Leap Blender: A Software Testbed for Bare Hand Input in 3D Graphics Editing

Bachelor's Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Sven Jung

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Torsten Kuhlen

Registration date: 28.05.2014
Submission date: 25.09.2014

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, September 2014
Sven Jung

Contents

Abstract **xiii**

Acknowledgments **xv**

Conventions **xvii**

1 Introduction **1**

1.1 Mainstream Graphical Input Device 1

1.2 Bare-Hand Input Device 2

1.3 User Studies for Graphical Input Devices . . 3

1.4 Contributions 4

2 Related Work **5**

2.1 Bare-Hand Input Devices 5

2.2 Leap Motion Controller in Research 6

2.3 Bimanual Interaction 7

3 Software Testbed **9**

3.1 Environment 9

3.2 Structure 10

3.3 Functionality 15

3.3.1 Displaying 15

3.3.2 Transformation 17

Leap Motion to Camera Transformation 18

Camera to Blender Transformation . . 19

3.3.3 Gestures 21

3.3.4 Pinch 21

3.3.5 Flathand 24

3.3.6 Modes 27

3.4 Extensibility 28

3.4.1 StudyLogger 29

3.4.2 Simulator 30

3.4.3 Settings 31

4 Case Study: Replicating Symmetric Bimanual Interaction Study 33

4.1 Experiment 33

4.1.1 Tasks 33

4.1.2 Setups 35

4.2 Metrics 36

4.3 Apparatus 38

4.3.1 Touchscreen Setup 39

4.3.2 Leap Motion Controller Setup 39

4.4 Participants 39

4.5 Design 39

4.6 Hypotheses 42

4.6.1 Each Setup 42

4.6.2 Overall 43

5 Results and Discussion 45

5.1 Results 45

5.1.1 Performance 45

5.1.2 Symmetry 47

5.1.3 Parallelism 49

5.1.4 Comparison of Setups 51

5.1.5 Comparison to Original Study 52

5.1.6 Limitations 52

6 Summary and Future Work 53

6.1 Summary and Contributions 54

6.1.1 Case Study 54

6.1.2 Software Testbed 56

6.2 Future Work 56

A Study user information 59

Bibliography **63**

Index **65**

List of Figures

1 Axis convention in 2D xviii

2 Axis convention in 3D xviii

3 Axis convention of Leap Motion Controller . xix

4 Graph convention for statistic evaluation . . xix

1.1 Overview of input devices 3

3.1 Overview of layer structure 11

3.2 Context of modularity 14

3.3 Overview of data-flow 15

3.4 Depth sense 16

3.5 Coordinate system transformations 18

3.6 Gestures 23

3.7 Customized scene example 28

4.1 Tracking task 34

4.2 Setup overview 36

4.3 M-metric for parallelism 38

4.4 Path overview 41

5.1 Evaluation of overall tracking performance . 46

5.2 Evaluation of symmetry (overview) 47

5.3 Evaluation of symmetry (hands) 48

5.4 Evaluation of symmetry (all factors) 50

5.5 Evaluation of parallelism 51

A.1 Consent form 60

A.2 User information form 61

List of Tables

2.1 Property comparison of bare-hand input devices. 6

5.1 Comparison of the hypotheses among the three setups. 51

5.2 Comparison of the results between our study and the original study. 52

Abstract

New bare-hand input devices have been invented to simplify the input for the user, especially in 3D manipulation tasks. These devices use free hand gestures as input. Many research questions about the behavior of the user using this new kind of input device are open. In this thesis the software testbed ”Leap Blender: A Software Testbed for Bare Hand Input in 3D Graphics Editing” is presented. It is a platform to manipulate objects in Blender with the Leap Motion Controller, which can be customized and extended. With this program tasks for user studies investigating bare-hand input can be created easily. No longer a new system for each research question and task has to be developed expensively, like it was done before.
In the end, a study which makes use of this testbed is presented to show how it can be customized. The study investigates the natural behavior of the user using a touchscreen and the bare-hand input device Leap Motion Controller to perform 2D and 3D tasks.

Acknowledgments

First, I would like to thank Chatchavan Wacharamanotham for his guidance and support.

