THE AR WORKBENCH: A COMPLETE COLOCATED REACH-IN MIRROR-BASED SEE-THROUGH AUGMENTED REALITY SOLUTION

A Master of Science thesis by

Marlon Richert
(MKE 9692077)

Friday, 23 February, 2007

MSc program of Media & Knowledge Engineering

Advisor: dr. ir. C.A.P.G. van der Mast

MMI Group
Department of Mediamatics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
The Netherlands
The AR Workbench: A Complete Colocated Reach-in Mirror-based See-through Augmented Reality Solution
Marlon M. Richert (MKE 9692077)

Thesis for Master of Science program of Media & Knowledge Engineering

Advisor:
dr. ir. C.A.P.G. van der Mast

Thesis Committee:
prof. dr. M.A. Neerincx
dr. ir. C.A.P.G. van der Mast
drs. P.R. van Nieuwenhuizen
J.A. Lozano Quilis, MSc, PhD

MMI group
Department of Mediamatics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
The Netherlands

ABSTRACT

The Medical Imaging and Computing Laboratory (MedICLab) of the Universidad Politécnica de Valencia (UPV) is the developer of many virtual reality (VR) solutions for use in many kinds of therapy, including a system for stimulating motor imagery in hemiplegics, called the VR Mirror. They are interested in creating augmented reality (AR) versions of many of their existing VR systems, as well as creating completely new AR therapy systems. To do this, the MedICLab requires a colocated reach-in mirror-based see-through AR framework on which they can base all of their future AR applications. In addition, the system will need to be inexpensive, yet complete, and be noninvasive and intuitive to the user. Furthermore, it should require minimal domain knowledge from the application programmer and not be tied to a particular development platform. This document describes an attempt to develop such a system, as part of the author’s Master of Science thesis project at the MedICLab, and presents an in-depth discussion of the results as well as making recommendations for the future.
This is my Master’s thesis. It describes the thesis project I worked on at the MedICLab of the Universidad Politécnica de Valencia in Spain. I did this as part of the Master of Science program of Media & Knowledge Engineering at Delft University of Technology in the Netherlands. The project consisted of creating a colocated reach-in mirror-based see-through augmented reality (AR) solution with which the MedICLab will be able to create AR versions of some of their existing virtual reality applications, as well realize new ideas of AR projects they have. It has been a long and arduous road and many cultural differences and language barriers have blocked my path, but I have arrived.

Some acknowledgements are in order. I would like to thank Charles van der Mast for being my advisor; we may have gotten off on a rocky start, but his guidance proved invaluable in the end. I would like to thank all of the wonderful people at MedICLab, especially José Lozano and Javier Montesa, for their hospitality and assistance; MedICLab also provided me with those fine photographs of the workbench that you may find in this thesis. I would like to thank Ivo Timmermans for proofreading the near-final manuscript and giving me invaluable feedback. I would like to thank Kirsi Virtanen en Jenni Leminen for being such good friends; without their help and support, I would have never been able to finish this thesis. Finally, I would like to thank myself for finally setting me free of the educational system. Thank you.

—Marlon Richert, Rijswijk ZH, the Netherlands, February 2007
# CONTENTS

Abstract ............................................................................................................ ii  
Preface ............................................................................................................. iii  
Contents ........................................................................................................... v  
1 Introduction ................................................................................................. 1  
2 Theoretical Background ................................................................................. 3  
   2.1 Mirror-Based See-Through Augmented Reality ........................................... 3  
   2.2 The Treatment of Hemiplegia ................................................................ 4  
      2.2.1 Motor Imagery ................................................................................. 4  
      2.2.2 Mental Practice ................................................................................ 5  
      2.2.3 Post-Stroke Rehabilitation ................................................................. 6  
   2.3 The Virtual Reality Mirror ........................................................................ 7  
   2.4 Implications ........................................................................................... 7  
3 Methods & Techniques .................................................................................. 9  
   3.1 IPF ...................................................................................................... 10  
4 Requirements Analysis ................................................................................ 13  
   4.1 Scenarios ............................................................................................. 13  
      4.1.1 Problem Scenarios .......................................................................... 13  
      4.1.2 Visionary Scenarios ........................................................................ 15  
   4.2 System Models ..................................................................................... 16  
   4.3 Requirements ....................................................................................... 17  
      4.3.1 Functional Requirements ................................................................ 17  
      4.3.2 Nonfunctional Requirements ........................................................... 17  
5 Design ....................................................................................................... 19  
   5.1 User Analysis ....................................................................................... 19  
   5.2 Conceptual Design ................................................................................ 21  
   5.3 Semantic Design .................................................................................. 21  
   5.4 Syntactic Design ................................................................................... 23  
   5.5 Lexical Design ...................................................................................... 25  
6 Implementation .......................................................................................... 29  
   6.1 Hardware ............................................................................................ 29  
   6.2 Software .............................................................................................. 31  
7 Evaluation & Results ................................................................................... 41  
8 Conclusion ................................................................................................. 43  
References ...................................................................................................... 47  
Appendix: Application Programming Interface .................................................... 51
1 Introduction

The Medical Imaging & Computing Laboratory (MedICLab) at the Center for Research and Innovation in Bioengineering (C2B or Centro de Investigación e Innovación en Bioingeniería) of the Universidad Politécnica de Valencia (UPV) in Spain has developed and is developing several virtual therapy tools—virtual reality (VR) tools for use in various kinds of therapy. One such tool is the Book of Life, intended for helping patients dealing with traumatizing experiences by letting them create a visual story out of key periods of their life using symbolic elements, under the guidance of a therapist. Another example is the Virtual Reality Mirror, a system for stimulating brain plasticity in hemiplegics, while minimizing the need for therapeutic guidance. Ideas for other virtual therapy tools are underway. One such idea, for example, is a food therapy tool in which the patient virtually prepares and eats food, in order to prepare them for doing so in the real world.

Right now, most of the MedICLab’s virtual reality tools use a head-mounted display (HMD) for stereoscopic viewing and a gamepad for user input. One problem with these systems is HMDs are quite invasive and can be uncomfortable to use, while a gamepad is not a very intuitive way of control for most users, especially those unfamiliar with them. The Virtual Reality Mirror is an exception to this as it does not require an HMD and users motion trackers instead of a gamepad, but this system does not provide stereoscopy and only limited immersion.

The MedICLab develops all of its VR applications—including systems other than virtual therapy tools, such as surgical simulations and computer-assisted medical intervention tools—using a proprietary third-party development tool called IPF, originally intended for creating virtual TV studio sets. The MedICLab has invested a considerable amount of finances and human resources in IPF. Unfortunately, much of IPF’s functionality uses a non-standard paradigm and has only poor documentation, making it hard for newcomers to MedICLab to master working with the tool. Additionally, although IPF is very suitable for creating static virtual environments, because of its origins, it has only very poor support for dynamic interactions. Instead, its main interaction focus lies mostly in managing simple buttons and triggers—such as those required for live TV coverage of national elections, for example—not the kind of complex interactions required by direct manipulation in virtual reality. Thus, creating these interactions often proves complex and time-consuming.

Finally, the MedICLab would like to move most of their virtual reality projects and ideas to augmented reality (AR). One reason for this is that they want to explore whether it increases immersion, which is important for many of their tools. Another reason is that, with AR, the patient can still observe the real world, including the therapist, making it a more comfortable and less alien experience. The MedICLab evaluated several AR approaches and decided that their preferred choice of AR would be mirror-based see-through AR. The main reasons for this are that it is noninvasive and easily allows for colocated reach-in interaction. Additionally, it has the potential to be relatively inexpensive, which is a requirement because they do not have a large budget. Finally, this kind of solution is the most suitable situation for most of their projects, with the Virtual Reality Mirror in particular.

Thus, the research question of this thesis is as follows: Is it possible to develop an inexpensive yet complete colocated reach-in mirror-based see-through augmented reality solution that is noninvasive and intuitive to the user, but requires minimal domain knowledge from the application programmer and is not tied to a particular
development platform? In other words, we would like to have the following: The AR system should be cheap and come with both hardware and software. It should restrict the user’s movements as little as possible and should be easy for them to use without prior knowledge or experience. Finally, it should require less knowledge of IPF to program than is necessary now and it should be possible to use it in the future with a development platform other than IPF without requiring a complete code rewrite.

This thesis discusses the development of just such a system, which I worked on at the MedICLab from September 2005 to June 2006. What you are reading now is chapter 1 and forms the introduction. The next chapter discusses the theoretical background, which includes a summary of the research assignment I did on the possibility of using AR to improve treatment of hemiplegia at the TU Delft preceding my stay in Valencia, Spain. Chapter 3 discusses the methods and techniques I used in the project, including an analysis of the IPF tool. Chapter 4 discusses the requirements of the project and includes a discussion on the projects and ideas that MedICLab would like to implement in AR. Chapter 5 details the design of the AR system, including a user analysis and design rationales. Chapter 6 discusses the implementation of the AR system and shows some examples of the source code. Chapter 7 evaluates the results of the project. Finally, the last chapter forms the conclusion.
2 Theoretical Background

Before starting on the project in Valencia, I did a research assignment on mirror-based see-through AR and how to best use it to create an AR version of the VR Mirror [31]. This chapter provides a summary of this research and shows how it relates to the project described in this thesis. First, it discusses mirror-based see-through AR. Then, it details the treatment of hemiplegia as it relates to the VR Mirror and the new AR system, followed by the specifics of the VR Mirror. Finally, it lays out the implications for the proposed system.

2.1 Mirror-Based See-Through Augmented Reality

With mirror-based AR, the user watches the reflection of a screen on a mirror, thus perceiving virtual objects displayed on the screen to be in fact located behind the mirror. For the system to be a see-through AR system, the mirror must be half-silvered and not fully opaque, so that the user can both observe the reflections of the virtual objects and see through the mirror to observe the real world behind it [5, 25]. In a colocated reach-in AR system, the user will be able to reach underneath this semitransparent mirror (reach-in) in order to interact with both virtual and real objects—the latter including the user’s hands—inhabiting the same space (colocated). Currently, two colocated mirror-based see-through AR systems are available that either support reach-in or seem to be suitable for modification to support reach-in: the Virtual Showcase and the Personal Space Station.

The basic design of the Virtual Showcase consists of a convex assemblage of semitransparent mirrors on top of a projection screen, or vice versa [5]. The mirrors reflect the objects displayed on the projection screen in such a way that viewers perceive these objects to be colocated with real objects inside the showcase. Shutter glasses and motion tracking provide the viewers with a view-dependent, stereoscopic perspective of the virtual objects, while a controllable video projector illuminates the real objects inside the showcase on a per-pixel basis [28]. The configuration of the Virtual Showcase in which the projection screen lies above the mirrors seems a more appropriate prototype to adapt for use in the computer-aided rehabilitation of upper-extremity hemiplegia (see 2.2), since this removes the projection screen from patients’ field of view when they are observing the virtual representation of their arm.

With the Personal Space Station, the user sits in front of a semitransparent mirror that reflects virtual objects displayed by a CRT screen above it [25]. Either a pair of shutter glasses or the combination of a polarization screen and polarized glasses provides the user with stereoscopic viewing. Two cameras monitor the area underneath the mirror, so that the user can reach there and interact with the virtual objects. The cameras are equipped with an infrared (IR) pass filter and IR LEDs around the lens, so that they can also track real objects marked with retro-reflectors that the user handles underneath the mirror.

