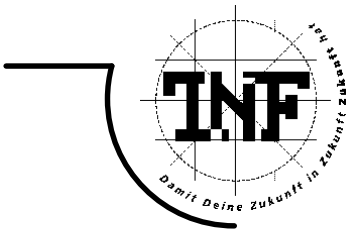
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



The Palmist Project. A new Approach of Interaction in Virtual Environments.

DIPLOMARBEIT

zur Erlangung des akademischen Grades

DIPLOMINGENIEUR

in der Studienrichtung

INFORMATIK

Angefertigt am *Institut für Wirtschafsinformatik*

Betreuung:

o. Univ.-Prof. Dipl.-Ing. Dr. Gustav Pomberger

Eingereicht von:

Wolfgang Ziegler

Linz, Juni 2004

Eine Diplomarbeit im Auftrag des

Ars Electronica Futurelab

Abstract

Interaction is a significant part of Immersive Virtual Environments (IVEs). Nevertheless, current interaction techniques are often inadequate and cumbersome, because they involve interaction with 2D Graphical User Interface (GUI) elements which use 3D interaction devices.

The *Palmist Project* is an approach to overcome these problems using a tracked pocket PC with wireless network access. The display of the pocket PC serves as an additional interaction space where users can work with familiar 2D GUI elements.

This functionality will be provided by a software framework called FATE [1]. This framework has three main objectives.

The first one is to enable an easy and fast creation of applications for pocket PCs.

Objective number two is to provide a framework-specific class library of GUI elements that have an intuitive programming API.

The third objective is to offer a set of software components that implement certain navigation and interaction functionality, which has been identified before.

Therefore, the main focus of the software framework FATE is to provide a component and class library for the creation of pocket PC applications in the domain of interaction and navigation in virtual environments with little programming effort.

FATE was developed in C++ using Microsoft Embedded Visual Studio. It was tested on Compaq iPAQ pocket PCs (series 36xx and newer) running Windows CE 3.0.

[1] **F**ramework for **A**dvanced **T**echniques of Interaction in Virtual **E**nvironments.

Contents

Chapter 1

Introduction

1.1 Virtual Environments

"The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be good enough to sit in. Handcuffs displayed in such a room would be confining, and a bullet displayed in such a room would be fatal. With appropriate programming such a display could literally be the Wonderland into which Alice walked." [1]

This is how Ian Sutherland described his idea about the perfect Virtual Environment in 1965.

However, for a long time, Virtual Environments (VEs) and Virtual Reality (VR) were broad terms which were described differently by a large number of scientific publications and conferences. It were Burdea Grigore and Coiffet Philippe in their book "Virtual Reality Technology" [19] who defined the term Virtual Reality, to put a stop to the confusion.

"Virtual Reality is a high-end user interface that involves real-time simulation and

interaction through multiple sensorial channels. These sensorial modalities are visual, auditory, tactile, smell, taste."

In terms of functionality, VR is a simulation that uses computer graphics to create a realistic-looking world. This world is supposed to respond to different forms of user input instantly and to provide real-time interaction.

Another, more precise, definition defines Virtual Reality and virtual environments as "a medium composed of highly interactive computer simulations that sense the user's position and replace or augment feedback of one or more senses - giving the feeling of being immersed, or being present in the simulation." [20] This is a very accurate definition for the reason that it includes the actual core points of virtual environments:

- *Immersion:* This is the degree of suspension of disbelief and is defined as the ability to give in to a simulation - to ignore its medium. [21][p.65]. This means the user is captured by the content delivered by the medium without being aware of the medium itself. Immersion is a highly subjective phenomenon and "science cannot supply a simple formula for calculating the degree of immersion someone experiences" [23]. Nevertheless, providing the feeling of immersion is essential for a good VR device. [24]

- *Presence:* This term describes the effect that a user in VR space experiences objects and processes in the virtual world temporarily more eminent than in the actual real world. [22][p.62]

- *Interactivity:* This is the degree of the system's reactiveness in response to the user's actions or how many possibilities the user has to modify the environment.

1.2 Interaction in Virtual Environments

"The types of tasks that users might perform in any interactive system are independent of the dimensionality of the system" [5], According to Foley *et al.* [4], these typical interaction tasks are:

- Selection: Making a selection from a number of given alternatives.
- Position: Indicating a position on the display or in the workspace.
- Orientation: Altering the orientation of an object in the workspace.
- Path: Generating a path, which is a series of positions and orientations over a time.
- Quantification: Specifying a value (i.e. a number) to quantify a measure.
- Text: The input of a text string.

Obviously, some of these interaction tasks (e.g. position or orientation) are intuitively 3D tasks, while others (e.g. quantification or text) are typical 2D.

1.2.1 2D vs. 3D Interaction Techniques

With all the success graphical user interfaces have had within the context of the desktop metaphor, their direct counterpart in the field of virtual environments has not been as widely accepted. There are several reasons for this. Pointing techniques in three-dimensional spaces are much more complex and error prone than in two-dimensional spaces. Typically, a six-degrees-of-freedom magnetic tracking device is used in combination with an instrumented glove device to enable pointing to three-dimensional user interface elements [32]. Magnetic tracking devices, even under the best of conditions, are noisy and require extensive filtering. This makes pointing difficult, especially if the user is at some distance from the target or if the target is small.

Another issue is the use of stereoscopy for depth perception. Target acquisition in three-dimensional space requires the preservation of visual depth cues. Often, immersive displays do not adequately produce images from which correct depth information can be found. Without these cues, the task of pointing (intersecting a ray projected from the finger with a target) is exacerbated.

Hand gestures are used to interact with the user interface in performing tasks such as activating buttons, selecting items in a menu, or triggering a selected menu item. These gestures are not intuitive and require some level of training and practice for efficient use. There is also a visibility and readability problem since text is inherently two-dimensional but is represented in three dimensions. It is likely that the user moves to far away from the textual information on buttons or menus, so it cannot be read any more. Or it is also possible that the user is positioned at some angle from which the text is skewed and therefore unreadable.

Most importantly, the use of direct pointing for selection in virtual space is awkward due to its inefficient use of the hands. In the real world, the hands are used for grasping and working as well as for pointing. In virtual environments, the hands should be used primarily for direct manipulation-related tasks. For example, the hands should be used to point at some object to select it or to grasp a virtual object in the space for movement or alteration. Since the hands are a fundamental mechanism by which humans interact with their world, they must be utilized efficiently. Without the use of effective tactile feedback, grasping of virtual objects can be difficult. After grasping a virtual object for manipulation or inspection, it is awkward to have to put it down in order to be able to interact with elements of a graphical user interface.

Also, as there is no passive haptic feedback in virtual environments and the challenge of the 3D interaction paradigm, users cannot interact as intuitively as they are used to in desktop interaction.

As soon as an interaction task, that is normally two dimensional, becomes three di-

mensional, the effectiveness of traditional interaction techniques decreases [10]. Even if users can interact with familiar user interface elements like sliders, buttons or menus, these elements are floating in space which makes it difficult to interact precisely with them.

Figure 1.1: Interaction in a CAVE-like environment. Picture courtesy of Rijksuniversiteit Groningen.

An example for this is given in figure 1.1 which shows a menu that partly covers the 3D view and may hide relevant information from the user (e.g. in industrial simulation applications) and is floating within the scene, which additionally hinders the possibilities of interacting with this menu.

Previous work has been done in addressing the problem with the use of 2D interaction techniques in 3D environments. Early works of Angus and Sowizral [11] or Wloka and Greenfield [12] have inspired many researchers to think of devices which make use of the 2D interaction paradigm in virtual environments. Many different interaction techniques

and devices have evolved, based on this research [13][14][15][16][17][18].
Another main issue when interacting in IVEs is the lack of a common interface [14]. One solution which addresses this, is the adaptation of the Windows, Icons, Mouse and Pointer (WIMP) interface to virtual environments. The WIMP paradigm is common on all computer platforms, which enables the user to easyily identify common interface elements and to start productive work with little or no learning curve [14].
This thesis presents the *Palmist Project*. An approach towards the goal of an ideal interface for virtual environments, concepts and visions related to this project as well as the underlying software framework FATE.

1.2.2 Pocket PC Interaction in Virtual Environments

The idea of this thesis is to overcome these burdens of ordinary interaction devices and to have an "intelligent" interface for VR systems. Additionally, this thesis provides a software framework for the development of pocket PC applications which are exactly intended for the domain of interaction and navigation in virtual environments. This intention reflects in a strong emphasis on software components and classes that cover aspects of communication (e.g. wireless networking) and serve as a basis for more sophisticated interaction patterns.
Such interaction scenarios are for example map-based navigation where the user interacts with the virtual environment via a map displayed on the touchscreen of the Palmist, or remote-controlling the virtual world using the Palmist in a joypad metaphor. These interaction patterns and the software components developed for them will be explained in more detail in the following section.

Generally, a "palm-fitting" VR interaction device like the Palmist should supply the user with the necessary data about the virtual environment and it should give him almost unlimited control over it. The idea is to have an easy to use interface which is

expandable in its capabilities and which does not decrease the mobility of the user to ensure unencumbered immersion.

Figure 1.2: Design on the fly tool.

The main approach is to use a tracked pocket PC with wireless network access as an interaction device. This opens up totally new ways in information presentation and interaction within the environment. Independent from this thesis, research in this field is done by others as well [5][18][33]. The big advantage of this method is that the user is familiar with the WIMP paradigm, and therefore finds the use of this device very intuitive and easy. There is a passive haptic feedback and the user is able to work precisely with both the dominate and the non-dominate hand. The interaction metaphor resembles the "pen-and-tablet" interfaces [30][16][31] and therefore offers all the benefits related to it and it is even better in some points. The input is absolute precise in opposite to tracked pen and tablet interfaces, where the problem of imprecise tracking makes them difficult to use. Moreover the use of a pocket PC instead implies that it is very easy to implement rich user interfaces for virtual environments, because graphical user interfaces can easily be implemented on pocket PCs and are an integral part of the FATE framework. Another big advantage over the pen and tablet interface

is that no screen space is lost for displaying the user interface [17].
The Palmist offers all the conventional functions like navigation and button interaction, but in addition its capabilities go further and lead to new perspectives in controlling virtual environments. The Palmist can be used to create and design the environment on the fly and the user is able to interact with its models and to make changes to them. On the other hand the user is able to get important data about the world and its components, e.g. measurements and materials of CAD models just by touching these objects with the Palmist.
In the following section some concepts for interaction with virtual environments are introduced and new possibilities for VR applications are shown on the basis of the Palmist project and its underlying software framework FATE.

1.3 Interaction Scenarios

1.3.1 General Interaction Techniques with the Palmist

This work mainly focusses on issues as navigation, selection and manipulation techniques in an immersive virtual environment which can be supplied by a pocket PC interface. For navigation the known joystick metaphor is used. This is done by using the keypad of the pocket PC. If additional spatial information about the interaction device is required, the pocket PC can be attached to a tracking sensor to get its position and orientation in space.
For selection with pocket PCs in virtual environments, different methods have been studied and implemented. The most basic approach which has been introduced by Watsen et al. [17] is to have an object list on the pocket PC, where the user can select from. For a vast VR application with a big number of objects this approach is very cumbersome.
Therefore, this method is combined with the well known selection technique of cone

casting. This means the user points in a certain direction with his tracked pocket PC and the objects which are in the field of the outgoing cone are displayed in an object list from which can be chosen then. The user is able to define the length and the radius of the cone easily on the Palmist interface. This method has been proven as very helpful, fast and precise and is used right now as selection mechanism in Palmist applications. All manipulation tasks are done over different graphical user interfaces displayed on the Palmist.
The software component which implements this interaction technique is described in more detail in the implementation chapter (see 4.5.6).

1.3.2 The Palmist as Virtual Environment Control

In virtual reality devices like the CAVE, there always exist a lot of different applications, which have to be started or closed whether over complicated and cumbersome 3D menus or even worse by running scripts on the graphic workstation.
The Palmist solves these troubles in an elegant way and the user has full control on which of these applications should be started or closed. So the user can switch through the applications in a straightforward manner like with a remote control for a television (see figure 1.3).
In addition the Palmist offers ways to change important values for the startup of an application (e.g. how many walls of a CAVE environment should be used for displaying the application). Furthermore the Palmist can be used to control the surrounding environment parameters at runtime, e.g. the volume settings of the audio system or the brightness and contrast of the projectors and similar settings.
This remote control paradigm has already been proven as very useful and is already a fix and tested part of the FATE framework. It is described in more detail in the implementation chapter (see 4.5.4).

Figure 1.3: Remotely start and stop VR applications.

1.3.3 Advanced Navigation in vast Virtual Environments

Navigation in vast virtual environments is somehow different from navigation in smaller dimensioned VEs. The user is very likely to lose orientation and can be fatigued by the long travels he has to take in order to come to the points of special interest within the VR applications.

To provide a fast and precise navigation the Palmist shows a two dimensional map of the area and visualizes the user's currently position in the virtual environment with a small symbol on the map (see figure 1.4). The user can scroll the map and is able to click on a certain point on the map where he is warped in the VR application.

This advanced navigation mechanism is very useful for vast environments where a user can easily lose orientation. In addition there is a bookmarking system included. This concept can be compared to a bookmarking system of an internet browser. The user can travel around in the environment and add bookmarks to a list in his Palmist application just by clicking on a menu symbol. So the user easily finds places of special interest again in the vast environment, just by choosing one of his saved bookmarks. This feature will become more and more important because the majority of virtual environments will

become vaster and more complex in the near future.

This advanced navigation is a readily developed component of the FATE framework and is explained in more detail in the implementation chapter (see 4.5.3).

Figure 1.4: Map-based navigation and interaction.

Chapter 2

Background

The *Palmist Project* is mainly intended for interaction in CAVE-like environments and not compatible with technologies like Head-Mounted-Displays (HMDs). The reason for this is that the user needs an unobstructed view to the screen of his pocket PC to be able to interact with the virtual world. This is only the case when used in CAVE-like environments.

2.1 CAVE

The idea of the CAVE system is to increase the immersion and the presence of the user by surrounding him with projection planes. The best way to do this would be a projection sphere in which the user is located. The problem with a sphere is that this would require a lot of computational power of the graphic workstations as well as very sophisticated adjustment of the projectors for proper edge-blending. Both factors increase the cost of this system dramatically and therefore, the CAVE uses a cube as an approximation.

The CAVE is a projection-based VR system, which makes it different from other VR systems, because the actual display device does not move with the user, as with e.g.

HMDs.
Further, the CAVE uses the paradigm of a room, which a user can enter and explore its content inside. This idea of a physical space that is filled or extended by virtual content leads to very high immersion and presence of the user. The user is able to move and look around and to see his own physical body or other users while seeing the virtual content.

Basically, the CAVE system consists of three types of hardware [24]:

- *Sensors* detect the operator's body movements or any other control actions such as pressing a button or moving a joystick.
- *Effectors* stimulate the operator's senses (mainly visual and auditory).
- A *graphics workstation* interprets input from the sensors and calculates the corresponding output, which is transmitted to the operator by the effectors.

2.1.1 Sensors

The most important sensor for the CAVE is the head tracking system. It provides the graphic workstation with accurate position and orientation of the user's head and current view of the scene. These values are needed to calculate a viewer-centered perspective, which is one of the most important factors in creating an immersive experience [25][26]. A head tracking system for an immersive VR system is in general a six-degrees-of-freedom tracker. High update rates and accurate tracking is desired because otherwise immersion is decreased [27][p.287] and simulation sickness [28] can occur.
Another important sensor to interact with the virtual environment is the hand tracker. The tracking techniques are the same as for head tracking. The data which is transmitted for the hand tracker is position and orientation. Additionally, a hand tracker

has several buttons and their state is transmitted as well.
The four basic techniques for tracking of position and orientation are based on electromechanical, electromagnetic, acoustic or optical technologies [27]. With the help of tracking middle-ware (see 2.2) the CAVE software supports various numbers of tracking systems.

2.1.2 Effectors

The aim of virtual reality systems is to stimulate the user's senses. Visual and auditory effectors are the most important effectors to increase the feeling of immersion.