Thanks to the people of the i10 for the nice work environment and to the people who participated in my study.

I would like to thank Prof. Dr. Jan Borchers and Prof. Dr. Torsten Kuhlen for their time.

Special thanks goes to my girlfriend, who supported me over the course of my thesis.

Thank you!

Sven Jung

Conventions

Throughout this thesis we use the following conventions.

Text conventions

The whole thesis is written in American English.

Definitions of technical terms or short excursus are set off in colored boxes.

Excursus:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
Excursus

Hypotheses are marked with a **H** and a number for identification.

H1:
Input device A is better than B.

Hypothesis:
H1

Source code and implementation symbols are written in typewriter-style text.

`myClass`

Experimental factors and conditions are emphasized:

factor

Graphical conventions

Two-dimensional axis convention:

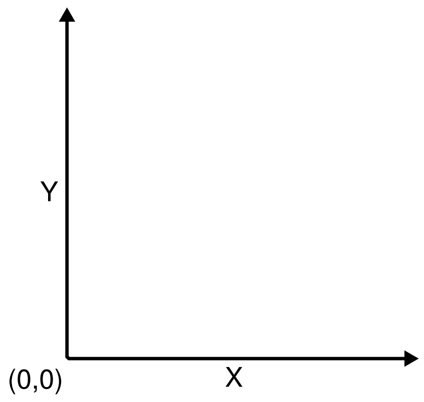


Figure 1: *Axis convention in 2D.*

Three-dimensional axis convention:

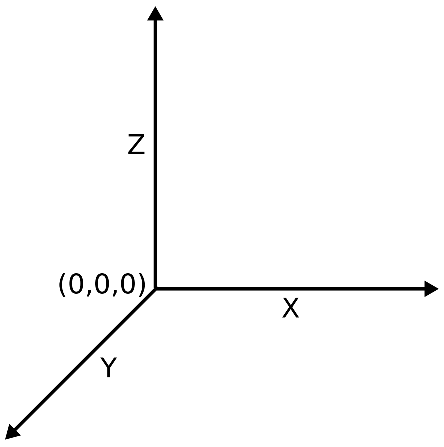


Figure 2: *Axis convention in 3D.*

Leap Motion Controller axis convention:

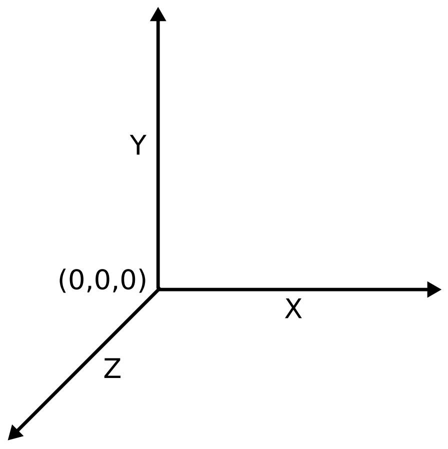


Figure 3: *Axis convention of the Leap Motion Controller.*

Evaluation conventions

Graphs with mean and 95% confidence interval are used for statistical evaluation:

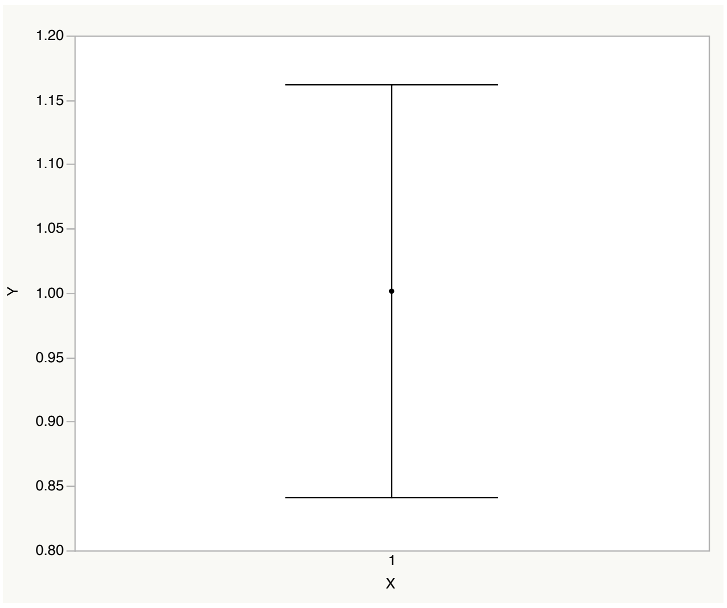


Figure 4: *Graph convention for statistic evaluation: using mean and 95% confidence interval.*