Mirror-based see-through AR has several advantages over using HMDs for this project [4, 5, 25]. First, using them is more ergonomic than using HMDs because the glasses used are lighter and less obtrusive. Second, since the focal distance to the virtual content is closer to its perceived distance than with HMDs, it provides the user with better convergence and accommodation of the eyes. Third, they are cheaper to construct than HMD systems. Finally, they can more easily colocate real objects with virtual ones.
Of course, the use of mirror-based AR also involves some trade-offs. First, unlike with HMDs, the user does not have a perpendicular view of the screen, so the system must transform all graphics so that the user does not perceive any distortion. If the rendering is to be stereoscopic or view-dependent, these transformations increase in complexity [5]. Second, occlusion artifacts may occur between real and virtual objects, because all virtual objects will seem transparent with respect to the real world, unless the system has a mechanism to compensate for this explicitly, which requires additional hardware and computational processing [25].

2.2 The Treatment of Hemiplegia

In the United States alone, each year more than 700,000 people suffer a stroke [26]. Nowadays, less than a quarter of these incidents result in death, but this then leaves almost 540,000 Americans per year in some state of stroke-related disability and requiring medical attention. Even after initial medical care, about two thirds of these have lingering neural deficits that continue to impair functioning in their daily lives [40]. In 2005, American society had to pay approximately 58 billion dollars for stroke-related medical costs and disabilities [2].

The most common disability resulting from stroke is paralysis [26]. Usually, this occurs only on one side of the body, a condition known as hemiplegia. It may affect either the whole side or just a part of it, such as an arm, a leg or half of the face. In particular, upper extremity hemiplegia (UEH) can seriously hinder patients’ independent functioning, as it makes them unable to do such everyday things like grasping objects. It is, unfortunately, a very common ailment, since only five percent of adults with UEH ever fully regain the functioning of their arm, while as much as 20% never regain any functional use at all.

One avenue of approach that research in post-stroke rehabilitation is currently exploring is the stimulation of brain plasticity through movement therapy, by forcing the patient to use their affected limbs [26]. In particular, using mental practice to augment physical practice has proven to be very fruitful and research in this area has come up with convincing results [37].

Unfortunately, many patients with severe hemiplegia not only have difficulty with physical practice, but with mental practice as well, because they are impaired in producing the required motor imagery [18]. Thus, several researchers have developed different means of providing computer-aided mental practice. The MedICLab’s VR Mirror is one of these. To understand how and why the VR Mirror aids the treatment of hemiplegia and to see how developing an AR version might improve on this, it is important to take a closer look at the topics of motor imagery, mental practice and post-stroke rehabilitation.

2.2.1 Motor Imagery

Imagery is the human ability to generate mental representations of perceptual and motor events spontaneously and at will. Human beings can call upon several types of imagery in their minds. Mental imagery is one type. This is what happens when someone actively relives an earlier sensation while the stimulus that caused it is no longer present. Another form of imagery is movement imagery, which involves someone imagining a person or object in motion. Motor imagery is then a specific kind of movement imagery in which the subject imagines moving their own body without performing any actual physical movements [9, 20].

Motor imagery should not be confused with visual imagery. This involves mentally manipulating visual images, such as imagining scenes in which people or objects are
moving. This might seem similar to motor imagery. The difference, however, is that while in visual imagery we imagine movement in our environment, in motor imagery we imagine the kinesthetic sensation involved in moving our own bodies [21].

Another important distinction made by several researchers is that between internal and external imagery [11]. Internal imagery on the one hand uses a first-person perspective and relies on motor-kinesthetic information processing. External imagery on the other hand uses a third-person perspective and relies on visuospatial processing [8]. Only internal imagery qualifies as motor imagery. Motor imagery represents the self in action and makes the subject feel as if they are executing the action, be it with their whole body or just part of it [24]. Therefore, motor imagery requires a representation of the body generating the action and not of the effects that these actions have on the world outside [21]. This obviously excludes external imagery from being motor imagery. This also has implications for the development of AR version of the VR mirror in that, to represent motor imagery in the most accurate way, it has to show it from as close as possible to a first-person perspective. Compared to the VR Mirror, an AR version should be able to do so more effectively, because with the latter, the virtual representation will be located spatially close to where the user’s actual arm would be located, thus providing an experience closer to a first-person perspective than the VR mirror can provide.

Finally, two key properties of motor imagery are of particular importance when trying to invoke motor imagery in subjects. First, motor images correspond to generalized movements and not specific muscular actions [3]. Second, it appears that to evoke motor imagery in subjects, using correct spatial movement is of the utmost importance, while providing detailed visuals seems to have no impact whatsoever [22, 29].

### 2.2.2 Mental Practice

Also known as mental or symbolic rehearsal, mental practice is a training method in which a subject intensively and repeatedly rehearses the internal mental reproduction of a specific action with the aim of improving performance. Mental practice usually involves motor imagery, but is not limited to it and may involve other kinds of cognitive processes as well [20]. Researchers usually measure the effectiveness of this method by comparing the improvement in speed or accuracy gained with the results of using physical exercise to practice the same actions.

It is interesting to note that mental practice can result in actual increases in strength [43]. It appears that these strength gains are the result of neural changes in motor planning [7, 21]. There is also considerable evidence that mental practice can have a significant positive effect on performance in learning motor skills [3]. Both behavioral and neurophysiological studies have reported significant positive transfer from mental to physical practice [8, 11, 15]. Mental practice thus not only appears to be capable of producing changes in the brain similar to those caused by physical practice, but actually seems to augment the efficiency of any ensuing physical practice. In fact, many studies have reported that a combination of both is more efficient than either is alone. In particular, adding even only a very small amount of physical practice to mental practice—say, a 10:1 ratio—already greatly increases the efficiency of the practice regime beyond that which can be achieved using one form of practice alone [20].

Besides measuring the effectiveness of mental practice, researchers can also rate the intensity of the motor imagery used in it by measuring the magnitude of the subject’s autonomic responses, such as changes in heart and breathing rate or skin resistance and potential [3]. It turns out that the more intensive the motor imagery
that subjects generate, the more effective their mental practice is [33]. Apparently, access to motor imagery that is more intensive leads to bigger improvements in motor skill learning. Therefore, to maximize the effectiveness of mental practice, it clearly is important to stimulate the use of intensive motor imagery as much as possible [20]. The reported vividness of motor imagery, however, does not appear to correlate with the effectiveness of mental practice. Vividness is very important for visual imagery, but not for motor imagery, for which being able to control what you are imagining is much more important.

2.2.3 Post-Stroke Rehabilitation

Post-stroke rehabilitation involves helping patients relearn skills they lost due to stroke. Well-focused, repetitive practice is the most important ingredient for post-stroke rehabilitation, experts agree [26]. Since many post-stroke patients are hemiplegic, initial treatment immediately focuses on promoting independent movement using passive or active practice. The difference between these two types of practice is that in passive practice the therapist actively assists the patient in moving impaired limbs in nonspecific ways, while in active practice the patient performs task-specific activities without receiving any physical help. Post-stroke rehabilitation generally focuses on goal-directed tasks, since they promote coordination.

Traditionally, post-stroke rehabilitation has used repeated physical practice, consisting mostly of compensatory training of the healthy arm and passive practice of the disabled arm to improving motor activity and stimulate brain plasticity [26]. A problem with this approach is that its effectiveness depends on the performance of the disabled limb [37]. In addition, since spontaneous recovery from stroke shows a plateau after three months, traditional stroke rehabilitation only focuses on those first three months [40].

Using active practice through forced use of the impaired arm, researchers have demonstrated that stroke patients can achieve functional gains and brain plasticity long after the three-month spontaneous recovery period [1]. Through the forced use of the affected arm, patients can significantly increase the functional ability and daily use of their weakened arm [40]. Auditory cueing can further enhance active practice, allowing forced-use task-oriented repetitive training to treat successfully much more severe cases of stroke than would otherwise be possible [38].

Mental practice is a form of active practice that is a cost-efficient and effective means of assisting motor recovery in stroke patients [17]. Indeed, mentally practicing a motor skill can result in the same kind of brain plasticity that would occur because of physically acquiring the skill [10, 13, 20, 42]. To stimulate the production of motor imagery involving the disabled limb in patients with UEH, therapists can use either video or a mirror box [37]. In the first case, patients watch a computer-generated movie showing a moving arm from varying angles and at varying speeds, after which they immediately have to perform the same movement mentally. When they have completed the imagined movement, they have to tap the space bar, enabling the therapist to obtain the time used to complete each movement. In the second case, patients sit at a table with a vertical mirror positioned in front of them in such a way that the reflection of the healthy arm visually replaces the disabled arm. The therapist instructs the patients to imagine that the reflection is in fact their disabled limb, moving around freely. Patients spent the first week learning to do this, after which they have to perform relatively simple exercises in the next two weeks, and complex maneuvers in the final week. At the end of the four weeks, patients will have made significant improvements in both
accuracy and speed of movements. Patients obtain the greatest functional gains during the period of intervention, indicating that these advances are the result of active mental practice.

Because of the lack of direct feedback, mental practice is still less effective than physical practice [14, 17, 30]. Combined with physical practice, however, subjects can improve their motor imagery by using the feedback gained from actual movements. In this way, mental practice enables subjects to increase the number of repetitions of a movement they can make without any increase in physical demands [20]. Further, even if patients are not able to physically produce any movement in their disabled limbs, they can still benefit from mentally practicing specific skills with motor imagery, because it rehearses the neural network involved, thus priming it for the future execution of these motor skills [27]. Patients have to be able, however, to produce motor imagery at all. Without it, mental practice is simply not very effective.

2.3 The Virtual Reality Mirror

As mentioned above, many patients with severe hemiplegia have difficulty to produce motor imagery [18]. Computer-aided mental practice can assist these patients by interactively stimulating them to generate motor imagery. The MedICLab—in collaboration with the Applied Technology for Neuro-Psychology Lab of the Istituto Auxologico Italiano in Milan, Italy—has developed just such a system, which makes use of VR to assist the rehabilitation of patients with UEH [17]. This system consists of two parts. The first part is composed of motion tracking equipment to track the movements of the patient’s healthy arm as they perform an exercise. The second part is a desk with a screen embedded into it, dubbed the VR Mirror, that displays a three-dimensional (though not stereoscopic) animation of a movement the patient performed earlier, but mirrored so that it appears to involve the disabled arm, with the intent to stimulate the production of motor imagery (see section 2.2).

The system has several claimed advantages for post-stroke rehabilitation [17]. First, providing the patient with a combination of visual and auditory cues makes it easier for the patient to produce the right kind of motor imagery for effective mental practice. Second, the system does not rely on pre-programmed exercises, which is a widely known drawback of traditional computer-aided post-stroke rehabilitation. Third, the external display keeps the patient free from having to wear any awkward or obtrusive equipment, such as a head-mounted display. Fourth, by keeping track of the durations of mental movements with the impaired arm, the system can dynamically keep the therapist updated on the patient’s performance and give the patient real-time feedback on their performance, which promotes effective motor learning [34, 35]. Finally, by combining physical and mental practice and using interactive technology to reduce the patient’s need for skilled support, the system provides a very efficient solution for post-stroke rehabilitation.