The *visual effectors* of a CAVE system are its projection screens which form a cube. The number of projection screens can vary. The basic configuration of a CAVE system consists of three walls and a floor, which are rear projected to save space. The visual effectors use the mechanism of stereopsis.
It is based on the fact that one of the main cues, that human brains use to gauge the distance to objects in their field of vision, is the difference in apparent position of the object in the views obtained from the two eyes (parallax). VR technologies take advantage of this by showing each eye a slightly different view [29]. By integrating these different views, the brain can calculate the spatial information of the scene [19]. The two keys for making this work are producing the correct views for the left and the right eye and delivering each view to the appropriate eye. One way of getting a stereoscopic display is to use a special stereo graphics card. Externally, these cards have a connection to LCD shutter glasses. The shutter glasses alternately black out the left and right eye in quick succession. The alternation of eyes is synchronized with the display screen so that the left eye sees one the "left image" and the right eye sees the other one. This method of providing stereo display is also called active stereo [29].

Auditory Effectors serve as a subconscious reinforcement of visual feedback [19]. There-

fore, adding sound to the simulation of a virtual environment can significantly improve the feeling of immersion. Auditory feedback can even enhance the illusion of depth.
There are various different systems and mechanisms to achieve realistic and three-dimensional sound effects.

2.1.3 Graphics Workstation

In order to achieve a realistic simulation of the virtual environment, the sensors and effectors have to be coordinated in real-time by reading the task-dependent user input and then calculating the corresponding scene for output via the effectors.
In traditional CAVE systems a high performance graphics workstation is used for this purpose, because the calculations for geometry and rendering require a lot of computational power and real-time requirements have to be fulfilled.
But with the rapid development of consumer graphics cards, such high performance graphics workstations can be replaced by commodity PCs. Therefore whole CAVE environments like the ARSBOX [3] can be built with a drastic cost reduction.

2.2 TRACKD

TRACKD by VRCO Inc. [36] is a widely used middle-ware for VR applications, developers, display manufacturers, motion tracking companies, and input device manufacturers in the immersive display industry.
The TRACKD software is a small "daemon" application that takes information from a variety of tracking and input devices and makes this information available for other applications to use.
In combination with VR systems, TRACKD collects data from navigation and interaction devices and forwards these data to the VR application. In this way the user can explore the virtual environment and interact with it.

Applications that use the TRACKD middle-ware do not need to know what type of hardware device is being used. These applications simply receive data from trackers and input devices in a highly abstracted and very generic way. This generic data access allows applications to support a wide range of hardware without having to modify source code, or writing specific hardware drivers. It also allows VR systems to change, replace or upgrade components without having to replace software, or losing functionality.

In fact, from the programmer's point of view, TRACKD only makes a difference between two different kinds of systems delivering data. These two systems are:

- *Trackers*

  This term describes the classical six-degrees-of-freedom tracking systems, which means systems that provide information about the tracked device's position (x, y, z) and its orientation (yaw, pitch, roll) in space. Such systems are e.g. "Flock Of Birds" or "MotionStar" by Ascension Technologies.

- *Controllers*

  Controllers describe interaction devices similar to joysticks, i.e. they normally provide information about two axes and several buttons. The most common Controller input system is the WAND (see figure 2.1).

For the reason that the majority of the existing VR systems uses TRACKD as tracking middle-ware, one of the first components developed for the FATE framework was a component which enables the pocket PC to communicate via the TRACKD network protocol (see 4.5.1). This software component enables the pocket PC to serve as a controller device. By using the buttons of the Palmist in a joypad metaphor the user can navigate through the virtual environment and interact with it. With the help of this component the whole possible interaction in existing VR applications (given that they use TRACKD as tracking middle-ware) can be done. As mentioned before,

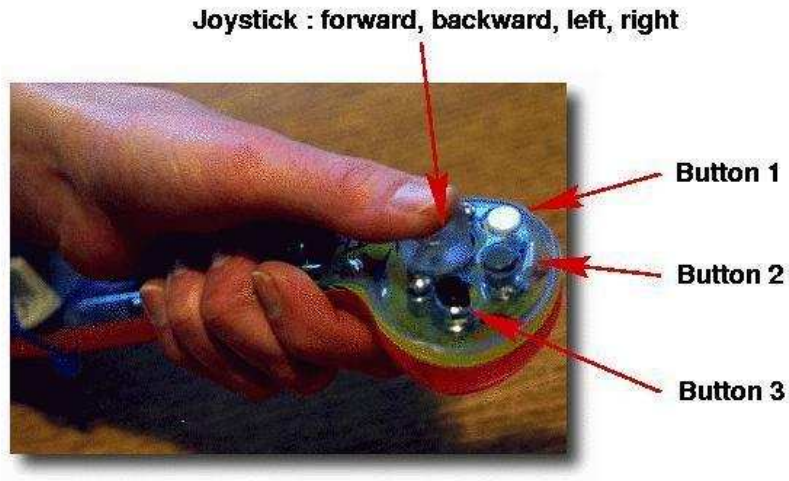


Figure 2.1: The WAND - An interaction device in virtual environments.

no additional software or drivers need to be installed on the VR system, because the network protocol, which is used by this component, is an existing part of the TRACKD software.

2.3 Ygdrasil

Ygdrasil is a high-level VR framework written in C++. Its primary features are: a distributed scene graph, dynamically loaded extensions and its scripting ability.
These features are intended to yield a dynamic VR authoring system, where creators of virtual environments can assemble a virtual world out of arbitrary existing components and bring new objects into a running world.
Ygdrasil has clearly defined structures for sharing data, so independent software components can "link up" and communicate with each other. The basis for this data and event exchange mechanism is the scene graph approach of Ygdrasil. The scene graph provides a basic structure, in the form of data associated with graph nodes, and so the communication mechanisms can happen automatically.

The possibility of loading objects dynamically and a scripting interface for defining the virtual worlds, make it possible for components to be easily shared and re-used by world authors [8].

Ygdrasil is built on top of SGI's OpenGL Performer graphics library [37], the CAVElib VR library [38] and the CAVERNsoft G2 networking and collaboration library [39].

- *Performer* provides the basis for a hierarchical scene graph representation of the virtual world database.
- *CAVElib* is a software API for creating applications for virtual environments. Some of the items that are covered by this library are viewer-centered perspective calculations, displaying to multiple graphics channels, multi-processing and multi-threading, cluster synchronization and data sharing, and stereoscopic rendering.
- *CAVERNsoft* is a networking toolkit for VR that emphasizes integrating VR with high-performance and data-intensive computing over high-speed networks. It distributes necessary data, such as lists of scene graph nodes, transformation matrices, and model information automatically among the participants in a networked application.

Ygdrasil uses a scene graph structure for the world database. It is a distributed scene graph, which does not require a central server for storage. Often, no single machine owns a complete copy of the original "master" scene. Conceptually, different subgraphs of the full scene can exist on different machines, and be linked over the network. Any particular machine will only have the parts of the scene graph that it controls, and proxies for any other parts that it needs for its calculations, rendering or whatever that machine is doing. Figure 2.2 shows an example scene graph for a world that contains two distinct scenes and demonstrates how this scene graph might actually be shared among multiple hosts.

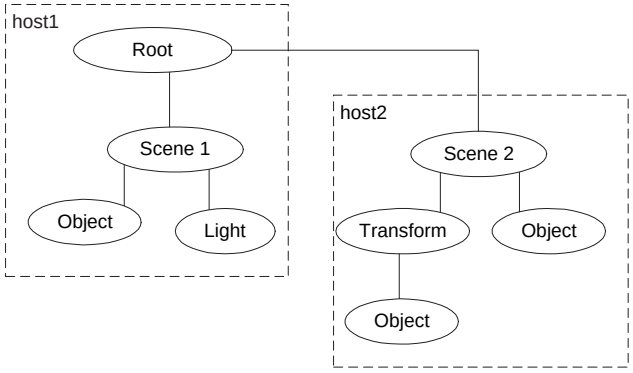


Figure 2.2: A distributed scene graph.

In addition to the basic graphical data used in Performer, any scene graph node can have behaviors attached to it. Each particular node is considered to be owned by the host that created it. This host executes any behavior associated with the node. All other hosts have proxy versions of this node, and only receive data for it. They do not directly modify the node, except by sending messages to the master copy to request changes. The basic scene graph data (i.e. data that is sufficient to render the scene) is shared automatically and for this reason new behavioral components generally do not have to include any networking code themselves [8].

The actual composition of a virtual world in Ygdrasil is done by using a higher level scripting layer. This scripting layer is a simple textual representation of the scene graph layout. It tells the system what kinds of nodes to create and includes commands with each node to control its behavior.

Ygdrasil's distributed scene graph mechanism as well as its scripting interface are both utilized by Palmist applications. If a Palmist application knows the scene graph of an Ygdrasil virtual environment, it can send messages to the host which owns the actual data. This host receives and processes these scripting messages and dynamically changes the nodes of the scene graph at runtime. This approach offers full access to an

application's scene graph and therefore unlimited possibilities of interaction.
The framework component which makes use of this concept is the `CFMsgServerProxy` component (see 4.5.2).

Chapter 3

The FATE Framework

3.1 Motivation and Summary of Results

The software framework FATE was designed and developed to overcome all the limitations discussed in chapter one that are caused by inadequate and cumbersome interaction possibilities in current VR systems. After having identified these limitations and having found solutions to them, the ideas behind these solutions were integrated into software components that are now part of this framework.

Basically, the FATE framework has three main objectives:
The first objective of this framework is to enable easy and fast development of applications for pocket PCs. For this reason FATE contains a class (see 4.2.9), that serves as an application template and additionally provides the whole basic functionality of an entire Palmist application. The advantage for programmers is that they no longer have to worry about platform specific initialization tasks (e.g. window creation), which are actually not related with the original purpose or semantics of the application but are nevertheless necessary. For the reason that this whole functionality like application initialization, window creation, setting up the window in fullscreen mode and handling user events is managed by framework mechanisms, programmers can put their focus on

the actual content of the Palmist application.
Even though, it is not the original intention of the FATE framework to serve as an application framework, it is nevertheless an important and useful part of it. Besides that, FATE does not serve as a general purpose application framework, for the reason that it has some built in constraints like the automatic fullscreen mode of each application. FATE is rather a domain specific application framework, that provides essential basic functionality, so that users of the framework do not have to take care of necessary but application-irrelevant tasks.
This part of FATE may be compared with the initialization functions of the GLUT [35] library. The actual intention and application domain of GLUT is the creation of 3D graphics applications. But for these applications it is also necessary to create an application window or provide event handling routines for mouse and keyboard. This functionality is also covered by GLUT mechanisms. For the FATE framework, the idea is the same. The focus and the original application domain is the creation of applications for interaction and navigation in virtual environments. But the necessary basic functionality of these applications and their initialization is provided and encapsulated by framework mechanisms.

Objective number two is to provide a framework specific class library of graphical user interface elements that have an intuitive programming API. Of course, there are already existing GUI libraries but there are several reasons why FATE provides is own set of user interface elements:

- From its basic design, FATE is not tied to any special operating system, but it provides its own programming API. This is a fundamental prerequisite that guarantees portability and is an important design feature of the framework. For this reason operating system specific GUI elements have not been used.

  At the moment there are two different implementations of FATE. The first one is

implemented on a Windows CE operating system and the second one on Windows 2000 / XP. Nevertheless, their programming API is absolutely identical.

- FATE has a proprietary event management system which also manages events triggered by GUI elements (e.g. a button was pressed or released). To be able to implement this event management system in a logical and consistent way, the GUI library had to be tightly integrated into the framework.

- The FATE framework only provides a limited set of GUI elements. These elements have been identified before as those that are most commonly used in applications and "special-purpose" elements have been omitted. But these selected GUI elements are designed to have a programming API which is as simple as possible so that they can easily be used in Palmist applications.

  Again, the intention was to avoid that programmers have to deal with problems like cumbersome programming interfaces of GUI elements and to decouple such tasks from the actual application logic.

- Many of the GUI elements of FATE serve as a basis for further specialized software components in the framework and are used in the object-orientated paradigm of reuse. To create a consistent class-hierarchy it was necessary to tightly integrate GUI elements into the framework.

- FATE also has its own drawing loop and automatically uses double-buffering for all GUI elements. It was necessary to create this framework specific GUI library, to avoid that drawing mechanisms of Windows CE GUI elements (or those of future platforms) interfere with the drawing mechanisms of FATE.

The third objective of the FATE framework is to offer a set of software components that implement certain navigation and interaction functionality, that has been identified before.

This is the original and main idea of this software framework and the actual application domain of FATE. The previously mentioned objectives provide services for programmers, so they can use the framework in a way that is as efficient as possible. Also, they provide programming interfaces, that fit for a wider spectrum of applications.
But, the core strength of FATE are its specialized software components that have already been explained in the introduction chapter (see 1.3). For that, certain scenarios have been identified, that are typical for interaction and navigation in virtual environments.
These scenarios are for example map-based navigation or virtual remote-controlling (see 1.3). FATE provides readily implemented software components for these typical interaction scenarios. They cover the whole functionality that is needed and provide it without any further programming effort. Programmers can simply configure these components in the appropriate way and integrate them into their Palmist application. It has shown, that already a few of these components suffice, to cover the basic mechanisms of the necessary interaction in virtual environments.
The basic request to a Palmist application is in almost every case to have the possibility of navigating through the VR scene. This can easily be realized by the joystick-functionality of the Palmist. Also, simple interaction can be done by using its buttons. This is already sufficient for a big part of the existing VR applications that have originally been developed for interaction devices like the WAND (see figure 2.1) that is also restricted to joystick and button interaction.
For faster and more precise navigation in virtual environments, map-based navigation is a helpful and often used navigation paradigm. Additionally, it increases the user's overview of the VR scene in a drastic way.
If also a more precise object-manipulation technique is needed, this can be covered by interaction mechanisms like those described in the introduction chapter (see 1.3.1) and realized by a single software component (see 4.5.6).

These domain specific software components of FATE have proven worthwhile in existing applications and cover a big part of the possible interaction in VR applications.

3.2 General Requirements

This section states the most important technical requirements for the FATE framework. As a part of the analysis, the hardware and software requirements had to be decided. Of course there is a given separation between the system on which the actual rendering process takes places (Graphic Workstation) and the system which is used for interaction (Palmist). To enable communication between these systems a network connection is necessary, which in most cases will be wireless.

1. Graphic Workstation

   This can be any kind of computer (PC, SGI Onyx, ..) which is able to render the virtual scene. Typically, CAVE applications create the visual output with the use of graphics APIs such as OpenGL or Performer. The main implementation and testing was done on the ARSBOX PC-CAVE system [3] running both OpenGL as well as Performer applications.

2. Palmist

   The system running the Palmist applications can actually be any kind of Windows CE based device. The systems which were used for implementation and testing were iPAQ pocket PCs with an ARM processor.

3. Network Connection

   For the reason that there are no big amounts of data transferred from the Palmist to the Graphic Workstation or vice-versa, no special networking technology was needed. The ARSBOX [3] PCs were connected via 100 MBit ethernet and the Palmist got connected using an 11 MBit WLAN network card and an access point.

3.3 Design Aspects

"The goal of the design phase consists of determining which system components will cover which requirements in the system specification and how these system components will work together" [2].
Generally, the FATE framework is designed using object-oriented concepts and this chapter shows how the classes and interfaces communicate and work together and how their functionality can be used.

3.3.1 Nomenclature

To indicate clearly, that a class or an interface is part of the FATE framework naming conventions are used as follows.
Classes which are part of the FATE framework have the prefix *CF* (e.g. `CFBitmap`).
Interfaces which are part of the FATE framework have either the prefix *IF* (e.g. `IFSocket`) or *IFate* (e.g. `IFateComponent`).
Many of the interfaces are not pure interface classes but provide already implemented methods. These classes are, what e.g. JAVA terminology would call abstract classes. Therefore the main difference between FATE interfaces and FATE classes is that classes can be instantiated whereas FATE interfaces own pure virtual methods. This means FATE interfaces have to be subclassed first and these pure virtual methods have to be overridden to be able to use their functionality.

