KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

CAMPUS GROUP T LEUVEN

Design and implementation of an application using motion input for controlling computer games as part of physical therapy

Christiaan VANBERGEN

Dimitri VARGEMIDIS

Supervisor: prof. dr. ir. L. Geurts

Co-supervisor: ir. K. Vanderloock

Master Thesis submitted to obtain the degree of
Master of Science in Engineering Technology:
Electronics-ICT, Internet computing

Academic Year 2016 - 2017

Acknowledgements

Designing and implementing a user interface utilizing the relatively new Kinect camera is a challenging, albeit fascinating subject. We encountered many problems along the way, for which we enjoyed coming up with creative solutions. In the process, we were supported by many people we would like to thank.

This thesis would not have been possible to write without our supervisor professor Luc Geurts and co-supervisor Karen Vanderloock from e-Media lab, who provided us with this subject, invested their time and shared their knowledge and experience with us, allowing us to come up with solutions that have lead to the result as it is today.

During the year, we've had several occasions to discuss the subject of support vector machines with our fellow students Victor Martens and Alexander Kerckhofs, who helped making us aware of the pitfalls and shortcomings of the used techniques and joined us to discuss the requirements and come up with a creative solution concerning the use of SVM.

User testing is very important when designing an application with a specific audience in mind. As such, we are grateful for the useful feedback of physical therapist Dries Lamberts of Windekind, who took the time to meet us twice and evaluate both our prototypes as well as the final result. But even more importantly, he shared with us his passion for taking care of children with disabilities, their need for physical exercise and the importance of making sure this is both meaningful and entertaining.

Last but not least, we would also like to thank our parents, family and friends, not only for their unwavering support during the course of the past year while working on this master's thesis, but also for trying out our application and giving valuable feedback.

Abstract

Using games as part of physical therapy improves patient motivation to perform the necessary exercises. The developed application, called OmniPlay, allows physical therapists to choose exercises that fit the needs of the patients and record them using the Kinect 2.0 camera. When the patient executes an exercise, machine learning algorithms are used to recognize the exercise. This activates a Windows command that virtually presses the keyboard key the therapist assigned to that exercise. This makes it possible to interact with any available game relying on key inputs. The designed and implemented OmniPlay application records gestures and allows the therapist to assign a key to them, which is virtually pressed by the operating system when a gesture is recognized while playing a game. The games patients can play are not a part of this application.

Research is conducted on the subject of human-computer interaction (HCI) involving the use of the Kinect camera to interact with the OmniPlay in what is called a natural user interface (NUI). Several prototypes are designed that focus on different interaction principles and are divided into two categories: the mime pattern and the hints pattern. The chosen implemented prototype of the interface allows users to interact with the menus and items appearing on-screen without the use of traditional buttons. Interactive elements like pulling ropes are present in the interface, providing both feedforward and feedback to the user in an effort to simplify the interaction process. Additionally, to provide the user with a more streamlined interaction process, the number of times he must switch between inputting with the Kinect or with a mouse is minimized.

As part of the user-centered design approach, the interface is evaluated by conducting user tests with a physical therapist and other persons. Feedback is obtained both during the prototyping phase, as well as after having implemented the coded prototype. The results of these tests allow to conclude that it is not immediately clear how all elements are interacted with and that it takes more time to understand how interaction is possible. With a short explanation, it is possible to decrease the time required to achieve this. The application as a whole has a low learning curve as the focus on simplicity results in the user being familiar with all required actions after recording an exercise once.

Key words: exergames, gesture recognition, Kinect 2.0, physical therapy, support vector machine, user interface

Extended abstract

Kinesitherapie is vaak essentieel bij revalidatieprocessen of om de spieren te trainen voor mensen met lichamelijke beperkingen. Computerspellen kunnen deze sessies en bijhorende oefeningen aangenamer maken om uit te voeren en betekenen een belangrijke vorm van motivatie, in het bijzonder voor kinderen. Het probleem met deze spellen is dat ze duur zijn om te ontwikkelen en aan te kopen en dat de oefeningen dat een bepaald spel aanbiedt op voorhand vastliggen. Daarnaast is het voor kinderen eentonig om telkens hetzelfde spel te moeten spelen. Ze verliezen daardoor snel hun interesse in het spel. Omwille van deze redenen zijn deze spellen niet altijd zinvol en effectief voor alle patiënten in het kader van de therapie.

De ontwikkelde applicatie, genaamd OmniPlay, biedt meer flexibiliteit en laat toe om reeds bestaande spellen te spelen door middel van bewegingen. OmniPlay is gericht op kinesisten en laat hen toe om bewegingsoefeningen op te nemen die zinvol zijn voor de patiënt. Het is dan mogelijk om naar eigen keuze een knop van het toetsenbord toe te kennen aan elke opgenomen oefening. Wanneer de patiënt vervolgens deze oefening uitvoert, herkent de toepassing deze beweging en wordt de knop die eraan toegekend is virtueel ingedrukt door Windows-commando's aan te roepen. Zo is het mogelijk om met bewegingen eender welk spel te spelen dat gebruik maakt van toetsenbordbediening.

De kinesist kan oefeningen opnemen met behulp van de Kinect 2.0-camera. OmniPlay gebruikt machine learning, meer specifiek support vector machines (SVM), om deze oefeningen aan te leren. Het geïmplementeerde algoritme splitst elke beweging op in een aantal kleinere deelbewegingen en classificeert deze elk onder een ander label. Tijdens het spelen herkent het algoritme een beweging als alle deelbewegingen ervan zijn waargenomen in de juiste volgorde. Deze benadering laat toe om de rekenkost van bewegingsherkenning te beperken en zorgt voor een betere reactietijd na het uitvoeren van een beweging.

Om bewegingen op te nemen, maakt de kinesist gebruik van de grafische gebruikersinterface. Hierbij is het de bedoeling dat de kinesist deze interface kan aansturen met behulp van de Kinect-camera, zonder herhaaldelijk te moeten wisselen tussen input met een computermuis en input via de Kinect. Interactie met de gebruikersinterface gebeurt door middel van interactieve elementen. Het is mogelijk om de interactieve elementen van de initiële prototypes van de gebruikersinterface onder te verdelen volgens twee categorieën: mime-elementen en hints-elementen.

Mime omvat elementen die herkenbaar zijn uit het dagelijks leven, zoals handvaten en trekkoorden. Door een combinatie van herkenbaarheid en affordances heeft de gebruiker reeds een notie van hoe interactie mogelijk is door middel van deze metaforen, waardoor het leerproces minder intensief is voor de gebruiker. Bijgevolg moet hij minder moeite doen om te onthouden hoe de interactie werkt. Hints-elementen verwachten handelingen van de gebruiker die meer abstracte gebaren omvatten en waarbij op voorhand aan de gebruiker moet uitgelegd worden wat de effecten zijn van een bepaald gebaar en waar deze gebruikt kunnen worden, zoals het kruisen van de onderarmen in een X om bijvoorbeeld een actie te annuleren. Bij gebruikerstesten met een kinesist zijn verschillende prototypes geëvalueerd die berusten op elementen van beide types.

Bij de uitwerking van dit prototype ligt de nadruk op een interface die de acties van de gebruiker koppelt aan de functies van de interactieve elementen. Met andere woorden is het de bedoeling dat interactie met deze elementen gebeurt op de manier die de gebruiker verwacht. Zo bewegen ze met dezelfde snelheid waarmee de gebruiker zijn hand beweegt over een element en reageren ze op interacties door bijvoorbeeld van kleur te veranderen. Daarnaast reageert de interface, perceptueel gezien, onmiddellijk op gebruikersinteractie met grafische elementen en geeft de gebruiker hierover feedback.

Het geïmplementeerde prototype van de OmniPlay is geëvalueerd door middel van een gebruikerstest. De focus van deze test ligt op het gebruiken van de OmniPlay-applicatie om bewegingen op te nemen, opnames te herbekijken en spellen te spelen, gebruik makende van de opgenomen bewegingen. De test maakt duidelijk dat interactie met de verschillende grafische elementen niet altijd natuurlijk aanvoelt. Zo is het bijvoorbeeld wel duidelijk dat de gebruiker aan een koord kan trekken ter hoogte van het handvat, maar niet dat het nodig is om de hand te sluiten om zo het effect van het vastgrijpen na te bootsen. Er is echter beperkte toelichting nodig om er zelf achter te komen hoe het mogelijk is om met de grafische elementen te interageren. Nadat de gebruiker dit eenmaal zelf heeft gevonden, is de benodigde tijd voor dezelfde interactie herleid tot een minimum. Om dit initiële leerproces te versnellen, is het mogelijk om de gebruiker een korte uitleg aan te bieden over manier waarop er interactie mogelijk is met de verschillende elementen. Verder toont de kinesist interesse in het gebruiken van de OmniPlay in de praktijk en ziet hij eveneens andere mogelijkheden die deze applicatie biedt, zoals het gebruik van gebarenherkenning bij kinderen met auditieve beperkingen of een interface voor de OmniPlay die gericht is op het gebruik door kinderen.

Het resultaat van deze thesis is een applicatie die toelaat om bewegingen op te nemen en deze te gebruiken om met een bestaand programma of spel te interageren. De OmniPlay werkt onafhankelijk van zowel de gekozen bewegingen als de toepassing die zij aanstuurt met deze bewegingen. Dit maakt het mogelijk voor de kinesist om voor de patiënt relevante bewegingen op te nemen. Opnemen is mogelijk via interactie met grafische elementen. Deze elementen communiceren aan de kinesist middels feedforward hoe deze interactie kan plaatsvinden. Er is een leerproces tijdens de initiële interactie met de interface, maar door het gebruiken van elementen die in het dagelijks leven kunnen voorkomen is het mogelijk om na enkele pogingen zelf te achterhalen hoe deze interactie mogelijk is. Door een minimum aan verschillende interactieve elementen, is het mogelijk voor de kinesist om na het opnemen van een beweging reeds vertrouwd te zijn met deze elementen en kan hij de OmniPlay autonoom en zonder verdere toelichting gebruiken.

Contents

List of Figures

List of acronyms

FPS	Frames Per Second
GUI	Graphical user interface
HTA	Hierarchical task analysis
IDE	Integrated developer environment
NUI	Natural user interface
SVM	Support vector machines
UI	User interface
WIMP	Windows, icons, menus, pointer
WPF	Windows presentation foundation

Chapter 1

Introduction

There are many different reasons for requiring physical therapy. They vary from recovering from a car crash to being born with physical disabilities. The goal of these therapy sessions is respectively to make sure that the patient can recover as much as possible from his injuries or to train the muscles, preventing the situation from deteriorating.

Especially for children, physical exercises performed during therapy can be demotivating. Combining these exercises with playing computer games leads to an increased willingness to continue with the exercises [1].

The purpose of this thesis is to present a more sustainable solution to this problem in a way that offers more flexibility to the therapist, both in terms of the exercises that need to be done by the patients and the games they can play by performing the exercises.

1.1 Discussion of the problem

Patients often see exercises as a part of physical therapy as being fatiguing, monotonous and tedious. This problem is even more prominent with young patients and can quickly demotivate them, especially when the exercises are uncomfortable or painful to perform [2].

Computer games already exist that can support physical exercises. For instance, there are games that run on Microsoft's XBox console and accept user input through the Kinect 3D camera, or Nintendo's Wii console that use a controller with accelerometers and gyrosensors. Additionally, games exist that can run on a computer, using any combination of sensors for user input, and are specifically made for use by physical therapists and their patients. An example of such a game is *Kungfu Kitchen*, which is a collection of mini-games that can be played for instance with a Kinect camera [3]. However, these kind of games have in common that they are very static by nature and are often not suited for the specific needs of a patient [4]. For instance, a game that focuses on arm movement is not very effective at exercising the leg muscles of a patient. In other words, concerning the therapy, this game is meaningless for a patient that had leg surgery.

In addition, these static games have fixed input gestures that patients have to perform in order to control the action on screen. Even if the gestures perfectly fit the exercise requirements for a specific patient, the decrease in attractiveness of playing the same game over and over again is problematic and loses its long-term effect. Physical therapist Dries Lamberts has experience with children with disabilities and uses computer games as a form of exercise. The initial reaction of the children to these games is positive. They are more engaged in doing physical exercises and feel motivated while doing so. On the downside,

this effect only lasts until the novelty of the game wears off. After that, the children lose interest in the game and, as such, in performing the exercises.

As games are expensive to develop [5], there is no wide variety of motion games that are tailor-made for people with specific needs and at the same time offer varied gameplay mechanics to remain interesting over longer periods of time. From the therapist's point of view, the game needs to support the right type of exercises and the game itself has to be considered fun as well by the patients for them to keep invested in it. If any of these conditions are not fulfilled, therapists can be less inclined to purchase the game and patients can be less motivated to play it. Because of these reasons, it is questionable if producing these kind of games is economically sustainable.

1.2 Purpose of the research

This thesis focuses on the design and implementation of an application called OmniPlay. OmniPlay provides a more flexible and dynamic solution to the problem discussed above. It lets the physical therapist choose what exercises the patient has to do and how these can be used as an input to control any computer game that uses keyboard input. All of this can be done without requiring the therapist to have any programming knowledge.

The goal is to have an application the therapist can control by mainly using the Kinect camera. To provide the user with a more streamlined interaction process, switching between Kinect input and mouse input is minimized. As such, the therapist can stay in the same spot in front of the camera. Interaction needs to feel simple, efficient and natural to do. This minimizes the setup time before a patient can start playing. It is necessary to research the type of interface that is required to meet these objectives, in addition to finding out the easiest way to interact with an on-screen application using a camera for input.

Chapter 2

Related work

The literature study focuses on the aspects of human-computer interaction (HCI) related to gesture-based input and natural interfaces. This is followed by a closer look on research done that is directly related to using exergames as part of physical therapy.

2.1 Human-computer interaction

Gesture-based interfaces gain popularity due to the advancing technology of sensors and processors [6]. Because of this, technologies like the Kinect camera become more affordable and accessible for individuals. While interfaces supporting gesture input are referred to as natural user interfaces (NUI), this is more of a marketing term than an actual scientific designation. Natural interfaces don't feel natural to interact with just because they rely on gesture-based input. The user does not actually touch the object shown on the screen to be able to interact with it naturally, so the feedback a user can get is limited [7]. Direct manipulation of objects that interact with an application offers more natural feedback [8]. The downside is that additional objects are required for interacting with the application.

Notwithstanding the limited feedback when not using additional objects, gestures are powerful tools for human-computer interaction. However, gestures as an input offer a wide variety of options and with increasing popularity, a wider variation of interfaces and gestures to interact with them emerge. This is problematic as users need to get familiar with a gesture-based interface each time they encounter a new one. The full potential of this kind of interfaces can only be achieved when some kind of standard is developed and widely used [7]. Frameworks based on reality-based interaction can be used to unify different research areas concerning interface design and to compare these designs to one another [6].

When designing an interface and ways to interact with them, it is essential to make clear what actions are required through feedforward and what the results of those actions are through feedback. Because of that, feedback and feedforward are powerful tools. Designing a gesture-based interface comes down to trying to have natural coupling between action and reaction. Natural coupling can be evaluated by six aspects: time, location, direction, dynamics, modality and expression [9]. The time aspect requires that there is no perceivable delay between the execution of an action and the result shown on the screen. The direction and location of action and reaction have to match as well. The modality is tightly coupled to the dynamics and direction, as it must be possible to visually perceive the result of an action. Modality however is not limited to visual cues. Auditive feedback on certain actions can improve the coupling between action and reaction as well. Visual representations of the reaction must be similar to the action in terms of dynamics, meaning that the speed and acceleration of both have to match. The aspect of expression is also related to the dynamics of the action. This means that the interface has to visually represent the way the user performed an action. If the action is done quickly and not very accurately, it must be reflected by the

interface. If feedback as well as feedforward are able to support these six aspects, the user perceives the interface as more intuitive to interact with and action and function become more closely related. The more aspects are fulfilled, the higher the natural coupling is. [9]

2.2 Exergames as part of physical therapy

Using games as part of physical therapy has been the subject of many research papers [4][10][11][12][13]. Since it is found that exergames improve motivation [1] and offer physical, social and cognitive benefits, they are more frequently used by physical therapists [14][15].

While the reliance of the patient on the therapist can be reduced by using exergames, the rate of rehabilitation does not increase [16]. Because of a decreased reliance on the therapist, he can focus on supporting the patient during play [2]. Since these games can be a tool of aid to the therapists, it should not hinder them in any way, nor increase the amount of work or effort they put into the preparation of a therapeutic session. For this reason, setting up everything needed before starting to play should not take up too much time. This includes but is not limited to: the time to set up the used sensors, the time the application takes to start up and the time needed for the application to process data.