2.4 Implications

Since motor images correspond to generalized movements and not specific muscular actions and when evoking motor imagery in subjects, as mentioned in section 2.2, using correct spatial movement is more important than providing detailed visuals. For the development of an AR version of the VR Mirror, this all implies that the screen resolution does not need to be that high and that the 3D models and textures are not required to be that detailed. This means that relatively low-end graphics hardware should be sufficient. It also means that having stereoscopy will improve
the system, as it makes the observed movements more spatially correct regardless of the angle from which you watch them and thus be more effective at stimulating motor imagery. A version of the VR mirror with stereoscopy will be an improvement over the original, as the latter does not provide this.

Further, as mentioned in the same section, to maximize the effectiveness of mental practice, it clearly is important to stimulate the use of intensive motor imagery as much as possible, while reported vividness of motor imagery, however, does not appear to correlate with the effectiveness of mental practice. This correlates with the conclusion in the previous paragraph, implying that an AR version of the VR Mirror should be able to aid mental practice more effectively than the original, while not requiring expensive graphics hardware or detailed modeling.

Finally, an AR version of the VR Mirror will effectively combine the best of both the mirror box and the video approach to post-stroke rehabilitation. Firstly, it will create a virtual version of the mirror box, but without requiring a training period, since the patient does not have to move the healthy arm at the same time, making it easier to focus on the virtual arm. Secondly, it will be able to provide the same timing feedback that the video approach does.
3 Methods & Techniques

The project development discussed in this thesis followed the human-computer interaction design process that is taught as a standard part of the Media & Knowledge Engineering program, as shown in Figure 1. Additionally, the project made use of aspects from scenario-based usability engineering as laid out by Rosson & Carroll [32], combined with object-oriented software engineering practices as suggested by Bruegge & Dutoit [6] and requirements engineering techniques as proposed by Kotonya & Sommerville [23].

![Diagram of human–computer interaction design process](image)

**Figure 1** Human–computer interaction design process [39]

Prototyping of the augmented reality workbench was done using 3D models, created in IPF (see 3.1) and 3D Studio Max. Prototyping of the software in absence of a finished workbench was done using an ISOTRAK II and a regular, non-stereoscopic CRT screen.

All software-modeling diagrams in this thesis use UML (unified modeling language). They were all created with the help of the freeware MagicDraw CE application. The software of the system discussed in this thesis was implemented using Python ([http://www.python.org/](http://www.python.org/)), a high-level programming language supporting functional, imperative and object-oriented programming paradigms with a strong but dynamic type system.
3.1 IPF

MedICLab develops all of its virtual reality application using IPF, a development tool created by the company Brainstorm Multimedia (http://www.brainstorm.es/) originally intended for developing virtual sets for TV shows. It is commercially available as eStudio and M3. A developer can create projects in IPF using both the GUI and by writing Python code, which then runs inside IPF itself.

Almost all of the Python classes that IPF provides for the developer’s code to interact with live in the ipf module. A select few classes, such as vector, live outside the ipf module in the default namespace. Figure 2 gives a partial class diagram of the ipf module, focusing on only those classes that are important to this project. Note that the class hierarchy presented in Figure 2 is not explicitly present in IPF as such. Rather, I have inferred it by analysis to give a better insight into the workings of the software described in this thesis.

Nearly every object in IPF is an Item in a List, indexed by strings that are the apparent names of the objects. As such, a List is more like a Python dict, rather than a Python list, except that a List does maintain a specified order between its items, while a dict does not.

The Obj (for object) class represents the virtual objects in IPF’s coordinate system. A Pivot is a single point in this system, but is different from a vector in that it not only has a displacement attribute DIS, but also an orientation attribute EUL (for Euler), which is a vector containing heading, pitch and roll. A Cond (for condition) checks whether a Pivot is inside a given Region, which it reflects in its STATE attribute. A Region is a bounded area in the virtual space. A Region can be one of several predefined shapes, but all of these, with the exception of a sphere, extend infinitely along one axis, thus making them generally unsuitable as bounding boxes.

![Figure 2](image-url)  
**Figure 2**  
Object analysis of relevant parts of IPF (class diagram).
with them. Thus, all Regions used by the software discussed in this thesis are limited to spheres only.

An object attribute in IPF can be either a reference to an item or a value. You can use a Bind object to turn a value attribute into a reference attribute. Each frame that IPF tells an active bind to update, it reads a value from its FUNCTION_ITEM and passes it, along with a reference to its ITEM_MODIFIED, to its EDITOR_MODIFIED. This Editor then actually updates the given item using the given value.

A Map is an object that, each time IPF draws a frame, maps one or more input values into an output value. For example, a FloatMultiplyMap takes two floats as input and gives their product as output. Instances of VoidPythonMap are an exception to this rule: They do not actually take any input variable but rather are told by a ButtonEditor when to execute their code.

Note that in IPF itself, subclasses of Bind, Editor and Map do not actually exist. Instead, the different types of binds, editors and maps are merely distinguished from each other by their parameters. The names I have given them here are not strictly present as such in the API of IPF but rather inferred from IPF’s user interface elements for them. From the viewpoint of object-oriented analysis, however, you can consider them subclasses.

![Diagram](image)

**Figure 3** Some examples of important list structures in IPF (instance diagram).

IPF uses a paradigm for its API that you could call *list-oriented*, rather than object-oriented. As mentioned, each Item is part of a List and, in fact, these Lists carry the sole responsibility for creating, retrieving and destroying Items in IPF. The application programmer cannot do this in any other way than through the
designated Lists. Figure 3 shows an overview of the lists that were used in the
development of the system described in this thesis.

All Items starting with “<>” are special singletons in IPF. The <>bind list, for
example, is the designated List for creating, storing and retrieving Binds. Similarly,
<>cond contains all Conds, <>editor contains all editors, etc. There is even a
special <>list List, which contains all of the other Lists in IPF. The Items in the
figure that start with “<>” and are in UPPER_CASE are all Editors. Nearly each
attribute of every Item has a dedicated Editor for accessing and modifying this
attribute. These are necessary for creating Binds. Some additional Editors for
accessing system variables exist as well. One example shown in the figure is
<>IPFMODE_FRAME_PULSE, which alternates between the values 0 or 1 every time
IPF draws a frame.
4 Requirements Analysis

The function of the proposed system is to be a complete AR solution that MedICLab can use to develop all of its future AR projects. The reasons for the development of the system are twofold. On the one hand, MedICLab currently does not have any such existing AR solution but desires to create new AR projects and AR version of existing VR projects and thus has a need for such a system. On the other hand, although MedICLab does have several VR systems, they do not share a common basis and MedICLab additionally desires an extra layer of abstraction on top of IPF in order to lessen its dependence on that platform. The objective of the project is thus to develop an inexpensive yet complete colocated augmented reality solution that requires minimal knowledge of IPF from the application programmer. Initially, it will work using IPF, but it should be straightforward to move it to a different development platform. The intention is that the system will serve as a common basis for all of MedICLab’s future AR projects.

This chapter details the requirements of the proposed system. Its structure and contents are based on the approaches to documenting requirements analysis outlined by Bruegge & Dutoit, Kotonya & Sommerville, and Rosson & Carroll [6, 23, 32]. First, it details scenarios that illustrate what people do and experience trying to use typical applications of the system. These include problem scenarios describing the current (VR) situation as well as visionary scenarios describing the future (AR) system. Second, it lays out the system models. These include the use cases, the object model and the dynamic models of the system. Finally, it details the actual requirements of the system, derived from the scenarios. These include functional requirements, nonfunctional requirements and pseudo requirements.

4.1 Scenarios

Below are the scenarios for the system. First are the problem scenarios. These describe the current situation, before the introduction of the proposed system. Last are the visionary scenarios. These describe the kind of situations that MedICLab would like to be able to create with the proposed system.

4.1.1 Problem Scenarios

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Alice uses the Book of Life (problem scenario).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Participating actors:</strong></td>
<td>Alice: patient; Dave: therapist</td>
</tr>
<tr>
<td><strong>Flow of events:</strong></td>
<td>Alice has come to Dave to help her to come to terms with a difficult period in her life. Dave feels that given the particulars of her case, Alice should try using the Book of Life. Dave hands Alice the gamepad used to navigate the Book of Life and asks her if she has ever used one before. Alice has played some computer games in the past, but those were with a joystick that had only one button and she finds the number of controls and buttons on the device intimidating. Dave explains her how to use the controller of the gamepad and what each button does, while Alice holds the device and familiarizes herself with it. After this, Dave shows Alice the head-mounted display that connects to his PC and explains her briefly how it works. Alice puts on the HMD with help of Dave. He asks her if she is comfortable and</td>
</tr>
</tbody>
</table>
hands the gamepad. Dave then starts the Book of Life. Alice now finds herself in a large, virtual pavilion set in a beautiful landscape. In the center of the pavilion is a pedestal with a book on it, while at the perimeter of the pavilion are pedestals with various symbolic objects on them. Dave tells her to use the controller of the gamepad to move around and familiarize her with the surroundings. Meanwhile, everything that Alice does in the virtual world, Dave can see on his PC screen. After this, they talk about what is bothering her and they conclude that her father’s death is a major event in her life that she needs to deal with. Dave asks her to pick a symbol in the pavilion to represent her feelings about his death and Alice picks the skull. Dave reminds her that she can use the buttons of the gamepad to change various properties about each symbol, such as the size or the color. Alice finds this a bit confusing, as she cannot see the gamepad while wearing the HMD. After some tries and with a little help from Dave she eventually manages to give the skull a size and color that feel right to her. She then has to press a button to pick up the skull, walk up to the book in the center and then paste the skull into the book using another button. After this, Alice and Dave are ready to move on to the next symbol. After having gone through several symbols like this, Alice has a kind of virtual picture book in front of her, symbolically describing important episodes in her life.

Table 2

Bob uses the VR Mirror (problem scenario).

| Participating actors: | Bob: patient; Ellen: therapist |
| Flow of events:       | Three months ago, Bob suffered a stroke. Though he has largely recovered by now, he is still suffering from severe upper-extremity hemiplegia. Ellen decides it will be beneficial for Bob to get therapy using the VR Mirror. Ellen shows Bob the VR Mirror and he takes a seat. She fixes two motion trackers on his healthy left arm, one around his wrist and one around his elbow. Ellen takes place behind her PC, which connects to the VR Mirror. She tells Bob to move his healthy arm in a specific way and she records his motions with the help of the VR Mirror software. She then tells Bob to watch the screen in the desk as she replays his movements and that he has to imagine he is performing them with his impaired right arm. Bob watches the screen and sees a computer-generated right arm that performs a mirror image of the movements he just performed with his left arm. Alice then tells him to imagine moving his right arm in the manner just shown. He has to depress a button on the desk the moment he starts mentally doing this and release it right away when he is finished. Alice monitors how long it takes him to make the mental motions. They then do eight more mental repetitions and then try a physical one, after which the cycle repeats with again nine mental practices followed by one physical one. |
4.1.2 Visionary Scenarios

Table 3  Alice uses the Book of Life (visionary scenario).