3.3.2 General Design

The classes and interfaces which form the FATE framework can be logically divided into four different categories. These categories originate from the functionality and semantic behavior of these classes and interfaces. Even though there are mutual relations and dependencies between these four parts, this categorization is useful and gives a better

understanding of how the framework's classes work together.

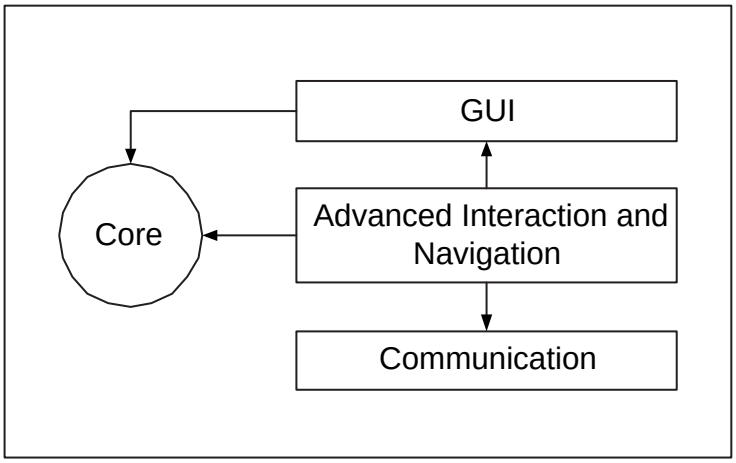


Figure 3.1: The main framework components

Figure 3.1 illustrates graphically how these four parts depend on each other and work together. The arrows in the figure illustrate a "uses" relation. An arrow pointing from one part to another one indicates that the first part uses services provided by the second one (e.g. the GUI part uses services provided by the core part).

The functionality of the individual parts and their inter-dependencies will be explained in more detail now:

1. **Core-Framework**

   The classes and interfaces which form this part, provide the framework's core functionality and cover most operating system dependent aspects. All of an application's startup, initialization, event-handling and finalization functionality is managed by this part of the framework. The basic objective of these core part is that an application developer does not have to care about operating system dependent issues and tasks like window creation or mapping of events like button or stylus interaction.

As figure 3.1 shows, this core part of FATE provides services for the framework's GUI library and the advanced software components. Basically, these services are event notification and drawing mechanisms.

2. **Communication**

   For the reason that communication between a VR system and a pocket PC is the main idea behind the *Palmist Project*, it is evident that this communication aspect has to be covered by the FATE framework somehow. Therefore it contains a set of classes which are intended for communication purposes. These contain socket classes and network protocols like TCP and UDP as well as data exchange over the PDA's serial port.

   Of course, this functionality is already covered by the operating system's programming interface, but FATE classes provides these communication services in an object-orientated way. This makes it easier for the programmer to use these services, because the classes that encapsulate them provide a less complex and more intuitive programming interface.

   For the reason, that these classes serve rather low-level communication purposes and provide services for more sophisticated software components (see figure 3.1), they have been implemented to be extremely robust and fail-safe.

3. **GUI**

   This set of classes implements a small GUI library. The most common GUI controls like buttons, item lists or menus have been identified and implemented for this part of the framework. The intention is to provide a GUI set with a programming interface that is as simple as possible.

   As shown in figure 3.1 the GUI classes use services provided by the core classes of FATE. These services contain all event handling notifications (e.g. when a button was pressed or a menu item was selected).

Also, these GUI classes serve as a basis for more sophisticated and specialized classes and software components, that are explained next.

4. **Advanced Interaction and Navigation**

   These are the most specialized classes within the FATE framework. Unlike the communication classes or GUI elements, which can be used in a very generic way, these classes serve more sophisticated purposes in the domain of interaction and navigation in virtual environments.

   Each of these classes is a readily implemented software component which can be used for a special purpose. Therefore, these classes form the part of FATE that actually covers the application domain, this framework was developed for.

   All of these classes cover one interaction or navigation paradigm that has been identified as being useful and necessary.

   Figure 3.1 shows that these classes form the center of the framework and use services that are provided by all other parts of FATE.

3.3.3 Control flow

FATE is a software framework that supports the development of applications on multiple levels. As it provides a readily implemented basis for applications, a deeper understanding of how the control flow within a FATE application works, has to be given.
The whole framework is compiled into a static library (`FATE.lib`) which, of course, consists of all classes and interfaces that are part of the FATE framework. But the significant characteristic of this library is, that it also provides the application's entry point (i.e. "main"-function) to the operating system.
This means that all applications that use the FATE framework must not have their own entry point ("main"-function), as this would usually be done. In FATE applications

all control flow is immediately taken over by framework mechanisms and application initialization is automatically handled by it.
All of the following initialization tasks happen inside the framework and are completely hidden from the programmer:

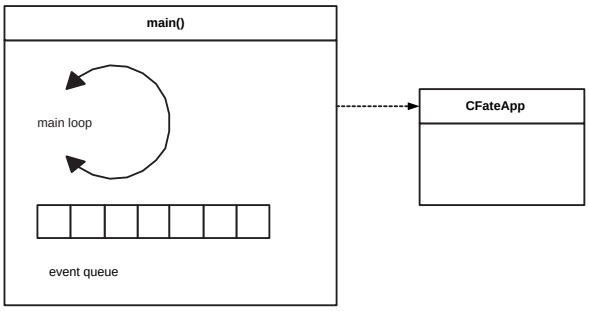


Figure 3.2: Application startup and initialization.

- An application window is created.
- The application window gets configured to be automatically displayed in full-screen mode.
- A double buffer for the frameworks GUI elements is created.
- The keys of the pocket PC are mapped appropriately, so they can be accessed by the framework.
- A message queue is set up for the application, to be able to process window messages and events sent to the application. This message queue is the basis for the event managing system of FATE.

In the next step, program control is handed over to the programmer for the first time. Because after all these initialization tasks, which are completely hidden from the programmer and happen inside the framework library, the framework calls a function that has this interface: "`CFateApp* CreateFateApp()`"

This function is only declared within the framework but not implemented. Therefore, the programmer is supposed to provide an implementation for it. The implementation of this function is also associated with a software contract.
As the return value indicates, this function is supposed to return a pointer to an object of the type `CFateApp` (see 4.2.9) or to a subclass of it.
This exactly is the way, how a programmer can add new functionality to this application template, when the returned pointer points to a subclass of `CFateApp`.

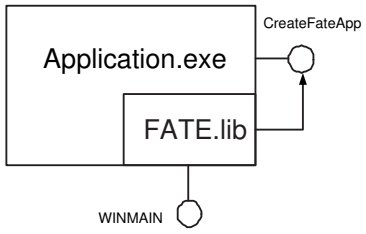


Figure 3.3: The layered architecture of a FATE application.

From the programmer's point of view, `CreateFateApp` is the new entry point to the application. Figure 3.3 also illustrates this concept of "double entry points" in FATE.

To sum up, an application based on the FATE framework, must not have its own application entry point, but must provide the `CreateFateApp` function and also ensure that its semantic behavior fulfills the specified contract. This means that programmers must derive an own class from `CFateApp`, create an instance of this class and return this instance in the implementation of `CreateFateApp`.
After that, the connection between the FATE library and the application-specific code has successfully been established.

3.3.4 The Event System

As Windows CE is an event-driven (also: message-driven) architecture, this is also true for FATE applications and therefore the application class implements base services for

message handling. All notification and communication mechanisms inside the framework happen via this event mechanism. If for example, a button was pressed or released, an event is put into the application's event queue.
Even though there are many framework classes that make use of this event notification system and a lot of different events exist therefore, the basic mechanism for event handling inside the FATE framework is always the same and works as follows:

1. An event is triggered and the appropriate event message is placed into the application's event queue.

2. The event is passed on to an internal event handling method of the FATE framework. This event handling method is hidden from the programmer and it depends on the kind of event whether the programmer is actually informed about this event or not.

3. Inside this event handling method there are three different possibilities how the message may be processed:

   (a) The event is processed inside the framework.

   In this case, programmers are not even informed about the fact that an event has occurred. Such events are e.g. the application startup event or the application shutdown event. They invoke certain actions inside the FATE framework but the programmer is not affected by them.

   (b) The event is unknown.

   FATE provides the possibility of processing user-defined event messages which gives programmers the possibility of using the asynchronous communication concept of an event queue.

   But, of course, user-defined messages cannot be processed by the framework automatically, because the semantic behavior of these messages is unknown.

For this reason an additional event handling method of the application object is called. The default implementation of this additional event handling method is to ignore the event and return immediately. But in derived application classes, this method is intended to be overridden and by using this technique, an application is able respond to events that are not automatically handled.

(c) The event is dispatched to registered framework classes.

This is the case for well-known events inside the FATE framework. Such events are e.g. the item selection within a menu or the pressing of a button. For all of these events, classes can register themselves as listeners and if one of these events occurs the registered listeners are informed.

The notification of these listeners happens in following order:

The application object is always the first listener to be informed about events. Depending on whether or not the event is consumed (indicated by the return value of the event handling method) the event gets further dispatched or not.

If the event is further dispatched, all registered listeners which are in enabled state (see 4.2.1) are informed about this even. Again, they may consume the event or not. If not and the listener object is a subclass of `IFateContainer`, it again dispatches the event to its enabled listeners.

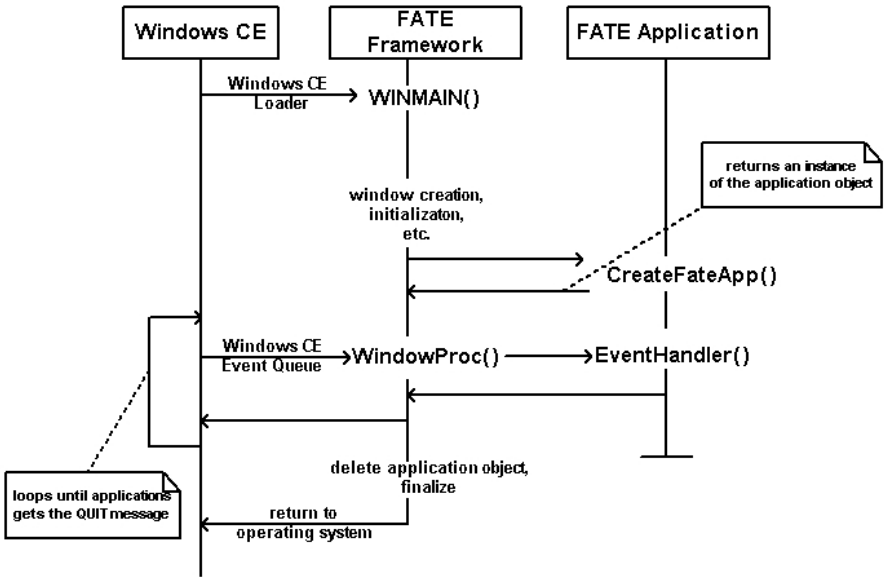


Figure 3.4: The life cycle of a FATE application.

Chapter 4

FATE - Implementation

4.1 Overview

"The goal of the implementation phase is to transform the products of the design phase into a form that is executable on a computer" [2].

The implementation chapter describes the functionality of the main classes and interfaces of the FATE framework in detail. Again, the four categories (Core-Framework, Communication, GUI, Advanced Interaction and Navigation) that were introduced in the previous chapter (see figure 3.1), are used to aggregate all of the framework's classes and interfaces.
Each of these categories is explained in its own subsection, which starts with a graphical representation of all the classes and interfaces that are part of this category. For this representation the Unified Modelling Language (UML) [34] was chosen in order to give an idea about the inheritance relationships and inter-dependencies of the frameworks individual classes and interfaces. Additionally, the UML diagrams provide a first overview of basic methods and attributes
In each of the subchapters, the individual classes and interfaces that are part of the appropriate UML diagram, are described in more detail. For this purpose, their main

methods and attributes are selected and the semantics are described.
For a few chosen methods, source code listings are used to explain their implementation in more detail. These source codes are listed in the implementation language C++.

4.2 Core-Framework

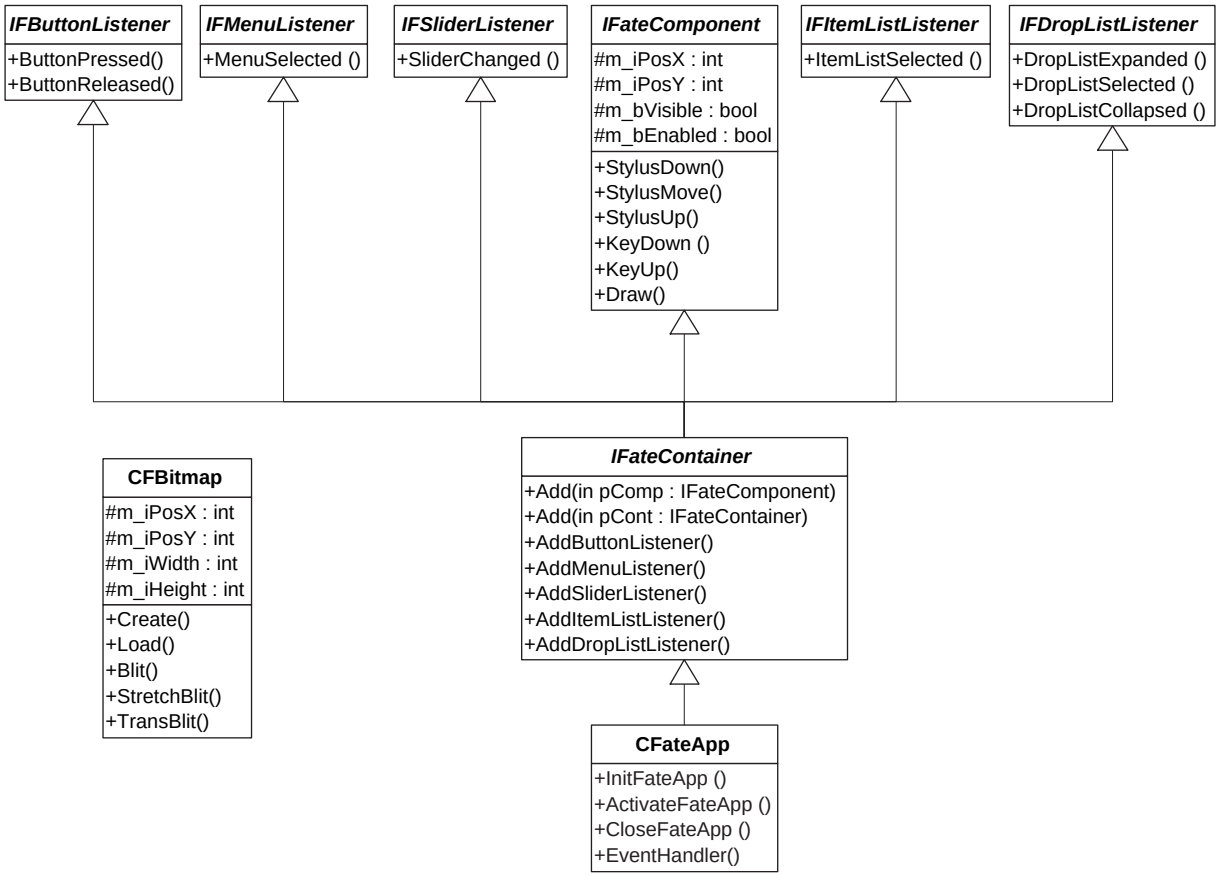


Figure 4.1: Core classes and interfaces of FATE.

4.2.1 IFateComponent

Together with `IFateContainer`, `IFateComponent` is one of the most important classes of FATE. It is implemented very generic and located at the top of the framework's class hierarchy.
It serves as a base class for all other FATE classes which are involved in the framework's automated process of redrawing and reacting to user interaction. For these purposes it provides following pure virtual methods as part of its interface:

- `void Draw()`

  Any subclass can override this method to redraw itself, when the framework sends a notification of drawing to its registered classes.

- `BOOL StylusDown(int xPos, int yPos)`

  This event is triggered by the framework when the user hits the PDA's touchscreen with the stylus. If any subclass of `IFateComponent` is supposed to react to this event, this method has to be overridden.