Within the same mindset of making life easier for physical therapists, the application can be a useful source of information for the therapist, as it can provide him with details of the progress his patient is making either during one session or over the course of long-term therapy. Again, this allows the therapist to focus on the essence of the therapy instead of on taking notes, keeping close track of how the patient is doing [2].

Chapter 3

Design

The main focus of OmniPlay is to record gestures and use them as an input for controlling computer games. In addition, it is important to simplify the setup process for the therapist, ensuring that he does not require any programming experience or knowledge about the underlying system.

The graphical user interface (GUI) allows the therapist to interact with the OmniPlay application, providing him with feedback and feedforward on inputting gestures for the patients. Gesture recognition is done by means of machine learning using support vector machines (SVM).

3.1 Description of the application

OmniPlay uses Microsoft's Kinect 2.0 camera as an input device. The principle is that the user can train the program to be able to distinguish multiple postures or gestures from each other. By assigning a virtual keyboard button to each posture or gesture, games can be played by performing them in front of the Kinect. In order to set up the application, there are a number of steps to be taken and guidelines to keep in mind.

3.1.1 Principle

When playing a computer game the conventional way, a person can interact with the game by pressing a keyboard key. Pressing the key then results in an action on the screen (see figure 3.1).

Figure 3.1: *Overview of the interaction between a person and a computer game under conventional use*

The OmniPlay application can be seen as an additional element between a person and the keyboard key for controlling a game. In short, OmniPlay allows a person to virtually press a key by performing a

gesture chosen by the physical therapist (see figure 3.2). This virtual key press is executed by the OmniPlay application, invoking the appropriate Windows operating system command. This invocation has the same effect as pressing the specified key on a physical keyboard. Henceforth, this action is referred to as pressing a virtual keyboard button.

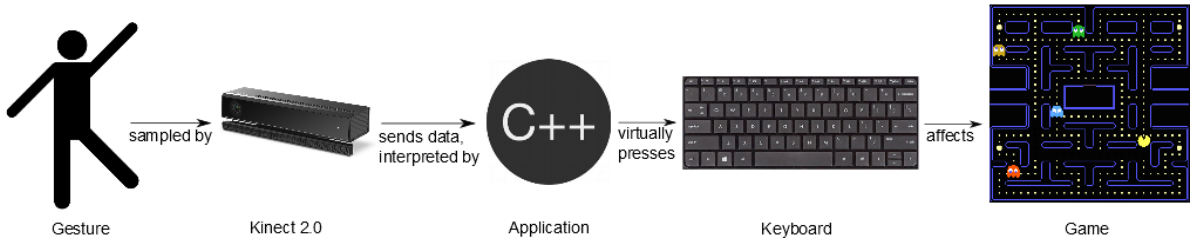

Figure 3.2: *Overview of interaction between a person and a computer game using the developed application*

More in detail, the physical therapist comes up with gestures that fit the needs of a patient. He uses the Kinect camera to interact with the application via a GUI. Next, the therapist uses the application to record what gestures need to be done by the patient. Using all of the recorded gestures, an SVM model is created, which is used to predict which gesture is performed by the patient while playing a game.

The therapist assigns a virtual keyboard button to each of the gestures. This means that when the patient mimics one of the therapist's gestures, a virtual keyboard button is pressed, as explained before. To start playing a game, the therapist can choose any game that uses keyboard input or let the patient choose his preferred game. This can be any type of computer game and includes, but is not limited to: browser games, games that need to be downloaded and installed, pre-installed games that come with the operating system,... If OmniPlay is running in the background while a computer game is opened, performing a gesture virtually presses the button that is linked to the gesture. By doing this, the patient can interact with the game.

By mapping gestures to virtual keyboard buttons, a vast amount of existing games can be played using gesture-based input. It is however limited to button presses. Pointing with the cursor like when using a mouse is not supported by the OmniPlay application. The reason is that every gesture performed is seen as one single action. The gesture of pointing to a specific spot on the screen could overlap with one of the gestures the therapist wants to use. This results in undesirable behavior and negatively impacts the gaming experience for the patient.

3.1.2 Setup

Before the application can be used, the user has to download and install the necessary drivers in order to use the Kinect 2.0 camera. To connect the Kinect 2.0 camera to a computer, an additional adapter is needed so it can be connected using a USB 3.0 port (see figure 3.3). The use of the Kinect camera requires a socket. For using the Kinect, it is necessary to install a driver. This comes as part of the *Kinect for Windows SDK 2.0*. Microsoft's download page [17] states the system requirements for using the Kinect 2.0 camera. These requirements include: a 64-bit processor, dual-core 3.1 GHz or faster processor, USB 3.0, 4 GB of RAM or more, graphics card that supports DirectX 11, Windows 8 or 8.1, Windows Embedded 8 or Windows 10.

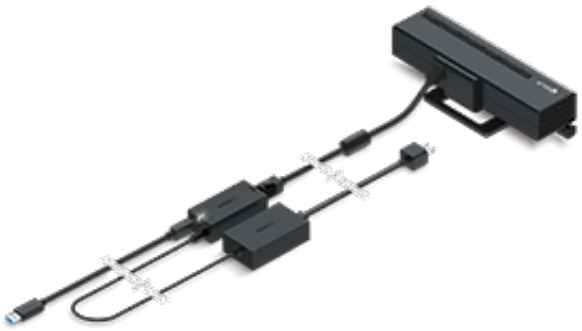

Figure 3.3: *The Kinect 2.0 camera connected to the adapter*

The camera has to be placed horizontally in order to function correctly. The distance between the user and the camera may vary between 1.5 and 4 meter, depending on the height of the user and the size of the used screen (see figure 3.4). The camera can be tilted upwards or downwards. The chosen angle depends on the height of the camera. Both can be chosen freely by the user, but the user has to ensure that he is fully visible at the position he stands. The GUI allows him to verify this. A possible configuration that works well is to have to Kinect camera pointing straight ahead on a table with a height of approximately 0.8 meter.

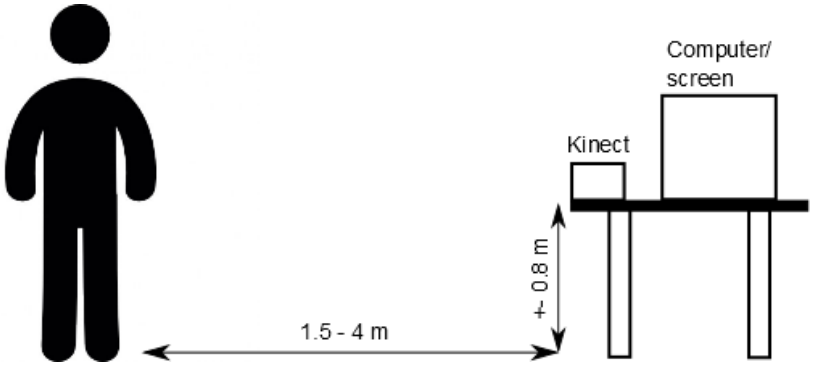

Figure 3.4: *Setup for the Kinect camera*

3.2 Graphical user interface

A large part of this thesis is focused on the design and implementation of a user-friendly interface that can be controlled via input from the Kinect 3D camera. The reasoning behind this is that the user does not want to run back and forth from the standing position in front of the Kinect to the keyboard. Many interfaces have been designed for use with the Kinect, but they are generally only used to navigate to the game and are therefore made up of standard WIMP elements (window, icon, menu, pointer) such as buttons and scrollbars to select your option. In this section, the process of developing the user interface and the properties of the coded prototype are discussed.

The goal in this part of the thesis is to find a new way of interacting with a user interface using the Kinect camera that feels natural to the average user. This despite the claims by Norman [7] that natural user interfaces (NUI), the term that is often used to describe interfaces controlled with 3D cameras, are not natural because there is not the same feedback as when acting on objects in real life. Standard WIMP elements definitely do not feel natural in a NUI. Pointing and clicking on buttons with a pointer may have

already become second nature to modern humans, but it is a different thing when they are large and float in the sky or are fixed to a wall and the user has to slap them or close their fist over them without getting any inherent feedback as Wensveen [9] called it. For this reason, elements such as push buttons, scrollbars and floating windows are avoided. Additional hardware is not desirable and voice control is also not considered as it is an area where plenty of research has been conducted already. Implementing such a system is impossible considering the limited time for this thesis. If possible, we wanted to find new ideas on how to control a NUI.

3.2.1 Hierarchical task analysis

To figure out what steps the therapist needs to perform to use the program, a hierarchical task analysis (HTA) is performed. This is a process in which all necessary tasks and subtasks are put into order to get a sense of the properties and elements that the UI must possess. This started as a very technical plan of the tasks that are needed, but was quickly distilled into a simplified visual plan that can be shown to the user to explain how OmniPlay can be used. This plan can be seen in figure 3.5.

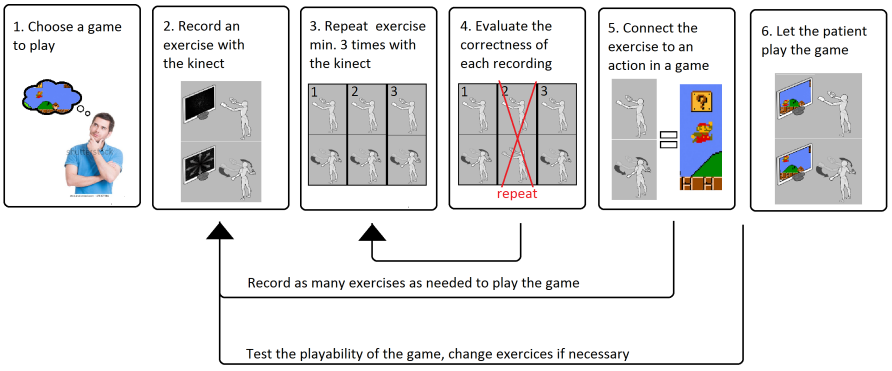

Figure 3.5: *The simplified HTA plan*

3.2.2 Experimental brainstorm technique

A brainstorm session with the targeted user, the physical therapist, would give us fresh ideas on how the user expects to interact with a user interface using his body or voice. The expectation was that two distinct observations would be made: how the user wishes to interact with the user interface and how he expects the program to function.

The idea is to prevent the subject from being influenced by our ideas or by obvious solutions. Initially, only the basic concept of OmniPlay and the abilities of the Kinect camera are explained. Then the subject is asked how he would imagine every aspect of the application. A few questions from the script are: What information would you expect to see on the screen? How would you start a recording? What would you want to see during recording?

To find out whether this experimental brainstorm technique would yield any viable results, brainstorm sessions with our relatives were conducted first. Two iterations were performed with four subjects, the first test was with two males and the second was with two females, of both genders there was one person around 25 years old and one around 55 years old. Each subject was interviewed completely separately to minimize the influence they had on each others ideas.

The sessions were very taxing for the test subjects, as the concept we tried to introduce was too abstract and foreign to them. This causes them to be shy with proposals, even after being ensured that there are no bad ideas. It seemed to give them the feeling that they are not smart enough to make good proposals. As a result they fell back on proposing familiar UI elements or concepts such as push buttons or voice control. Getting locked into a mindset that we were designing games to play with the Kinect was also a minor issue, thereby answering the questions with that in mind, which was not the point. After two iterations with minor changes between them, we decided not to subject the physical therapist to this kind of questioning. Setting up a meeting would waste too much of his time with little gain to us. Worse, it might instill the physical therapist with a feeling of dread for our next meeting which might reduce his level of co-operation.

Instead the decision was made to brainstorm with each other and our supervisor to develop a few prototypes that we can show to the physical therapist to see which one seems the most user-friendly. The two major interaction patterns that came out of this brainstorm session are, for the purpose of this thesis, named the hints and mime patterns. In the rest of this chapter, the mention of user interface (UI) elements refers to buttons, scrollbars or any other element that the user can see on the screen. When discussing the different concepts, we refer to a movement or position that is taken by the user to activate a function in the program as a sign.

The hints pattern implies that the screen is largely devoid of UI elements and actions are performed when the user does a certain movement or takes a certain position. For example, the user forms a cross with his arms to delete a recording. It is called the hints pattern because it is similar to how a game of Hints is played. This is a game where one player tries to depict a concept or activity without using his voice. The other players have to guess what he is trying to say. In this case, the user is the player who is depicting the concept or activity and the program needs to guess what the user means. There is a slight difference here in that the program already knows which signs the user can choose from. Though not always possible, it is better when a sign activates a function in the program that is similar to the concept or activity that this sign refers to in everyday life, as in the last example where an X is related to delete because it generally means bad, wrong or unwanted in Western culture. However, this might be different for other cultures as Norman [7] pointed out. The implied lack of feedforward means the user should learn all of the signs at the start of the application, putting a large strain on the user's cognitive ability which is not ideal. One could compare it to the shortcut keys on a keyboard such as CTRL+ALT+DEL. There is no indication on the UI that this key combination performs a special function. Norman [7] made these complaints and also stated that it is hard for the user to find out what he is doing wrong if he does the sign incorrectly. Alternatively, all possible actions could be displayed on the screen around the user with a depiction of the sign. Though this would put less strain on the user's cognitive abilities, he would still have to focus on these pictures and try to figure out what sign they depict. But what if the sign is a movement? It is impossible to depict it statically, so how can you make this clear without distracting the user with constant moving pictures? A more significant problem might be the fact that positions and movements might be recognized by accident. An undo movement could partly solve this, but you also need a redo button to compensate for any accidental undo-activations. The only real benefits to using the hints pattern is that activating an action can be done from anywhere and it could be faster once the user has mastered the signs. Though this depends on the ease of performing the signs and the delay between a sign and the corresponding action in the program.

The mime pattern, as the name suggests, is the act of mimicking an action that is needed to act on everyday objects such as opening a door. But where a mime artist does this without any indication of that object being there, in this case an image of, for example, a door is displayed on the screen. The technical term for such a UI element is a metaphor because the function is similar to what a user would do with the object in real life. The power of this pattern is that the user should immediately recognize the UI element and know how to interact with it because it has the same affordances as the object in real life. To achieve the best effect, the UI elements that are chosen should require signs that the user has done

frequently already, so that doing them feels natural and familiar. Linking a sign to a function that is similar to the use of that sign in real life, such as going to a different screen when opening the door, improves the interaction, but it is less important than in the hints pattern. Wensveen [9] already said that for this to work correctly, feedback is crucial. The reaction of the UI element must match the expectations of the user. Otherwise, he will be confused and the flow of the program is interrupted. Disadvantages are that the user cannot activate every action from anywhere and it might be slower because of that. Both Wensveen [9] and Norman [7] state that inherent feedback is lacking here. The user does not feel the UI element with which he is trying to interact.

One should prevent mixing the two patterns because the user expects to interact with everything on the screen when using the mime pattern. If one action is activated using the hints pattern without indication, the user will be confused when he has forgotten the sign. Even if there is an indication, the user will likely try to interact with the indicator of the sign as a button, which will not have the desired effect. A combination with regular WIMP UI elements, such as push buttons or scrollbars, is possible for both patterns, but it combines much better with the mime pattern. This is because the user is already interacting with elements of the UI. Pointing at a specific area in the UI does not fit well in the concept of the hints pattern and detracts from the idea of being able to do every action everywhere. To sum up, in the mime pattern the behavior of the UI elements are simple and obvious but possibly more inefficient, like pulling a handle on a slot machine to spin the slots. The hints pattern is more abstract and complex, but allows more efficient interaction once mastered.

3.2.3 Prototypes

With these patterns in mind, the design of the prototypes focuses more on designing UI elements and methods with which the user can interact. They have to be simple, fast, ergonomic, feel natural and not be taxing to the user physically or cognitively while being practical and not easy to activate accidentally. The regular way of testing a concept UI is by doing paper prototyping, which means that the user is presented with a paper version of the UI. Whenever the user touches a UI element the designer manually changes the paper prototype to reflect the activated action. This kind of testing is not suited for the goal of this project. It is more useful for assessing the placement and shape of UI elements and the general flow of a program. What is needed here is a way to interact with the prototypes in a way similar to how the real program would work, but without having to code the whole program. For that reason, we took two example programs from Microsoft's Kinect SDK and made some adjustments to display the user on-screen, in front of the prototype.