<table>
<thead>
<tr>
<th>Participating actors:</th>
<th>Alice: patient; Dave: therapist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow of events:</td>
<td>Alice sits down behind the AR workbench. Dave fixes a headband on her head that contains a motion tracker so that the wire hangs down from the back of her head. He hands her a handle that contains another motion tracker with a wire hanging from the bottom. He tells to place the tip of it between her eyes, look straight ahead and hold it as perpendicular to her face as possible. While she holds still in this position, he presses a button on his PC, so that the software records the position and orientation of her eyes relative to the motion tracker on her head. Once they have finished, he gives her a pair of polarized glasses to wear. Alice can now start using the system. Dave tells her to place her hand holding the handle well in front of her on the desktop and to look at it through the transparent mirror suspended above the desk. When she looks through the mirror, she sees that there are several transparent virtual objects sitting on the top. The largest and most prominent is a life-sized book in front, lying open with blank pages. Further, around the perimeter of the desk are several virtual objects that are symbolical miniature versions of real-world items. Finally, she notices that the real handle she is holding has extended itself with a metallic-looking virtual rod that ends in a point and has a forward-facing hook protruding from one side. Dave tells that she can use the point to touch a virtual object to activate it, which will make virtual controls appear around it that she can also manipulate using the rod. He also tells her she can lift objects by hooking into them and that she can also use the rod turn the pages of the book. Alice uses the point of the rod to touch the skull. It starts to glow and several controls appear around it. She drags the slides labeled “Size” to change the size of the skull. She then touches the “Color” button and a palette of colors appears, floating in the air. She touches a color, after which the palette disappears and the skull assumes the selected color. She lifts up the skull using the hook and drops it onto the page.</td>
</tr>
</tbody>
</table>

Table 4  Carol gets virtual food therapy (visionary scenario).

<table>
<thead>
<tr>
<th>Participating actors:</th>
<th>Carol: patient; Frank: therapist</th>
</tr>
</thead>
</table>
| Flow of events:       | Carol is suffering from an eating disorder and Frank wants her to try a virtual food therapy program on his AR workbench to see if this can help reduce her anxiety about food. On the desktop of the workbench, Carol sees various virtual imitations of real-world edible objects. She also sees several virtual pieces of cutlery on
the desk, including a fork, a knife, and a spoon. Frank tells her that she can touch any of the pieces of virtual cutlery and it will automatically attach to the real handle she is holding. If she already has a piece of cutlery on the handle, it will at that point simply fall off and the new one will attach in its place.

Frank suggests that she uses the virtual knife to cut a piece off the virtual apple. Carol touches the virtual knife with the real handle. She makes a slicing motion with the knife thru the virtual apple and a piece of the apple falls off. Franks then asks Gloria if she can use the virtual fork to move the virtual piece of apple to her mouth. Gloria does so and the piece disappears. They then continue to pretend to eat other items of virtual food.

<table>
<thead>
<tr>
<th><strong>Table 5</strong></th>
<th>Gloria calibrates the display (visionary scenario).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Participating actors:</strong></td>
<td>Gloria: therapist</td>
</tr>
<tr>
<td><strong>Flow of events:</strong></td>
<td>Gloria has just recently moved her AR workbench to a new room and needs to recalibrate the display. She puts the video projectors in place and starts the calibration software. Each projector now projects a grid onto the projection screen. Gloria adjusts the projectors until their grids align. She then takes place behind the work and, while looking through the mirror, takes one of the motion trackers and places it in turn on the nearest two corners and the center of where she perceives the reflected projection to be. Each time she does so, she presses a key on her PC keyboard. The display is now calibrated and ready for use.</td>
</tr>
</tbody>
</table>

### 4.2 System Models

This section details the models that describe the system. Figure 4 represents the functional model, which describes the functionality of the system from the user’s point of view. Figure 5 represents the analysis object model, which describes the structures of the system in term of concepts that are visible to the user.

**Figure 4** Functional model of the system (use case diagram).
4.3 Requirements

Below are the elicited requirements of the system. These include functional and nonfunctional requirements. Firstly, the functional requirements detail the high-level functionality of the system. This includes the interactions between the user and the system. This list is quite short, the result of the desire to have a very generic system. Secondly, the nonfunctional describe constraints placed on aspects of the system that are visible to the user, as well as restrictions that the client (MedICLab) imposes on the system but that do not directly influence the user’s view of the system.

4.3.1 Functional Requirements

The functional requirements are as follows:

- The system shall be a complete, stand-alone AR workbench.
- The system shall use mirror-based see-through AR.
- The system shall be noninvasive to the user.
- The system shall include a software library that enables the application programmer to set up interactions easily between virtual objects.
- The system shall provide the application programmer with a high-level interface to the development environment, be it IPF or otherwise.
- The system software shall reduce the time and knowledge of the development environment, be it IPF or otherwise, required to create highly interactive AR applications.
- The system shall be able to provide the user with real-time, viewpoint-dependent stereoscopy.

4.3.2 Nonfunctional Requirements

The nonfunctional requirements are as follows:

- The system shall provide the user with a colocated reach-in AR interface.
- The user shall be able to interact with virtual object using direct manipulation.
- The system shall use one motion tracker to keep track of the user’s gaze and one motion tracker to keep track of the user’s action in the colocated space.
- No prior knowledge or expertise with AR systems shall be required of the user to be able to use the system, but before starting a session, an expert will have to calibrate the system and register the user with it to ensure proper functioning.
• The system shall use the hardware available for the project, which includes a Polhemus ISOTRAK II with two receivers, two consumer-grade LCD video projectors with polarization filters, and a 3-year-old PC.
• The system workbench shall be made of a wooden frame, a semitransparent mirror and a projection screen that will preserve the polarity of the light projected onto it.
• The user shall wear polarized glasses.
• The system shall be adaptable to different hardware.
• The system software shall be modifiable by the MedICLab to use a different development environment than IPF.
• The system shall be extendible so that the MedICLab will be able to add new features easily.
• The system shall require dimmed lights but not complete darkness to function.
• The system shall be small enough to fit into an office environment.
• All system software shall be implemented in Python.
• The system shall be developed using IPF.
• The system shall be developed for the Microsoft Windows PC platform.
5 Design

To explain the design of the proposed system, this chapter first presents a user analysis, presenting the anthropometric data and heuristics used in the design of the workbench. It does not discuss the psychological traits of the target group, as this includes a simply too broad range of people and because this project does not include the development of user application. Next, it lays out the actual design in a manner loosely based on Foley & Van Dam’s four-level approach to documenting user interface design [16]. Note, however, that because the system software is intended to be a tool for creating a user interfaces, instead of being a user interface itself, it is designed with the application programmer in mind, rather than the end user, as is usual. As such, the design is more technical and abstract than is normally the case with Foley & Van Dam’s approach and deliberately takes some liberties with it.

5.1 User Analysis

We would like a large variety of people to be able to use the AR workbench, spanning the spectrum of human dimensions from a small woman to a large man. Clearly, it is impossible to be able to accommodate everyone perfectly. Instead, the design of the workbench aims to be suitable for approximately 90% of the civilian population.

Unfortunately, not a lot of civilian anthropometric data is available and what is available often carries a hefty price tag for use. Since this project had only limited monetary resources, the design of the system had to make do with what was readily available, which was a summary in Wickens et al. [41] of the anthropometric data collected by the Eastman Kodak Company [12]. Table 6 lists the anthropometric data used in the design of the AR workbench. The percentiles assume a normal distribution.

<table>
<thead>
<tr>
<th>Table 6</th>
<th>Anthropometric data used in the design of the workbench [12, 41].</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5\textsuperscript{th} percentile female</td>
</tr>
<tr>
<td>Hand</td>
<td>Digit two length</td>
</tr>
<tr>
<td></td>
<td>Hand length</td>
</tr>
<tr>
<td>Head</td>
<td>Biocular breadth</td>
</tr>
<tr>
<td>Seated</td>
<td>Elbow rest height</td>
</tr>
<tr>
<td></td>
<td>Elbow-to-fit length</td>
</tr>
<tr>
<td></td>
<td>Eye height</td>
</tr>
<tr>
<td></td>
<td>Normal sitting height</td>
</tr>
<tr>
<td></td>
<td>Midshoulder height</td>
</tr>
<tr>
<td></td>
<td>Popliteal height</td>
</tr>
</tbody>
</table>
Digit two length is the length of the index finger. Hand length includes the outstretched fingers. Biocular breadth is the distance between the left corner of the left eye and the right corner of the right eye. Seated elbow rest height is the vertical distance between the seat and the elbow when the upper arm is left to hang but the lower arm is kept at a 90° angle. Seated elbow-to-fit length is the horizontal distance between the center of the elbow joint and the knuckles of the hand. Seated eye height is the vertical distance between the center of the eye and the top of the seat. Normal sitting height is the vertical distance between the top of the head and the top of the seat. Seated midshoulder height is the vertical distance between the top of the shoulder joint and the top of the seat. Seated popliteal height is the vertical distance between the top of the seat and the floor when the user’s upper and lower leg are at a 90° angle. Seated thigh clearance height is the highest vertical diameter of the thigh. Seated upper-arm length is the vertical distance between the top of the shoulder and the bottom of the elbow when the upper arm is left hanging but the lower arm is kept at a 90° angle. Standing forward functional reach of abdominal extension to functional pinch is the horizontal distance between the abdomen and the pinching fingers of the horizontally outstretched arm.

In addition to the data above, the design takes several anthropometric guidelines from other sources into account:

- Standing work surface height should be at 5 to 10 cm below elbow level while seated work surface height should be at elbow level, unless the job requires precise manipulation or the application of great force [41]. The recommended work surface height for standing light work is 85–90 cm for women and 90–95 cm for men [19].
- The seated forward reach envelope, measured horizontally from the nearest edge of the work surface when sitting upright, for a fifth-percentile female’s arm has a radius of about 42 cm [12]. This maximum horizontal reach occurs at a height of about 25 cm above elbow height. When reaching 52 cm to the left or right of the centerline, she can reach forward by about 8 cm across the work surface, located at elbow height. When she reaches 38 cm to the side of the centerline, her forward reach on the work surface is about 27 cm.
- The work surface should be at most 100 cm wide [36]. Primary controls should be located within 15 cm to the left and right of the centerline.
- Humans cannot comfortably look down more than 30° from the horizontal plane without tilting their heads [19].

<table>
<thead>
<tr>
<th></th>
<th>5th percentile female</th>
<th>50th percentile</th>
<th>95th percentile male</th>
<th>99th percentile male</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thigh clearance height</td>
<td>11 cm</td>
<td>17 cm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upper-arm length</td>
<td>30 cm</td>
<td>35 cm</td>
<td>42 cm</td>
<td>43 cm</td>
</tr>
<tr>
<td><strong>Standing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forward functional reach</td>
<td>49 cm</td>
<td>77 cm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>of abdominal extension to</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>functional pinch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.2 Conceptual Design

The user’s mental model of the system is very similar to the analysis object model (see section 4.2), but with some simplifications. Figure 6 lays out the objects of the user’s mental model and their properties, relations between each other and operations on them. All objects in the model represent things that the user will actually perceive.

The names of the objects in the figure refer to nouns in the scenarios presented in chapter 1. The Patient class represents, of course, all patients that will use the system. A patient using the system will control a real handle, which is a real object. Attached to this handle is a virtual tool, which is a virtual object. Moving the handle will move the tool with it. With the use of the virtual tool, the patient can interact with other objects present in the workspace. Finally, all objects in the workspace, both real and virtual, are three-dimensional objects, with a position and orientation in space.