- `BOOL StylusMove(int xPos, int yPos)`

  After `StylusDown`, this method will be called as long as the stylus keeps contact to the touchscreen and gets moved around.

- `BOOL StylusUp(int xPos, int yPos)`

  When the stylus loses contact with the touchscreen again, this method will be called by the framework.

- `BOOL KeyDown(PdaKey key)`

  This method gets called by the framework, whenever one of the PDA's hardware buttons was pressed.

  Typically, PDA's have a joypad-like navigation interface that corresponds to a button for each of the four points of a compass, and at least four further buttons

which can be used for additional user interaction (see figure 4.2).

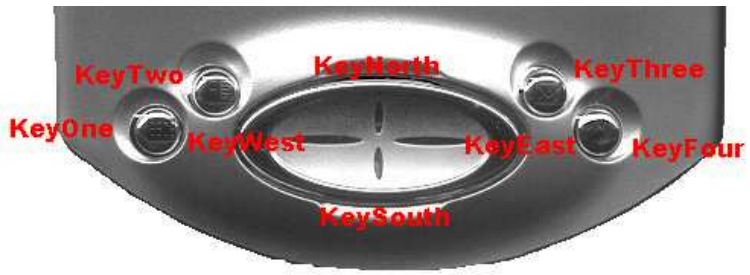


Figure 4.2: Interaction keys of a Compaq iPaq 3750.

The method has a single argument `key` and its type `PdaKey` is an enumeration which consists of the members `KeyWest`, `KeyEast`, `KeyNorth`, `KeySouth`, `KeyOne`, `KeyTwo`, `KeyThree` and `KeyFour` which correspond to the mentioned buttons.

- `BOOL KeyUp(PdaKey key)`

  By calling this method, the framework notifies that a pressed button was released again.

The following two methods also play a vital role within the event-handling mechanism of FATE, because it depends on these methods whether the event-handling methods introduced above are called or not.

- `void SetEnabled(BOOL bEnabled)`

  Determines whether an `IFateComponent` object is enabled or disabled, and this state is responsible for the participation of the object in the event-handling mechanism. A disabled object will not be able to react to events of any kind, because its event-handling methods will not be called by the framework.

  Automatically, if a subclass of `IFateComponent` is instantiated, its state is set to disabled, so a call to `SetEnabled` has to be made, before an event of any kind can be processed.

- `void SetVisible(BOOL bVisible)`

  This method influences the redrawing-behavior of an `IFateComponent` in the same way as `SetEnabled` does for the event-handling-behavior. This means, that only objects with their state set to visible will automatically be redrawn by the framework.

  Again, at instantiation time, objects derived from `IFateComponent` are set to be invisible. A positive visible state has to be set manually.

Another especially important method is `SetDC(HDC hdc)`. The fact that this method has been called, indicates to an `IFateComponent` that it can safely perform drawing operations from this time on.
Whenever an `IFateComponent` is added to a parent container (see 4.2.8), this method is automatically called by the framework. It passes a handle to a windows device context (HDC) [9][p.4] as argument, which corresponds to a valid drawing surface.
The default behavior of this method is to store this handle in a member variable. But often this method is overridden by subclasses to perform graphical initialization, because this is the earliest possible moment for an application to do so.
In earlier stages of the application's control flow, the application does not have a valid device context and it is not safe to perform drawing or other graphical operations.
If this design pattern is taken on in self-written subclasses, it should be implemented as follows.

```
void MyComponent::SetDC(HDC hdc)
{
  // Store the handle to the device context in a member variable.
  m_hdc= hdc;

  // Do whatever we want with the device context.
  // For example, create a bitmap based on it.
  CFBitmap *bmp= new CFBitmap(hdc);
  ...
}
```

`IFateComponent` also has some member variables which have a protected visibility state, so that they can also be accessed by subclasses. Anyway, for most of these attributes, there exist virtual get- and set-methods which can also be overridden to change or extend their behavior.

- `int m_iPosX` / `int m_iPosY`

  These attributes determine the position of each `IFateComponent` object and are initialized to zero in the constructor.

- `DWORD m_dwID`

  This attribute is particularly important for `IFateComponent` objects that are involved in the event-handling mechanism as event-sources (e.g. see 4.4). Every object that has triggered an event (e.g. a button was pressed) sends its ID together with information about the event itself. Therefore, an event-handler can later identify which object has caused the event.
  This ID can be set manually by the programmer via a call to `SetID(DWORD dwID)` or it is automatically set when an object is added to a container.

- `CFateApp* m_pApp`

  An attribute which points to `NULL` at construction time of the object. After adding the object to a container, this pointer will point to the current application object (see 4.2.9).

- `IFateContainer* m_pParent`

  An attribute which points to `NULL` at construction time of the object. After adding the object to a container, this pointer will point to exactly this container (see 4.2.8).

4.2.2 IFButtonListener

This is a pure interface class which is responsible for handling events triggered by interaction with the graphical user interface class `CFButton` (see 4.4.2).
Each time a `CFButton` is pressed or released, registered subclasses of this interface are informed about these events. (To see how registration for certain events works, see 4.2.8.) Every class which wants to process events caused by `CFButton` has to implement this interface, which consists of following two methods:

- `BOOL ButtonPressed(DWORD dwBtnID)`

  This method is called by the framework for each registered subclass of `IFButtonListener` when the `CFButton` with the previously assigned ID `dwBtnID` was pressed. Each subclass can process this event then in its individual way and then either return `TRUE` or `FALSE`. This return value determines how the event handling is continued by the framework. `TRUE` indicates that the event has been processed and event handling must not be continued and when `FALSE` is returned, this method is called for the next registered `IFButtonListener`.

- `BOOL ButtonReleased(DWORD dwBtnID)`

  This method is called by the FATE framework following the same principle as `ButtonPressed()`.

  Each time the `CFButton` with the ID `dwBtnID` is released, this method is called for all registered subclasses of `IFButtonListener`.

4.2.3 IFItemListListener

This is a pure interface class which is responsible for handling events triggered by interaction with the graphical user interface class `CFItemList` (see 4.4.4).
Whenever an entry of the item list is selected (touchscreen interaction using the pocket PC's stylus), registered subclasses of this interface are informed about this event by

calling their method:

`BOOL ItemListSelected(DWORD dwListID, LPITEMLISTENTRY pEntry)`

This method is called by the framework for each registered subclass of `IFItemListListener` when a selection in the `CFItemList` (or one of its subclasses) with the previously assigned ID `dwListID` was made.

The second argument `pEntry` is a pointer to a structure of the type `ITEMLISTENTRY`, which has following members:

ITEMLISTENTRY	
TCHAR*	pszItem
TCHAR*	pszAddInfo
DWORD	dwIndex
DWORD	dwUser
CFBitmap*	pBmp
ITEMLISTENTRY*	pNext

Figure 4.3: Members of the structure ITEMLISTENTRY

The structure's first member `TCHAR pszItem` is the string which is displayed by the item list and which was selected by the user.

`TCHAR pszAddInfo` is an additional text string that may have been specified for the given entry (e.g.: a full path for a file name). This member may also be NULL.

`DWORD dwIndex` is the entry's (zero-based) index in the item list.

The structure's member `DWORD dwUser` is an extra value that may have been specified by the programmer. Its default value is 0.

The member `CFBitmap* bmp` is used for framework internal purposes and not intended to be accessed or modified during event handling.

`LPITEMLISTENTRY next` is a pointer to the next structure in the item list and also not intended to be used or modified by event handling methods.

Each subclass can process this event in its individual way and then either return `TRUE` or `FALSE` indicating whether event handling has to finish or not.

4.2.4 IFDropListListener

This is a pure interface class which is responsible for handling events triggered by interaction with the graphical user interface class `CFDropList` (see 4.4.5).
Registered subclasses of this interface class can handle three different events triggered by `CFDropList`. These event handling methods are called by the framework:

- `BOOL DropListExpanded(DWORD dwListID)`

  This method is called by the framework for each registered subclass of `IFDrop-ListListener` when the drop list expands caused by user interaction. Each subclass can process this event then in its individual way and either return `TRUE` or `FALSE`.

- `BOOL DropListSelected(DWORD dwListID, LPITEMLISTENTRY pEntry)`

  The behavior of this method is the same as in `ItemListSelected` (see 4.2.3).

- `BOOL DropListCollapsed(DWORD dwListID)`

  This method is called by the framework for each registered subclass of `IFDrop-ListListener` when the drop list is closed (collapsed) again by the user. Each subclass can process this event then in its individual way and either return `TRUE` or `FALSE`.

4.2.5 IFMenuLister

This is a pure interface class which is responsible for handling events triggered by interaction with the graphical user interface class `CFMenu` (see 4.4.7).

If the user opens a menu or selects an item within this menu, following method gets called for each registered subclass of this interface:

```
BOOL MenuSelected(DWORD dwMenuID, int iSelMain, int iSelSub)
```

The first argument `dwMenuID` is the ID which was assigned to the `CFMenu` object. The integer arguments specify which entry of the menu was selected, where `iSelMain` is the zero-based position within the main menu and `iSelSub` the zero-based position within the corresponding sub-menu.

4.2.6 IFSliderListener

This is a pure interface class which is responsible for handling events triggered by interaction with the graphical user interface class `CFSlider` (see 4.4.8).
If any user interaction with a slider object occurs, this method gets called for each subclass of this interface that was registered:

```
virtual BOOL SliderChanged(DWORD dwSliderID, int iValue)
```

Argument one `dwSliderID` specifies the ID of the event-source and `iValue` is the current value of the `CFSlider` object.

4.2.7 IFConnectionListener

This is a pure interface class which handles events caused by incoming network connections. The event itself gets triggered by the application object (see 4.2.9) if it has an open server socket (see 4.3.6) and a client wants to connect to it.
Then, for each registered subclass of this interface, this method is called:

```
virtual BOOL ClientConnect(CFServer *pServer)
```

The connection can then be accepted and communication to the client may start (see 4.3).

4.2.8 IFateContainer

This class is one of the framework's absolute core classes because it implements a significant part of the event-handling functionality of FATE. It extends `IFateComponent`, but also the previously mentioned interface classes

- `IFButtonListener`
- `IFItemListListener`
- `IFDropListListener`
- `IFMenuListener`
- `IFConnectionListener`
- `IFSliderListener`

This is due to experience reasons, which have shown that the processing of these events is mainly done at the level of container classes (e.g. in reaction to a pressed button, a container sets some of its child components into a visible or invisible state). Therefore, these event-handling methods have been made a part of the container's interface.

Another important part of this class's interface and a main extension to the functionality of `IFateComponent` are its `Add`-methods:

`BOOL Add(IFateComponent *pComp)`
This method establishes the actual parent-child-relation between an `IFateContainer` and an `IFateComponent` object.
Within this method, several important values are set for the child component, as following code sequence shows.

```
BOOL IFateContainer::Add(IFateComponent *pComp)
{
  // set main initialization values
  if (pComp->GetID() == 0xFFFFFFFF) pComp->SetID(m_app->GetNextID());
  pComp->SetParent(this);
  pComp->SetAppObject(m_app);
  pComp->SetWnd(m_hWnd);
  pComp->SetDC(m_hdc);

  return(m_pCompList->Append(pComp));
}
```

At first a unique ID is assigned to the component, if it has not already got one (the hexadecimal value 0xFFFFFFF is a default initialization value and indicates an invalid ID).

Then, the container is made the parent object of this component by calling its `SetParent` method.

In the next step, a pointer to the actual application object is set.

Every child component gets a handle to the application window and to the main device context [9][p.4].

Finally, this instance of `IFateComponent` is appended to a list, in which each `IFateContainer` stores its child-objects.

`BOOL Add(IFateContainer *pCont)`

This second overload of `Add` expects an `IFateContainer` object as its argument and had to be implemented because of the different treatment an container gets when it is added to another container.

Again, the same initialization as in the first overload happens, but with the difference that the list to which the object `pCont` is added, is a list which mainly consists of `IFateContainer` objects.

As additional difference, an `IFateContainer` instance gets added to a number of other lists. Each of these lists store a different kind of event-listener (see above). For the

reason, that `IFateContainer` implements all these listener interfaces, it therefore has to be added to all of the corresponding lists.
Following methods are called within `Add` to realize that:

- `BOOL AddButtonListener(IFButtonListener *pBtnLst)`
- `BOOL AddMenuListener(IFMenuListener *pMenuLst)`
- `BOOL AddItemListListener(IFItemListListener *pItemListLst)`
- `BOOL AddDropListListener(IFDropListListener *pDropListLst)`
- `BOOL AddSliderListener(IFSliderListener *pSliderLst)`
- `BOOL AddConnectionListener(IFConnectionListener *pConnLst)`

It is also the suggested design pattern to manually add a listener for a certain event to a container if one of the listener interfaces was implemented in a newly written class. Following code sequence demonstrates this.

```
public class MyListener : public IFButtonListener
{
  ...
};
...
MyListener *pMyListener= new MyListener();
pContainer->AddButtonListener(pMyListener);
...
```

The last big set of methods are container's *notification-methods*. These methods are called by the event handling mechanism, before a container's actual event-handling methods are called.
This happens because it is the default behavior of a container to dispatch any message to its registered child-controls or child-containers first, before it handles the event itself. Therefore, a corresponding notification-method exists for each event-handling method.

- `BOOL StylusDownNotify(int xPos, int yPos)`
- `BOOL StylusMoveNotify(int xPos, int yPos)`
- `BOOL StylusUpNotify(int xPos, int yPos)`
- `BOOL KeyDownNotify(PdaKey key)`
- `BOOL KeyUpNotify(PdaKey key)`
- `BOOL ButtonPressNotify(DWORD dwBtnID)`
- `BOOL ButtonReleaseNotify(DWORD dwBtnID)`
- `BOOL MenuSelectNotify(DWORD dwMenuID, int iSelMain, int iSelSub)`
- `BOOL ItemListSelectNotify(DWORD dwListID, LPITEMLISTENTRY pEntry)`
- `BOOL DropListExpandNotify(DWORD dwListID)`
- `BOOL DropListSelectNotify(DWORD dwListID, LPITEMLISTENTRY pEntry)`
- `BOOL DropListCollapseNotify(DWORD dwListID)`
- `BOOL SliderChangeNotify(DWORD dwSliderID, int iVal)`
- `BOOL ClientConnectNotify(CFServer *pServer)`
- `void DrawNotify()`

Using the example of the notification method for a button-press-event, following code sequence shows, how event propagation within an `IFateContainer` works.

```
BOOL IFateContainer::ButtonPressNotify(DWORD dwBtnID)
{
  CFListIterator<IFateContainer*> *pContIt;
  CFListIterator<IFButtonListener*> *pBtnIt;
  IFateContainer *pCont;
  IFButtonListener *pBtnLst;

  // first iterate through registered containers
  pContIt= m_pContList->GetIterator();
  while(pContIt->HasMore()) {
    pCont= pContIt->GetData();
    // containers have to notify their registrees again
    if (pCont->ButtonPressNotify(dwBtnID)) return(TRUE);
    if (pCont->ButtonPressed(dwBtnID)) return(TRUE);

    pContIt->Next();
  }

  // now iterate through listeners
  pBtnIt= m_pBtnList->GetIterator();
  while(pBtnIt->HasMore()) {
    pBtnLst= pBtnIt->GetData();
    if (pBtnLst->ButtonPressed(dwBtnID)) return(TRUE);

    pBtnIt->Next();
  }

  return(FALSE);  // event not handled
}
```

The first step is that a container checks its datastructure of registered child-containers. An iteration through these child-containers is done and again the `ButtonPressNotify` method for each of them is called. If any of these calls to `ButtonPressNotify` returns `TRUE`, which means that event has been processed somehow, the dispatching of the event is stopped immediately and the method returns. Otherwise, the direct event-handling method (in this case `ButtonPressed`) of the child-container is called.

If none of the child-containers had returned `TRUE`, event propagation is continued with the registered `IFateComponent` objects. Again an iteration through them is done and their `ButtonPressed` method is called. If none of these components processes the event, the method returns `FALSE`, which is again an indication for the parent container of this object, that the event propagation may continue.