When designing a motion controlled UI, it is very important to let the user know what the camera sees of their signs. The manner in which this feedback is given has a great impact on how natural the user's experience with the UI is. There are four viable options:

1. A real image of the person as the camera films it (figure 3.6 a).
2. A semi-transparent shadow of the user (figure 3.6 b).
3. A puppet that copies the user's movement (figure 3.6 c).
4. No body is shown, only images that portray the position and state of the hands (figure 3.6 d).

A real image of the person is the most identifiable for the user, because it is literally an image of himself. This doesn't mean that it is the best option to use in interacting with the UI. Though it has the advantage that it is clear which movement the user carried out, which is good for recording and replaying gestures.

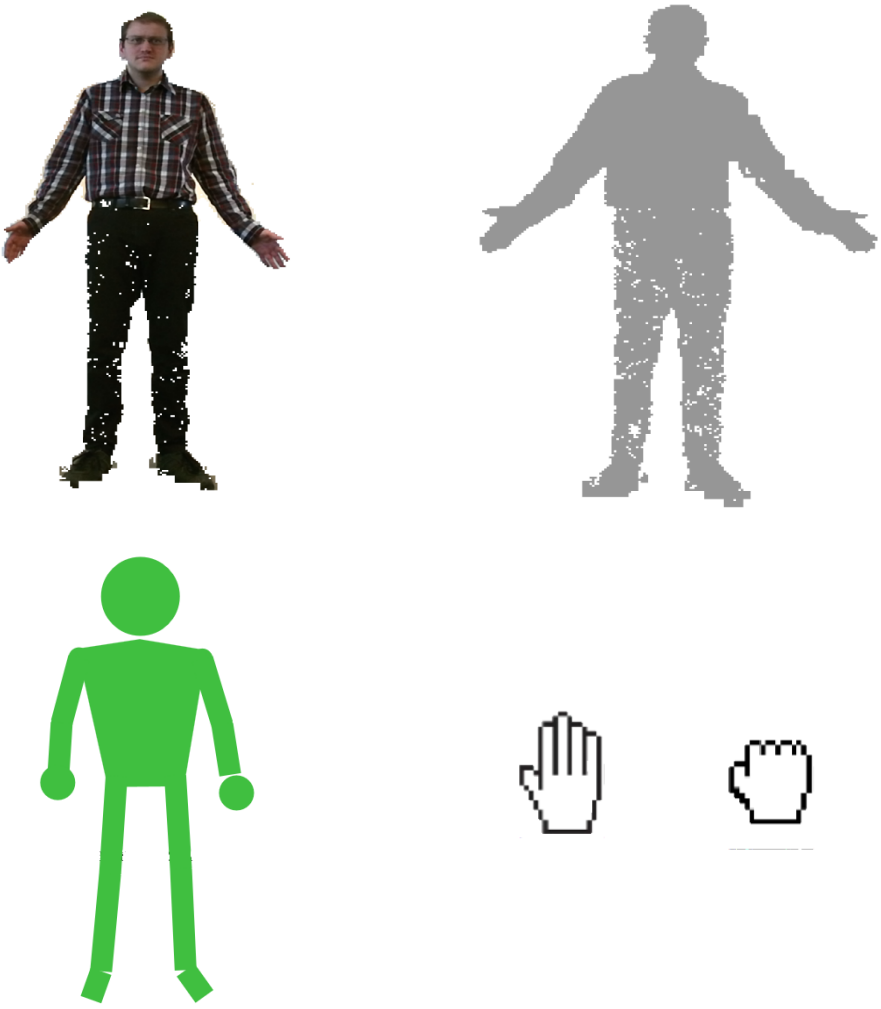

Figure 3.6: *(top-left) a) UI with a real image of the user, (top-right) b) UI with a semi-transparent shadow, (bottom-left) c) UI with a puppet, (bottom-right) d) UI with only hands visible*

The most important issue is that the image is taken from the front, so the image of the screen is facing away. Unless the UI elements are drawn on top of the user's image, it is hard to imagine for the user that he is grabbing a handle or pushing a button on the UI. Placing the Kinect camera behind the user to solve this problem is technically not possible because an important part of the Kinect's person recognition is based on recognizing the facial features. This would also require much space to operate because a distance of at least 1 meter from the screen in front of the user and at least 1.5 meter from the Kinect at the back needs to be maintained. When the UI elements are drawn behind the user, he will also obscure any elements directly behind him. Another issue that became apparent during the user tests was that most people are not comfortable with an image of themselves being displayed on the screen, which detracts from the user's experience. The physical therapist we interviewed, who works in a school for disabled children, also noted that this might cause legal problems with parents who do not want unauthorized footage of their child being stored. A final technical issue is that displaying the real image of the user is very resource intensive and in this implementation it would sometimes cause the program to lag.

A semi-transparent shadow can solve the legal issue and the discomfort experienced by users. It also solves the problem of the image facing away from the UI, because there are almost no features that indicate which way the image is facing, making it look like the shadow is facing towards the UI. By making the image semi-transparent, any UI element behind the user can still be seen. A disadvantage is that it is harder to identify subtle differences between similar recordings. It is also still very resource intensive. Trying to optimize this is beyond the scope of this thesis.

A puppet that copies the movements of the user has similar benefits to the shadow and can be transparent or non-transparent. It is far less resource intensive, but makes it even harder to distinguish similar recordings from each other.

Showing no body and only hands can obviously only be used to interact with the UI and not for the recording of a gesture. It removes all distractions from the screen and shows only the information that is essential for the user to interact with the UI. But because of this the interaction might feel less natural to the user.

The shadow type of representation has the most advantages so it is chosen to make the most prototypes with. Though the more abstract prototypes use the representation that only shows images of the hands of the user to see whether a more abstract representation really detracts from the user experience.

In this thesis, three prototypes are discussed, ranging from obvious UIs using mostly mime pattern elements and metaphors to more abstract UIs where more elements of the hints pattern and standard WIMP concepts are used with less mime pattern elements. The first two prototypes are only the screen in which the user can record one gesture multiple times to provide the data for the SVM (step 2 to 4 in figure 3.5). They use a shadow image of the user to control the UI. The other prototype is a menu to browse to a recording and replay it while the user only sees images of his hands. There are also other prototypes but only these three are discussed to show the kind of elements that were experimented with. These can be ordered as follows:

1. A UI with mostly mime pattern elements and metaphors.
2. A UI that combines mime patterns with WIMP elements.
3. A UI that combines the hints pattern with WIMP elements.

The first prototype (see figure 3.7) mostly uses the mime pattern and metaphors. The orange color is used to indicate a possibility for an action. On the right the door on the side says exit and would swing open when the user grabs the orange handle and pulls to the left. On the top of the screen, a collection of recordings can be seen pinned to a scrollbar with orange thumbnails. The arrows on the sides imply that the recordings can be moved. The user would grab the orange indicator bar to browse the recordings. To delete a recording the thumbnail of that specific recording can be grabbed and dragged to the recycle bin to release it there. The user can display the recording by dragging it to the projector on the left side of the screen which would display the recording on the projector screen where the large ”1” is displayed. In the middle, a camera with its lens pointed towards the user is displayed, by moving his hand forward on the “opnemen” button the user could start a recording.

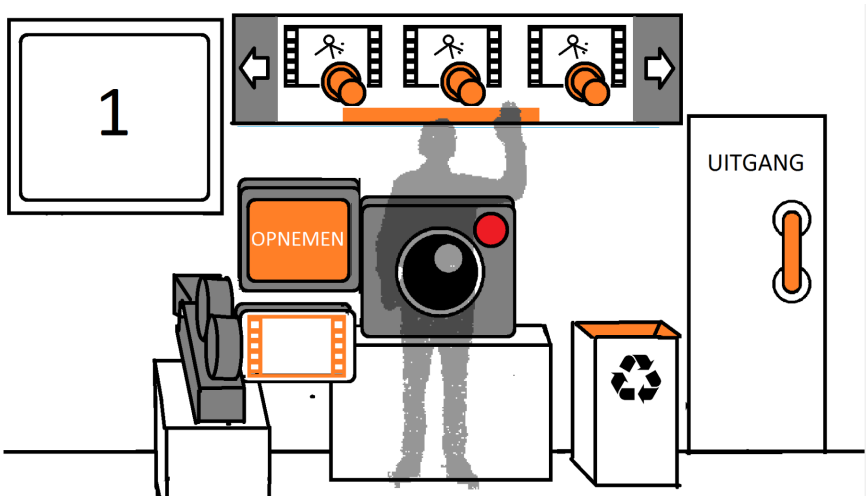

Figure 3.7: *The first prototype screen using metaphors*

The second prototype in the list combines mime pattern elements with standard WIMP elements to streamline the process (see figure 3.8). On the right of the screen is a lever that is activated by grabbing the orange part of the lever and pulling it downwards, causing the user to leave this screen. The record button is a push button that the user activates by moving his hand forward over it. This action activates the record screen (see figure 3.9). The window shows what is going to be recorded. The red one indicates where the program will count down from three until the recording starts, allowing the user some time to get to his starting position. All other UI elements are grayed out to indicate that they are not available to interact with. Once the recording is finished, the program returns to the screen in figure 3.8. On the left is a scrollbar that contains elements, which represent previous recordings of which they show static representations. On the right of it, there is an orange scroll wheel with which the user can scroll through the scrollbar by simply hovering his hand over it. When the user stops scrolling, the elements are locked into a position (see figure 3.8). The user can slide the top element into the “TRASH” or “PLAY” area by hovering over it in that direction to play or delete the element.

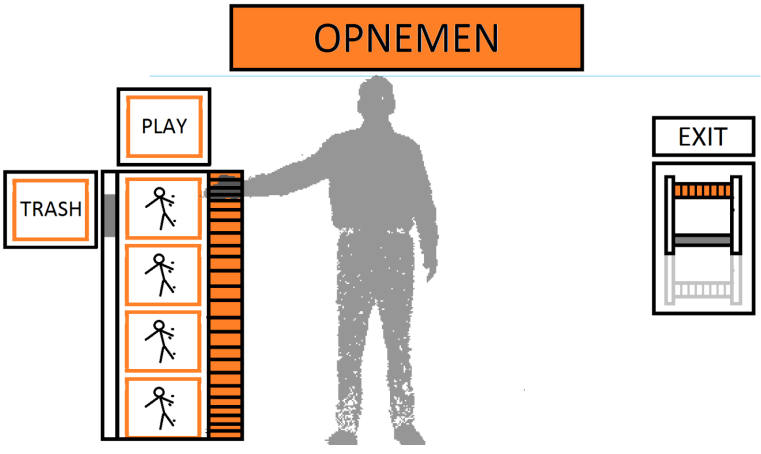

Figure 3.8: *The standard screen of the second prototype focusing on WIMP elements*

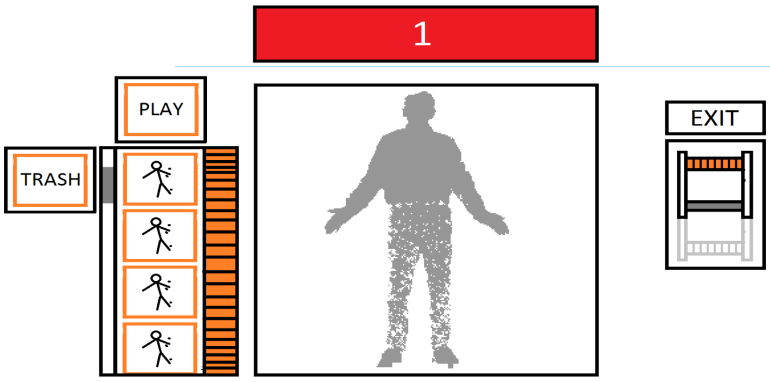

Figure 3.9: *The record screen of the second prototype*

The third prototype uses the hints pattern. The user has to close his hand over the element with which he wants to interact. This opens a WIMP element, called a context menu, with which the user can interact. The user can only see images of his hands. It starts off with a small tutorial screen (figure 3.10). When the user closes his hand within the orange box, the context menu opens (figure 3.11). When the user slides his closed fist towards any of the elements, it indicates that the option is chosen (figure 3.12). Throughout the prototype the user can interact with the chosen recording as seen in figure 3.13. In this instance, he can choose between playing the recording, deleting it or canceling the current menu.

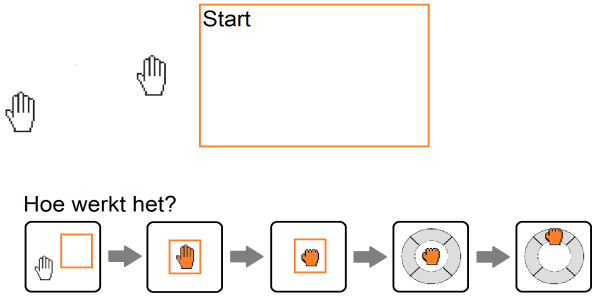

Figure 3.10: *The first tutorial screen of the third prototype*

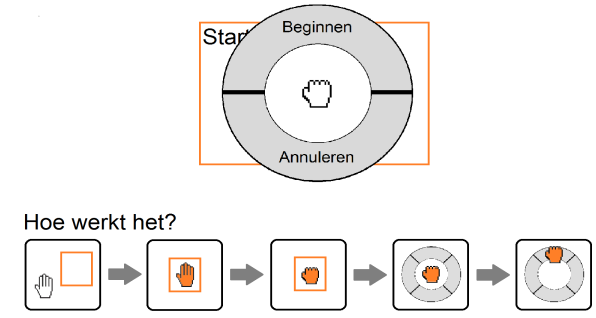

Figure 3.11: *The second tutorial screen of the third prototype*

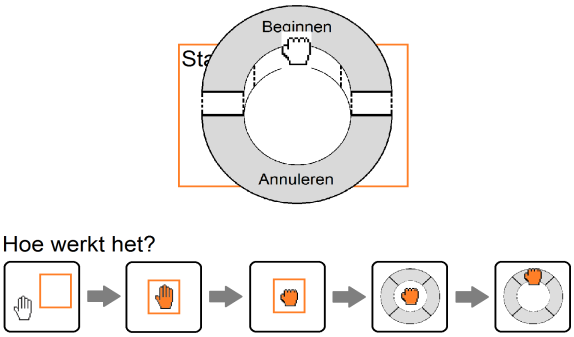

Figure 3.12: *The third tutorial screen of the third prototype*

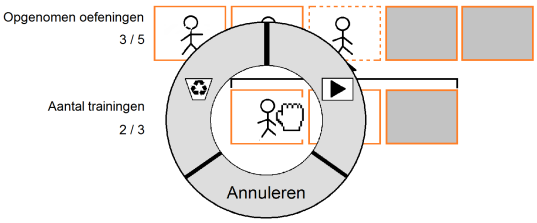

Figure 3.13: *An example of how the user can interact with an element of the third prototype*

3.2.4 Prototype user test

The main goal of this user test is to see how the user interacts with the different prototypes and how fast can he do the required signs. Additionally, we verify if they feel natural and which one of the prototypes is favored by the user.

For these tests, the Kinect is placed on a table of regular height and the user is asked to stay at a distance of 2 to 4 meter from the camera. The screen with the UI is placed directly behind the Kinect camera.

The user is first introduced to the concept and flow of our program through the use of the HTA plan as seen in 3.5. Then, a small demo program is presented showing the capabilities of the concept and the Kinect to give him an idea of the end goal and the possibilities with this concept. After some more introductory questions, the user is assured that he is never at fault and that we are responsible for any misunderstandings he might have. Before the actual tests, permission is asked to film his proceedings. During the tests, the user is reminded that we like to hear his thoughts on what he sees and thinks.

Then, the user is asked to interact with the prototypes using the Kinect. To mimic the functions of the UI, a variation of the Wizard of Oz method [18] is used to change the screens whenever the user performs the required action. Some prototypes still require some explanation, but generally, the user is only assisted when he is completely stuck. At the end of each prototype, an explanation of the intent of each UI element is given. After all the tests, each prototype is gone over again without the Kinect to reflect about the performance of each prototype. The user is asked to select a personal favorite.