![Figure 6](image)

**Figure 6** Conceptual design of the user’s mental model of the proposed system (class diagram).

5.3 Semantic Design

The virtual part of the conceptual design needs to be realized as a software library. Given the requirements, this will be a module in Python. Figure 7 details the static structure of this module, called roles. The design of the roles module tries to solve the problem of generality: we not only want the design to be able to implement the situations sketched in chapter 4 and the section 5.2 above, we want it to be generic enough to be easily adapted to be applied to other, unforeseen applications of the AR system. Further, the design tries to be extendable and modular, so that developers may easily add functionality or change the underlying implementation without breaking existing code or violating interface contracts.

Some elaboration on the contents of the figure is in order. First, please note that the Actor class in the figure is not the same as the Actor class in the conceptual design or the analysis object model. The Actor here rather represents any object...
that can act on other, virtual objects. A virtual Tool from the conceptual design or the analysis object model would be an Actor here. An Actor can be what the user perceives as a tool, but does not have to be, in order to keep the system open-ended and adaptable. An Actor could also be, for example, an object not directly controlled by the user or even a real object, instead of a virtual one. In the case of a real object, its Actor would still have a virtual Obj representing the exact shape, location and orientation of the real object in the virtual space, but it would be invisible so that the user would perceive only the real part of the Actor and not the virtual part.

![Semantic design of the roles module (class diagram).](image)

Such Actors may act differently on different objects and thus can play different Roles towards different Target objects. An Interaction object defines what Role an Actor plays toward a particular Target object. The application programmer determines what exactly it is that a Role does; Role is an abstract class and cannot be instantiated. The application programmer will need to subclass Role and implement the act(...) method to define Role behavior. In summary, these four classes (Actor, Role, Target and Interaction) are at the core of the functionality of the roles module and are the main classes that the application programmer will deal with.

Additional classes are required, however, to interface with IPF. Each required list item in IPF gets an object-oriented representation in the form of an Item. Each Item has a set(...) and a get(...) method to set and get the values of properties of its IPF counterpart. It also has a name, which is identical to its name in IPF. Then, each Actor gets an Obj visually representing it in the virtual space. Each Obj represents at most one Actor and this Actor can be retrieved through the class method Actor.get(...). Each Target also gets an Obj representing it, but also several Regions
defining its spatial bounds, used in collision detection. Just like with Actors, each Obj
represents at most one Target, which can be retrieved through the class method
Target.get(...). Because collision detection in IPF is not possible between Regions,
however, but only possible between Pivots and Regions, each role gets several
clouds of Pivots, called Sensors, which determine when an Actor’s Role is interacting
with a Target. An Interaction between a Role and a Target takes places when all
Pivots of a Sensor of the Role are located together inside a Region of the Target.
Thus, you can think of a Sensor as a logical conjunction ("and") of Pivots, while all
Sensors of one Role together form a logical disjunction ("or"). Thus, different visual
parts of an Actor can play different Roles. This effectively means that a user can
make a tool act differently towards a virtual object by manipulating it differently. For
example, in a real-world analogy, you can use a hammer both to hammer in nail and
to pull them out, simply by using a different part of the hammer. The hammer itself
would be an Actor, while these two functions of the hammer would be two different
Roles, each with their own collision Sensors.

Finally, each Interaction has a number of Bind, Conds and Maps. The Conds in IPF
check whether the Pivots of the Sensors of the associated Role have entered the
Regions of the associated Target. The Binds and Maps in IPF are then used to link
these together to form a callback structure that will call the Interaction’s perform() method. This method then tells the Role to act. The Binds of an Interaction can be
enabled or disabled through its enabled attribute. The callback structure retrieves
its Interaction by its id using the class method Interaction.get(...). The fact that
items in IPF cannot hold references to Python objects makes this necessary.

5.4 Syntactic Design

Figure 8 and Figure 9 show the design of the display of the workbench. The figures
show two polarized video projector beams projecting unto a screen that preserves
the polarity of the projected light, located at the top of the workbench. A
transparent mirror at front reflects this projection to the user. The perceived location
of the projection extends beyond the work surface; this is to provide more vertical
room for virtual items lifted above the work surface.

Note in Figure 8 that the beams do not project all the way to the edge of the
screen nearest to the user. As a result, the perceived reflection of the screen does
not reach to the user’s edge of the work surface either. This is intentional; the user
will not be able to view this area of the work surface through the mirror anyway. In
addition, thus moving the projection further away from the user allows the
projectors to project at an angle that is virtually perpendicular to the screen. This is
important as it minimizes the number of pixels lost to keystone correction.

The mirror sits at a zero-degree (0°) angle, perpendicular to both the screen and
the work surface, for several reasons. First, in order to avoid the user accidentally
seeing their own reflection, the reflective surface of the mirror has to tilt away from
the user; the farther it tilts away, the less likely it is that the user will see their own
reflection. There is, however a limit to this: the farther you tilt the mirror away, the
greater foreshortening you will get in the user’s view of the reflected screen. Prototyping has shown that a horizontal mirror tilts away sufficiently without too
much foreshortening of the reflected screen. Second, the reflective surface of the
mirror sits about halfway between the projection screen and the work surface and
projection screen are horizontal as well, it allows the reflected screen and the work
surface to be very near to each other, making it easier to calibrate the display.
Third, this angle allows for easier insertion and removal of the mirror than an angle
tilted toward the user would, because there is no need to fasten it. You can simply slide it in and out of the frame when necessary.

**Figure 8** Syntactic design of the display of the workbench. Visible are the projection screen (white, top), the four edges of each of the two projection beams (thin diagonal lines in red and cyan respectively), the transparent mirror (center), the transmitter unit of the magnetic tracker (gray, center), and the position and orientation of projection screen as perceived when looking through the mirror (bottom).

**Figure 9** Syntactic design of the display of the workbench, as seen from the back. The two projectors (not present in the image) would be located where the projection beam edges (red and cyan respectively) converge at the bottom of the image.
5.5 Lexical Design

Figure 10 shows the design of the exact dimensions of the workbench. The two faces drawn in the picture (not to scale) represent the eye height of a 95th-percentile male and a 5th-percentile female when seated as high as possible. It evaluates several design options with the use of the anthropometric data mentioned in chapter 4. For the 5th-percentile female, it shows her nose height above the surface (44 cm), the height above the work surface of her maximum horizontal reach (25 cm), her maximum horizontal reach (42 cm) and her eye height above the work surface (48 cm). For the 95th-percentile male, it shows his eye height above the seat, which is positioned for the 5th-percentile human thigh clearance. Some of the values listed, such minimum mirror and screen depth, ended up discarded because they were the result of assumptions that prototyping later showed to be invalid.

Most of the resulting dimensions—though not all—then functioned as guidelines for the virtual prototype of the workbench. One of the nonfunctional requirements, however, is that the workbench should fit into an office environment, so constraints relating to that were also taken into account. For example, the workbench is 75 cm deep because the standard issue desks in the MedICLab are also 75 cm deep. Further, because only limited accuracy was possible in the construction of the wooden frame, all dimensions were rounded to multiples of 5 cm.

Figure 10  Lexical design of the workbench, showing the field of view of a 95th-percentile male and a 5th-percentile female, as seen from the side.

Figure 11 and Figure 12 then show the virtual prototype based on the lexical design. At the top, it contains support strips for the projection screen. There is nothing located above the projection screen; this way, it can be easily removed and replaced. In the center of the frame are two crossbars that uphold the transmitter of
the magnetic tracker. The transmitter has two magnetic hemispheres whose coordinate systems are each other’s mirror images. In other words, when on receiver crosses from one hemisphere to the other, its coordinates flip. Unfortunately, there is no way of telling in which hemisphere a receiver is currently located. Therefore, the transmitter sits at an angle to make sure that both receivers are in the same hemisphere and stay there.

The mirror sits as high as possible in the frame, in order to provide the largest possible interaction space. It still sits low enough, however, that even a 5th-percentile female will be able to use the system. If we want the reflected screen to be very near the work surface, putting the mirror as high as possible also makes it possible to the projection screen as high as possible, creating more space for tall people to use the system. Indeed, a 95th-percentile male has ample space above his head when using the system. Putting the screen as high as possible also makes the distance between the screen and the projectors as big as possible and thus the projected image as large as possible. To enable the user to make full use of the large virtual space enabled by this projection area, the frame sports a very wide work surface, thus giving the user plenty of space to move around and position the virtual objects with which they interact. The work surface is purposely shallow, though, so a 5th-percentile female will be able to reach all of the virtual content even when it sits at the far edge of the work surface. This is to minimize the chance of the user bumping into or pushing against the mirror.

The human-shaped dummy in the figure is about 1.78 m tall. Note, however, that it is sitting too low: When it looks at the far side of the works surface, it is not looking through the mirror. To be sitting correctly, the user should be looking through the mirror when looking at their hands. Clearly, most people will need to sit a bit higher than what they are accustomed to, even though the work surface is located at the same height as the MedICLab’s standard issue desks (75 cm). Making the work surface lower, however, would not allow enough space for tall people to
use the system. Thus, the user will need a well-adjustable desk chair to be able to use the workbench. Very short people will also require footrests to be able to sit comfortably.

**Figure 12** Virtual prototype of the workbench frame, as seen from the side, showing the projection screen (white, top), transmitter unit of the magnetic tracker (blue, center), and the projectors (blue, bottom).
6 Implementation

This chapter discusses the implementation of the system. I implemented the hardware that I designed in collaboration with others at the MedICLab. I implemented the software that I designed by myself. The exception to this is the registration module, the basics of which I designed and implemented, but which uses an algorithm created by the MedICLab.

6.1 Hardware

Figure 13 shows the finished workbench in a setup that is very close to how it would be in actual use. The main difference is that the PC would likely be at the therapist’s desk instead of underneath the workbench. Most important in the configuration, however, are the positions of the LCD video projectors and the motion-tracking transmitter. Figure 14 and Figure 15 show close-ups of these.

![Figure 13](image)

Figure 13 The final workbench with a person sitting in the right position for usage. In actual use, the motion-tracking receiver lying on the work surface would not have its wire run through the workspace so as not to interfere with user. Missing from the picture is the tool handle that will contain the receiver and the other receiver that the user will wear on their head.

MedICLab had outsourced the construction of the workbench frame to a carpenter. Once we received the built workbench from the carpenter, however, we discovered we had to make several modifications to it because of several practical problems. First off, we replaced the glass mirror with a tinted, transparent plastic sheet, because of safety concerns. The plastic sheet adequately reflects the screen projection. Because the light from the screen’s reflection, however, is significantly less bright than the light reflected off the user’s hands even under dimmed lighting conditions, we chose to use a tinted sheet to dim the visibility of the real objects in the scene and thus enhance the visibility of the virtual objects. We considered several tinted plastic sheets and tried to find one that would work under normal
office lighting conditions. This, however, proved to be impossible with the different shades of plastic available to us, so we settled for one that provided almost no discernible difference in perceived brightness between virtual and real objects under the dimmed lighting conditions available to us.

![Figure 14](image1.png)

**Figure 14** The LCD video projectors. Missing from this image are the polarization filters that would sit over the lens openings.

![Figure 15](image2.png)

**Figure 15** The motion-tracking transmitter sitting on the support beams behind the mirror sheet. Note its angle.

The plastic sheet turned out to be too flexible and thus did not stay planar on its own. At the back of the sheet, this is not a problem as it rests on the support for the transmitter, but at the front, extra support is required. We achieved this by adding an extra support beam below the front of the sheet. You can see this in Figure 16. Obviously, this does impede the user’s vision somewhat, but fortunately, I designed the mirror to be somewhat oversized as to allow for a reasonable margin of error. The net effect is that the user needs to sit a little bit higher than before and bend slightly forward to look over the support beam.