Formula conventions

Formulas are indented and italicized:

$$a^2 + b^2 = c^2$$

Vector notation:

$$\overrightarrow{vec} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

Matrix notation:

$$Matrix_{purpose}$$

The coordinate system of a vector is marked with an inferior character, L for the Leap Motion coordinate system, C for the camera coordinate system, and B for the Blender coordinate system:

$$\overrightarrow{vec}_B$$

Chapter 1

Introduction

This chapter is an introduction to the topic of input devices and the problem of finding the best use of them in programs to provide a most natural usage to the user. Furthermore, the topic of symmetric bimanual interaction is introduced and the contributions of this thesis are outlined.

Necessity of an input device

To communicate with a computer, an input device is necessary. Many kinds have been invented and analyzed concerning the natural use by a user in order to integrate them in programs and make the usage easy and intuitive.

1.1 Mainstream Graphical Input Device

Mouse is a proven standard input device.

The first and still mostly used device is the mouse (figure 1.1a), which in combination with a keyboard is the standard input device for the computer. Most of the graphical user interfaces are optimized for a mouse and every new kind of input device has to fight against it - a proven, accurate, and fast hardware. But there are also negative aspects, like the time for homing[1] and a relative mapping[2].

[1]Time to switch from the mouse to the keyboard

[2]The movement of the mouse is transfered to a movement for the cursor. The position of the mouse and the position of the cursor are not interdependent

Touchscreen allows direct manipulation.

Another kind of input device is the tablet or touchscreen (figure 1.1b). Even if it has not reached to replace the mouse completely, it is an established way of communicating with a computer today. The advantage of them is a direct input[3], what provides an intuitive usage and a faster pointing. But they have a lack in accuracy - programs have to be adapted or designed especially for this input device to provide a GUI[4], which can be operated easily. They are used where it is annoying to have a mouse or an intuitive usage is required - for example, hand-held tablets or info-screens.

1.2 Bare-Hand Input Device

Bare-hand devices add a 3rd dimension and are more intuitive.

People always dreamed of using the 3D space for input and with that the most natural mapping. They are called bare-hand input devices and let the user use hand gestures for interaction. Conceptual models have been invented, like the Mockup Builder from Ara´jo et al. [2012], which is a combination of a touchscreen and a 3D input above. The problem is that the hardware is expensive and restricts the free movement of the user. More commercial products are the Kinect[5] from Microsoft or the Vicon[6] system. Both are expensive and require good programming skills to integrate them into an application. The Leap Motion Controller[7] (figure 1.1c) is small, less expensive, and makes use of the space above the keyboard. It also provides a simple API for several languages. Even if bare-hand devices are still expensive and not that accurate, the further development will go on. The great advantage for 3D graphics editing is that this kind of device adds a 3rd dimension.

[3]Objects can be manipulated directly by touching them on the display.
[4]Graphical User Interface
[5]www.kinect.com
[6]www.vicon.com
[7]www.leapmotion.com