In total, four candidates were subjected to these tests. One person is a physical therapist and three are relatives or friends. However, this paragraph focuses on the results with the physical therapist supplemented with the results of the others. Thereby, the physical therapist is referred to as *the user* and the others as *the other users*. It is important to mention that the user has previous, albeit limited, experience with Kinect games which sometimes influenced his reaction. For instance, he did not know that he could move his hand forward to mimic pushing a button because in his previous experiences, this was never used. For the first prototype (figure 3.7), the user was trying to interact with UI elements that were not intended to be interacted with, he later indicated that he did not see the metaphors as they were intended. The screen was too hectic and he didn't see which ones were buttons and which were not. He also found that some signs required him to reach too far. Most of the other users also made the same remarks about the first prototype. A rope in one of the prototypes on which the user could pull was clear to him, but he had no idea how to interact with the scroll wheel, as seen in the second prototype (figure 3.8), to scroll through the scrollbar. In one of the prototypes, the screen changed based on where the user was positioned. This caused a great deal of confusion. The same goes for the other users. The lever on the third prototype, as discussed above, to leave the screen was clear but he did not like it, as the metaphor was too foreign to him. The user was quick to pick up the controls of the last two prototypes that only showed the hands, but he later indicated that the representation of the hands only didn't feel right to him and that he much preferred to see all his options on the screen. Other users indicated that they expected the UI to react as if it were a touchscreen. When asked how they preferred to see the themselves on the screen, all users experienced discomfort with the real image of themselves and the therapist made the comment about legal issues with the patient as seen before in the discussion of the user's representation. They all preferred the shadow representation. Remarkable was that all the users chose the third prototype, as seen in figure 3.8 earlier on, as their favorite, mostly due to the simplicity and clearness of it.

The conclusions are that the most important characteristics are that the screen needs to be simple and straightforward with all of the options displayed on the screen permanently. Modes dependent on subtle parameters, such as the user's position, need to be avoided. The preferred representation of themselves is the shadow. For the purpose of simplifying the UI, regular WIMP elements are still allowed to be used without interfering with the user experience, though clearer affordances for elements that do not look like any object in real life are needed.

3.2.5 Description of the interface

The end design looks as seen in figure 3.14. Starting from the left, the user can see a gray window in which a recording can be replayed, this is a typical WIMP element because it does not have any elements to it that can be seen as a metaphor to anything the user is familiar with in real life. Right of this there is the rope, which is a very distinct mime pattern element. Further right is a simplified version of the scrollbar containing the recordings with the currently selected recording highlighted in orange. The user can slide this element to the "DEL" bar to delete the element. The black bordered box at the feet of the puppet represent the ground on which the puppet walks so it does not appear to float in mid-air.

Though we had previously established that the shadow was the most natural way of representing a user in the interface, performance issues forced us to resort to using the puppet image. The reason why the puppet is not semi-transparent is because then the joints are more visible than the rest because two semi-transparent images are drawn on top of each other which combine into a less transparent color. This breaks the unison of the puppet's coloring making it seem less human which would be a distraction that detracts from the experience of the user. Another detail of the puppet is that the centerpoint of the spine is fixed in the vertical direction but not in the horizontal direction. This eliminates any dependency on the position of the camera so that, as long as the user is within the field of vision of the Kinect, he

can always reach every UI element on the screen. This measure introduces two problems. The biggest problem is that when the user tries to squat the puppet will lift its feet instead of lowering its torso, this might cause confusion and does not feel natural. The other problem is that the user cannot reach any UI element that is placed where it could only be reached by bending downwards. This is not seen as a big problem because such a sign is very taxing for an adult to perform and thus not wise to use in a UI. The puppet is free to move in the horizontal direction but there is an offset on where the puppet is drawn so that the puppet appears to stand between the rope and the scrollbar when the user is standing right in front of the Kinect. In a normal setup the screen will stand behind the Kinect so that the user can look straight ahead when interacting with the UI. The depth coordinates in 3D space are also set to a fixed distance before conversion to a 2D screen to ensure that the puppet is always drawn to the same size no matter how far the user is standing within the field of vision.

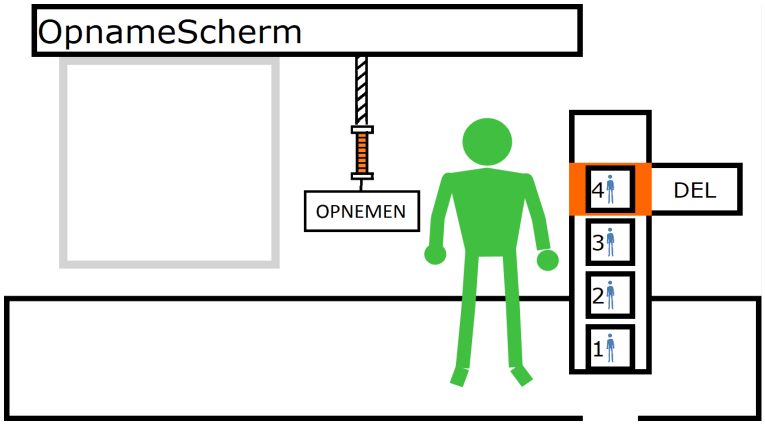

Figure 3.14: *The regular view of the end design of the UI*

An important aspect of the UI is the color coding that is consistent throughout the UI except on the puppet itself though it should be apparent that the puppet is not much more than a representation of the user. This should give the user extra feed forward about what the consequence of his action will be. Orange is indication of interaction possibilities, green means a hand is hovering over the element, blue gives information about what happened or will happen after an action and red indicates a more serious action is about to be activated. The exact meaning of all these color codes will become clear during the description of the UI.

Since it is important for the user to know when the system sees that it is grabbing the rope, the UI provides feedback to the user by coloring the hands green when they are open and red when the user makes a fist.

In figure 3.15 a can be seen that when the user hovers his hand over the orange handle of the rope the handle turns green indicating that it can now be grabbed. When the user closes his hand, the handle turns red to show the user that he is pulling it is shown in figure 3.15 b. The dynamics for pulling the rope are simple, between the starting position and the end position, which is below the starting point, the rope will make the same relative movements in the vertical direction as the user makes with the center of his hand. In this way the interaction is linked in time, speed, direction, dynamics and location of the user's hand. When the rope is near to his end position the record screen is activated. There is no progress bar to indicate how far the user is from activating the function because generally the action of pulling a rope is very sudden and fast, as long as the function is activated before the user runs out of momentum this kind of feedback is not needed here. In the normal state the handle guards are not colored because the affordance is clear enough and a person would not normally grab a handle by the handle guards, they are however colored green and red to make it more apparent to the user that they are acting on it.

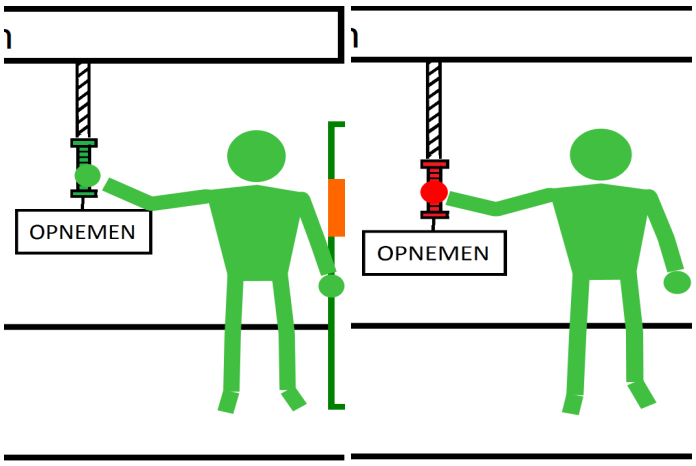

Figure 3.15: *Reaction of the rope handle: a) when hovered over, b) when grabbed*

The rope activates the record screen seen in figure 3.16. In this screen a separate window is opened with a scaled version of the puppet, another typical WIMP element. The large change in the UI clearly indicates that the user is in a different mode. Here the puppet is also fixed in the horizontal direction to indicate correctly how the program will see the gesture. To avoid confusion the window is opened on top of the last position in which the user was before pulling the rope. Since the goal of this screen is to record one gesture multiple times, two static puppets are drawn behind the user, the light blue puppet shows the starting position and the dark blue puppet shows the end position of the gesture. Care has been taken to choose a color that does not conflict with the green of the user's puppet and doesn't overpower it by being to bright while still being clearly visible and distinguishable. The colors also follow the color code defined earlier in this chapter since it gives information about the first recording. The message "Get Ready" is shown for 1 second before changing to a count down of "3","2","1","Go!". At "Go!" the window's edges turn red to clearly indicate that the program has started recording the user's movements. An example of how this would look can be seen in figure 3.17. The program stops recording when the user remains still for certain amount of time. After that the text changes to "Done!" and after half a second the screen returns to figure 3.14. The result is that a new element in the scrollbar is added to the top. It is given the next number and set into the orange selection area.

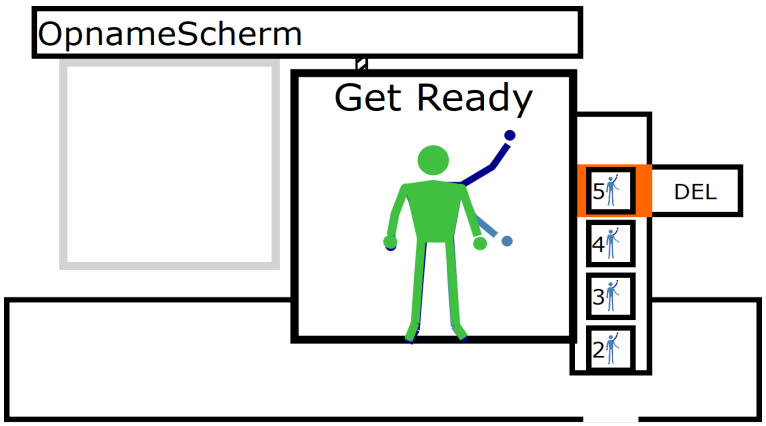

Figure 3.16: *First view of the recording screen*

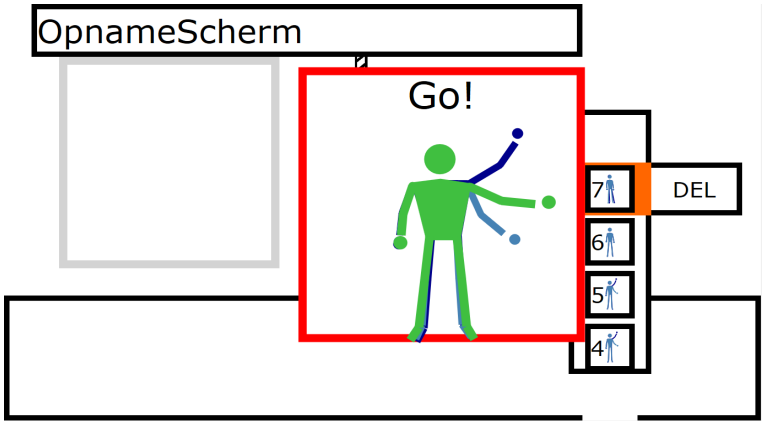

Figure 3.17: *View when the program starts recording, between this and the first view there is a countdown from 3, after the recording is started the user must stand still for a certain amount of time to stop, after which the text in the window will read "Done!"*

As described before, the scrollbar contains elements that are each linked to a recording of the current gesture. In these boxes a small preview can be seen of the start and end position of the linked recording in the same color scheme as during recording. The numbering of the boxes goes from the first recording as one counting up to the last recording. These numbers are not the same as the IDs of the gesture class as seen further on during the discussion of the implementation of the back end. The order of the numbers is still the same though. When the user hovers one of his hands over the recorded element within the orange selection box the effects as seen in 3.18 are triggered.

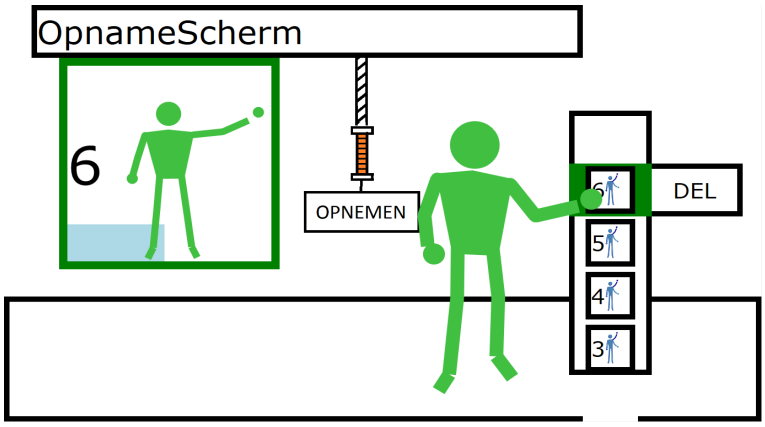

Figure 3.18: *The recorded element six is replayed to the user by hovering over it with any hand.*

The specific recorded element is replayed within the previously grayed out window on the left. The blue bar on the bottom of the window indicates the progress in playing the recording. As long as the user hovers over the element in the scrollbar, the recording is repeated with half a second delays between repeats. This was a deliberate choice to make sure the user is fully aware of which recording is playing in the window because he should notice that the recording is only playing when he touches it. For the same reasons the number of the recording is also displayed on the left of the screen where it does not obstruct the user in reviewing his recording but is there when the user is uncertain of which recording is playing. Additionally both the orange selection box in the scrollbar and the edges of the display screen are colored green, the color previously linked with hover over information. The redundancy in linking the replay with the recording comes from the fact that the screen is so far from the actual element it is connected to. The scrollbar is on the right because right handed users are more common, so it is easier for them to operate

the scrollbar. The delete action is already on the right of it and can hardly be moved to somewhere where it is still as easy to operate. Following both the proximity and similarity law of the Pragnanz law's would make the most sense but placing the screen on the right of the delete button would still have the delete button between it and the recorded element and would place the user's puppet too far from the center of the screen, out of the focus of the user.

To activate the scroll function on the scrollbar, the user has to hover his hand within the boundaries of the black bordered box but not within the orange selection box, the black borders of the scrollbar turn green to indicate that the user is hovering over the scrollbar, this is shown in figure 3.19 a. Scrolling only starts when the user has moved a small distance up or down to avoid to scrollbar being immediately stuck to the user which would be annoying. Once scrolling is started it continues as long as the user keeps moving in the same direction, this also counts when going through the selection box area and even when going outside of the scrollbar's borders. This makes sure that the user is not surprised by a sudden stop of the scrolling action when accidentally going inside of the selection box or outside of the scrollbar. The whole reason why the user cannot initiate a scroll within the selection box is because the element that is placed there has to be able to be hovered over to display it, as explained earlier, and it has to be able to be moved to the side for deletion. Accidental activation of the scroll function is minimized by this measure. Though there is more to this, informal tests showed that users who are unfamiliar with the UI often try to scroll down or upward starting from the selection box, the large delay in reaction confuses them. To counter this the activation area of the scroll function is expanded into the orange area for about 15 percent of the selection box height on both the top and the bottom. Once activated the elements stay on a fixed distance from each other but all elements are moved with the same dynamics as used in the record rope, they perform the same vertical relative movement as the hand that is hovering over it. During scrolling all recorded elements are locked to any interactions and the element that will land in the selection box is indicated by a light blue coloring of the white background. The example of scrolling can be seen in figure 3.19 b. When the user stops scrolling the recorded elements are locked into positions as seen in 3.19 a. Only the element in the selection box can be interacted with.

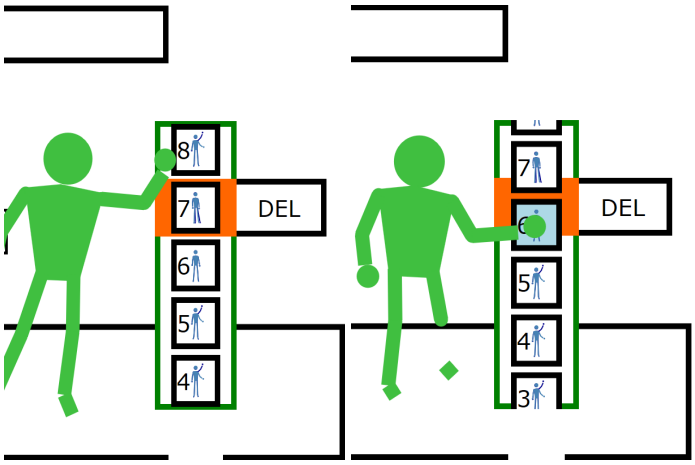

Figure 3.19: *Reaction of the scrollbar: a) when hovered over, b) when scrolling is activated*

The delete sequence of the recorded element in the selection box can be started by hovering over the middle of it. After this the element will follow the relative horizontal movement of the interacting hand between start point and end point with the same dynamics as the scrollbar and the rope. As seen in figure 3.20 a, The "DEL" box is filled up with a red progress bar to indicate how far the user still has to move until the item is deleted. When the end point is reached the recorded element turns red and fades away within a halve second to clearly indicate that this recording is deleted. Once the endpoint is reached the

action is irreversible and for the duration of the fading the scrollbar and all the elements are locked for any further interaction. An example of this is shown in figure 3.20 b. The reason why this action is only started from the middle is to increase the difference between initiating the delete sequence and replaying the recording and also heighten the threshold to start such a serious action as deleting an item. After deleting an element the other recorded elements are renumbered.