Finally, the execution of the workbench has some slight differences in configuration and dimension from the design. The foremost of these is that at his
discretion, the carpenter executed the support beams for the transmitter and the back of the mirror slightly differently, because of technical limitations. This, however, has no impact on the functioning of the system.

Figure 16  The plastic mirror sheet of the workbench, under fully lit conditions. Under the dimmed lighting conditions required for use, it will appear significantly darker. When actually using the system, the user will need to sit a little bit higher and lean slightly forward, with their hands in front of them on the work surface.

6.2 Software
At the start of each session, the user needs to register with the system so that it can properly calibrate the relative positions of the trackers, the user’s eyes and the reflection of the screen. The user first needs to place the handheld receiver squarely between their eyes and press a button on the keyboard, so the system can from there estimate the relative positions of the eyes with respect to the receiver worn on the head. After this, the screen displays a border around itself and a crosshair in its center. The user must place the handheld receiver at the two nearest corners and the center of where the screen reflection appears to be spatially located, and press a button on the keyboard each time when doing so. From this, the system calculates the relative position of the screen with respect to the transmitter. The system then uses the built-in CAVE support of IPF to take care of the stereoscopy. It simply projects what would be the front CAVE wall.

The most complicated part of the software implementation is the Interaction callback structure. Figure 17 shows what is conceivably the simplest case of an Interaction callback structure in IPF that is still complete and representative of the generic case: A Interaction callback structure for a Sensor with just two pivots. Each Pivot is a point in space. The callback structure orientation ignores each Pivot’s EUL, though the application programmer can still check for this, if desired. The location DIS of each Pivot is bound to the PTO attribute of a Cond. Each Cond has an associated Region (not shown) and the STATE of a Cond is 1 when the point represented by its PTO is located inside its Region but otherwise 0 (zero). The STATEs of the two Conds feed into a FloatMultiplyMap (as there is no such thing as an IntMultiplyMap in IPF). A MapBind then tells VoidPythonMap to execute its code whenever the product of the multiplication is 1, effectively simulating a logical
conjunction. Finally, the VoidPythonMap executes the actual call to the Interaction that created the callback structure.

Figure 17 Interaction callback structure in the case of two Pivots (object diagram).

Figure 18 then shows a model of the generic case. Let us call this structure a callback unit. Then, you can combine several of these units to construct a callback structure like the one shown in Figure 17 but then for an arbitrary number of pivots as follows: Create a callback unit for all Pivots except one. Next, chain the units together by substituting each unit’s MapBind for the undefined Bind on the right side of the figure. Next, replace the only remaining undefined Bind with an EditorBind and link the only remaining Pivot to in the same way that the other Pivots in Figure 17 and Figure 18 connect to an EditorBind. Finally, link the only remaining MapBind to a VoidPythonMap, as shown in Figure 17. Thus, all Pivots link together in a logical conjunction that executes the code in the VoidPythonMap whenever it evaluates to True.

Table 7 shows the code that implements the Interaction constructor, which creates the callback structure. It takes away many repetitive tasks the application programmer would normally have to perform and automates them. In addition, these tasks are quite complicated to perform and prone to error. Automating them reduces the risk of errors, reduces the amount of domain knowledge required and frees up the programmer’s time for implementing the actual ways in which the virtual objects will interact by implementing subclasses of Role.

An explanation of the code in Table 7 is in order. Figure 19 shows the first part of the Interaction system. Every frame cycle—that is, every time before it draws a frame on screen—IPF tells all of its Binds to update. Each Bind checks whether its inputs depend on the output of another bind. If so, it waits for the Binds it depends on to update their output first before updating itself. Thus, in the Interaction system,
the PivotBinds are the first to update. Each PivotBind gets the position of its Pivot and forwards it, along with a reference to its Cond, to <<COND_PTO. This then updates the PTO attribute of the given Cond to the given vector.

![Image](image.png)

**Figure 18** Generic case of the Interaction callback structure (object diagram).

**Table 7** Interaction constructor, which creates the callback structure in IPF (Python code).

```python
def __init__(self, role, target):
    
    """
    Creates a Interaction between the given Role and Target, and adds it to them and to this class's dict of instances, silently replacing any existing instance this class has for the same (Role, Target) pair.
    NOTE: All Interactions are initially disabled. After creation, you will have to call setEnabled(True) to enable them.
    """
    self.__enabled = False
    self.__role = role
    self.__target = target
    roleName = "a.%s/rl.%s" % (role.getActor().getObj().getName(), role.getName())
    targetName = target.getObj().getName()
    self.__id = "%s/t.%s" % (roleName, targetName)
    self.__maps = []
    self.__binds = []
    self.__conds = []

    # Create callback code for each (Sensor, Region) pair.
    # For each Sensor, require all pivots to fire before making the call.
    for region in target.regions():
        regionName = region.getName()
        for sensor in role.sensors():
```

33
maps = []
conds = []
cmBinds = []  # cond-map binds
mmBinds = []  # map-map binds
pcBinds = []  # pivot-cond binds
prefix = "%s/rg.%s/s.%s"
    % (self.__id, regionName, sensor.getName())
for pivot in sensor:
    pName = pivot.getName()
cName = "%s/p.%s" % (prefix, pName)
ipfspecial.itemnew("cond", cName)
    cond = Cond(cName)
    cond.set("COND_REGION", regionName)
    conds += [cond]

    # bind from pivot.xyz to cond.pto
bName = "%s/PC.%s" % (prefix, pName)
ipfspecial.itemnew("bind", bName,
    "BIND_XYZHPR", ".XYZ",
    "BIND_EDITOR_MODIFIED", "<>COND_PTO",
    "BIND_FUNCTION", "Pivot",
    "BIND_FUNCTION_ITEM", "pivot[%s]" % pName,
    "BIND_ITEM_MODIFIED", "cond[%s]" % cName,
    "BIND_ACTIVE", self.__enabled)
bind = Bind(bName)
pcBinds += [bind]

    # Step 1. Create a map for each cond.
    mName = cName
    ipfspecial.itemnew("map", mName)
    map = Map(mName)
    map.set("MAP_EXPTYPE", "Multiply")
    maps += [map]

    # Step 2. Bind one cond to each map.
    # bind from cond.state to map.float_op2
bName = "%s/CM.%s" % (prefix, pName)
ipfspecial.itemnew("bind", bName,
    "BIND_EDITOR_MODIFIED", "<>MAP_FLOAT_OP2",
    "BIND_FUNCTION", "Editor",
    "BIND_FUNCTION_EDITOR_ITEM",
    "cond[%s]" % cName,
    "BIND_FUNCTION_ITEM",
    "editor[<>COND_STATE]",
    "BIND_ITEM_MODIFIED", "map[%s]" % mName,
    "BIND_ACTIVE", self.__enabled)
bind = Bind(bName)
cmBinds += [bind]

    # Step 3. Move the LAST cond-map bind to the FIRST map’s
    # float_op1.
    cmBinds[-1].set("BIND_EDITOR_MODIFIED",
         "<>MAP_FLOAT_OP1")
    cmBinds[-1].set("BIND_ITEM_MODIFIED",
        "map[%s]" % maps[0].getName())

    # Step 4. Bind <>ipfmode.IPFMODE_FRAME_PULSE to the last
    # map’s float_op2. If we don’t include
    # IPFMODE_FRAME_PULSE in the equation, the callback will
    # fire only on initial collision. With FRAME_PULSE
# included, the callback will keep firing on prolonged
# contact, which is what we want.
bName = "%s/FramePulse" % prefix
ipfspecial.itemnew("bind", bName,
    "BIND_EDITOR_MODIFIED", "<>MAP_FLOAT_OP2",
    "BIND_FUNCTION", "Editor",
    "BIND_FUNCTION_EDITOR_ITEM", "modes[<>ipfmode]",
    "BIND_FUNCTION_ITEM",
    "editor[<>IPFMODE_FRAME_PULSE]",
    "BIND_ITEM_MODIFIED",
    "map[%s]" % maps[-1].getName(),
    "BIND_ACTIVE", self.__enabled)
bind = Bind(bName)
self.__binds += [bind]

# Step 5. Add a python map to make the callback.
mName = "%s/PythonMap" % prefix
ipfspecial.itemnew("map", mName)
map = Map(mName)
map.set("MAP_EXPTYPE", "Python")
map.set("MAP_TYPE", "void")
map.set("MAP_CODE",
    "import roles
    roles.Interaction.get('%s').perform()" % self.__id)
maps += [map]

# Step 6. Bind each map to the next one.
i = 0
bName = "%s/MM.%i" % (prefix, i)
prev = maps[0]
for m in maps[1:-1]:
    # Bind the PREVIOUS map to this map's FLOAT_OP1.
bName = "%s/MM.%i" % (prefix, i)
    ipfspecial.itemnew("bind", bName,
        "BIND_EDITOR_MODIFIED", "<>MAP_FLOAT_OP1",
        "BIND_FUNCTION", "Map",
        "BIND_FUNCTION_ITEM",
        "map[%s]" % prev.getName(),
        "BIND_ITEM_MODIFIED",
        "map[%s]" % m.getName(),
        "BIND_ACTIVE", self.__enabled)
    bind = Bind(bName)
    mmBinds += [bind]
    prev = m
    i += 1

# bind from last multiply map to the python map's
# EXP_EXE
bName = "%s/ToPythonMap" % prefix
ipfspecial.itemnew("bind", bName,
    "BIND_EDITOR_MODIFIED", "<>EXP_EXE",
    "BIND_FUNCTION", "Map",
    "BIND_FUNCTION_ITEM",
    "map[%s]" % maps[-2].getName(),
    "BIND_ITEM_MODIFIED",
    "map[%s]" % maps[-1].getName(),
    "BIND_ACTIVE", self.__enabled)
bind = Bind(bName)
mmBinds += [bind]
Next, shown in Figure 20, each EditorBind asks `<>COND_STATE` for the STATE value of its Cond. It then tells `<>MAP_FLOAT_OP1` or `<>MAP_FLOAT_OP2` to update the FLOAT_OP1 or FLOAT_OP2 attribute of its FloatMultiplyMap, respectively. Which operand attribute an EditorBind updates depends on its position in the callback structure, as shown in Figure 17.

After this, after a FloatMultiplyMap has had both of its operands updated, the MapBind that monitors it get the resulting product and uses it to update one of the operands of the next FloatMultiplyMap. The exception to this is the final MapBind in the chain, which updates the VoidPythonMap. This MapBind, as shown in Figure 21, gets the product from the FloatMultiplyMap it monitors and passes it to `<>EXP_EXE`, along with a reference to the VoidPythonMap it wants to update. If the product is not zero, it tells the VoidPythonMap to execute its code; otherwise, it does nothing.

Whenever the logical conjunction evaluates to True, the VoidPythonMap executes its code. It retrieves the Interaction object that created it by passing its id to the class method Interaction.get(id). It then tells its Interaction to perform(). Finally, the Interaction then tells its Role to act on it. The Role then knows what to do because it can get from the Interaction that called it with which Target it is interacting.
Figure 21 Interaction callback mechanism, part 3: Evaluating the logical conjunction (activity diagram).