4.2.9 CFateApp

This class provides the actual application functionality of the FATE framework. It extends `IFateContainer` and implements all of its interfaces. Therefore this class can be instantiated as is (for it has no pure virtual methods), but it does not have any further functionality beyond the ability of displaying a black screen.
To add additional behavior to this class, it is supposed to be subclassed and have its virtual methods overridden (see the sample application 5).
At instantiation time, an argument can be passed to the class's constructor which decides on the orientation the application's screen is going to have.
`CFateApp(UINT uDrawMode= DM_PORTRAIT)`
Four possible orientations of the screen exist (see figure 4.4):

- `DM_PORTRAIT` (default orientation)
- `DM_PORTRAIT_FLIPPED`
- `DM_LANDSCAPE`
- `DM_LANDSCAPE_FLIPPED`

An application's screen orientation can still be changed at runtime by calling the method `void SetDrawMode(UINT uDrawMode)`.
Following methods are called during the life cycle of a `CFateApp` object by framework mechanisms:

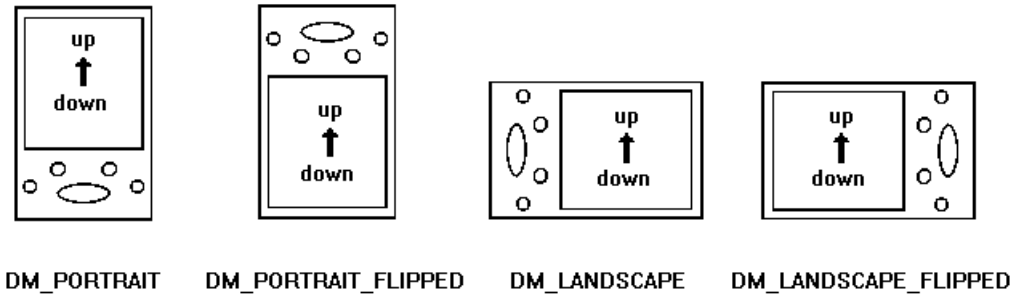


Figure 4.4: The four possible orientations of the application's screen.

- `BOOL InitFateApp()`

  This method is intended for all graphical initialization tasks because as soon as this method is called, it is guaranteed that the application has already created its internal doublebuffer and owns a handle to a valid windows device context (HDC) [9][p.4]). This handle is necessary for instance, for the creation of `CFBitmap` objects and can be accessed via the application's member variable `m_hdc`.

  The second reason for performing initialization tasks within this method is, that its boolean return value gets checked by the calling routine. If any problems occur during this initialization phase, the programmer can indicate this to the framework by returning `FALSE` and the application will immediately be shut down properly.

- `LRESULT EventHandler(...)`

  As from this method's interface can be seen, this is the typical Windows CE event-handling procedure.

  For every message that is sent to the applications event queue, this method will be called. It provides handling routines for some Windows CE system messages, but also for user-defined ones (see 3.3.3).

- `BOOL ExtraEventHandler(...)`

  This method is an additional event-handler which is called by FATE if messages are in the system's event queue that will not be handled by `EventHandler`.

  It is implemented empty (i.e. it returns `FALSE`) and it is supposed to be overridden by subclasses if any additional events have to be processed. A return value of `TRUE` indicates that the event was processed successfully.

- `void DrawDoubleBuffer()`

  If this method is called, the application's doublebuffer (member variable `CFBitmap *m_cbDblBuffer`, see 4.2.10) is drawn to the PDA's screen.

  This normally happens after a notification to redraw themselves has been sent to all child controls. But calls to `DrawDoubleBuffer` may also happen periodically if the applications drawing-loop is enabled (see next item).

- `void EnableFateLoop(BOOL bEnabled)`

  A call to this method enables or disables the framework's internal drawing-loop, which causes the application to redraw its doublebuffer as often as possible (i.e. as soon as the PDA's processor is idle).

  It is recommended to disable the drawing-loop in performance-critical situations and to manually call `DrawDoubleBuffer` if necessary. This makes sure that the doublebuffer is redrawn if its contents have changed asynchronously. This may be the case if drawing operations outside the `Draw` methods have occurred.

- `BOOL CloseFateApp()`

  This is the last method which will be called by the FATE framework before the `CFateApp` object gets deallocated. Any cleanup work can be done within this method or within the objects destructor.

An additional feature of each `CFateApp` object is its ability to function as a simple TCP server. An application's server port is automatically checked for incoming connections and the framework's event-mechanism is used to inform registered listeners about these connections (see 4.2.7).
Using this concept avoids possible problems of blocking socket operations or synchronization of concurrent server-threads.
The methods involved in these mechanisms are:

- `void AddServer(int iPort)`

  This method expects a port number as single parameter and automatically creates a new `CFServer` object that listens on this specified port (see 4.3.6). Each subsequent call to this method creates a new server socket and enqueues it in a list.

- `void CheckServers()`

  Performs an iteration through all listening server objects and if an incoming connection on any of them is detected, the message `WM_CLIENTCONNECT` is posted to the applications message-queue. As a result to that, the framework calls each registered `IFConnectionListener` about the pending connection.

The proper way to shut down an application based on the FATE framework is calling the method `BOOL Exit()`. After a call to `Exit()` the appropriate deinitialization and destruction routines are executed and all memory and resources which were allocated internally are properly deallocated again.

4.2.10 CFBitmap

This class is primarily a wrapper for windows device contexts but also supports access to the bitmaps pixels. The internal doublebuffer of FATE applications is also an instance of this class.

One of the main methods is the `Load()` method. It works on the pixel layer and allocates memory and resources for the specified bitmap and loads it into memory. The bitmap which will be loaded can be specified in three ways:

1. `BOOL Load(const char* pszFileName)`

   Using this method, the bitmap file specified by `pszFileName` will be loaded.

2. `BOOL Load(WORD wResourceID)`

   With the help of this method the resource specified via `wResourceID` can be loaded.

3. `BOOL Load(char *pszData, DWORD dwSize)`

   This method loads the bitmap file from the specified memory location `pszData` and the second argument `dwSize` gives the size of this memory block.

   This method is internally also used by the previous two overloads. It also validates the specified memory block whether it really has a valid bitmap format.

4.3 Communication

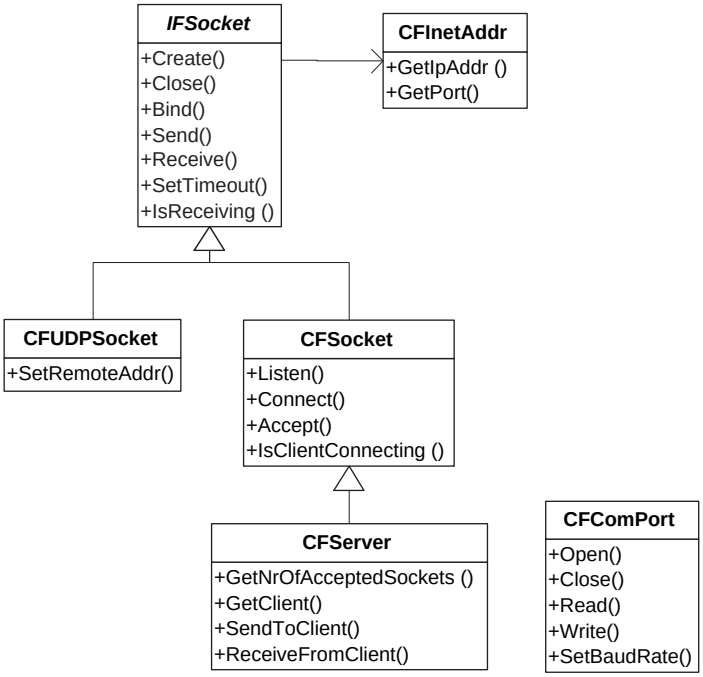


Figure 4.5: Communication classes and interfaces of FATE.

The FATE classes and interfaces described in this section are intended for use of communication and data exchange with other systems and provide rather low-level communication mechanisms like serial port connections or network sockets.
Nevertheless, these classes enable a more abstract view to these communication interfaces and encapsulate real low-level programming mechanisms.
As network connections and data exchange are an important topic when using a PDA as interaction device to other systems, stable and robust communication mechanisms are an essential part in the implementation of the framework.

4.3.1 CFComPort

This class encapsulates the functionality of accessing the pocket PC's serial port, sending and receiving data through it and adjusting the different connection settings that are available (e.g. baudrate, parity-checks, start and stops bits, etc.). The methods most commonly used are:

- `BOOL Open(int iPort= 1, BaudRate brBaud= BD_9600)`

  Opens a COM port at a specified baudrate and returns either `TRUE` or `FALSE` whether the operation has succeeded or not.

- `int Read(char *pszBuffer, int iBuffSize)`

  This method reads a maximum amount of `iBuffSize` into the buffer specified by `pszBuffer`. The return value is the number of bytes, which where actually read from the port.

- `int Write(const char *pszBuffer, int iBuffSize)`

  This method attempts to write `iBuffSize` bytes from the buffer specified by `pszBuffer` to the port. The return value is the number of bytes, which were actually written to the port.

4.3.2 CFInetAddr

A simple wrapper class for internet addresses, used by the frameworks socket classes (see next subsections) for specifying endpoints of network connections.
The class is mainly used by passing a dotted IP string and a port number to its constructor `CFInetAddr(const char *pStrIP, const USHORT usPort)` so that an appropriate internet address is created.

4.3.3 IFSocket

`IFSocket` is an abstract class which serves as baseclass for `CFSocket` (implementation of a TCP socket) and `CFUDPSocket` (implementation of a UDP socket). It extracts and implements the common functionality of these derived socket classes.
Socket methods which are readily implemented by this class are:

- `BOOL Bind(...)`

  This method binds the socket to a specified port which enables a UDP socket to simply receive data on this port and a TCP socket to accept incoming connections on it.

  There exist to overloads for this method.

  The first one simply expects a number in its argument list and binds the socket to the port specified by this number. The IP address to which the socket is bound, gets chosen automatically which is usually the desired behavior for PDAs, normally just having one network interface.

  The second possibility is to provide an instance of `CFInetAddr` as argument which allows the programmer to specify both a port and an IP address to which the socket is bound.

- `BOOL IsReceiving()`

  A boolean value is returned by this method which indicates whether another socket is currently trying to send data to this one. The implementation is the same for UDP and TCP sockets.

- `void SetTimeout(DWORD dwTimeout)`

  The sending and receiving behavior of the framework‘s socket classes can be altered by this method. These classes can both work in blocking as well as in non-blocking mode.

The sockets' default behavior is to act in non-blocking mode which means that all send- and receive-operations time out automatically after a certain amount of seconds (see constant `DEFAULT_TIMEOUT`) if the opposite socket does not respond to this operation. The number of seconds that has to pass until a timeout occurs can be set for each individual socket by the `dwTimeout` argument.

If the socket is supposed to work in blocking mode, the constant value `NO_TIMEOUT` has to be specified as argument to `SetTimeout`. This means that if an opposite socket does not respond to send or receive operations, the calling socket will hang as long as the connection is open.

- `BOOL Close()`

  This method closes the socket and automatically shuts down any existing connection. A socket that was closed has to be reopened by a call to `Create` before any other operations (e.g. connect, send, receive, ...) can be performed on it.

The actual sending and receiving methods are left purely virtual for the reason that they are implemented differently for TCP and UDP sockets.
The interface however, provided by `IFSocket` is as follows:

- `BOOL Create()`

  This is always the first operation that has to be performed for a socket. Without a prior call to `Create()` all other methods will fail.

- `int Send(const char *pBuff, const int iSize)`

  A maximum count of `iSize` bytes in the buffer `pBuff` are sent to the other endpoint of this socket connection.

  The return value is the number of bytes that were actually sent, `SOCKET_ERROR` in case of an error and `SOCKET_TIMEOUT` in case of a timeout.

- `int Receive(char *pBuff, const int iSize)`

  A maximum count of `iSize` bytes is received into the buffer specified by `pBuff` from the other endpoint of this socket connection.

  The return value is the number of bytes that were actually received, `SOCKET_ERROR` in case of an error and `SOCKET_TIMEOUT` in case of a timeout.

4.3.4 CFUDPSocket

The functionality of UDP sockets is implemented by this class. It basically adds three methods to `IFSocket`, which are necessary because of the connection-less behavior of the UDP protocol.

- `void SetRemoteAddr(LPINETADDR addrRemote)`

  For the reason that UDP is a connection-less protocol, the receiving endpoint has to be specified to the sender. By setting the remote address explicitly, via passing a pointer to a `CFInetAddr` object before the actual sending operation, the interface method `Send()` declared in `IFSocket` can also be used for UDP communication.

- `int Send(const char *pBuff, const int iSize, LPCINETADDR pInetAddr)`

  This overloaded `Send()` method expects an additional argument, which is a pointer to a `CFInetAddr` object, which specifies the socket address where the data is sent to. Due to this, no prior call to `SetRemoteAddr()` is necessary.

- `int Receive(char *pBuff, const int iSize, LPINETADDR pInetAddr)`

  The third argument of this overloaded `Receive()` method also expects a pointer to an instance of `CFInetAddr`. As soon as data is received from another UDP socket, the data fields of the `CFInetAddr` object are automatically filled with the IP address and the port number of the sending socket.

4.3.5 CFSocket

This class encapsulates the behavior of connection-orientated TCP sockets and the operations that may be performed on them.

- `BOOL Connect(...)`

  A call to this method puts a socket into client state (which is the common behavior for network applications on PDAs, as experience shows).

  Two overloads have been implemented for this method:

  The first one simply expects an IP address in string format (e.g. "192.168.5.110") and a port number in its argument list. It internally creates a `CFInetAddr` object out of these arguments and tries to connect to this address.

  The second possibility is to immediately provide an instance of `CFInetAddr` as argument. This is the implementation that is also called by the previous overload after its arguments have been transformed.

- `BOOL Listen()`

  Puts the socket into listening state. This means that the sockets acts as server and therefore a call to `Bind()` must have occurred before. After a successful call to `Listen()` the socket is able to accept incoming connections.

- `BOOL IsClientConnecting()` This method is only adequate for sockets which are in listening state, i.e. a prior call to `Listen()` is necessary. The return value shows whether there is a client, that is currently trying to connect to this socket and can be accepted.

- `BOOL Accept(CFSocket &sock)`

  If there is a pending connection from a client socket, a call to this method actually establishes the connection. The argument is a reference to another `CFSocket`

object, which serves as the communication socket to the client after the connection has been established successfully.

The method's blocking behavior also depends on the timeout-settings that were made for this socket. This means, that if no connection is pending it will either timeout after the specified interval or block forever.

4.3.6 CFServer

This class is derived from `CFSocket` and comes to use when a PDA takes on server functionality. It extends its baseclass with some server-specific functionality and behavior. By overriding the `Accept` method of `CFSocket` it dynamically manages a list of connected clients (`CFSocket` objects). Additional methods for enhanced communication to these clients are:

- `BOOL IsReceiving(int iClient)`

  This method checks for the connected client number `iClient` whether it is currently sending data to the server.

- `int SendToClient(char *pBuff, int iSize, int iClient)`

  The behavior is identical to `Send` in `CFSocket` but with the additional parameter `iClient` the zero-based index of the client can be specified to which the data will be sent.

- `int ReceiveFromClient(char *pBuff, int iSize, int iClient)`

  The behavior is identical to `Send` in `CFSocket` but with the additional parameter `iClient` the zero-based index of the client can be specified from which the data is going to be received.