An important requirement in HCI design is that it should be possible to undo actions. Due to the technical complexity of an undo function and the limited time for this thesis, such a function is not included.

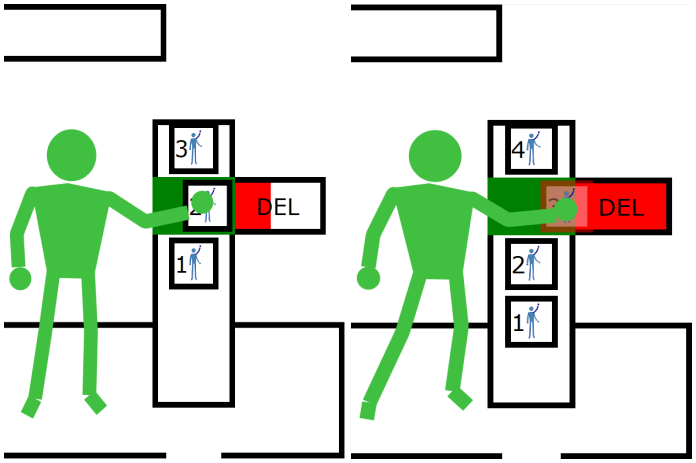

Figure 3.20: *The delete sequence: a) user in the process of deleting, the red progress bar indicates how far until deletion the user has to go, b) the endpoint of the action is reached, the item turns red, fades out and is deleted*

3.3 Gesture recognition

While the main focus of this thesis is on the user interaction with the OmniPlay application, it is necessary to have a basic algorithm that makes it possible to recognize gestures.

In order to provide enough flexibility concerning the type of gestures to be recognized, SVM is used. SVM supports supervised machine learning and its use encompasses two modes: train and predict. After a discussion of the way SVM can be useful for this application, the approach relying on SVM to predict the executed gesture is explained.

3.3.1 Support vector machines

A model can be trained with a given data set and labels to classify each gesture. The data set contains all of the recordings of each gesture. A gesture consists of multiple recorded frames. Each frame is classified by using a label. This makes it possible to specify which frames belongs to a certain gesture. Each frame consists of 25 data points, called joints, which are measured by the Kinect camera. A single point is defined in 3D space, meaning that one frame is defined by 75 coordinates. In other words, one frame has 75 features and, as such, one frame can be seen as a single point in a 75-dimensional space.

It is possible to record the same gesture more than once and include all of these recordings in the data set. All recorded frames of the same gesture receive the same labels to indicate that they are basically the same gesture. While these gestures may appear to be identical, they contain slight variations due to noise during the measurement of a gesture or a slightly varying execution of the gesture by the user.

After performing all required gestures, an SVM model is created using an available implementation [19]. Given a single frame of a gesture, this model can predict which of the recorded gestures the frame belongs to. SVM does this by finding the class of frames that are the most similar to the given frame and returns the label of that class as a result. This also implies the necessity of having multiple trainings recorded for each gesture. Errors in a single training due to noisy measurements of the Kinect camera can lead to wrong predictions. Having multiple trainings minimizes the influence noise has on the prediction and keeps into account that the user executes all gestures with slight variations. As a result, the model can predict more accurately which gesture is performed.

However, there are some things to keep in mind when applying this strategy. Firstly, inputting multiple repeats of the same gesture helps creating a better model, which improves predicting the gesture after a model is created. The downside is that it takes more time for the therapist to record all of these gestures. Secondly, when trying to predict a gesture, the generated SVM model always returns the label of the most similar gesture, even if they are not related at all.

These problems are tackled as part of the approach to using SVM to predict gestures. Also, the next section discusses how gesture recognition works for the OmniPlay application in more detail.

3.3.2 Approach

Two approaches are considered when it comes down to learning how to recognize gestures. By directly comparing these approaches, it is easier to identify their advantages and disadvantages.

The first approach is to add a time stamp to each frame as an extra feature, which indicates the time relative to the first frame of the gesture. This results in having 76 features per frame. The additional time stamp feature can be used to make sure that the gesture is not just similar in space, but also in time. If performing a certain gesture takes t seconds and is captured at a rate of f frames per second, this amounts to $76 \cdot t \cdot f$ features. For a 4-second gesture recorded at 30 frames per second, which is the highest sampling speed of the Kinect camera, this amounts to 9120 features for a single training of the gesture. In other words, the number of features of one training depends on the duration of the gesture. The entire gesture is classified as a single gesture. During prediction, a gesture with the same number of features can be input. As a result, SVM returns the class of gestures that is the most similar to the given gesture.

There are several problems with this first approach. If the number of features of the gesture used for predicting does not match the number of features of the training gestures, the prediction is not accurate and should be discarded. Discarding is necessary as SVM always returns the label of a gesture, even if the predicted gesture is completely unrelated to any of the gestures used for training the model. This poses a problem as different gestures can have a different duration. Even different recordings of the same gesture can take for instance 4 seconds and 4.1 seconds, implying that they have a different number of frames and thus a different number of features. A possible solution is to recalculate the entire gesture and use interpolation to convert the set of frames to a new set with a known, fixed size and preferably with equidistant time stamps. Using time stamps also implies that it is necessary to know when exactly a gesture starts and ends.

Furthermore, not just the different recordings of one gesture, but all of the gestures need to have the same number of frames. The gesture to predict is not known beforehand and can only be predicted accurately if the number of features for both the predicted gesture and trained gesture are equal. As a result, it is required that all gestures have an equal amount of features. This means that short gestures need to be mathematically extended with additional frames to match the size of longer gestures. Another side effect is that all gestures need to have the same duration, not just the same amount of features. This limits the therapist in choosing exactly what gestures he wants the patient to perform. Because of this, the action of predicting a gesture can only start when enough time has passed for the player to perform the gesture with the longest duration. More in particular, assume the longest gesture is 5 seconds in length. It then follows that it takes about 5 seconds before prediction can be started. If prediction is started sooner, the gesture has not yet fully been performed by the user. This also means that patients that control the game can only perform one in-game action every 5 seconds. As the games to be played are unknown beforehand, a slow reaction time of the application can render some games unplayable. As such, this approach is not viable.

The second approach is to split up one gesture into n smaller gestures and classify each of them differently, assigning a different label to each part of the gesture. Assume that a gesture is split up into 4 smaller parts so that each part contains approximately an equal amount of frames (see figure 3.21). Consider a 4-second gesture sampled at 30 frames per second. The complete gesture consists of 120 frames. By splitting it up into 4 parts, each part contains 30 frames. All first 30 frames are labeled with the same label, for instance 1. The next 30 frames receive a label 2, and so on. In other words, the considered gesture is split up into 4 postures with each having 30 slightly varying trainings. In contrast to the first approach, as described above, prediction does not happen for an entire gesture, but for separate postures. The condition for having executed the entire gesture is that each of the postures are predicted correctly and in the right order. To put it differently, n *pictures* are taken of the gesture and if at some point during prediction all pictures are executed in the right order, the application acknowledges the execution of the gesture.

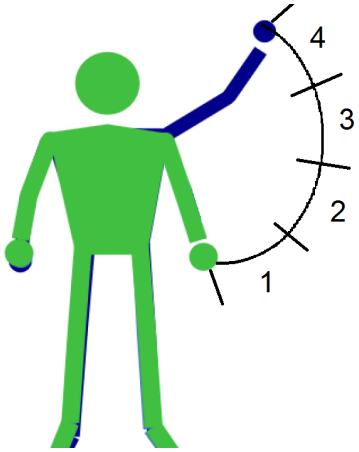

Figure 3.21: *An executed gesture, split up into four parts*

The biggest advantage of this approach is the flexibility of it. Each posture corresponds to a single frame, which contains 75 features. This eliminates the need of having gestures with the same length or having to modify training data to have gestures with an equal number of features. It is even possible to not just link a gesture, but also a posture to a virtual keyboard button, which allows the physical therapist to choose the gestures that fit the needs of a patient the best. Additionally, the application reacts immediately to an executed gesture, not after a fixed amount of time. This allows for a wider variety of games being playable using this application.

Another advantage is that it solves the issue where SVM always returns the label of a predicted gesture class, even when the patient is not doing anything. A gesture is only recognized when all parts of it are executed. In other words, gestures are not recognized involuntarily and, as such, actions like button presses that influence the game are not executed by accident.

A limitation of this second approach is that the physical therapist has no way to choose how fast the patient should perform a gesture. There is no strict requirement for the entire gesture to be executed in about the same time as the recorded gesture used to train the SVM model. If a gesture is executed faster during prediction compared to during training, it is recognized as the same gesture being executed. However, it is possible to solve this timing issue to some extent when the gesture during predicting is performed slower. The gesture can be ignored if more than a certain amount of time passes. This time threshold takes the gesture with the longest duration into account and is chosen higher than this in order to allow all gestures to be executed and successfully recognized.

Chapter 4

Implementation

The application is developed in C++ using Microsoft's Visual Studio IDE and can run on computers with a Windows operating system. On an implementation level, platform-specific features are used for saving and loading data files and running the GUI. The implementation consists of three big parts: back-end software, graphical user interface and gesture recognition.

4.1 Back-end software

4.1.1 Class diagram

The focus of the application is reflected by the structure of the code. The class diagram shows a model-view pattern to make a distinction between the graphical interface of the application and the back-end (see figure 4.1).

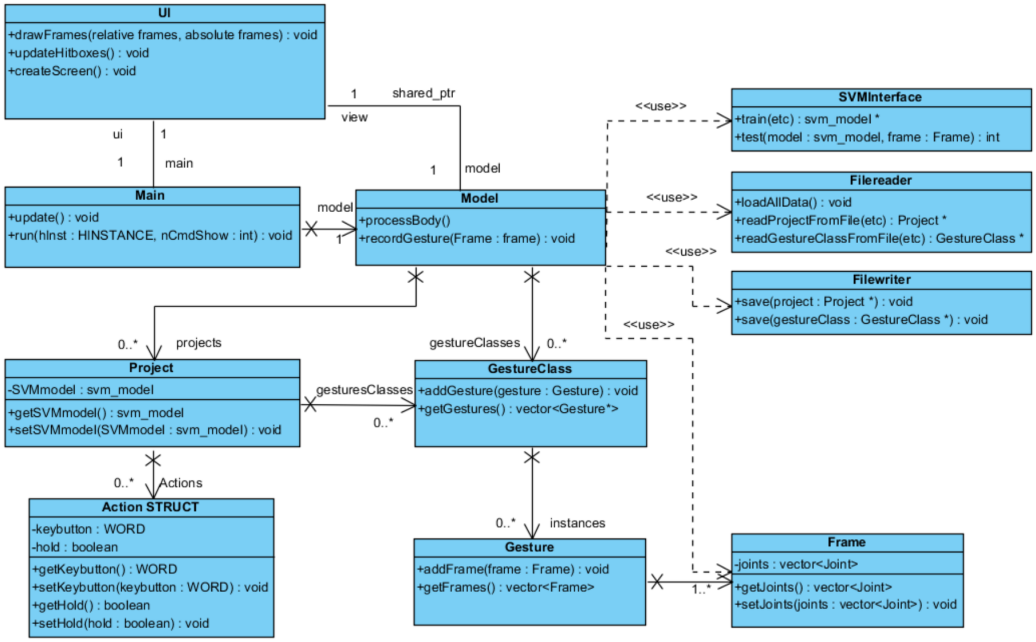

Figure 4.1: *The class diagram of the application, focusing on the back-end*

The Kinect camera has the ability to identify and measure the position relative to the camera of 25 points of a person, for instance: left elbow, right elbow, head, center of the spine, left wrist,... All of these points are referred to as `Joint`s and can be accessed using the libraries that come as part of the Kinect SDK installation. Each `Joint` has an x, y and z coordinate, so it is unambiguously defined in space.

When the Kinect measures all of the `Joint`s at one specific point in time, these 25 `Joint`s together form one frame. This can be seen as a single picture of a person taken by the camera. Analogous to a movie, which is nothing more than a quick succession of pictures, an instance of `Gesture` collects all `Frame`s, which all together, contain information about how the person moved over a period of time.

In order to improve the application's ability to recognize a gesture, each of the gestures must be trained more than once by the physical therapist. For each gesture that is trained, an instance of `Gesture` is created. The `Gesture`s that contain data about different executions of the same gesture are grouped into a `GestureClass`. In other words, one `GestureClass` object contains all trainings of the same gesture.

After setting up a `Project` object, it contains all data that is needed for playing a game using the application. It has a `map` that maps a label for each gesture to a `GestureClass` and the actions linked to that `GestureClass`. An `Action` contains information about the virtual keyboard button that needs to be pressed and if that button should be held down or be quickly pressed and released. The `Model` class forms the core of the application. It controls the flow of the program and contains all important objects, such as the `Project` and the `GestureClass`es. By storing the `GestureClass` objects in `Model` rather than `Project`, it is possible to reuse the same `GestureClass`es in multiple `Project`s.

`SVMInterface` takes all gesture data and converts it to a format that is accepted by the LibSVM library [19]. This is an existing library that provides a C++ support vector machine implementation and is used in this application for gesture recognition.

To prevent having to train all gestures again each time the application is started, all necessary data is saved into files in the data folder using the `Filewriter` implementation. This includes the data of the `Gesture`s, `GestureClass`es, `Project` and the computed SVM model. The `Filereader` is responsible for reading data from the saved data files when the application is started.

4.1.2 Flow of the program

After initialization of variables like the instances responsible for the user interface and the communication with the Kinect camera, the program locks into an infinite loop while the application is running. This main loop consists of three processes: fetching new data from the Kinect camera, analyzing this data in order to use it for recording or predicting gestures and updating the GUI with relevant changes.

It is important that none of these three processes block the continuous flow of the main loop. The update rate of the interface is only as fast as the rate with which the main loop is executed. If it is slowed down with a blocking or long-running loop, the GUI appears to stutter or freeze and no new data is collected from the Kinect camera.

The Kinect can only provide new data as fast as 30 times per second. If no new data is available at the moment the main loop requests the data, for instance when the main loop is executed faster than 30 times per second, the program skips this process and continues with the other two processes.

To control the flow of the program without reverting to blocking loops, software flags are used. These flags keep track of the current state of the program. By evaluating the state of the flags, it is possible to indicate if the program is predicting a gesture, recording a gesture or animating specific graphical elements of the GUI.

4.2 Graphical user interface

In this section, the implementation of the UI is discussed. First, the used technologies are given, then the class diagram is discussed per class. Finally, the general flow of the program is described.

The Windows presentation foundation (WPF) is used to create buttons and input fields to use for the developer's UI, seen on the right side of figure 4.2. The interface for the actual users is created in Direct2D. Because this part of the program is used to experiment with different types of feedback and UI elements, it needs to be easily adaptable.

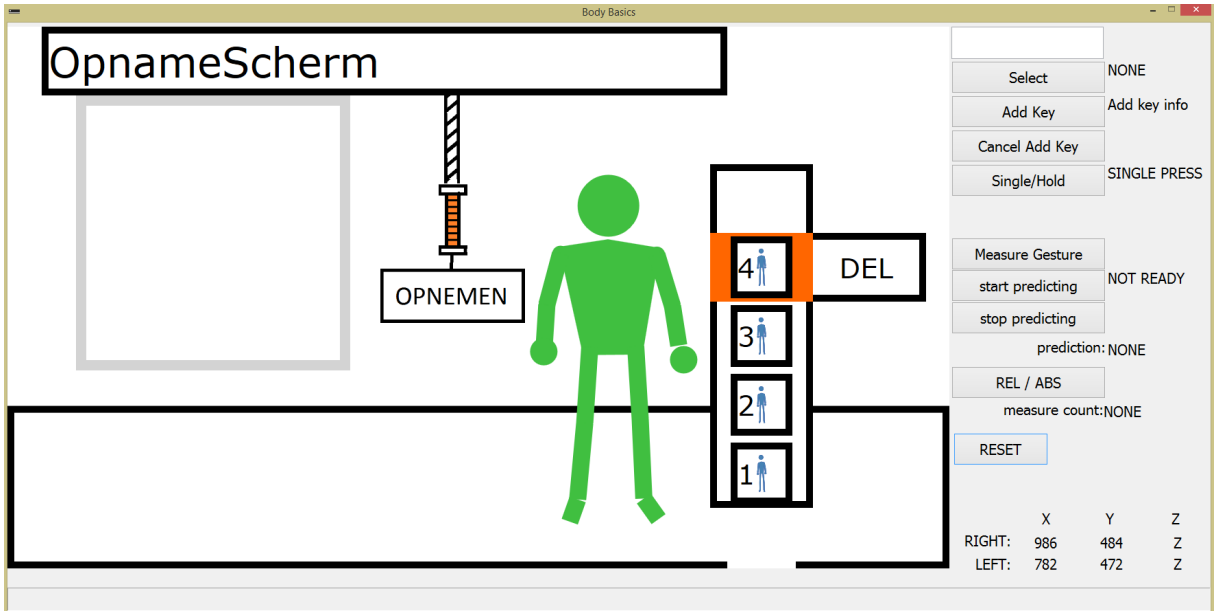

Figure 4.2: *The full screen of the user interface*

4.2.1 Class diagram

The program is built out of a publicly available example by Microsoft called *BodyBasics-D2D* that contains all code to draw 2D skeletons of people seen by the Kinect camera on an empty background. The original program has all of its code grouped into one class, but is refactored into the structure shown by figure 4.3. The original code is relocated to the `Main`, `UI`, `Model` and `D2D_Graphics` classes to fit into a model-view structure. The `Main` class initiates the resources for the Kinect, creating an instance of the `UI` and `Model` classes and starting the program loop. The `UI` class consists of all the elements related to the view and the controller, while the model received all code related to the processing of the Kinect input. The `D2D_Graphics` serves as an interface between the `UI` classes and the Direct2D canvas.