Figure 22 Interaction callback mechanism, part 4: Making the callback (sequence diagram).

Table 8 shows an example of how you can use the roles module to set up Interactions between virtual objects. This example is also included with the module itself. Note that the two Roles that the example defines are not very sophisticated, but that is not the point of the example. The point is that to subclass Role, the only thing the application programmer needs to do is implement the act(…) method. The programmer can then reuse their Roles for different actors. Of course, as you can see, implementing an act(…) method still requires a fair amount of domain knowledge. This is, however, unavoidable as it is very hard to automate because the details of what the programmer wants to act achieve with a certain Role will vary wildly between applications.

Table 8 Example of how to use the roles module (Python code).

```python
# In IPF:
# Create objs to represent your actors and targets.
# Create regions to define the boundaries of your target objs.
# Create pivots to define the sensors of your actors' roles.
# Create binds and maps to make sure your regions and pivots stay current.

# Add the path where roles.py is located to the system path
import sys
sys.path.append(".")
```
# Import the roles framework in IPF
from roles import *

# Create Items for your objs, regions, and pivots
objs = dict()
objs['actor1'] = Obj("actor1")
objs['target1'] = Obj("target1")
objs['target2'] = Obj("target2")

regions = dict()
regions['target1'] = Region("target1")
regions['target2'] = Region("target2")

pivots = dict()
pivots['actor1/000'] = Pivot("actor1/000")
pivots['actor1/001'] = Pivot("actor1/001")
pivots['actor1/010'] = Pivot("actor1/010")
pivots['actor1/011'] = Pivot("actor1/011")
pivots['actor1/100'] = Pivot("actor1/100")
pivots['actor1/101'] = Pivot("actor1/101")
pivots['actor1/110'] = Pivot("actor1/110")
pivots['actor1/111'] = Pivot("actor1/111")

# Create Targets
targets = dict()
targets['target1'] = Target(objs['target1'], [regions['target1']])
targets['target2'] = Target(objs['target2'], [regions['target2']])

# Create Actors
actors = dict()
actors['actor1'] = Actor(objs['actor1'])

# Derive subclasses of Role that override act(...)
class Pusher(Role):

    def __init__(self, name, actor, sensors):
        super(Pusher, self).__init__(name, actor, sensors)

    def act(self, interaction):
        target = interaction.getTarget()
targetObj = target.getObj()
dt = targetObj.get("OBJ_DISPLACEMENT")
actor = self.getActor()
actorObj = actor.getObj()
da = actorObj.get("OBJ_DISPLACEMENT")
d = dt - da
d.z = 0.0
dt = dt + d.normalize()
targetObj.go("OBJ_DISPLACEMENT", dt, 0.5)

class Lifter(Role):

    def __init__(self, name, actor, sensors):
        super(Lifter, self).__init__(name, actor, sensors)

    def act(self, interaction):
        target = interaction.getTarget()
targetObj = target.getObj()
dt = targetObj.get("OBJ_DISPLACEMENT")
actor = self.getActor()
actorObj = actor.getObj()
da = actorObj.get("OBJ_DISPLACEMENT")
d = dt - da
if d.z > 0:
    dt.z = dt.z + 0.1
    targetObj.go("OBJ_DISPLACEMENT", dt, 0.5)

# Create Roles for your Actors
Pusher("pusher1", actors["actor1"], [
    Sensor("001", [pivots["actor1/001"]]),
    Sensor("011", [pivots["actor1/011"]]),
    Sensor("101", [pivots["actor1/101"]]),
    Sensor("111", [pivots["actor1/111"]]),
])
Lifter("lifter1", actors["actor1"], [
    Sensor("x0", [
        pivots["actor1/001"],
        pivots["actor1/011"]
    ]),
    Sensor("x1", [
        pivots["actor1/101"],
        pivots["actor1/111"]
    ]),
    Sensor("y0", [
        pivots["actor1/001"],
        pivots["actor1/101"]
    ]),
    Sensor("y1", [
        pivots["actor1/011"],
        pivots["actor1/111"]
    ]),
])

# Create an Interaction for each Role-Target interaction you want to enable
Interaction(actors["actor1"]["pusher1"], targets["target1"])
Interaction(actors["actor1"]["lifter2"], targets["target2"])

# Enable your Interactions
for actor in actors.values():
    for role in actor.roles():
        for interaction in role.interactions():
            interaction.setEnabled(True)
7 Evaluation & Results

The final prototype of the workbench turned out reasonably well. It is now safe to use, as it no longer has any glass parts, with a plastic sheet replacing the glass mirror. During the middle of the day, with closed Venetian blinds, both real and virtual objects are clearly visible with approximately equal brightness through the glossy, shaded piece of plastic that serves as the mirror. The executions of the wooden workbench and the special projection screen have a slight mismatch in dimensions, however, with the screen being slightly wider than the workbench allows for, leading to an unsuitably tight fit. Future versions of the workbench should allow for a bit larger margin of error in production size. Additionally, due to a miscommunication that led the carpenter to believe that the projection screen was actually another mirror, he left out the support beams for it. In that light, it is actually a boon of sorts that the workbench frame fits so tightly around the projection screen frame or otherwise the projection screen would not even stay up there at all. If you look closely, you can barely see this in Figure 23.

![Figure 23](image)

**Figure 23** Backside of the workbench, showing the projection screen at the top and the video projectors at the bottom. On the left side of the picture, you can just make out the Polhemus ISOTRAK II.

Unfortunately, the current hardware setup has some shortcomings with respect to some its envisioned usages, a result of some of the constraints that the nonfunctional requirements impose on it. First, if we want to use the system to stimulate the production of motor imagery in treating hemiplegia, having only one receiver to track the motions of the arm is simply not enough. As mentioned in chapter 1, the more spatially correct the observed movements, the stronger they invoke motor imagery. Unfortunately, this is nearly impossible to provide anything than merely the spatially correct movement of the hand with only one motion tracker. For correctly recording and recreating the motions of the entire arm, at least two additional trackers are necessary: one for the upper arm and one for the lower
Second, the current setup is inherently limited in the kind of tools it can simulate, since the handle does not contain any moveable parts. For example, it is quite unfeasible to use it to control virtual scissors, wire cutters, clamps or tongs, since there is no way of indicating whether the tools should be opened or closed. Adding a simple button to the handle that the user can press with their thumb should alleviate this problem. Unfortunately, the requirements did not foresee in this.

Performance of the roles module when using only a few Role–Target combinations with only few Sensors and Regions, respectively, is good, as the system causes no noticeable additional delay beyond that already present in the real-time stereoscopy. However, adding more combinations with a greater number of Sensors and Regions causes the number of Binds necessary to implement the logical conjunctions to grow exponentially due to the number of possible Sensor–Region permutations, dramatically increasing startup times and memory usage of the system—though fortunately not responsiveness. The system could avoid this if IPF supported intersection checking between Regions, instead of only between Pivots and Regions, but unfortunately, it does not. In addition, you normally do not want all Pivots of one Role to be in one Sensor but rather only have adjacent Pivots in the same sensor, since callbacks only execute when all Pivots in one Sensor have entered a Region. Depending on the shape and function of the acting object, this can become a combinatorial problem leading to an explosion in the number of Sensors as well, further compounding the problem. Thus, the system currently works best with virtual tools that have very specific and local interaction points, such as pointers, hooks and forks.

One of the main sources of complication with the implementation of the software is the fact that IPF was simply never intended for highly interactive virtual environments. It has good support for triggers, buttons and timers, but no real support for, for example, collision checking, bounding boxes or continuous interaction. This has been an ongoing source of frustration during the development of the software. All in all, IPF turned out to be a significantly limiting factor in the development of the system, necessitating complicated workarounds for the absence of certain features in IPF, such as, for example, the creation of Sensors in order to compensate for the lack of true collision detection support. The creation of these and others workaround caused a lot of overhead that could have perhaps been better spent on elsewhere had the project made use of a development platform better tailored to VR in general and highly interactive VR in particular.

Notwithstanding all the above, however, the system can still definitely be used to implement all kinds of application. First, the system is definitely suitable for implementing the visionary scenarios outlined in chapter 4. Indeed, food therapy and the Book of Life definitely belong to the realm of possibilities with the final workbench prototypes. Second, many other applications also come to mind. One example would be an application with which an architect could view a 3D model of a blueprint and lift off the roof or remove a wall with the use of a virtual tool. Another example would be games, such as virtual air hockey or table tennis. In short, the system might have its shortcomings, but it is still definitely functional and useful.
8 Conclusion

This thesis discussed the development of an augmented reality (AR) system that the MedICLab commissioned from the author of this document. In chapter 1, we asked the question: *Is it possible to develop an inexpensive yet complete colocated reach-in mirror-based see-through augmented reality solution that is noninvasive and intuitive to the user, but requires minimal domain knowledge from the application programmer and is not tied to a particular development platform?* To answer this, we need to split it into further sub questions. This conclusion will discuss these in turn.

*Is the system inexpensive?* This is of course a very subjective question, but we can put it in perspective by summing the costs of the components and relating it to the envisioned usages of the system. First, the most expensive component is the motion tracking system. Polhemus has by now discontinued its ISOTRAK II, but while it was available in the 90s, it cost about 2200€. The company’s current replacement for this model is the FASTRAK, which is virtually identical but for the addition of two extra receivers and costs around 4300€. A similar product from a different manufacturer would be Ascension’s Flock of Birds, priced around 2600€ for two receivers and 5000€ for four receivers. Second, LCD video projectors similar to the ones used in the prototype are available commercially for around 1000€ apiece. Third, PCs at least as powerful as the one used in this project are already available from as little as 500€. Finally, the rest of the hardware adds at most about an estimated 300€. Thus, the total cost of the system is only around 4000€. To be on the safe side, suppose we estimated this cost a little on the low side and it turns out to be more like 5000€—or 7000€ should we opt for the two additional motion tracking receivers. Considering that many different kinds of intensive medical treatments can make very effective use of this system, it does not seem like a very high price to pay.

*Is the system a complete colocated reach-in mirror-based see-through AR solution?* That is, can we use it as-is to immediately create AR applications of the kind mentioned without requiring additional components? As chapter 6 demonstrates, a developer can easily create interacting objects with only very few lines of code. On the hardware, the system provides everything necessary to immediately colocate real and virtual objects in a space that the user can reach into, using a see-through mirror. There is still a deficiency, however, in that the software library does not provide an obvious way to connect a motion tracker to an Actor object. The actual procedure of how to do this is not very complicated, but it is unfortunately not immediately obvious from the architecture of the software, because part of it still requires manual interaction with the user interface of IPF, the development environment that MedICLab uses. A future version of the system will need to address the further automation and clarification of setting up this connection.

*Is the system noninvasive to the user?* The equipment the user has to wear is very light and nonrestrictive, consisting only of a headband and a pair of glasses. In addition, the user can wield or hold the tool handle in any way they like and put it down at any time, since it is not attached to their body, making the system very noninvasive. Of course, the user will still have to contend with the wires coming out of both motion trackers. Indeed, if we were to expand the system by adding still two more motion trackers and attaching these to the user’s arms, as would be necessary for the scenario hemiplegia treatment, we would add even more wires and objects.
attached to the user. Even in this case, however, the user will still enjoy great freedom of movement and it will not significantly encumber them.