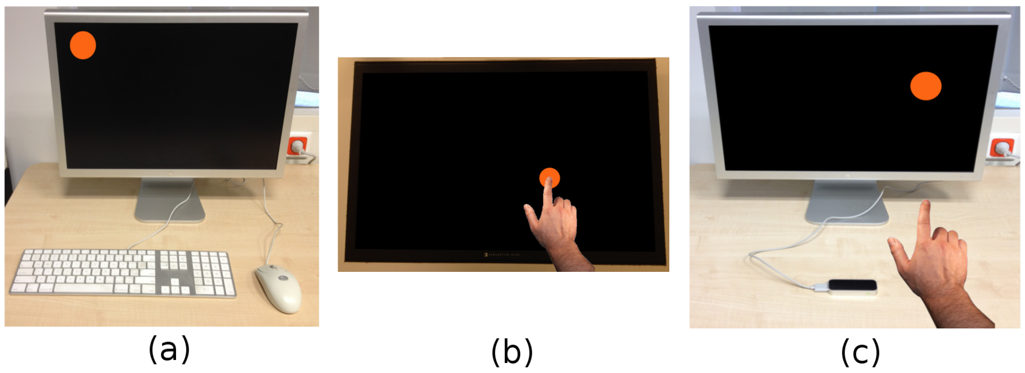


Figure 1.1: *Overview of input devices: (a) mouse, (b) touchscreen and (c) Leap Motion Controller.*

1.3 User Studies for Graphical Input Devices

Time intensive studies for investigating how to interact with a device. Testbed shortens the implementation phase.

In order to compare different input devices and find the best method of handling an input device in computer programs, studies are necessary. Independent from the input a study needs a well structured task to gather data. Besides, hypotheses, the right experimental setup, and the implementation of the task is a main factor in the planning of a study. Even if the structure of the input is often the same (e.g. coordinates of the input device), every study group implements its own system. The drawback of implementing everything from scratch is that common parts have to be reimplemented all the time - for example, logging, coordinate transformation, and gestures. Additionally, a reimplementation is error-prone. These factors take a lot of time of a study process. This is where the testbed Leap Blender comes into, which provides base functionality for implementing tasks of a study. That shortens the implementation process to study specific requirements. Details are given in chapter 3 “Software Testbed”.

A study of symmetric bimanual interaction tests the testbed.

In order to test the quality and the real benefit of the testbed, a case study was conducted. Especially bare-hand devices are not yet studied much and studied behavior of the mouse and touchscreen has to be checked for its validity with bare-hand devices. So, there are many open research questions. Because the interaction with bare-hand input devices is no longer bound to a physical device, bimanual interaction[8] came up. This new interaction principle enables the user to perform two tasks at the same time. Whereas asymmetric assignment[9] is already studied much, the less studied symmetric assignment[10] of tasks to the hands is very effectively in 3D editing. The study of how attention, task speed, and visual integration affect the performance and the motions of the user enabled us to derive patterns for designing interfaces. Further information are given in chapter 4 "Case Study: Replicating Symmetric Bimanual Interaction Study".

1.4 Contributions

The main contributions are the following:

- Design and implementation of the testbed "Leap Blender" to provide fast task development and study conduction for bare-hand input devices (Chapter 3 "Software Testbed").
- A user study about the natural behavior of the user performing a symmetric bimanual object tracking task with a touchscreen for 2D input, the Leap Motion Controller for 2D input, and the Leap Motion Controller for 3D input (Chapter 4 "Case Study: Replicating Symmetric Bimanual Interaction Study").

In the following, existing systems and related studies are presented. Then the implementation of our own system is described and the setup and conditions of the study is presented. Afterwards, the study are discussed and evaluated. In the end, a summary of the thesis, future work, and possible additional studies are given.

[8]Two-handed input
[9]Two different roles for each hand
[10]Identical role for each hand

Chapter 2

Related Work

The previous chapter pointed out the most common input devices and the bare-hand devices as the current research area. Many different approaches came up in the last years with different advantages and disadvantages. This chapter discusses different bare-hand input devices and their comparison. Besides, information about the by this new kind of device enabled bimanual interaction are given.

2.1 Bare-Hand Input Devices

There are several bare-hand input devices.

The most known bare-hand input device is the Kinect[1] from Microsoft. Actually developed as input device for games it is used for any kind of body movement recognition today. The 3Gear[2] system, for example, provides a software kit to get movement and gesture information from the Kinect device for further use in applications. A wearable approach are Data Gloves[3], which recognize finger movements of the user. In 2012 a controller especially for mid-air gestures was published, the Leap Motion Controller[4] . These devices vary in different properties and have different advantages and disadvantages (see comparison in table 2.1).