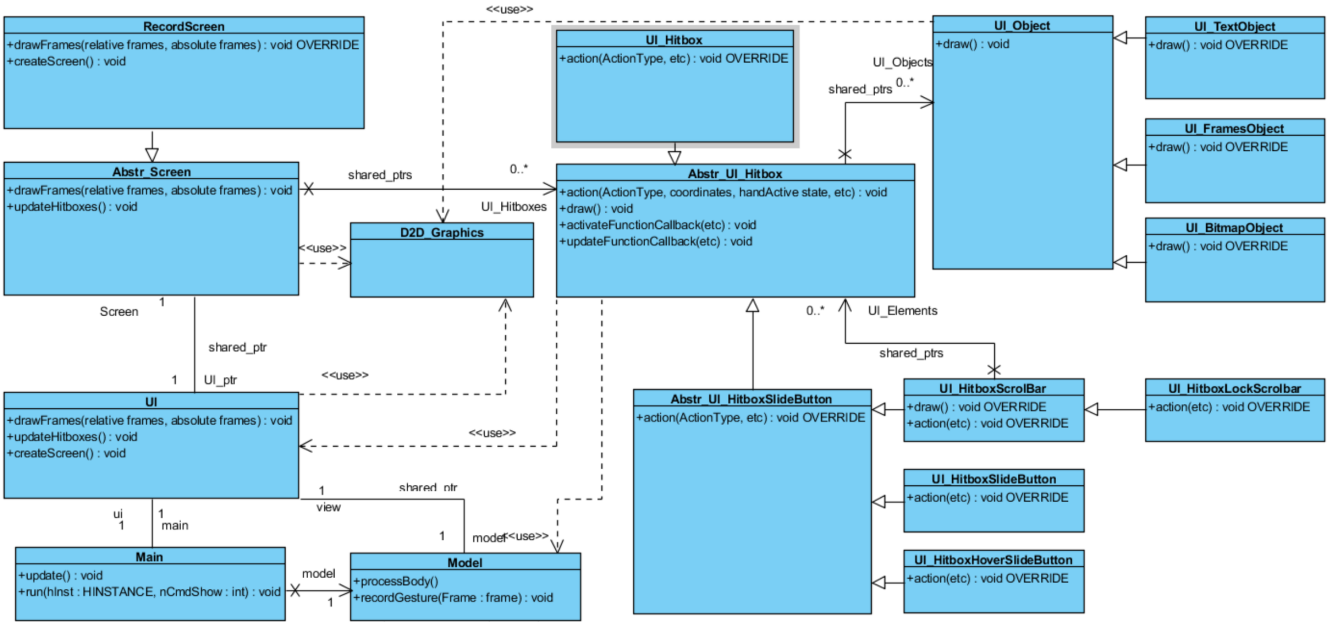

Figure 4.3: *The class diagram of the application, focusing on the GUI*

The `UI` class is the main class for the GUI part of the program. It serves as the main manager of the view and controller-related classes and is the only one through whom the `Model` class can activate UI elements. It creates the main window in which the application is displayed and handles the input from the WPF UI elements and the keyboard input. At construction, a global instance of the `D2D_Graphics` class is created so that all other classes can have direct access to the screen. It also initializes the resources for the Direct2D canvas. It contains a pointer to the current screen that is displayed. Every time a new screen is opened, the object of the old screen is freed and replaced with a new object of the current screen. This class is responsible for creating the instance whenever a new screen is opened. Though at this point there is only one screen so the pointer is never changed. Three important methods are: `drawFrames`, `updateHitboxes`, `createScreen`. `drawFrames` manages the Direct2D resources and starts and ends the drawing on the canvas of `D2D_Graphics`. It also passes the draw command to the `Abstr_Screen` class. The method is called by the `Model` class to redraw the screen. `updateHitboxes` is used by the `Model` instance whenever changes in the model data need to be displayed in the UI. The `createScreen` method is called by the `Main` instance. When all resources are initialized and the `Model` is created, the UI class passes it on to the `Abstr_screen` class to instruct it to build the elements for the current screen.

The `D2D_graphics` class contains methods from the original example program, methods to initialize the `D2DResources`, to convert 3D joint coordinates to 2D screen coordinates and to draw the body. Some of the original methods were adapted to fit the requirements of the application. It also contains new methods to draw bitmaps, text or scale 2D skeleton coordinates to different sizes.

To explain the rest of the structure, the most basic class is considered first. After that, the relation with other classes is explained.

The `UI_Object` class and its children are an important part of the view organized in a strategy pattern, where each child adds properties and defines the `draw` function differently. It contains information about how a rectangle element is drawn using a center, width and height format whilst also containing additional properties such as color, border color and border thickness. The most important function is `draw`, which uses the `D2D_Graphics` to draw their rectangle. As mentioned before, each child class overwrites this function to draw their own specific type of image.

The UI_TextObject class is a child of the UI_Object class that is used to draw text on the D2D_graphics. It contains only the necessary properties of the text. The rectangle within which the text is drawn is defined by the parent class.

The UI_FrameObject class is also a child of the UI_Object class that is used to draw images of puppets other than the one directly controlled by the user. The puppet is scaled to the dimensions of the rectangle defined by the parent class.

The UI_BitmapObject is the last child of the UI_Object class. It is used to draw bitmaps scaled to the dimensions of the rectangle defined by the parent class. This is the only UI_Object that uses states to decide which image is drawn. Three states are defined: standard, hover and *handActive*. Standard behavior is to draw the standard image for all states, but each state can be assigned to a different image.

The Abstr_UI_Hitbox class represents a rectangle that takes on different states depending on the position and state of the hands of the user. The different states that the class can attain are: *hover* when a hand is hovering over the borders of the defined rectangle, *handActive* when the hovering hand is in the *activeHand* state, such as a closed hand and *activeHandOutside* state when the user's hand moves outside of the rectangle while the hand is in the *activeHand* state. The states are indicated by flags in the class.

The action method of the Abstr_UI_Hitbox class defines behavior for the entering, holding and leaving of a each state using enumerations. Additional flags allow the usage of special circumstances such as when the hand enters the rectangle while in the *activeHand* state. This action method is the main method that is redefined by every child to obtain more complex behavior using these states. This means that every hitbox that has a different effect on the UI must have his own class that is a child of this class. This is the reason why the strategy pattern is used. The class contains a vector of shared pointers to UI_Objects. The class can access and change these objects, like moving them and change their color. It is also used to have the UI_Objects draw themselves when the method draw is called. Multiple hitboxes can have a pointer to the same UI_Object, so they can all change it. It also contains two callback functions. One is called activateFunctionCallback, which defines the action performed on the Model and UI when the criteria for activation of the hitbox have been reached. The other is called updateFunctionCallback, which defines the behavior when the data in the Model is changed and the UI is updated accordingly.

The Abstr_HitboxSlideButton class is a different abstract class that adds behavior to the parent class. Specifically it adds functions to calculate how far the hand that is interacting with the hitbox has moved since the last frame. This information can be used by its children to move hitboxes and UI_Objects relative to the hand's movement. Parameters within this class define how far the hand can move from the hitbox's original position and when the activation callback function is called.

The UI_HitboxScrollbar class adds a vector of Abstr_UI_Hitboxes and uses the behavior defined in Abstr_HitboxSlideButton to move them up or down according the movement of interacting hand.

The UI_HitboxLockScrollbar class adds the act of locking each Abstr_UI_Hitbox derived object in its vector into predefined positions when the user stops scrolling. It also manages the behavior of the orange selection box, as discussed in chapter 3, where this class of hitbox is used as the scrollbar in the end design.

The UI_HitboxSlideButton defines the behavior to slide a button until an activation point is reached. In this specific case, the user has to put his hand into the *handActive* state, before he is able to slide the button. The rope described in chapter 3 is an instance of this class.

The UI_HitboxHoverSlideButton does the same as the UI_HitboxSlideButton, but doesn't require the user to put his hand into the *handActive* state. A hand hovering in any state on the right side of the hitbox rectangle activates the sliding towards the activation point. This class is used for the delete slide button for the recorded elements in the scrollbar.

The Abstr_Screen class is an abstract class for the creation of different screens. It holds a vector of shared pointers to all hitboxes on the screen, defines how the puppet of the user is drawn and provides functions to draw background and top UI elements to be defined by his children. For every different screen, a new child of this abstract class needs to be added.

The RecordScreen class is a child of Abstr_Screen that defines the type and the properties of the hitboxes and UI_Hitboxes that are used to create the screen as it is described in chapter 3. The hitboxes are also given their callback functions that are used when the hitbox is activated and when the Model calls for an update of the UI representation.

4.2.2 Flow of the program

At the start of the program, after the creation of the UI and Model objects, the UI instance is instructed to fill his pointer to the Abstr_Screen class with an instance of the RecordScreen class and then call the method createScreen on it to create all the hitboxes and UI_objects that belong to that particular screen. Then, the program loop is started and the processBody method in the Model class calls the drawFrames method in UI, which passes it along to its pointer of Abstr_Screen that is now pointing to a RecordScreen object. This calls the draw method on all the hitboxes belonging to the background and recursively passes on the draw function to any Abstr_UI_Hitbox child or UI_Object child it contains. At this stage, only the UI_Object class and its children call the D2D_Graphics object to draw something on the Direct2D canvas. The drawFrames function then continues by calling the D2D_Graphics object to draw the puppet representing the user and on top of that the hitboxes of the UI are drawn in the same manner as the background. Lastly, the coordinates and states of the hands of the user are given to every hitbox in the screen. These hitboxes use this information to determine and execute any changes in their state or their UI_Objects based on their criteria. If a hitbox determines that its activation criteria are met, it calls its activateFunctionCallback that makes calls directly to the Model. This happens every cycle when the processBody function in the Model class is called, which is approximately 30 times per second.

Whenever something changes in the Model structure that affects the representation in the UI, a flag is set and during execution of the processBody method, the updateHitboxes function is called on the UI object before the UI is drawn. The UI passes this on to the Abstr_Screen child, which calls an update function on all of the hitboxes it contains. Each hitbox calls their own updateFunctionCallback to retrieve all information related to them from the Model and process it. Hitboxes that contain other hitboxes call the update function recursively.

4.3 Gesture recognition

It is possible to split up the implementation of gesture recognition into two parts: the recording of a gesture by the physical therapist and the prediction of a gesture executed by a patient.

4.3.1 Recording gestures

Code snippet 4.1 shows the code for recording a gesture. Recording starts when the user starts moving. In order to know when the user moves, the first frame is stored in the variable `firstFrame`. The variable can then be used to compare the first frame to the current frame. If the two frames are not equal, the user has moved. Two frames are considered equal when all of the joints of both frames in all three dimensions have a difference of less than 10 centimeter. In other words, if the person moves more than that, the frames are not equal. The threshold of 10 centimeter is chosen due to limitations of the Kinect camera, where small variations in measurements are possible due to noise. Additionally, it is hard for a person to stay perfectly still, so the choice of the threshold can eliminate recognizing small tremblings as the user having moved.

When actual recording is initialized, the frames measured by the Kinect camera are stored in a buffer at a rate of 30 frames per second. The user now has two options.

The first option is to stand still in a certain posture. If the user does not move for approximately 1.3 seconds, the recording stops and the frames stored in the buffer represent the recorded posture. The constant `NOT_MOVING_FRAME_DELAY` contains the duration of the delay and is expressed in number of frames. The chosen number of frames translates to a delay of 1.3 seconds, as mentioned before. Several variations for this delay are tested, but 1.3 seconds has been empirically proved to be a good balance between giving the user enough time to react to on-screen feedback, while it does not make the user stand still too long to end the recording.

The second option is to start moving after recording has started. The user can perform a gesture and then stop the recording by standing still for 1.3 seconds. After that, the frames stored in the buffer represent the recorded gesture.

```
void Model::recordGesture(Frame & currentFrame) {
  if (!initialized) {
    firstFrame.setFrame(currentFrame); //Set the first frame on the first call
    initialized = true;
  }
  framesBuffer.push_back(currentFrame); //Add the frame to the buffer

  if (!startedMoving) {
    if (! currentFrame.equals(firstFrame)) {
      //The user moved, recording of the gesture is started.
      startedMoving = true;
      framesBuffer.clear();
    }
    else if (framesBuffer.size() > NOT_MOVING_FRAME_DELAY)
      addRecordedGesture(); //User did not move while recording; user recorded a posture.
    else //User not moving and not enough frames stored in the buffer.
      return;
  }

  if (framesBuffer.size() > NOT_MOVING_FRAME_DELAY &&
  framesBuffer.back().equals(framesBuffer.at(framesBuffer.size()-NOT_MOVING_FRAME_DELAY)))
    addRecordedGesture(); //User stopped moving, the gesture is stored.
}
```

Code snippet 4.1: method to record a gesture

4.3.2 Predicting gestures

Code snippet 4.2 shows the method for predicting a gesture. This method is called recursively. Gestures are split up into a number of smaller parts, referred to in the code snippet as `NB_OF_LABEL_DIVISIONS`. This constant equals four, so it means that a gesture is split up into four parts and each part gets assigned a label. To explain how a gesture is split up, consider a recorded gesture that has a duration of 100 frames. The first 25 frames get assigned a label that equals 1, the next 25 frames get label 2 and so on. In other words, gestures are divided into a number of different classes.

Each recorded gesture has a `vector` containing the four labels that refer to each part of the gesture. The labels are also stored in the correct order, representing the way the gesture is executed. The 100-frames gesture from the example has a `vector` with following labels: 1, 2, 3 and 4. For a posture, this `vector` contains, for instance, following labels: 5, 5, 5 and 5. The labels are the same, so this implies that the user did not move during recording, which is how a posture can be recorded.

Gestures are recognized when all four labels are predicted in the correct order. To predict which class is most similar to a given frame, an SVM model is required. The LibSVM library [19] is used to train an SVM model. The model returns a label that corresponds to the class the current position of the user is most similar to. This occurs 30 times per seconds, thus at the same rate the Kinect camera can process new data.

```
bool Model::isGestureExecuted(std::shared_ptr<Gesture> gesture, int posInBuffer, int
    recursiveCounter) {
  for (int i = posInBuffer; i >= 0; i--) {
    if (labelsBuffer.at(i) == gesture->getLabelOrder().at(labelOrderPosition)) {
      if (NB_OF_LABEL_DIVISIONS - recursiveCounter <= 0)
        return true;
      return isGestureExecuted(gesture, i--, recursiveCounter++);
    }
  }
  return false;
}
```

Code snippet 4.2: method to verify if given gesture is executed

Chapter 5

Results

Usability tests make it possible to verify if the design meets the expectations of the target group and offer a way to discover unexpected problems that are hard to track by the developers themselves. Test persons have a fresh look on the application and can provide useful insights in the way they interpret visual elements of the user interface.

In addition, the technical evaluation considers the effect of the implementation, ensuring that the user is not hindered by performance issues or similar problems.

5.1 Usability test

The graphical user interface is designed for use by physical therapists. As such, acquiring feedback from therapists is essential in developing an interface that is both useful and easy to use. The process of the usability test is described, followed by an overview of the most important results of the evaluation itself and an interpretation of these results.