**Is the system intuitive to the user?** This of course largely depends on the kind of application developed with the system. In general, however, the kind of direct manipulation that the system provides to the user should be very intuitive to use because it largely corresponds with interaction situations in the real world. In addition, the system has an actual physical component that the user can grasp and control—the tool handle—whose real position and orientation translate to the virtual world in the most direct way possible. Thus, when manipulating objects in the virtual world, the user can apply much of the expertise they have with manipulating objects in the real world, thereby enabling positive transfer between the two.

**Does the system require only limited domain knowledge from the application programmer?** That is, will a developer with only very limited prior knowledge and experience with IPF be able to use the system to successfully and easily create AR applications? This is a difficult question to answer, as the topic matter is very complex. We can glean a great deal of information, however, from the sample code presented in chapter 6. What is immediately clear is that the developer is still required to know how to be able to create objects in IPF. This is, however, straightforward and does not require that much time to learn. The real crux, though, lies in implementing the `act(...)` method in subclasses of `Role`. This will still require a great deal of knowledge of IPF. The real merit of the `roles` module, however, lies in allowing the developer to think of and create interaction structure on a much higher level of abstraction that is normally possible in IPF. If you take a close look at the implementation of the Interaction constructor, it is immediately clear that it takes a tremendous amount of tedious, low-level work out of the hands of the application programmer. In addition, if one constructs such interaction structures manually in IPF, they can quickly become unmanageable because of their opacity and the sheer numbers of objects necessary, whose relationships often quickly become quite obscure to the casual observer. The `roles` module effectively allows the application programmer to take a step back and work at the big picture instead of having to tinker with the niggling little details. In that respect, perhaps the software does not have such a significant impact on the amount of knowledge of IPF required to create applications, but rather benefits the programmer by significantly streamlining and speeding up the development process, as well as making the code more human-readable and maintainable.

**Is the system not tied to a particular development platform?** Although the design tries to put plenty of abstraction between the application programmer and the underlying platform, many aspects of IPF still shine through in many different places. This turned out to be unavoidable for all practical purposes. First, IPF is simply too large a system that it would be completely unfeasible to fully hide it from the application programmer. It is also not completely desirable to do so, because the graphical user interface of IPF is rich with features that hiding would make inaccessible. Second, being able to create virtual objects in a visual such as allowed by the user interface of IPF is simply too convenient a feature to discard, but this of course means that we need some way of dealing with the primitives that result from this process. The system, however, is by no means completely stuck with IPF. Changing from IPF to another system will obviously require modification of the links that Actors and Targets have with Items in IPF and reimplemention of the `act(...)` methods of subclasses of Role because these will likely interact directly with IPF. The main structures created with Actors, Roles, Targets and Interactions, however, will be able to persist, as the contracts of their interfaces will remain unchanged. Only
their implementation will need to change, but this will not invalidate the API and thus have very little effect on existing code. Thus, while not making the developer completely independent from IPF, it at least makes future migrations a little less effortful.

In hindsight, given the limited time and resources available to the project, perhaps its original goals were a tad too ambitious. Several features originally planned did not make it into the final prototype because of time constraints. In other cases, the complexity of the problems facing the development turned out to be much higher than anticipated, which made it impossible for the project to fully achieve its original objectives. In particular, during the course of the project it has become quite clear that IPF is perhaps not the ideal platform for developing such a system at all. In the end, however, the project produced a working prototype that should prove fully functional for the MedICLab to develop their intended applications.
REFERENCES


38. Thaut, M.H., McIntosh, G.C. and Rice, R.R. Rhythmic facilitation of gait training in hemiparetic stroke rehabilitation. *Journal of the Neurological*
APPENDIX: APPLICATION PROGRAMMING INTERFACE

This appendix documents the application programming interface (API) of the roles module for future use. For each class, it only documents those methods that are not standard features of Python, with the exception of __init__(...), the constructor.

```python
class Actor(__builtin__.object)
    An Obj wrapper capable of playing Roles.

    Methods defined here:

    __init__(self, obj)
        Creates an Actor for the given Obj.
        param obj      an Obj

    add(self, role)
        Adds the given Role to this Actor. If this Actor already has a Role with the same name, it silently replaces it. Automatically called by Role.__init__(...).
        raises ValueError if role already has a different Actor

    getObj(self)
        Returns the Obj that this Actor represents

    roles(self)
        Returns a list of this Actor's Roles.
```

```python
class Bind(Item)
    See Item.

    Methods defined here:

    __init__(self, name)
```

```python
class Cond(Item)
    See Item.

    Methods defined here:

    __init__(self, name)
```

```python
class Interaction(__builtin__.object)
    An interaction context between the given a Role and a Target. An Interaction maintains a callback structure in IPF that causes it to call its Role's act(...) method whenever ones of its Role's Sensors is wholly inside
```
one of its Target's Regions. When an Interaction goes out of scope, it automatically deletes its callback structure.

Note: All Interactions are initially disabled. After creation, you will have to call setEnabled(True) to enable them.

Methods defined here:

__del__(self)
Deletes all of this Interaction's items in IPF. Called by the garbage collector when it is about to destroy this Interaction.

__init__(self, role, target)
Creates an Interaction between the given Role and Target, and adds it to them and to this class's dict of instances, silently replacing any existing instance this class has for the same (Role, Target) pair.

getID(self)
Returns a string generated from the names of the Objs of this Interaction's Actor and Target.

getRole(self)
Returns this Interaction's Role.

getTarget(self)
Returns this Interaction's Target.

isEnabled(self)
Returns whether this Interaction is enabled or not. When a new Interaction is created, the default value is False.

see setEnabled(...)  

perform(self)
Causes this Interaction's Role to act on this Interaction's Target.

setEnabled(self, boolean)
Enables or disables this Interaction. Disabling this Interaction temporarily deactivates its callback structure; enabling it reactivates the callback structure again.

param boolean True to enable or False to disable

Static methods defined here:

get(id)
Returns the Interaction instance with the given ID.

param id a string
raises KeyError if no such instance exists
class Item(__builtin__.object)
An IPF list item wrapper. Abstract class for representing items in IPF. Cannot be
instantiated. Programmers should instead implement subclasses of Item and
instantiate those. The name of a subclass of Item should exactly match IPF's
internal name for the list it corresponds to, except that it may differ in
capitalization. For example, a subclass of Item whose instances serve as proxies
for objects in IPF can be called obj, Obj, or OBJ, because each of these names
forms a case-insensitive match with IPF's internal name obj.

Methods defined here:

_init__(self, name)
Creates an Item with the given name. Note that an Item never checks
whether the actual IPF item it represents really exists or not. It merely relays
messages to the item in IPF.
Subclasses should override this method as follows:

def MyItem(Item):
    def __init__(self, name):
        super(MyItem, self).__init__(name)
    param name a string
    raises TypeError if name is not a string
    raises NotImplementedError if type(self) == Item
get(self, property)
Returns the results of invoking IPF's itemget(...) method on this item.

    param property a string
    raises AttributeError if the given property string is not correct for this
    Item's list
getName(self)
Returns a string equal to this item's name in IPF.

go(self, property, value, transition, delay=0.0)
Invokes IPF's itemgo(...) method on this item with the given property and
value.

    param property a string
    param value the value you want to assign to the given property
    param transition the time you want to spend in seconds to reach the
given value
    param delay (optional) delay in seconds
    raises AttributeError if the given property string is not correct for this
    Item's list
listcheck(self, property)
Checks whether the given property string starts with the right prefix for this
Item's list in IPF. For example, if this Item's list is obj, all of its property
strings start with \texttt{OBJ\_}.

\textbf{raises} \texttt{AttributeError} if the given property string is not correct for this Item's list

\begin{Verbatim}
set(self, property, value=None)
Invokes IPF's \texttt{itemset(...)} method on this item with the given property and value.

\textbf{param} property \quad a string
\textbf{param} value \quad optional parameter
\textbf{raises} \texttt{AttributeError} if the given property string is not correct for this Item's list
\end{Verbatim}

class Map(Item)
\hspace{0.5em} See Item.

\emph{Methods defined here:}

\begin{itemize}
  \item \texttt{\_\_init\_\_}(self, name)
\end{itemize}

class Obj(Item)
\hspace{0.5em} See Item.

\emph{Methods defined here:}

\begin{itemize}
  \item \texttt{\_\_init\_\_}(self, name)
\end{itemize}

class Pivot(Item)
\hspace{0.5em} See Item.

\emph{Methods defined here:}

\begin{itemize}
  \item \texttt{\_\_init\_\_}(self, name)
\end{itemize}

class Region(Item)
\hspace{0.5em} See Item.

\emph{Methods defined here:}

\begin{itemize}
  \item \texttt{\_\_init\_\_}(self, name)
\end{itemize}

class Role(__builtin__.object)
\hspace{0.5em} A function assumed or part played by an Actor. A \texttt{Role} evaluates what effect, if any, its Actor has on a given Target. Abstract class. Cannot be instantiated. Subclasses should override and implement \texttt{act(interaction)}.
Methods defined here:

__init__(self, name, actor, sensors)
Created a new Role with the given name and Sensors for the given Actor.

  param name an alphanumeric string
  param actor an Actor
  param sensors an iterable of Sensors

act(self, interaction)
Tell this Role to evaluate the given Interaction and act on behalf of its
Actor on the Target of the Interaction.
Automatically called by Interaction.perform(...).
Abstract method. Subclasses should override and implement this method to
provide the desired evaluation behavior.

  raises NotImplementedError if a subclass does not override this method

addInteraction(self, interaction)
Adds the given interaction to this Role. If this Role already has an
Interaction for the same Target, it silently replaces it.
Automatically called by Interaction.__init__(...).

  raises ValueError if interaction.getRole() does not return this Role

getActor(self)
Returns this Role’s Actor.

getInteraction(self, target)
Returns this Role’s Interaction for the given Target.

  raises KeyError if this Role does not have an Interaction for the given
  Target

getName(self)
Returns this Role’s name.

getSensor(self, name)
Returns this Role’s Sensor with the given name.

  raises KeyError if this Role does not have a Sensor with the given name

interactions(self)
Returns a list of this Role’s Interactions.

sensors(self)
Returns a list of this Role’s Sensors.

targets(self)
Returns a list of the Targets for which this Role has Interactions.
class Sensor(__builtin__.object)
    A non-empty, unordered, immutable collection of Pivots without duplicates

    Methods defined here:

    __init__(self, name, pivots)
    Creates a Sensor from the specified collection of Pivots. Each Pivot in
    pivots is added only once; duplicates are ignored. pivots is not allowed to
    be empty.

    param name  an alphanumeric string
    param pivots an iterable whose iterator returns Pivots; len(pivots) > 0
    raises AttributeError if pivots's iterator does not return Pivots
    raises TypeError if pivots does not implement __len__(...)
    raises TypeError if pivots does not implement __iter__(...)
    raises TypeError if pivots contains items that are not Pivots
    raises ValueError if not name.isalnum()
    raises ValueError if len(pivots) < 1

getAddress(self)
    Returns this Sensor's name.

class Target(__builtin__.object)
    An Obj wrapper that a Role can act upon.

    Methods defined here:

    __init__(self, obj, regions)
    Creates a Target with the given Obj and collection of Regions.

    param obj  an Obj
    param regions  an iterable whose iterator returns Regions
    raises TypeError if regions contains an element that is not a Region

getObj(self)
    Returns the Obj that this Target represents.

regions(self)
    Returns a list of this Target's Regions