[1]www.kinect.com
[2]www.threegear.com
[3]www.vrealities.com
[4]www.leapmotion.com

	Kinect	3Gear	Data Gloves	Leap Motion
Price	200 €	220 €	150 € -1500 €	100 €
API	C#	C++, C#, Java	C	6 common, web port
Space needed	large distance	big stand	none	small above keyboard
Accuracy	good	good	just fingers	middle, but improved
Applications	Games	No	No	App Store
Gestures	Grip, Point	No	No	Circle, Swipe, Tabs

Table 2.1: Property comparison of bare-hand input devices.

Leap Motion Controller is a good choice as representative for bare-hand devices.

The Leap Motion Controller is suitable as representative for bare-hand input devices. It is commercially available and the software is improved continuously to provide better accuracy and more gestures. "Comparable controllers in the same price range, e.g., the Microsoft Kinect, were not able to achieve this accuracy," said Weichert et al. [May 2013]. Furthermore, it is quite inexpensive compared to other input devices. The dimensions of the controller are just 7.6 x 3 x 1.3 cm on a desktop where already a keyboard and a mouse are. It has a very good compatibility by supporting common programming languages and a web port. Since it was developed only for motion and gesture recognition, the provided information are presented very well: wrapped into gesture, finger, and hand objects with certain attributes. These advantages, compared to the other devices, make the Leap Motion Controller interesting for several studies.

2.2 Leap Motion Controller in Research

Several studies used the Leap Motion Controller.

The Leap Motion Controller has been used in many different studies investigating bare-hand input.

Vatavu et al. [2014] used the controller to evaluate different gestures for controlling a tv. Potter et al. [2013] conducted a study about the potential of the Leap Motion Controller to recognize the Australian Sign Language. To improve creativity, Sutton [2013] used the controller to create an air painting application. Zubrycki et al. [2014] compared the Leap Motion Controller to the 3Gear system in order to control a 3-finger gripper.

Leap Motion Controller is used for 3D editing.

Nunnari et al. [Last visited: Juli 2014] wrote a Blender plug-in to use the Leap Motion Controller for manipulation of 3D scenes. Since Blender is a ready-to-use 3D editing environment, its functionality can be used to model objects and create scenes easily. The plug-in then uses the data from the Leap Motion Controller to manipulate objects.

2.3 Bimanual Interaction

Symmetric interaction outperforms asymmetric interaction.

Buxton et al. [1986] and Balakrishnan et al. [1999] showed that bimanual techniques outperform one-hand techniques. In our study we used bimanual interaction as a test case. A model to describe this interaction technique is the kinematic chain model of Guiard [1987] , which says that naturally one hand follows the other. Bimanual interaction can be symmetric or asymmetric - symmetric bimanual interaction means an identical assignment of roles to the hands, asymmetric a different assignment of roles. Results of Casalta et al. [1999] showed that symmetric assignment results in better performance and parallelism.

Several factors influence symmetric bimanual interaction.

Balakrishnan et al. [2000] investigated how symmetric bimanual tasks are performed and which potential factors may influence symmetric bimanual interaction. They found out that increased difficulty, divided attention, and missing visual integration lead to a more sequential conduction of a symmetric task. Also, there is a slight asymmetry of 8% between the left and right hand, regarding to the error.

This chapter showed that the Leap Motion Controller is suitable as representative for bare-hand input devices. There is a great interest of using the Leap Motion Controller as bare-hand input device in studies to answer questions of many different kinds. Therefore, a testbed which provides basic functionality would be a great simplification and save time for further studies. In the next chapter our testbed is presented and detailed design decisions are given.

Chapter 3

Software Testbed

In order to avoid the explained drawbacks of a reimplementation and to save time when designing a study, we implemented a testbed with some base functionalities which can be easily customized. Because there are already some studies with the Leap Motion Controller, there seems to be the need of such a testbed. This chapter is about the software testbed we developed and gives detailed information about design decisions and the structure of the program. Furthermore, steps how to customize the testbed for a study are explained. Please find an instruction of how to install the software in the “install” directory of the program files.

Customizable testbed safes time and avoids drawbacks of reimplementation.

