to group some objects together since it is really complex and time-consuming to do an equal manipulation to a multiple number of objects. So, if the user wants to move e.g. ten related objects about the same amount in the same direction, he has to select every object and perform the operation object by

object. If the objects could be defined as a group, the manipulation can be done in a single step. Additionally, the relations between each other will not change in this case.

Furthermore it is imaginable to allow the user the sharing of single objects by the use of an additional global *Object Explorer*. So, it would be possible to share not only whole scenes according to a specific place, but also to share single objects with other users. Of course, a one-to-one reuse of the local *Object Explorer* will meet its limits shortly because there is no possibility to get an overview of all stored objects, and it is not practicable to step through a huge number of objects step by step. Here is room for further considerations to this topic.

During our test directly on the mobile device, we have seen that writing some annotations is not very comfortable. Even if the *Freeze Mode* is in use, writing a good-looking text nearly seams to be impossible. So, the possibility of entering a text using a virtual keyboard on the touch screen would raise the comfort and the usability.

Due to the evolution of the localization technique described in [AWK+09], a usage of this system would reduce the necessary communication with different servers since the new method runs directly on the mobile device. An upgrade to the new technique will result in an acceleration of the initial tracking process.

Also some real extensions are possible. Actually, we generate only static content. But it is also imaginable to extend the system and allow the usage of video content in our scenes. There are a lot of web-based platforms providing a different kind of videos. They could be used as a source to include moving images in the scenes. So, the application itself could be even more attractive for different users.

Analog to the usage of video content, the integration of audio content is also a possible extension for the current scenes. They could be used as background music to enhance scenes or containing the information itself, e.g. to add short spoken messages to special objects and to provide these messages for the other users.

And last but not least, the evaluation of our prototype with real users is necessary because the acceptance of an application by the user can only be found out by integrating the user in the evolution process. Needless to say that we try to build up a user-friendly interface and an intuitive menu. Whether the user agrees with us, however, has to be decided by the user.

Bibliography

[AWK+09] Clemens Arth, Daniel Wagner, Manfred Klopschitz, Arnold Irschara, and Dieter Schmalstieg. Wide area localization on mobile phones. In *Proceedings of the 2009 8th IEEE International Symposium on Mixed and Augmented Reality*, pages 73–82, 2009. 16, 54

[BMC08] Pished Bunnun and Walterio W. Mayol-Cuevas. Outlinear: an assisted interactive model building system with reduced computational effort. pages 61–64, 2008. Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality (ISMAR'08). 4

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. VIII, 32, 46, 48

[HFH+01] Tobias Höllerer, Steven Feiner, Drexel Hallaway, Blaine Bell, Marco Lanzagorta, Dennis Brown, Simon Julier, Yohan Baillot, and Lawrence Rosenblum. User interface management techniques for collaborative mobile augmented reality. *Computers and Graphics*, (25):799–810, 2001. 3

[MGDB04] Blair MacIntyre, Maribeth Gandy, Steven Diw, and Jay David Bolter. Dart: A toolkit for rapid design exploration of augmented reality experiences. In *Symposium on User Interface Software and Technology*, 2004. 4

[NKG04] Michael Bang Nielsen, Gunnar Kramp, and Kaj Gronbak. Mobile augmented reality support for architects based on feature tracking techniques. In Martin Bubak, Geert D. van Albada, and Peter M. A. Sloot, editors, *Computational Science - ICCS 2004*, pages 921–928. Springer-Verlag Berlin Heidelberg, 2004. 4

[Pie06] Wayne Piekarski. 3d modeling with the tinmith mobile outdoor augmented reality system. *IEEE Computer Graphics and Applications*, pages 14–17, January/February 2006. 4

[UTO+04] Kengo Uratani, Daisuke Takada, Takefumi Ogawa, Takashi Machida, Kiyoshi Kiyokawa, and Haruo Takemura. Wearable augmented reality system with annotation visualization techniques using networked annotation database, 2004. The 3rd CREST/ISWC Workshop on Advanced Computing and Communicating Techniques for Wearable Information Playing. 5

[WCVH08] Jason Wither, Chris Coffin, Jonathan Ventura, and Tobias Höllerer. Fast annotation and modeling with a single-point laser range finder. pages 65–68, 2008. 3

[WDH09] Jason Wither, Stephen DiVerdi, and Tobias Höllerer. Annotation in outdoor augmented reality. *Computers and Graphics*, 2009. doi:10.1016/j.cag.2009.06.001. 5

[WRM+08] Daniel Wagner, Gerhard Reitmayr, Alessandro Mulloni, Tom Drummond, and Dieter Schmalstieg. Pose tracking from natural features on mobile phones. In *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*, pages 125–134, 2008. VIII, 17