4.4 GUI

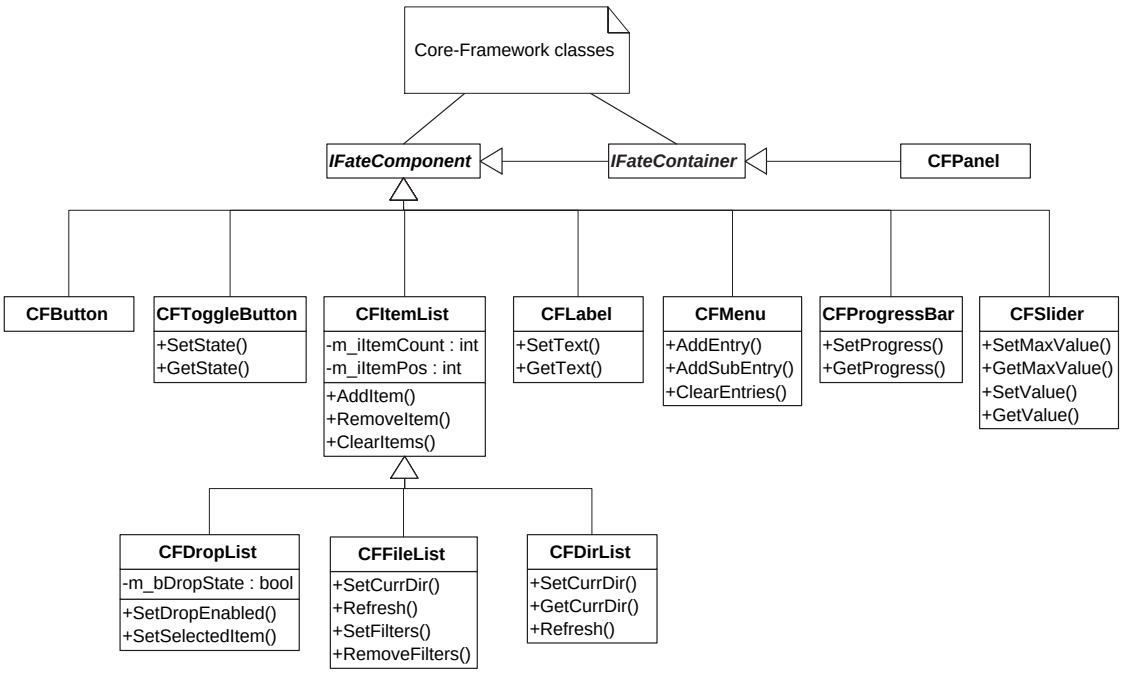


Figure 4.6: GUI classes and interfaces of FATE.

These classes provide the functionality of basic GUI elements, which are needed for most applications that require some user interaction.

All these classes are directly or indirectly derived from `IFateComponent` which is necessary to be automatically redrawn by the framework, respectively to be involved in the framework's event processing mechanism.

4.4.1 CFLabel

This is a simple class which only has the purpose of displaying text on the screen.

The text to be displayed by the label can either be specified to the class's constructor or specified later with a call to `SetText()`.

Internally, it uses a `CFBitmap` object on which the text is drawn and each time the text

buffer gets changed, the bitmap is also altered.
The graphical appearance of such a text label can be changed by various methods which are:

- `SetColText(COLORREF col)`: Changes the foreground-color of the actual text which is displayed.
- `SetColBack(COLORREF col)`: Changes the background-color of the actual text which is displayed.
- `SetColBorder(COLORREF col)`: Specifies the color of the border around the displayed text.

Each of these methods (as well as `SetText`) cause a call to the private method `CreateBitmap()` which renews the buffered bitmap of the label and displays it on the screen again.
`CFLabel` objects do not respond to any kind of user interaction and neither trigger any events.

4.4.2 CFButton

This class provides the well-known functionality of a GUI button and can be constructed in two different ways:

- Text: The button's caption is specified to the constructor and it is drawn in the framework's default colors for the button's non-pressed, pressed and disabled state.
- `CFBitmap`: Bitmap objects can be specified in the class's constructor which determine the button's appearance in its non-pressed, pressed and disabled state.

A `CFButton` can trigger two kinds of messages which are processed by the framework's event mechanism:

- The method `StylusDown()` from `IFateComponent` was overridden and is called each time when the button gets pressed by user-interaction. This triggers the message `WM_BUTTONPRESS` and classes that implement the `IFButtonListener` interface (see 4.2.2) can process this event.
- The method `StylusUp()` from `IFateComponent` was overridden and is called each time when the button gets released. This triggers the windows message `WM_BUTTONRELEASED` and classes that implement the `IFButtonListener` interface (see 4.2.2) can process this event.

4.4.3 CFToggleButton

The functionality of this class is different from a `CFButton` in a way that it toggles between two states, like a switch or a checkbox. This means that there is only one message that is triggered by this class.
The method `StylusDown()` from `IFateComponent` was overridden and is called each time when the button gets pressed and changes its state therefore. This triggers the message `WM_BUTTONPRESS` and classes that implement the `IFButtonListener` interface (see 4.2.2) can process this event.
The button's state can also be set programmatically by a call to `SetState(int iState)` and can be retrieved via calling `GetState()`.

4.4.4 CFItemList

This class implements the functionality of a GUI item list. It displays a list of textual entries, from which a selection can be made.
Following methods provide this behavior:

- `BOOL AddItem(TCHAR *pszItem, TCHAR *pszAddInfo=NULL)`

  The textual item `pszItem` is added to the list. An additional string (which is not displayed but can be retrieved programmatically) can be specified via the argument `pszAddInfo` to enhance the entry with further information.

- `BOOL AddPicItem(CFBitmap *pic, TCHAR *pszItem, TCHAR *pszAddInfo)`

  The arguments `pszItem` and `pszAddInfo` are the same as mentioned before and additionally a picture can be specified which is displayed in the list.

- `BOOL RemoveItem(int index)`

  A call to this method removes the item at the (zero based) position `index`.

- `BOOL ClearItems()`

  This method removes all items from the list.

- `void ItemSelected(LPITEMLISTENTRY pEntry)`

  This method is called as soon as an entry in the list is selected by user-interaction. The message `WM_ITEMLISTSELECTED` is triggered and classes that implement the `IFItemListListener` (see 4.2.3) interface can process this event.

There are two further subclasses of `CFItemList`, which hardly add extra functionality but are used for more specialized purposes. The events triggered by these classes are identical to `CFItemList`. These classes are:

- `CFFileList`
- `CFDirList`

4.4.5 CFDropList

This is a subclass of `CFItemList` which adds extra functionality to the item list.

It consumes less space on the screen because it can be expanded and collapsed again and normally displays just the selected item.
Due to this behavior it triggers other events than `CFItemList` which all can be processed by classes that implement the `IFDropListListener` (see 4.2.4) interface.

- `WM_DROPLISTEXPANDED`
- `WM_DROPLISTSELECTED`
- `WM_DROPLISTCOLLAPSED`

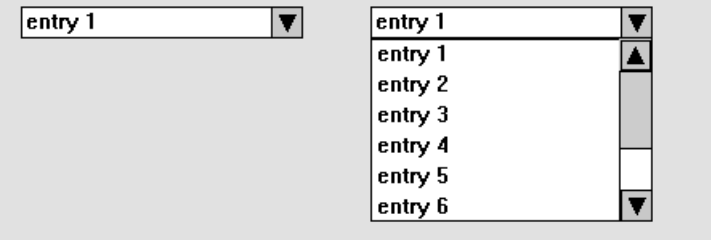


Figure 4.7: A CFDropList in collapsed and expanded state.

4.4.6 CFPanel

This class extends the IFateContainer interface and provides basically an empty implementation for all the pure virtual methods of this interface.
This means that `CFPanel` does not respond to any kind of user interaction, neither does it have a graphical representation.
Nevertheless, it can be instantiated and used in applications in two different ways:

- Firstly, it can serve as an invisible aggregation component. This means, that due to its `IFateContainer` functionality, it can serve as parent object for other classes derived from `IFateComponent`. This is done by calling its `Add()` method

and specifying the respective child-component. By using this aggregation mechanism it is possible to link the visibility and/or enabled-state of multiple components logically. This logical association is realized via the appropriate calls to `SetVisible()` respectively `SetEnabled()` of their parent `CFPanel`.

- The second main purpose this class was designed for, is to serve as baseclass for more specialized panels in terms of GUI components.

  By overriding this class's `Draw()` method a visual representation can be given to a panel, so it can serve as a child window to the main application. Further, its event-handling methods can be overridden as well, so the panel is able to respond to any kind of user interaction or events triggered by other GUI components. (An example of this subclassing design-pattern is the class `CFVEObjPanel`.)

4.4.7 CFMenu

The functionality of a drop-down menu is implemented by this class. Following methods can be used to create the menu:

- `BOOL AddEntry(LPCTSTR pText)`

  Adds top-level entry to the menu which is displayed as `pText`.

- `BOOL AddSubEntry(LPCTSTR pText, int iPos)`

  Adds a sub-entry to the (zero-based) top-level entry at position `iPos`. If no entry at this position exists, the call fails.

- `BOOL ClearEntries()`

  This method clears the whole menu.

If any entry in the menu was selected, the message `WM_MENUSELECTION` is triggered and classes that implement the `IFMenuListener` (see 4.2.5) interface can process this event.

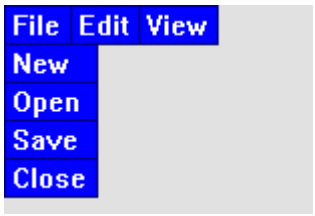


Figure 4.8: The CFMenu GUI class.

4.4.8 CFSlider

This GUI control allows the user to make a selection between a minimum (always 0) and a maximum value.

Calls to `GetMaxValue` and `SetMaxValue` can alter this maximum value and `GetValue` and `SetValue` influence the actual value represented by this control.

Each time the value of this control is changed (either manually or programmatically) the message `WM_SLIDERCHANGE` is posted and classes that implement the `IFSliderListener` (see 4.2.6) interface can process this event.

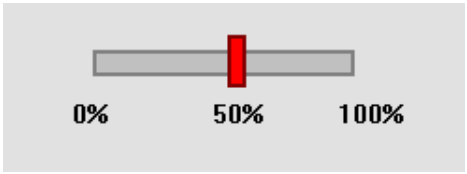


Figure 4.9: The CFSlider GUI class.

4.5 Advanced Interaction and Navigation

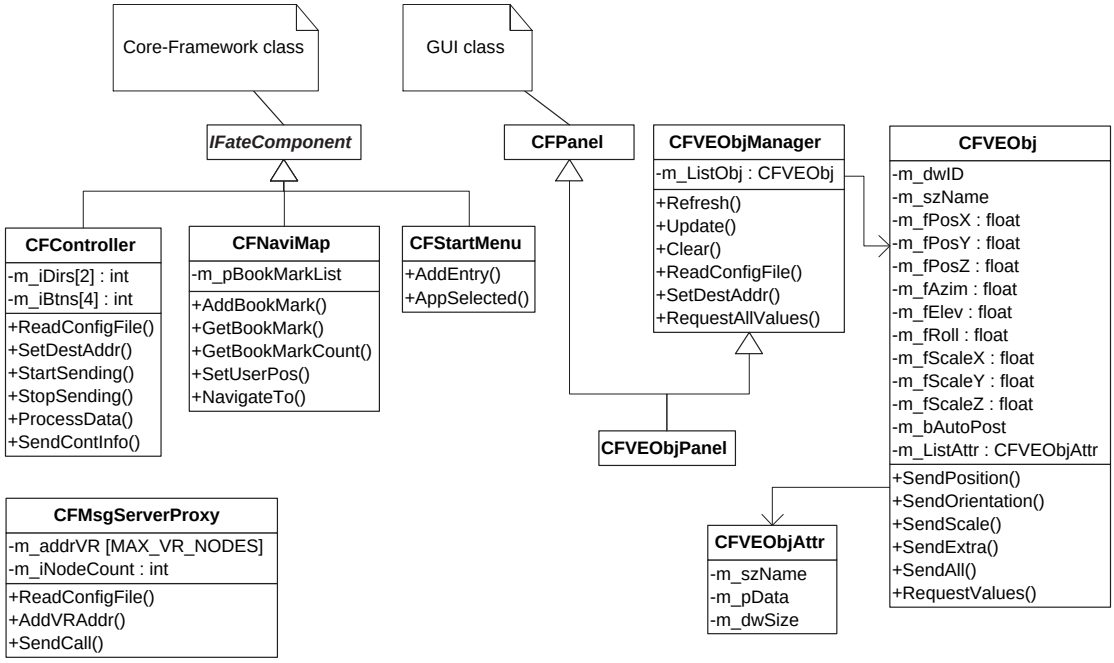


Figure 4.10: Interaction and navigation classes and interfaces of FATE.

Following FATE classes and interfaces provide possibilities of advanced interaction and navigation in virtual environments. Most of these classes are advanced user-interface controls which have a readily implemented graphical appearance and are supposed to provide a solution for more specific problems (e.g. map-based navigation).

4.5.1 CFController

This class provides the full functionality of turning a pocket PC into an interaction device like a WAND.

It is intended to work within an architecture that uses TRACKD. Therefore it implements the TRACKD controller protocol to propagate key interaction to the VR system. In case of an iPAQ pocket PC, its navigation button can be used as a joystick and its four hardware buttons serve as interaction buttons (see figure 4.2).

`CFController` is derived from `IFateComponent` and can therefore be added to any `IFateContainer`. It has no graphical representation, but it overrides following methods to process user interaction:

- `BOOL KeyDown(PdaKey key)`
- `BOOL KeyUp(PdaKey key)`

There are two ways of setting the destination IP address for this component:

- `BOOL ReadConfigFile(LPCTSTR pszFileName)`

  In this case, the IP configuration is done based on the file specified with the argument `pszFileName`. If the file contains an entry in the form of `TRACKD: IP-address: Port` (e.g. `TRACKD: 192.168.5.110: 9010`) this is the address where the controller values are sent to.

- `void SetDestAddr(CFInetAddr *pAddr)` With the help of this method the IP address can be set programmatically.

For propagating the controller values to the VR system, these are the important methods:

- `BOOL StartSending(DWORD dwMillis=200)`

  This method initiates the process of continually sending the controller data to the previously specified IP address. The argument `dwMillis` specifies the interval in which the data is sent.

  Internally, a new thread is started which periodically sends the controller data.

- `BOOL StopSending()`

  The sending process is stopped after this method is called.

- `BOOL SendContInfo()`

  The actual sending process happens within this method. It is automatically called in the interval that was formerly specified by `StartSending`. The controller data is assembled into a UDP datagram according to the TRACKD protocol and sent to the formerly specified IP address.

If another form of event propagation instead of TRACKD is desired, this can easily be achieved by subclassing `CFController` and overriding its virtual method `void ProcessData()`. Within this method the protected attributes `int m_iDirs[2]` and `int m_iBtns[4]` can be accessed and processed in the desired ways.
The default implementation of `ProcessData()` just calls `SendContInfo()`.

4.5.2 CFMsgServerProxy

The basic idea of this class is to serve as a kind of proxy between the pocket PC application and the VR system. Particularly, this class is intended for message server architectures like Ygdrasil (see 2.3).
For this reason, Ygdrasil was extended by a TCP-Server component, which waits for incoming commands. These commands are supposed to have a certain syntax and according to this syntax are interpreted as scripts and dispatched to the corresponding objects within the scene graph. This mechanism enables FATE programs to directly manipulate scene graphs of VR applications.
For the IP configuration of this class two different possibilities exist:

- `BOOL ReadConfigFile(LPCTSTR pszFileName)`

  In this case, the IP configuration is done based on the file specified with the argument `pszFileName`. For each entry in the form of `VR: IP-address:Port` (e.g. `VR: 192.168.5.110:2233`) in this file, another IP address is added to a list datastructure. This multiple address configuration is intended for cluster

architectures and PC CAVE systems like the ARSBOX [3], because in systems like these, each node of the cluster owns its local copy of the scene graph. For this reason, scene graph manipulations have to occur on each of the nodes individually.

With the help of this IP address list, `CFMsgServerProxy` automatically propagates scene graph changes to all of the registered addresses.

- `void AddVRAddr(CFInetAddr *pAddr)`

  With the help of this method an IP address can be added to the list of VR hosts programmatically.

Following methods are intended for the actual communication with the message servers.

- `BOOL SendCall(char *pData)` This method expects one argument which is a readily assembled string, that is sent to all registered IP addresses. The format of this string depends on the used message server architecture.

  The correct syntax for the Ygdrasil message server extension is as follows:

  `"ObjectName|MethodName(Argument1, Argument2, ...)\n"`. If e.g. the scene graph of a VR application owns an object with the name `"spaceship"`, which has a method `"RotateX(float)"`, this method can be executed remotely like this:

  `SendCall("spaceship.RotateX(2.5)\n")` (provided that the IP configuration is correct).