3.1 Environment

As environment the open-source[1] 3D graphics editing software Blender[2] is used. It is free for everyone and any kind of use. Besides, the main distributions are supported - Windows, Mac, and Linux - and the source code is open. Blender provides functionalities for creating complex scenes, modeling objects, animations, and rendering. Although no very special tools are needed for creating a study

Blender is used as environment, because of 3D graphics editing functionality.

[1]GPL licensed

[2]www.blender.org

task scene, the base functionalities of displaying, modeling, and moving objects are given and an own implementation would be error prone and take a long time. Furthermore, Blender provides a Python[3] integration, so that manipulations can be done by using a large and well-documented API with an interactive console and a text editor to write scripts. This allows to create complex tasks with the tools of Blender and add logic by scripts.

3.2 Structure

Layer structure encapsulates the different parts.

There are four main actors in this program: Blender as environment, the scene of Blender for handling the objects and the appearance, the Leap Motion Controller which provides the data of detected hands and fingers on a web port, and finally Python to manipulate the scene by the provided API in order to add logic. Figure 3.1 shows the layer structure of the testbed. This layer structure encapsulates the design from the logic and the data part. The functionality of the testbed is implemented in Python to process the data of the Leap Motion Controller. The object data are exchanged between Python and Blender in order to manipulate the current state of objects. Blender automatically updates the scene according to the states set by the Python module or uses the scene to get the information requested by the Python module.

Interface structure for adaptability of the testbed.

The explained layers and an interface structure are chosen to make the testbed easily customizable (see figure 3.1). To use the testbed for a study, first the appearance of a task has to be designed in the scene with the functionality of Blender. Afterwards, the `Simulator` can be modified to add custom logic to the scene. The `StudyLogger` can be customized to gather task specific information for an evaluation. Of course the structure of the different Python modules of the testbed can be extended to add more base functionality. A more detailed explanation of how to customize these modules can be found in section 3.4.

[3]Interpreted high-level programming language

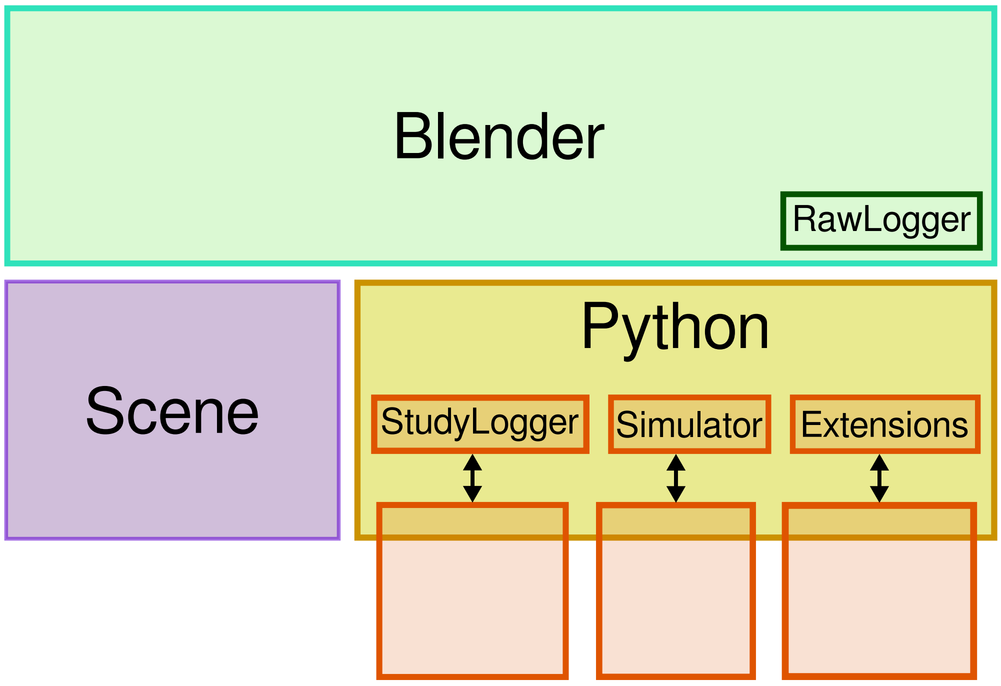


Figure 3.1: *Overview of the layer structure, which encapsulates each part from the others.*

For the Python part of the testbed a modularization is chosen to make the testbed understandable and extendable. Each of the modules has its own area of responsibility (see classes and association in functionality in figure 3.2). Figure 3.3 shows the data-flow of the data of the Leap Motion Controller between the python modules and the concurrency of receiving and processing.