5.1.1 Description of the test

A usability test is conducted with physical therapist Dries Lamberts at Windekind Leuven, an organization that focuses on the education of children with disabilities and offers a full program to assist them with their specific difficulties. For this test, the Kinect camera is connected to a computer that runs the application and, positioned on a desk at hip height. The user indicates that he is familiar with using the Kinect camera and motion-based games.

A short introduction on the purpose of the application is given to the test person for reference. After that, he is presented with several tasks and is asked to apply the think-aloud protocol in order to obtain as much information as possible about the interaction with the application and the thoughts of the user while doing so.

In preparation of the usability test, gestures are pre-recorded that both serve the purpose of testing the robustness of the application as well as having a fallback plan in case problems arise during the test.

One task is to watch the pre-recorded gestures and control a game called Space Invaders using the pre-recorded gestures. This is a game that has an avatar move left or right on the screen and shoot bullets (see figure 5.1). As such, four gestures are pre-recorded: one gesture that allows the avatar to move to the left, one to move to the right, one to shoot bullets and a neutral gesture that is linked to no keyboard key. At this point, it is explained to the user that a neutral gesture is required in order to have a gesture

to which no keys are linked. This is explained without going into detail about the specifics of SVM. The goal of this task is twofold. Firstly, the task is used to verify if the test person is able to watch replays of the pre-recorded gestures without any guidance and to learn what gestures are pre-recorded without additional information. After this is done, it is explained to the user how the gestures and keyboard keys are linked together. Secondly, it makes it possible to confirm if a correct prediction of a gesture execution does not depend on the person performing the gesture, as problems can arise due to the person recording the gestures and the person playing a game having a very different physique.

Figure 5.1: *Screenshot from Space Invaders*

Another task is to play the game Sokoban Geek. This puts the gamer in control of an on-screen avatar that can move up, down, left or right. Walking into boulders allows the player to move them in order to solve a puzzle (see figure 5.2). The game is shown to the test person and after trying the game, he decides how many gestures are required to play the game. Then, he comes up with different gestures for each input and records three trainings for each of them. The test person is asked to delete a recorded gesture and record it again. After recording all gestures, he tries to play the game. The goal of this task is to check how the test person reacts to the recording elements of the interface and if it is clear how to interact with them without assistance.

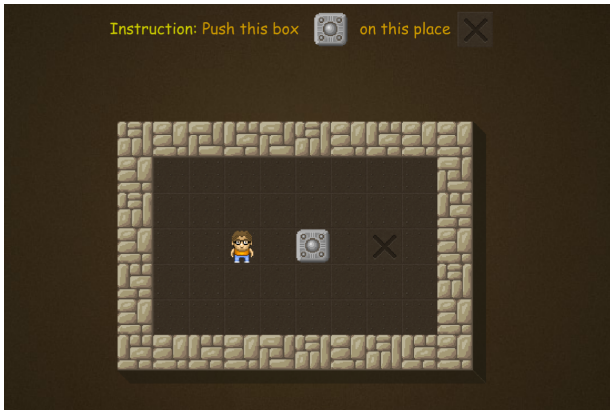

Figure 5.2: *Screenshot from Sokoban Geek*

With permission of the test person, during the entire test, the computer screen is recorded, as well as everything said, for analysis purposes.

5.1.2 Results

This section describes how the physical therapist interacted with the UI that was seen in section 3.2.5, where all relevant figures of the interface are shown.

For the first task, the user notices the recorded gestures in a list on the right-hand side of the screen. By trying to move its avatar's hand to align with the previews of the recordings, he notices that it is possible to scroll through the list of recordings. He also notices the delete option next to the recordings list and indicates that he is anxious about deleting one of the recorded gestures because of the rather aggressive red color of the progress bar.

Watching a replay of the recorded gestures initially is not clear to the user. He tries to drag one of the gestures to the square on the left hand-side of the screen, which is intended to be the display screen. He notices that this is not possible, but makes several attempts do this. After that, he indicates that he can't find it and looks at the interface without trying to interact with anything. To replay a recorded gesture, the user can point with the avatar's hand to the recorded gesture in the orange area of the scrollbar, previously referred to as the selection box area. At that point, a replay of that gesture is displayed automatically. The replay stops while scrolling through the list or when not pointing at the gesture in the selection box area. The user did this several times during this task subconsciously, without noticing that a replay started playing at the left side of the screen. A factor adding to this problem is the fact that the recordings were displaying a posture rather than a gesture, which in fact is a static image instead of an animation. The user required assistance to complete this task. He was able to do this after knowing that the gesture in the selection box area is displayed at the left-hand side square.

After watching replays of the recorded gestures, the user is able to replicate in real life what gestures are recorded and displayed on-screen. When the Space Invaders game is started, the user has no problem interacting with the game and does not require any assistance. He is able to control the in-game avatar subtly and meet the game objectives. After asking for his experience with the controls, he indicates that the controls are responsive and that he does not experience any lag.

For the second task, the user gets the chance to see what the Sokoban Geek game looks like and how it is played. After that, he states that he needs five gestures to play the game. Four gestures are linked to a directional keyboard key and one gesture that serves as the neutral gesture. To record a gesture, it is required that the user hovers the hand of the on-screen avatar over the grip of the pulling cord, makes a fist, and move his fist down to start recording. This translates to pulling the displayed cord. When trying to start recording the first gesture, the user pulls the cord without any problems. The recording screen appears and he starts performing a gesture when the interface indicates that recording has started. He stands still when he is done with the gesture and the recording stops automatically.

Upon being asked to delete the gesture he just recorded, he needs no assistance to push the gesture object to the right into a box indicating that the gesture can be deleted. He indicates that his earlier anxiousness of deleting a gesture by accident is unnecessary, as it requires some effort to perform this deleting action and it is hard to do it by accident.

When trying to record a gesture the second time, the user is unaware that he is required to close his hand into a fist to pull the cord. He tries hovering over the grip and moving his hand down and indicates that he isn't sure why recording does not start this time. Only after several tries, he tries closing his hand to pull the cord, activating the recording screen. He states that he did not know that closing the hand was required and that he did that by accident to record the first gesture. He indicates that he has experience using the Kinect camera for other games and that these games never required to close a hand, so he was unaware that the Kinect is able to make a distinction between open and closed hand.

The user notices clearly that the recording of a gesture stops when he stands still. Because he was not aware of this, one gesture was not recorded the intended way, but the user was able to delete this gesture and record a new one without assistance. The user also noticed that the light blue and dark blue puppets behind his puppet during the recording screen indicate the beginning and ending position of his first gesture. He states that this feature is very useful for repeating the original gesture.

When trying to watch a replay of a recorded gesture, he is not sure if he needs to point at the gesture with open or closed hand. Although both cases function in the same way, the user states that using a closed hand feels like the application responds better. He also wondered whether the recording would distinguish the difference between open and closed hand, but concludes that it does not because the replay of the recording does not show any difference. This is as intended. While watching the replay, he noted that it is not possible to accurately tell what a gesture looks like when stretching for instance an arm straight ahead. The replay does not provide any information about depth.

After the user records all gestures, he is able to play the game. However, the gesture that is linked to the left arrow key does not respond. The user is asking if the key is linked correctly to the gesture and thinks this is the problem. He misses feedback about what gesture is linked to what key, though it should be noted that the concept is that keys would be assigned on a different screen in a full application. After verifying for the user that the key is linked correctly, we present him with the question of how he would handle this problem without any assistance. He suggested to delete the recorded gestures and record them again. After doing this, the user is able to play the game without further problems. Analogous to the situation of playing the first game, the user plays the second game with similar ease.

The entire user test approximately took 45 minutes. After the test, the user asked because of his own experience with disabled children if very small gestures are supported by the application and if the application can be used for other purposes than playing games. An example he thought of was using gestures to type words using sign language. He also added that he was glad that use of the space bar is supported by the application, as he has experience with similar projects that do not support the space bar, while it is one of the most used keys in computer games. However, a lot of games the children are interested in require the ability to point with a cursor on-screen. This is a feature he would appreciate. Finally, he stated that he is very interested in using the application, allowing disabled children to play games and exercise while doing so.

5.1.3 Discussion

The physical therapist proved to understand the concept of the application. This shows that it is possible for persons without programming knowledge to use the application for recording gestures and controlling computer games. He also understands that the use of the application is not limited to games and can also be used for other computer applications that require keyboard input.

A few tries are required to get familiar with how to start recording and how to end it. In particular, without prior knowledge or experience, more feedforward is required to communicate that the hand needs to be

closed when pulling the cord and that the user needs to stand still to stop recording. However, after a few tries, it is possible to record multiple gestures in succession without any of them not being recorded as intended.

Replaying recorded gestures requires the user to align his on-screen hand with the gesture inside the selection box area of the scrollbar. There is no form of feedforward indicating that performing this action results in the gesture being replayed. The feature is intended for the user to be discovered while scrolling through the gestures or while trying to delete a gesture. As such, this can be confusing to the user.

Both the actions of recording gestures and replaying recorded gestures are performed faster after having experienced this firsthand. The provided feedforward is too subtle to immediately make it clear what actions are expected. To improve this, it is either possible to provide more explicit feedforward, or to introduce the user to the most important functions using a tutorial.

When playing the games using gestures, the test person appears to be very engaged in them and the way they are controlled. This is an indication that not only the application succeeds in its purpose of playing a game with gesture-based input, it also can make for a pleasant experience.

5.2 Technical evaluation

The application is developed with the Microsoft Visual Studio integrated development environment (IDE) in C++ and can run on Windows operating systems. This limitation is due to the use of the IDE's visual interface editor and Direct2D, as well as the feature of saving and loading data files. The application is tested to run on both Windows 8.1 and Windows 10 machines.

All of the required files for running the application, including resource images, take up approximately 20 MB of space. Saving all gestures needed for playing a game takes up approximately an additional 0.6 MB of space. This depends on the number of gestures recorded, the number of trainings done for each gesture and the duration of each gesture.

The application refreshes the interface at a rate of 30 frames per second (FPS), which is limited by the output of the Kinect camera. The Kinect also introduces limitations related to the environment and the user. These limitation are that overly lit rooms make it harder for the camera to track a person. The same is true for persons wearing black or reflecting clothing. These findings are in accordance to Microsoft's guidelines for using the Kinect [20].

Chapter 6

Discussion

6.1 Reflection on the results

Usability tests are important tools for the evaluation of a prototype. However, with a limited number of test persons, a quantitative analysis is not possible due to the necessity of a test group that is sufficiently large to produce statistically useful data. Even for a qualitative analysis, using the opinion of only one physical therapist may result in responses that are biased towards a certain view the therapist has on the subject. To distinguish between genuine design defects and problems only encountered due to individual bias, it is necessary to have a large enough test group. For the scope of this thesis, the number of user tests is limited, so a more careful approach is needed when interpreting the results of the test.

The test users, including the physical therapist, require time to get familiar with non-traditional gesture-based interface elements. The time needed depends on the person and his knowledge and experiences with other gesture-based interfaces. An important part of this is knowledge about the Kinect camera and its possibilities. The test users are worried of doing something that the application does not know how to process. A common problem is that it is unknown to the users that the Kinect can make a distinction between an open and a closed hand, even while there is visual feedback provided when opening or closing a hand: the hands on-screen turn respectively green or red. This means that the user does not consider opening or closing his hands when trying to interact with something on the screen. This problem is related to the features supported by the used sensors. The Kinect camera supports this feature, but similar cameras may not. Because of this, it is not possible to simply assume user knowledge of this operation.

When test persons get the time to experiment with the interface at their own pace, they eventually find a way by themselves to interact with the application. Due to the limited amount of different interactive elements, the application has a gentle learning curve. A tutorial can help bringing new users up to speed with how to use the application. However, a tutorial can also be a source of frustration or confusion for the user if not done well. As such, tutorials should be seen more like a last resort instead of the go-to solution for any interface-related problem. More and better feedback and feedforward is generally the preferred option to choose, as it increases natural coupling between the required actions and its function. Also, providing redundancy concerning feedforward for an action can help make the interaction more clear as well. For instance, the color of an element changes when the users hover over that element, implying that interaction is possible. This can be expanded by having a sound effect play at the same time the color changes.

A potential problem is the limited amount of space the user has in the room where he is using the application. Unless the user is required to move left or right during the use of the application, all interactive elements must be placed within reach of the user's avatar. This means that non-interactive elements can be placed further away from the avatar. This introduces the problem of limited interface space on the screen and can result in elements that are functionally coupled together to be separated. For instance,

the scrollbar with recorded gestures is on the right of the user's avatar, but the playback of the gesture is shown on the left. It limits the locational aspect of the functional feedback, but at the same time indicates that no specific action is required for interacting with the gesture display on the left as it is out of reach. For instance, to some extent, this can be compared to having to interact with a Blu-ray player, but receiving visual feedback from another place, i.e. the television screen.

Unrelated to the interface design, but an important part of the application is gesture recognition. Due to the chosen approach, it is possible to record and train gestures as well as postures. However, the way SVM works limits the requirement for high accuracy when executing a gesture or a posture. The executed gesture is always recognized as the recorded gesture that is the most similar. If all of the recorded gestures are very dissimilar, gestures do not need to be accurately executed in order for them to be recognized as a specific gesture. This means that it is not possible to tell if a gesture is executed correctly. It requires the physical therapist to keep a close look at the way the patient performs a gesture during play.

Furthermore, the speed with which a gesture is executed is not checked. Only if the execution is too slow, it is not recognized as a recorded gesture. Faster execution does not affect the gesture recognition, on condition that the Kinect's sampling limitations are not surpassed. To solve this problem, as well as the previous problem of inaccurate executions, the therapist needs to record not only the required gestures, but also examples of how gestures should not be executed, i.e. gestures that are performed too slow, too fast or just not the way the gesture has to be done. This requires the therapist to record many examples of wrong executions, which requires a lot of time to do and eventually leads to a less pleasant experience during play as far fewer executions are accepted as being correctly executed. In addition, it requires the therapist to have some understanding of how machine learning works, which does not match the goal of designing an application that can be used without any programming knowledge.

Playing some games found online with OmniPlay is not possible if they require mouse input. A way to solve this problem is to predefine a specific gesture that activates a *pointing mode*. In this mode, it is only possible to point to something on the screen or switch back to a *gesture mode*. However, this gives more responsibility to the therapist, who has to remember what that specific mode-changing gesture is and that he cannot reuse that gesture as a gesture for the patient. In addition, defining a preset gesture means that not every patient has the ability to perform it, as some may not be able to move an arm or a leg, for instance. Since, from an end-user point-of-view, this setup is confusing and undoes the application's elegance of simplicity, only games can be played that require only button presses as an input.

6.2 Analysis using Wensveen's design framework

Wensveen [9] created a design framework that designers can use to make their product more tangible and interactive. Though his work was already discussed in chapter 2, it is interesting to elaborate on the criteria he has set out and to look at the measure in which the UI design in this project fulfills these criteria.

The article states that the key to a more tangible and interactive interaction is by creating a natural coupling between the action that is performed by the user and its associated function in the program through feedback and feed forward. This can be done on six different aspects: time, location, direction, dynamics, expression and modality. A coupling in time can be achieved by having no delay between the action of the user and the feedback of the program. For instance, when pushing a button, it changes color immediately. Location coupling is achieved by placing the feedback for the program in the proximity of the action of the user. In the previous example, the button is coupled in time but also in location because the location where the user pressed is also where the feedback appears. The direction is coupled when the feedback of the program moves in the same direction as the action of the user, such as a scrollbar that moves up when it is dragged up. A coupling in dynamics is achieved by giving feedback with the same position,

speed, acceleration and force as the action of the user. For example, when dragging a scrollbar, it follows the cursor with the same acceleration and the same speed so that the relative position of the cursor on the scrollbar doesn't change. Modality is a more abstract term that describes what a user expects to see, feel or hear when a certain action is performed. Wensveen explains modality by giving this example: *the touching of objects can cause a sound or moving an object can be visually perceived*. So to couple action and function, the feedback must react in a way that the user expects it to react based on his experience in the real world. To further clarify this, we present the example of a slide button in an indent. The user expects the button to move within the indent when he drags it, but he also expects it to stop at the end of the indent. The coupling in expression also needs some more clarification. What is meant with a coupling in expression is that the feedback has a metaphorical link to the way the user performs the action or to the function to which the feedback belongs. Wensveen's best example is that of the LED on an Apple Powerbook®. He states that *the light is an indication of the sleeping state of the system and has the same expression as a relaxed breathing rhythm*. How this is interpreted in this thesis is made clear by the following example. On a Motorola Moto G® 3rd generation with Android®, an overview of all the active apps can be shown. To close an app, the user has to swipe it either left or right. The swiping expresses the act of throwing something aside because it is no longer needed which clearly links to the function of closing the app.