- `BOOL SendCall(char *pObjName, char *pMethName, char **ppArgs)`

  This method is particularly intended for the Ygdrasil message server extension.

  Its expected arguments are the individual components of the call string. The call string gets assembled from these components and `SendCall(char *pData)` is called.

  When using this method, it has to be made sure that the last entry of the `ppArgs` array is a NULL pointer.

The main intention of how to use this class is to create a new class and to derive it from `CFMsgServerProxy`. This newly created class and its methods should be named equally to the scene graph object which is intended to be manipulated, so it can be seen as a proxy for the remote object.
Taken the example above, a class `CSpaceShip` (derived from `CFMsgServerProxy`) could be created. Then the method `void RotateX(float)` can be added to this class. This method could be implemented as follows:

```
void CSpaceShip::RotateX(float fVal)
{
  char szCall[256];

  // create the call-string
  sprintf(szCall, "spaceship|RotateX(%f)\n", fVal);
  // send the call
  CFMsgServerProxy::SendCall(szCall);
}
```

Using this design pattern, the source code of FATE applications become more readable and communication to VR systems is easy and transparent.

4.5.3 CFNaviMap

This class is a ready-to-use graphical component derived from `IFateComponent` and was designed and implemented especially for the purpose of navigation in vast virtual environments. The class's constructor `CFNaviMap(CFBitmap *bmpMap)` expects an argument of `CFBitmap` type, which is expected to contain a 2D map-representation of the VR scene. With this map the user should be able to navigate through the virtual environment.
For each interaction with the map itself, the virtual method `void NavigateTo (int iNaviX, int iNaviY)` is called. To be able to respond to this interaction a new class has to be derived from `CFNaviMap` and the `NavigateTo` method has to be overridden.

Following code snippet demonstrates how this navigation information could be processed and sent to another host (e.g. a tracking server) via network.

```
class MyNaviMap : public CFNaviMap
{
public:
  ...
  virtual void NavigateTo(int iNaviX, int iNaviY)
  {
    // coordinates could be transformed here ...
    // copy coordinates into a string
    char szBuff[128];
    sprintf(szBuff, "%d,%d", iNaviX, iNaviY);
    // send them
    CFUDPSocket sock;
    sock.Send(szBuff, strlen(szBuff) + 1, m_addrRemote);
  };
private:
  CFInetAddr *m_addrRemote;
  ...
};
```

Another useful feature of this class is its bookmarking system, which can be compared to the bookmarking system of an internet browser. The user can add a bookmark at a position of certain interest to be able to return to this position later.
For the administration of this bookmarking system, following methods have been implemented:

- `AddBookMark(LPCTSTR pszRemark= NULL)`

  This method adds a bookmark at the user's current navigation position. Additionally, via `pszRemark`, a text string can be specified to add descriptive information to this bookmark.

- `AddBookMark(int iPosX, int iPosY, LPCTSTR pszRemark= NULL)`

  The method's functionality is the same as above. The only difference is, that the

bookmark position can be specified programmatically via the arguments `iPosX` and `iPosY`.

- `GetBookMarkCount()`

  This method retrieves the number of saved bookmarks.

- `GetBookMark(int iNr)`

  Retrieves the bookmark at the specified position.

- `DeleteBookMark(int iNr)`

  Deletes the bookmark at the specified position.

- `ClearBookMarks()`

  Delete all saved bookmarks.

- `SaveBookMarks(LPCTSTR pszFileName)`

  This method exports all currently saved bookmarks into the file specified by `pszFileName`. The export format is an xml format which is described by following *Document Type Definition (DTD)* [6][7].

```
<!ELEMENT bookmarks (bookmark*)>
<!ELEMENT bookmark (remark?)>
<!ATTLIST bookmark xpos CDATA #REQUIRED>
<!ATTLIST bookmark ypos CDATA #REQUIRED>
<!ELEMENT remark (#PCDATA)>
```

  A correctly exported file could look like this:

```
<?xml version= "1.0"?>
<bookmarks>
  <bookmark xpos= "100" ypos= "50">
  </bookmark>
  <bookmark xpos= "200" ypos= "80">
    <remark> A place of special interest. </remark>
  </bookmark>
</bookmarks>
```

- `LoadBookMarks(LPCTSTR pszFileName)`

  This method loads a set of bookmarks from a previously saved file. The format must match the specifications above.

4.5.4 CFStartMenu

`CFStartMenu` is a graphical component and therefore derived from `IFateComponent`. It was implemented as a graphical component which is ready to be used and serves as an interactive menu for launching applications in a virtual environment system.
This is done with a list of symbols, that can be configured programmatically. Each of these symbols represents one application in the VR system and by pressing these symbols a user can switch between the individual VR applications which are represented by these symbols.
The `CFStartMenu` component can be seen as a "remote control" to VR environments like the CAVE, where the user can switch between the individual "channels" (i.e. the VR applications) by using the Palmist.
This class is intended to be subclassed to be able to override its method `void AppSelected(int iNr)`. By calling this method, the framework indicates that the user wants to launch the application at position `iNr`.

```
class MyStartMenu : public CFStartMenu
{
public:
  ...
  virtual void AppSelected(int iNr)
  {
    // copy application number into a string
    char szBuff[32];
    sprintf(szBuff, "%d", iNr);
    // send the data
    CFUDPSocket sock;
    sock.Send(szBuff, strlen(szBuff) + 1, m_addrRemote);
  };
private:
  CFInetAddr *m_addrRemote;
  ...
};
```

4.5.5 CFVEObjManager

This class is intended for the actual exchange of information about 3D objects with virtual environment systems.

Instances of this class can be integrated into both Palmist applications as well as into the actual VR applications.

A `CFVEObjManager` object, that exists within a Palmist application, can connect to another object of this type via network. If it connects to an object within a VR application, it can request data about all the 3D objects and models that are part of this application. This is done by calling the object's `Refresh()` method. The Palmist application gets a list of these 3D objects with their main attributes.

Each object which is transferred from the graphic workstation to the Palmist or vice-versa is described by the structure `CFVEObj` (see figure 4.11).

With the help of this mechanism, the Palmist can alter the 3D values of a VR application in a precise way. All changes to these values that were made by the Palmist

CFVEObj	
DWORD	m_dwID
char	m_szName[MAX_OBJ_NAME_LEN]
float	m_fPosX
float	m_fPosY
float	m_fPosZ
float	m_fAzim
float	m_fElev
float	m_fRoll
float	m_fScaleX
float	m_fScaleY
float	m_fScaleZ
CFVEObjManager*	m_pMgr

Figure 4.11: Main members of the class CFVEObj

application can be sent back to the VR system, where these manipulations are visualized immediately.

4.5.6 CFVEObjPanel

This component is derived from `CFVEObjManager`. It inherits all of its functionality but additionally provides a complete user interface (see figure 4.12) to access this functionality.

The user interface has an item list which contains all of the `CFVEObj` objects, that the application has received from the VR system. It also provides the possibility to change the attributes of these objects via a graphical user interface.

With the help of this graphical component no complicated programming steps are necessary to establish a connection to a VR system and to exchange data. The only

thing that is left to do for a programmer is to create an instance of this object and to configure it appropriately. Following code sequence demonstrates this.

```
...
// Create and configure the CFVEObjPanel.
CFVEObjPanel *pPanel= new CFVEObjPanel();
// Set the network address where the VR application is running.
pPanel->SetDestAddr("192.168.0.198", 5612);
// Add the panel to application.
Add(m_pPanel);
...
```

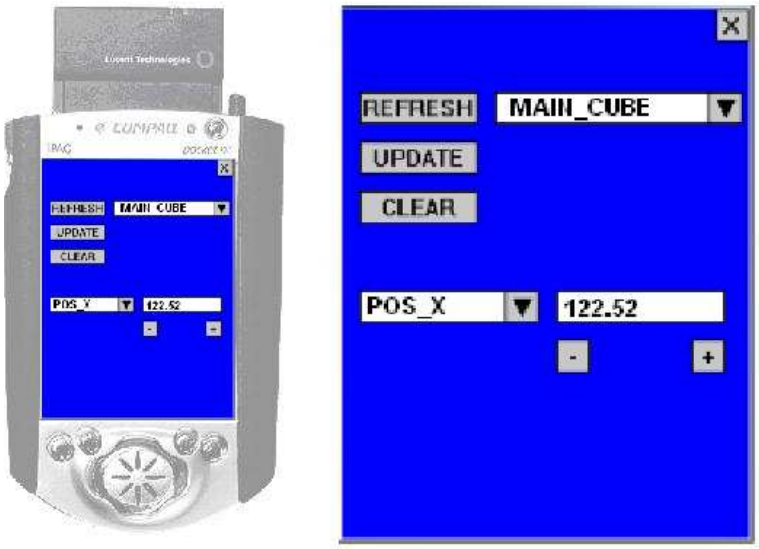


Figure 4.12: CFVEObjPanel - a ready-to-use GUI component.

Chapter 5

Example

5.1 Problem Description

The following example shows the "Picasso" application - an actual application for the Palmist based on the software framework FATE. It demonstrates the possibilities of the Palmist as a "design on the fly" tool for virtual prototyping [40].
"Picasso" is a VR visualization application, that was developed for a car manufacturer. The application shows a 3D model of a car and with the help of the Palmist, the user can navigate around this model and examine it from different distances and perspectives. This navigation can be performed in a familiar joystick metaphor.
To illustrate the advantages of virtual prototyping in a better way, the Palmist application provides an additional interaction feature, which gives the user the possibility of changing the color of the car body dynamically.
For this purpose the Palmist application provides a basic user interface, with which the user can compose an RGB color value. This value is automatically transferred to the VR application and the 3D model of the car is displayed in the appropriate color. This mechanism provides very accurate interaction and the user gets immediate feedback about the applied changes when the virtual car body changes in response to the actions

performed in the Palmist application.

The application which performs the actual visualization of the 3D car model is running on a VR system like the CAVE [21] or the ARSBOX [3] and uses TRACKD (see 2.2) for collecting tracker data and Ygdrasil (see 2.3) and Performer [37] for the VR graphics.

The following subchapters give more detailed information about the structure and the implementation of the Palmist part of this application.

5.2 Solution Approach

The Palmist application that was developed for this purpose has a simple graphical user interface with which the basic interaction can be performed.
The user interface of the "Picasso" application consists of three slider bars and each of these three slider bars is associated with one component of an RGB color value (see figure 5.1). This RGB color value represents the current color of the car visualized by the VR application. By manipulating this RGB color value on the Palmist, the car's color is changed accordingly.

Figure 5.1: The Palmist as design on the fly tool.

To be able to navigate around the 3D model of the car, the cursor keys of the Palmist can be used.
For interaction with the VR application following components of the FATE framework are used:

- `CFController`

  This software component (see 4.5.1) is responsible for navigation in virtual environments. It detects when the cursors keys of the Palmist are pressed and sends the appropriate navigation commands to the VR system. In this case the TRACKD tracking middle-ware (see 2.2) is running on the VR system and receives this commands to perform the navigation operations in the VR application.

- `CFMsgServerProxy`

  This software component (see 4.5.2) applies the actual changes to the 3D model of the car in the VR application using the mechanism of the Ygdrasil message server (see 2.3).

  Whenever the RGB color value is manipulated on the Palmist, these changes are forwarded to the VR system by this software component. The application running on the VR system interprets these changes and visualizes the car in the corresponding color.

5.3 Selected Details of the Solution

This chapter shows some selected code lines that cover the main functionality of the "Picasso" application. Some code parts which are not primarily relevant for the actual application logic are skipped, but the main part of the source code is listed on the following pages.

```
//--------------------------------------------------------------------
// Global function for creating the application object.
CFateApp* CreateFateApp()
{
  // Return an instance of our application class.
  return(new CPicasso());
}
```

The first code snippet shows the obligatory entry point for each application based on the FATE framework. This function is called by framework mechanisms and is supposed to return an instance of a class which is derived from `CFateApp` (see 4.2.9). In the case of this application a `CPicasso` object is returned by this function.

```
//--------------------------------------------------------------------
CPicasso::CPicasso() : CFateApp()
{
  // Initialize some member variables.
  m_iRedValue=   0;
  m_iGreenValue= 0;
  m_iBlueValue=  0;
  ...
}
```

The constructor of the `CPicasso` class performs initializations of its member variables, that store the RGB color values.

```
00 //--------------------------------------------------------------
01 BOOL CPicasso::InitFateApp()
02 {
03   // Create and configure the controller.
04   m_pCont= new CFController();
05   m_pCont->ReadConfigFile("config.txt");
06   // Add controller to application.
07   Add(m_pCont);
08   m_pCont->SetEnabled(TRUE);
09   m_pCont->StartSending(100);  // update ever 100 milliseconds
10
11   // Create and configure the proxy for the message server.
12   m_pMsgServer= new CFMsgServerProxy();
13   m_pMsgServer->ReadConfigFile("config.txt");
14
15   ...
16
17   // Create the red slider.
18   m_sliderR= new CFSlider();
19   m_sliderR->SetMaxValue(255);
20   m_sliderR->SetID(ID_SLIDER_R);
21   m_sliderR->SetVisible(TRUE);
22   m_sliderR->SetX(40);
23   m_sliderR->SetY(120);
24   Add(m_sliderR);
25
26   // ... same for green and blue slider
27
28   // Create the labels for current color value RED.
29   m_labelR= new CFLabel(30, 20);
30   m_labelR->SetValue(0);  // initial value
31   m_labelR->SetVisible(TRUE);
32   m_labelR->SetX(m_sliderR->GetX());
33   m_labelR->SetY(m_sliderR->GetBottom() + 20);
34   Add(m_labelR);
35
36   // ... same for GREEN and BLUE
37
38   return(TRUE);
39 }
```

The `InitFateApp` method is called by the framework at the initialization stage. It is recommended to perform all main initialization tasks (e.g. opening files or network connections, creating user interface elements, etc.) within this method, because the return value of this method is taken as an indicator whether the initialization of the application was successful or not.
A return value of `TRUE` indicates that the initialization was successful and that the execution of the application may continue.
Otherwise, the framework stops the execution of this application and deletes the application object again.

In the "Picasso" application this initialization consists of three main steps:

1. *Step one - The CFController object*

   Code lines 3-9 demonstrate how to create and correctly initialize a `CFController` object (see 4.5.1).

   After constructing the object, a configuration file ("config.txt") is specified, from which the object parses the IP address and the network port on which the tracking middle-ware is receiving data.

   Then, the `CFController` object is added to the application container and its state is set to enabled, which involves the object in the event management system of the FATE framework.

   Finally, the sending interval is set to 100 milliseconds, i.e. that 10 times per second the current status of the Palmist's navigation keys is sent to the VR system.

2. *Step two - The CFMsgServerProxy object*

   In code lines 12 and 13 it is shown how to construct and initialize a `CFMsgServer-Proxy` object (see 4.5.2).

Again, a configuration file is specified, from which the object parses the necessary IP address and network port on which the actual server is receiving the commands which are applied to the 3D model.

3. *Step three - The user interface*

   Starting with code line 18, several examples for the initialization of user interface components are shown.

   First, the three sliders are constructed which are used to alter the red, green and blue components of the RGB color value. In the code listing above, only the code for the "red slider" is shown, because the code for the "green" and "blue" slider is almost equal and can easily be reconstructed.

   After having the `CFSlider` object constructed its maximum value is set to 255, to ensure that the possible value for the red component is between 0 and 255.

   Then, the unique number `ID_SLIDER_R` (an integer constant) is assigned to this object, to be able to identify events that are triggered by this object later.

   Finally, the slider object is made visible, its position on the screen is set and it is added to the application container. This involves the slider in the event management system of the FATE framework and makes it able to react to user interaction automatically.