Each aspect can be coupled through three kinds of feedback or feedforward information: functional, inherent and augmented. Functional information is perceived when the activated function is actually performed. For example, the appearance of the letter *A* when the *A* keyboard button is pressed. Inherent information is feedback inherent to the medium that is used, such as the movement and clicking of a physical keyboard button when it is pressed. The previous examples are all examples of feedback. An example of inherent feedforward is the presence and shape of the *A* key on the keyboard which indicates that it can be pressed. Augmented information is information that is added by the designer and is only bound to the action or function due to the context it appears in. For example, in most operating systems, the progress of moving a file from one drive to another is indicated by a progress bar. This progress bar is not inherent to the function or the medium, it is only bound to the function by the context it appears in.

In his article, Wensveen already stated that a NUI, such as designed in this thesis, has almost no inherent information because there is nothing that the designer did not specifically add to inform the user of an action possibility or an action's effect. There is no physical object with which to control the NUI. The only true inherent information on a Kinect is the presence and direction of the camera which indicates that something is registered in that direction. The original article groups all augmented information together to make the link between action and functional information. In this situation, there are two layers of augmented information: the information given by the puppet and the information given by the UI elements such as the rope and the scrollbar. How this works will become clear after the analysis of the puppet and the first UI element. The tool is best used to analyze every action with its respective function separately, but to keep this discussion relatively short only the connections between the user and the puppet, the puppet and the pulling of the rope, the rope and the start of the recording are discussed in detail. The rest is done in more general terms. All links described in the following sections are visualized in figure 6.1.

The coupling between the user and the puppet is crucial because it is the user's way of interacting with other UI elements. If the puppet does not feel natural to the user, the rest of the interaction also feels unnatural. The puppet is coupled to the user in time because there is no delay between the user's movement and the puppet's movement. The hand of the puppet also turns red immediately when the user closes the respective hand. Of course, this is only true when the computer is able to process the information fast enough. This is where the choice for a puppet over a shadow plays a major role, as discussed in chapter 3. There is no coupling in location because the puppet is not near the same three-dimensional location as the body of the user. The coupling in direction is strong because the puppet follows every move of the user, though it breaks down when the user tries to reach down below. The dynamics are coupled very

well because the puppet's limbs make the same proportional and relative motion as the user so that the position, acceleration and speed are all proportional to the user's. The link in modality is weaker because there are many situations where the puppet differs from the user's reality, such as the limbs disappearing when the user moves out of the field of vision and the fact that the user cannot feel any of the UI elements that the puppet touches. There is also no expression because there is no metaphorical link between the action of the user and the puppet.

The rest of the UI elements only interact with the puppet. They are represented in figure 6.1 as the second augmented information block. The rope, as seen in figure 3.15, to start a recording is coupled to the puppet in time because it changes color and moves only when the puppet touches it. It is also coupled in location because it only changes around where the hand of the puppet is located. The coupling in both direction and dynamics is apparent because the rope moves downward with a downward move of the hand at the same acceleration and speed. Besides the fact that the rope can be grabbed like a real-life rope, the coupling in modality is weak because it doesn't react to the user's touch with the same physics that a rope would. There is no link in expression because it doesn't matter how the puppet pulls the rope. The effect is always the same.

The activated function, i.e. starting the recording, is only coupled in time with the rope UI element because it doesn't matter where, in which direction or with what speed the user pulls the rope. It always results in the recording window appearing. There is however a more direct coupling between this function and the puppet: the record window appears on the location of the puppet.

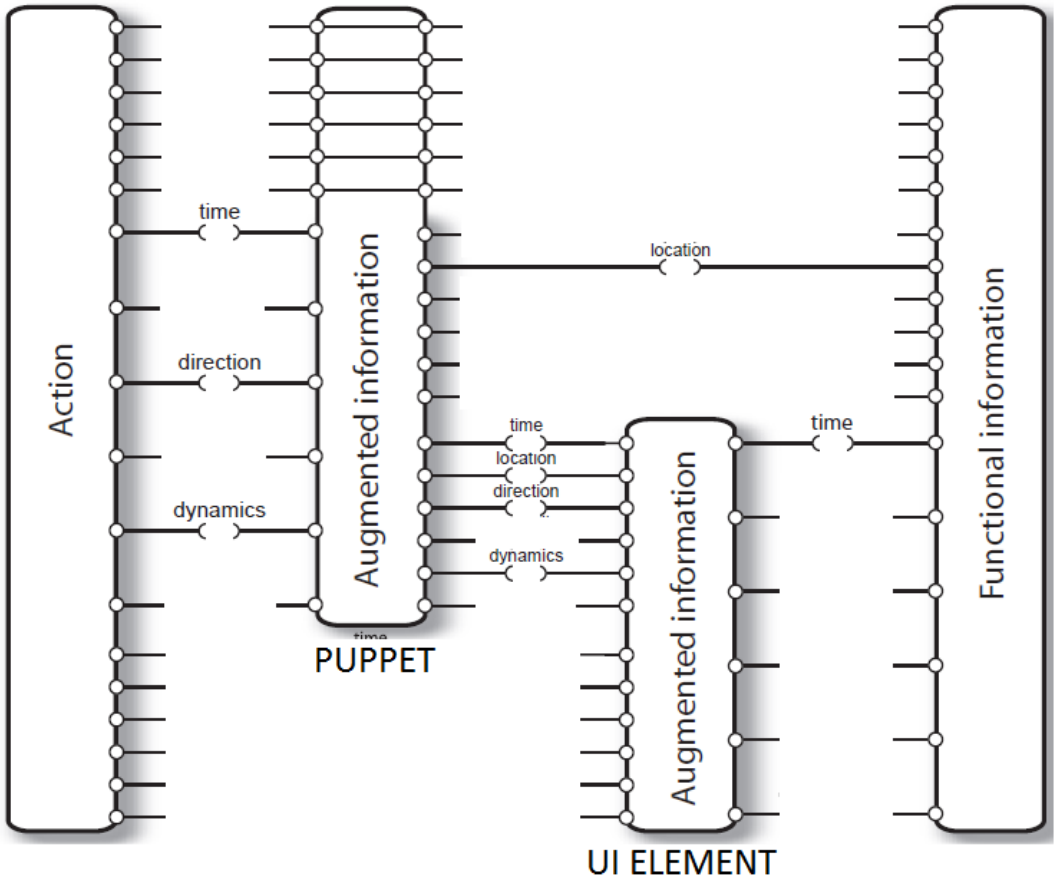

Figure 6.1: *Overview of the framework couplings for the puppet and the rope UI element*

The links from the user to the puppet stay the same for all of the UI elements. Links from puppet to UI element or function can be different. Assuming the computer can handle the workload, most of the UI elements are coupled in time with the puppet. A notable instance of time coupling is when the user wants to replay a recording, as seen in figure 3.18. Only when the puppet's hand hovers over the recording in the selection box area the recording starts to play. The change to green of the selection box is coupled to the puppet's hand in location. It makes the bridge to the replay screen because it also changes to green at the same time. The number of the recording is also shown on the screen at that time. The thorough link in time is because the coupling in location is so weak as mentioned before and no other couplings can be made. Both the scrolling and the delete slide have the same location, direction and dynamic coupling as the rope due to the general color coding and the fact that the movement of the UI elements are always the same as that of the hand that acts upon it. Though the delete action, seen in figure 3.20, has a special expression between the UI element and its function, the red color refers to signals like a red traffic sign or red traffic light that demand attention because they contain important information.

6.3 Reflection on the process

There are different approaches for designing an interface, but it always comes down to not treating the interface as an afterthought. As such, integrating the interface design right from the start with the back-end software design is essential. For instance, requirements concerning the responsiveness of the interface are for a big part set by the way the back-end software is designed and implemented. In other words, if the back-end is programmed inefficiently, this is reflected by a lack of responsiveness in the GUI.

However, it is necessary to switch fast from interface prototyping to an actual implementation. Since the early prototypes are static and non-interactive, it is only possible to really evaluate a decision after having an implementation of the concept and to obtain feedback from the users it is intended for. In other words, while static prototypes can undergo several iterations until they are *perfected* in theory, implementations in practice are needed to really convey the feel of the interface.

To come up with ways of interacting with the application other than relying on traditional controls, an experimental approach is tested. The idea is to interview a physical therapist in order to obtain ideas from him concerning the interaction with the application. This makes it possible to find solutions that fit the needs and expectations of the user. Early tryouts of this experimental technique show that persons are reluctant of coming up with ideas they are not familiar with. It is more assuring and easier to fall back on traditional, well-known elements.

In the same sense, user testing often leads to results implying that more conventional interactive elements are easier and quicker to understand than others, even if they rely on making use of the mime pattern discussed before. This can be related to years of experience with other types of interfaces and confirms that there is need of a more streamlined way for gesture-based interaction that unifies all interfaces of this kind [7].

Including a scrollbar in a gesture-based interface is an example of this. The user has experience using scrollbars, so there are no problems to recognize the visual element as a list that can be scrolled through. However, interacting with it is more difficult initially. This can be related to differences in the way gesture-based and touchscreen-based interfaces work. On touchscreens, it is possible to swipe with a finger on the screen to scroll, then lift it and touch the screen again to scroll again. With gesture-based interfaces, the action analogous to lifting the finger is less obvious to achieve. One way is to have the user move his hand out of the scrollbar before he can scroll in the same direction again. This is how it is implemented for the OmniPlay application. A different approach is to use composite gestures. The user can close his hand to grab the scrollbar and scroll through it, then open his hand, reposition it and close it to scroll again. This

allows the action of scrolling to be faster, but it also adds the burden of having to remember more actions for interacting with this element. Additionally, it can be tiresome to close and reopen the hand many times in succession. In the end, this comes down to personal preference and requires more user tests before a more general conclusion can be made.

6.4 Future work

Physical therapist Dries Lamberts indicated that the action of recording and training gestures can be pleasant for patients to do it themselves. This is especially useful if the developed application is not exclusively used for rehabilitation, but also as a way to have some physical exercise on a regular basis, as is the case for children with physical disabilities. As the GUI of the application is focused on being used by therapists, a different interface is required to simplify and optimize its use for children. This includes features like pressing a keyboard key to be designed appropriately.

Another possibility for future work is to use the current back-end structure to allow the user to create patient-specific projects, to which all gestures are coupled. On startup of the application, it can be possible to require the user to enter the name of a patient to load the project with all gestures they used before. This decreases the time needed before a patient can start exercising and playing a game.

Finally, the application can be expanded by adding screens to the interface that allow the user to manage different gesture classes and projects, reusing assets that are present in the developed interface.

Chapter 7

Conclusion

The developed application, called OmniPlay, succeeds in offering a way to incorporate exercises with playing games. A physical therapist can come up with gestures that fit the needs of the patient and link each of these gestures to a keyboard key. As such, any game that can be played with keyboard controls can be played with this application, independent of the chosen gestures. This makes OmniPlay potentially more profitable than custom designed games, as it gives patients a very large pool of games to choose from and helps keeping them motivated to do all needed gestures.

A machine learning algorithm using support vector machines is used to evaluate the gestures performed by the user. When a recorded gesture is recognized, OmniPlay virtually presses the keyboard button that is assigned to that gesture. By employing a system of splitting the gesture into a number of postures, the OmniPlay can be used to effectively play games using gestures.

Using prototypes consisting of mime or hints patterns of interaction combined with WIMP elements, user tests are conducted to come to the conclusion that a combination of elements following the mime pattern and WIMP elements is preferred. The most important aspects of the UI are its simplicity, that all possible actions are permanently on screen and usage of clear and familiar metaphors. From these criteria, a coded prototype is developed where the user can record and evaluate gestures to be used to play any game. A user test demonstrates that the physical therapist is able to input all required gestures without needing any programming knowledge. After some introductory explanation, he is able to find effective gestures on his own that he can use to play a game found online.

In the end design of the UI, a proper color coding of the UI elements combined with a well chosen activation threshold for important actions such as the delete button can give the user easy access to these functions without causing accidental activations. An analysis of the design using Wensveen's framework shows that a significant amount of coupling between the action and the function is done via two layers of augmented feedback and feedforward.

Bibliography

[1] Brauner P, Valdez AC, Schroeder U, Ziefle M. Increase Physical Fitness and Create Health Awareness through Exergames and Gamification. Human Factors in Computing and Informatics. 2013;7946(1):349 – 362.

[2] Annema JH, Verstraete M, Vanden Abeele V, Desmet S, Geerts D. Video games in therapy: a therapists's perspective. International Journal of Arts and Techonology. 2013;6(1):106 – 122.

[3] e Media Lab. KungfuKeuken. KULeuven; 2011. https://projects.groept.be/~emedia/?page_id=344.

[4] Geurts L, Vanden Abeele V, Husson J, Windey F, Van Overveldt M, Annema JH, et al. Digital games for physical therapy: fulfilling the need for calibration and adaptation. ACM; 2011. p. 117 – 124.

[5] Nakevska M, Vos C, Juarez A, Hu J, Langereis G, Rauterberg M. Using Game Engines in Mixed Reality Installations. Entertainment Computing - ICEC 2011. 2011;LNCS 6972:456 – 459.

[6] Jacob RJK, Girouard A, Hirshfield LM, Horn MS, Shaer O, Solovey ET, et al. Reality-based interaction: a framework for post-WIMP interfaces. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 2008;CHI 08(1):201 – 210.

[7] Norman DA. Natural User Interfaces Are Not Natural. Interactions. 2010;17(3):6 – 10.

[8] Shneiderman B. Direct Manipulation: A Step Beyond Programming Languages. IEEE Computer. 2010;16(8):57 – 69.

[9] Wensveen SAG, Djajadiningrat JP, Overbeeke CJ. Interaction frogger: a design framework to couple action and function through feedback and feedforward. Proceedings of the 5th conference on Designing interactive systems. 2004;DIS 04(1):177 – 184.

[10] Hondori HM, Khademi M. A Review on Technical and Clinical Impact of Microsoft Kinect on Physical Therapy and Rehabilitation. Journal of Medical Engineering. 2014;2014(1):16.

[11] Lange B, Koenig S, McConnell E, Chang CY, Juang R, Suma E, et al. Interactive game-based rehabilitation using the Microsoft Kinect. Virtual Reality Short Papers and Posters. 2012;p. 171 – 172.

[12] Chang YJ, Chen SF, Huang JD. A Kinect-based system for physical rehabilitation: A pilot study for young adults with motor disabilities. Research in Developmental Disabilities. 2011;32(1):2566 – 2570.

[13] Verbouw B, van den Bergh W. Machine Learning to create a motion controlled application [Master Internet computing]. KU Leuven, GroepT; 2015-2016.

[14] Peng W, Lin JH, Crouse J. Is Playing Exergames Really Exercising? A Meta-Analysis of Energy Expenditure in Active Video Games. Cyberpsychology, Behavior, and Social Networking. 2011;14(11):681 – 688.

[15] Staiano AE, Calvert SL. Exergames for Physical Education Courses: Physical, Social, and Cognitive Benefits. Child Dev Perspect. 2011;5(2):93 – 98.

[16] Dahl-Popolizio S, Loman J, Cordes CC. Comparing Outcomes of Kinect Videogame-Based Occupational/Physical Therapy Versus Usual Care. Games Health J. 2014;3(3):157 – 161.

[17] Microsoft. Kinect for Windows SDK 2.0. Microsoft; 2014. https://www.microsoft.com/en-us/download/details.aspx?id=44561.

[18] Green P, Wei-Haas L. The Wizard of Oz: a tool for rapid development of user interfaces; 1985. Available from: https://books.google.be/books?id=XhWEpS1OooAC.

[19] Chang CC, Lin CJ. LIBSVM - A Library for Support Vector Machines; 2016. https://www.csie.ntu.edu.tw/~cjlin/libsvm/.

[20] Microsoft. Human Interface Guidelines v2.0. Microsoft; 2014. http://download.microsoft.com/download/6/7/6/676611B4-1982-47A4-A42E-4CF84E1095A8/KinectHIG.2.0.pdf.

FACULTY OF ENGINEERING TECHNOLOGY
CAMPUS GROUP T LEUVEN
Andreas Vesaliusstraat 13
3000 LEUVEN, Belgium
tel. + 32 16 30 10 30
fax + 32 16 30 10 40
fet.groupt @kuleuven.be
www.fet.kuleuven.be