   On code line 29 the construction of a `CFLabel` object is shown. This label is responsible for showing the current value of the "red slider". Initially, it shows a value of 0, which is set programmatically on line 30. The following code lines are again responsible for visibility, positioning and adding the object to a parent container.

```
//-----------------------------------------------------------------
BOOL CPicasso::SliderChanged(DWORD dwSliderID, int iVal)
{
  char szBuff[128];  // send buffer

  switch(dwSliderID) {
  case ID_SLIDER_R:
    m_labelR->SetValue(iVal);
    m_iRedValue= iVal;
    break;

  case ID_SLIDER_G:
    m_labelG->SetValue(iVal);
    m_iGreenValue= iVal;
    break;

  case ID_SLIDER_B:
    m_labelB->SetValue(iVal);
    m_iBlueValue= iVal;
    break;
  }

  // send changes to message-server
  sprintf(szBuff, "car|setColor(%d, %d, %d)\n",
          m_iRedValue, m_iGreenValue, m_iBlueValue);
  m_pMsgServer->SendCall(szBuff);

  return(TRUE);
}
```

The code above shows how to implement an event handling method in a FATE application.

This method handles events triggered by `CFSlider` objects. The arguments passed to this method are the ID (`dwSliderID`) of the actual slider object, where the event was triggered, and the current value (`iVal`) of this slider.

Depending on the ID of the slider, the value is stored in the corresponding member variable for red, green or blue, and the label which displays this value gets updated.

Then, a string is assembled (e.g. "car|setColor(0, 98, 255)") and sent to the VR system via the `CFMsgServerProxy` object.
On the VR system, this string is received by the Ygdrasil software (see 2.3). This software searches for an object called "car" in the VR application and calls its "setColor" method with the specified arguments. This causes the actual change of the color of the 3D model.

```
//--------------------------------------------------------------------
BOOL CPicasso::KeyDown(PdaKey key)
{
  if (key == KeyOne) {
    m_pMsgServer->SendQuit();
    return(TRUE);
  }
  return(FALSE);  // event handling may continue
}
```

The `KeyDown` method is another example of event handling in a FATE application.
In this case, the event of a pressed key is handled. If the user presses the first key of the Palmist (see figure 4.2) the VR application receives a special "QUIT" command, that terminates the application.
This is done by calling the `SendQuit` method of the `CFMsgServerProxy` object.
The return value `TRUE` indicates that this "button pressed" event has successfully been handled and that the framework must not call any other interested listener objects for this event (event handling mechanism: see 3.3.3).

5.4 Evaluation of the Solution

The "Picasso" Application demonstrates the advantages and efficiency of the FATE framework in a simple way. With just a few lines of code, powerful applications can be developed. The framework provides the necessary components for creating interactive

applications without requiring special programming skills. If a traditional programming approach was chosen and this application had been designed and implemented from the scratch, the programming effort would have been significantly bigger. For example, network protocols would have to be specified, all communication mechanisms would have to be implemented, and the whole data exchange over network would have to be completely implemented by the programmer. Additionally, all these tasks would have to be performed for both the pocket PC as well as for the graphics workstation platform. For these purposes the FATE framework provides readily implemented software components, that can be integrated into new applications. The remaining task for programmers simply is to configure these components correctly and use them in their applications.

The "Picasso" application illustrates this concept in a demonstrative way. With FATE it is easily possible to assemble an application with which the user is able to navigate in a VR application and to interact with a 3D model in it. All data exchange, which enables this navigation and interaction, happens over a wireless network connection, even though programmers do not have to write a single line of networking code.

Experience has shown that certain navigation and interaction patterns occur repeatedly in VR applications and it is therefore possible to abstract from these. For this reason, these patterns can be covered by software components of a framework. FATE provides these components for programmers and shortens the development cycle of interactive VR applications, because developers do not have to "reinvent the wheel" each time they develop an application.

Additionally, FATE reduces the time that is spent for debugging such applications, for the reason that many framework components encapsulate critical and error prone pieces of code (like network code).

Chapter 6

Conclusions and Future Work

For conventional two-dimensional graphical user interfaces, common user interface elements like buttons, sliders or menus, have proven to be an effective technique for interaction. Utilizing human pointing abilities, this method provides clearly visible graphical components, with which all kinds of interaction tasks are trivial. These are desirable properties for user interfaces in any environment.
However, using a direct analogy from two-dimensional user interfaces to three dimensional virtual environments, the visibility of the user interface elements is often poor and interaction is difficult. Visibility problems are primarily due to the fact that user interface elements like menus do not always face the viewer. The user is free to move independently of them and item selection is largely related to the performance of a six-degrees-of-freedom pointing task. And furthermore, the difficulty of this task is compounded by tracker noise and imprecision in the devices being utilized.

To provide an alternative solution, different visions and possibilities have been shown, related to the approach of using a pocket PC as an interaction device for immersive virtual environments.
The great benefits and advantages of the *Palmist Project* are obvious. It offers wireless mobility for the user of a VR system, which increases the user's immersion in a positive

way.
Additionally, the 2D paradigm is much better for data presentation like values and long texts and increases the usability of virtual environments.
All in all, there are a bunch of new perspectives in interaction with VR devices and VR applications. But besides that, the Palmist is also an "intelligent" device, which means that it is a computer which can be expanded in its capabilities and have all the functionality, that ever can be realized by software.
There are, however, some drawbacks and limitations to this approach which already have been discussed in [5][18]. But in general, empirical studies [16][30] and research efforts have proven that the use of a 2D interaction metaphor, especially the use of hand-held windows [30][16] or pocket PCs [5][18] as interfaces, increases productivity in immersive virtual environments and finds acceptance of the users.

This current status of the *Palmist Project* and its software framework FATE provides a fair basis for application development as well as a set of instruments for advanced interaction techniques in virtual environments.
Nevertheless, several aspects of possible improvement have been identified and following functionality and features are suggested to be added:

- Linux port:

  An open item, which is currently in progress, is to port the FATE framework to Linux platforms. This operation is still at an early stage but the core system as well as some of the framework's classes have already been compiled on desktop Linux distributions. The target platform however, is a Linux distribution that is running on pocket PCs.

- Scene graph management:

  As most VR systems are based on scene graph APIs like Performer or OpenSG, it would be an extremely useful improvement to FATE if it was given the possi-

bility to transfer the application's scene graph to the PDA. On the PDA then, attributes of the scene graph could be changed, scene graph nodes could be added or deleted and these changes could immediately be communicated back to the VR application. The user would experience an immediate feedback on the changes made to the scene graph.

This extension to the framework would be a big improvement in prototyping situations and the Palmist could be used as a highly sophisticated design-on-the-fly tool.

- PalmOS port:

  Currently, FATE was only designed for pocket PCs running a Windows CE operating system but porting it to PalmOS PDAs would be a promising way to continue this project as soon as the other open items are finished successfully.

Bibliography

[1] Sutherland I. E.:, "The Ultimate Display", Proceedings of IFIPS Congress 1965, Vol. 2: pp. 506-508, 1965.

[2] Pomberger G., Blaschek G.:, "Object Orientation and Prototyping in Software Engineering", Prentice Hall (Translation of: "Grundlagen des Software-Engineering - Prototyping und objektorientierte Software-Entwicklung", Carl Hanser Verlag): page 34, 1996.

[3] Hoertner H., Lindinger C., Praxmarer R., Riedler A.: "ARSBOX with Palmist - Advanced VR-System based on commodity hardware", ACM SIGGRAPH 2002, Emerging Technologies: page 64, 2002.

[4] Foley J.D., Wallace V.L., Chan P., "The Human Factors of Computer Graphics Interaction Techniques", IEEE Computer Graphics & Applications: pp. 13-47, 1984.

[5] Watsen K., Darken R., Capps M.: "A Handheld Computer as an Interaction Device to a Virtual Environment", Proceedings of the Third Immersive Projection Technology Workshop, 1999.

[6] Harold, Eliotte Rusty: "The XML-Bible", IDG Books, 1999.

[7] Bray T., Paoli J., Sperberg-McQueen C. M., Maler E.: "Extensible Markup Language (XML) 1.0 (Second Edition)", available from the World Wide Web: http://www.w3.org/TR/REC-xml#dt-doctype, 2000.

[8] "Pape D., Anstey J., Dolinsky M., Dambikc E.J., "Ygdrasil - a framework for composing shared virtual worlds", Future Generation Computer Systems, Vol. 19, pp. 1041-1049, 2003.

[9] Kruglinski D., Shepherd G., Wingo S.:, "Programming Microsoft Visual C++ Fifth Edition", Microsoft Press, 1998.

[10] Bowman D., Kruijff E., LaViola J., Poupyrev I.: "An Introduction to 3-D User Interface Design", PRESENCE: Teleoperators and Virtual Environments, 10(1): pp. 96-108, 2001.

[11] Angus I., Sowizral H.: "Embedding the 2D Interaction Metaphor in a Real 3D Virtual Environment", Stereoscopic Displays and Virtual Reality Systems, SPIE 2409: pp. 282-293, 1995.

[12] Wloka M.M, Greenfield E.: "The Virtual Tricorder", Technical Reports CS-95-05, Department of Computer Science, Brown University, Providence RI, USA, 1995.

[13] Serra L., Hern N., Guan C. G., Lee E., Lee Y. H., Tasi Y. T., Chan C. and Kockro R. A.: "An Interface For Precise and Comfortable 3D Work With Volumetric Medical Datasets", Proceedings of Medicine Meets Virtual Reality, San Francisco, 1999.

[14] Rorke M., Bangay S., Wentworth E.: "Virtual Reality Interaction Techniques", Proceedings of the first Annual South African Telecommunication Networks and Application Conference (SATNAC), Cape Town, South Africa: pp. 526-532, 1998.

[15] Cutler L., Frohlich B., Hanrahan P.: "Tow-handed direct manipulation on the responsive workbench", Symposium on Interactive 3D Graphics, 1997.

[16] Lindeman R., Sibert J. and Hahn J.: "Hand-Held Windows. Towards Effective 2D Interaction in Immersive Virtual Environments", IEEE Virtual Reality, 1999.

[17] Watsen K., Darken R., Capps M.: "A handheld computer as an interaction device to a virtual environment", Proceedings of the Third Immersive Projection Technology Workshop, 1999.

[18] Hill L., Cruz-Neira C.: "Palmtop Interaction Methods for the Immersive Projection Technology VR Systems", 4th Immersive Projection Technology Workshop, Ames, Iowa, 2000.

[19] Burdea G., Coiffet P.: "Virtual Reality Technology", John Wiley & Sons, Inc., 1994.

[20] Sherman B., Craig A.B.: "Literacy in Virtual Reality: a new medium", Computer Graphics, Vol. 29, No. 4, ACM Press, 1995.

[21] Cruz-Neira C., Sandin D.J., DeFanti T.A.: "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE", SIGGRAPH '93 Proceeding, Vol. 27, pp. 135-143, ACM Press, 1993.

[22] Slater M., Usoh M.: "Modelling in Immersive Virtual Environments: A Case Study for the Science of VR", Virtual Reality Applications, Academic Press, UK, 1995.

[23] Pimentel K., Teixeira K.: "Virtual reality: Through the new looking glass", Windcrest / McGraw-Hill, 1994.

[24] Ellis S.R.: "What are Virtual Environments?", IEEE Computer Graphics and Applications, IEEE, 1994.

[25] Pape D., Cruz-Neira C., Czernuszenko M.: "CAVE User's Guide", EVL, Chicago 1998.

[26] Cruz-Neira C., Sandin D.J., DeFanti T.A, Kenyon R.V., Hart J.C.: "The CAVE: Audio Visual Experience Automatic Virtual Environment", Communication of ACM, Vol. 35, No. 6, ACM Press, 1992.

[27] Kalawsky R. S.: "The Science of Virtual Reality and Virtual Environments", Addison Wesley, Reading Massachusatts, USA, 1993.

[28] Bolas M. T.: "Human Factors in Design of Immersive Displays", IEEE Computer Graphics and Applications: pp. 55-59, 1994.

[29] Zelle J. M., Figura C.: "Simple, Low-Cost Stereographics: VR for Everyone", SIGCSE04, Norfolk, Virginia, USA, 2004.

[30] Rorke M., Bangay S.: "The Virtual Remote Control - An Extensible, Virtual Reality, User Interface Device".

[31] Robert W., Lindeman R., Sibert J. L., Hahn J. K.: "Towards Usable VR; An Empirical Study of User Interfaces for Immersive Virtual Environments.", CHI 99 Proceedings, pp. 64-71.

[32] Jacoby R.H., Ellis, S.R.: "Using Virtual Menus in a Virtual Environment", Proceedings of the SPIE Technical Conference 1666, 1992.

[33] Fitzmaurice G. W., Zhai S., Chignell M. H.: "Virtual Reality for Palmtop Computers", ACM Transactions on Information Systems, Vol. 11, No. 3: pp. 197-218, 1993.

[34] Rumbaugh J., Jacobson I., Booch G.: "The Unified Modeling Language Reference Manual", Addison-Wesley, 1998.

[35] "GLUT - The OpenGL Utility Toolkit", http://www.opengl.org/resources/libraries/glut.html.

[36] "TRACKD", http://www.vrco.com/products/trackd/trackd.html.

[37] Rohlf J., Helman J.: "IRIS performer: a high performance multiprocessing toolkit for real-time 3D graphics", Proceedings of the SIGGRAPH'94 Computer Graphics Conference, pp. 381-395, 1994.

[38] Pape D., Cruz-Neira C., Czernuszenko M.: "CAVE: User's Guide", EVL, Chicago.

[39] Park K., Cho Y., Krishnaprasad N., Scharver C., Lewis M., Leigh J., Johnson A.: "CAVERNsoft G2: a toolkit for high performance tele-immersive collaboration", Proceedings of the ACM Symposium on Virtual Reality Software and Technology, pp. 8-15, 2000.

[40] Bullinger H., Breining R., Bauer W.: "Virtual Prototyping State of the Art in Product Design", Faunhofer Institute for Industrial Engineering.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

St. Oswald, am 17. Juni 2004 Unterschrift:

Lebenslauf

Persönliche Daten:

Name	Wolfgang Ziegler
Adresse	Promenade 12 A – 4271 St. Oswald b. Fr.
Telefon	+43 650 8415432
Geburtsdatum	10.03.1980
Geburtsort	Linz
Staatsbürgerschaft	Österreich
Glaubensbekenntnis	röm.-kath.
Familienstand	ledig

Schulbildung:

1986 – 1990	Volksschule St. Oswald b. Fr.
1990 – 1998	BG Freistadt (Abschluss Matura)
1999 – 2004	„Diplomstudium Informatik" an der Johannes Kepler Universität Linz (Abschluss mit Dipl. Ing.)

Berufspraxis:

Juni 2001 - September 2001	Werksvertragliche Tätigkeit im Ars Electronica Futurelab http://www.aec.at/en/futurelab/future_office_table.asp
Februar 2001 - April 2001	Werksvertragliche Tätigkeit im Ars Electronica Futurelab http://www.aec.at/en/futurelab/projects_sub.asp?iProjectID=2855
Mai 2002 - Oktober 2003	Freier Dienstnehmer im Ars Electronica Futurelab http://www.aec.at/en/futurelab/team.asp
November 2003	Anstellung im Ars Electronica Futurelab
Februar 2004 - Juni 2004	Diplomarbeit mit dem Titel „The Palmist Project. A new Approach of Interaction in Virtual Environments" als Auftragsarbeit für das Ars Electronica Futurelab

Besondere Kenntnisse und Fähigkeiten:

Fachkenntnisse Informatik	System- und Anwendungsprogammierung für die Plattformen Windows 2000/XP, Windows CE, Linux Programmiersprachen: C/C++, C#, JAVA, Visual Basic (.NET), Delphi, x86 Assembler, ARM Assembler APIs und Frameworks: MFC, WIN32 API, .NET Klassenbibliothek, Qt, COM, DCOM DirectX (speziell DirectShow und Direct3D) OpenGL, OpenGL Performer, OpenSG OpenCV, Intel IPP
Sprachen	Deutsch (Muttersprache) Englisch (verhandlungsfähig) Spanisch (Grundkenntnisse)
Sonstige Fähigkeiten	Führerschein Klasse „B“ Abgeleisteter Zivildienst (01.10.1998 - 30.09.1999)
Interessen	Ich bin ehrgeizig und lernbereit. Ich bin ein kommunikativer Mensch und arbeite daher gerne mit anderen im Team zusammen.

St. Oswald, 17.06.2004