Modularization of the Python part supports extensibility of the testbed.

The main Python module is the `LeapModal` operator,[4] which handles the singletons, the data handling, and the sequential processing of the data. This operator is continuously called by Blender (50 Hz; can be changed in the settings) in order to update the scene according to the data of the Leap Motion Controller. A separate `LeapReceiver` thread receives and stores the data of the Leap Motion Controller, so that the stored data are always the newest. Since the `LeapReceiver` works with a blocking socket connection, we assume that the update rate equals the up-

Detailed description of the modularized structure.

[4]Blender Operator: modal execution

date rate of the Leap Motion Controller, which is up to 200 fps and depends on the number of detected objects. The data are inquired by the main operator in each update step and given to the `Manipulator`. The `Manipulator` uses the data structure of the Leap Motion Controller to update and create `Finger` and `Hand` objects, which build the data model of the testbed. These objects implement the functionality of independently displaying the fingertips as cursors. At initialization time the `Finger` and `Hand` objects get passed the data structure of the Leap Motion Controller and then use the `LeapSelector` to extract necessary information. The `Manipulator` also calls the `GestureHandler`, which detects gestures and creates gesture objects by using the available `Finger` and `Hands` objects. The `GestureHandler` then updates the gesture objects in each update step (see section 3.3.3 for more information about the gesture recognition). A gesture object handles the task and the end of the gesture by using information provided by the `Hands` and the `Fingers`. This structure of independent gesture objects and a central gesture management enables an easy extensibility. After the call of the `Manipulator` the `LeapModal` operator calls the `Simulator`, where the scene can be manipulated in each update step, like moving an object around (see section 3.4.2 for more information about adding logic to the scene). Finally, the main operator calls the `StudyLogger` module, where study related states of objects can be logged into a file or on a console (see section 3.4.1 for information about logging). The structure of calling the `Simulator` and the `StudyLogger` frequently during an update step provides the interface for a customization of the two main requirements for a study: to animate the task and to log necessary information. Four singletons can be called by every module and provide general functionality. This avoids a confusing structure of passing the objects to a module which needs them. The function `getLogger(name)` returns a logger object, which can be defined in the settings and used in a certain scope to have different logger for different modules to log debug information to the console. A `Transformator` provides functions to transform coordinates from one coordinate system to another (for explanations of the used transformations see section 3.3.2), to get projections of a vector, or to get camera parameter. The

`LeapSelector` offers functions which take the data provided by the Leap Controller and return requested parts of it. This module is necessary to have the structure of the input device data independent and replaceable from the structure used by the testbed. The last singleton is the `ObjectHandler`, which handles the objects used for displaying and provides functions to manipulate them in order to be able to manage the objects at a central place and keep the scene clean. All parameters are handled by a `settings` file, which is included by every module to provide a place where the testbed can be adjusted. In this file parameters like the update rate of the `LeapModal` operator, the debug level of the logger of each module, or the thresholds for starting a gesture can be adjusted. For the full list of parameters have a look into the settings file, each parameter is documented there.

GUI for controlling the testbed.

The GUI is designed by a Panel with buttons to call different operators. For example, the "Start" button calls the `LeapModal` operator to start the program. Some operators set global flags to handle if different modules of the testbed are called during the sequential processing of the `LeapModal` operator. Therefore, the "Start logging" button calls an operator which sets a logging flag. Then, during the update step the `StudyLogger` is called.

`RawLogger` in Blender source code for basic logging functionality.

In order to get a better performance, basic logging functionality is outsourced to Blender itself so that with the `StudyLogger` module only task related logging has to be done (See figure 3.1). This `RawLogger` is called internally in Blender and can be started and stopped separately from the GUI. It logs selections and movements of objects and points of meshs. Also button and mouse presses are captured. For that, a logging module is added to the source code of Blender. This module checks in each update step if the selection or the position of an object has changed and logs changes into a log file.

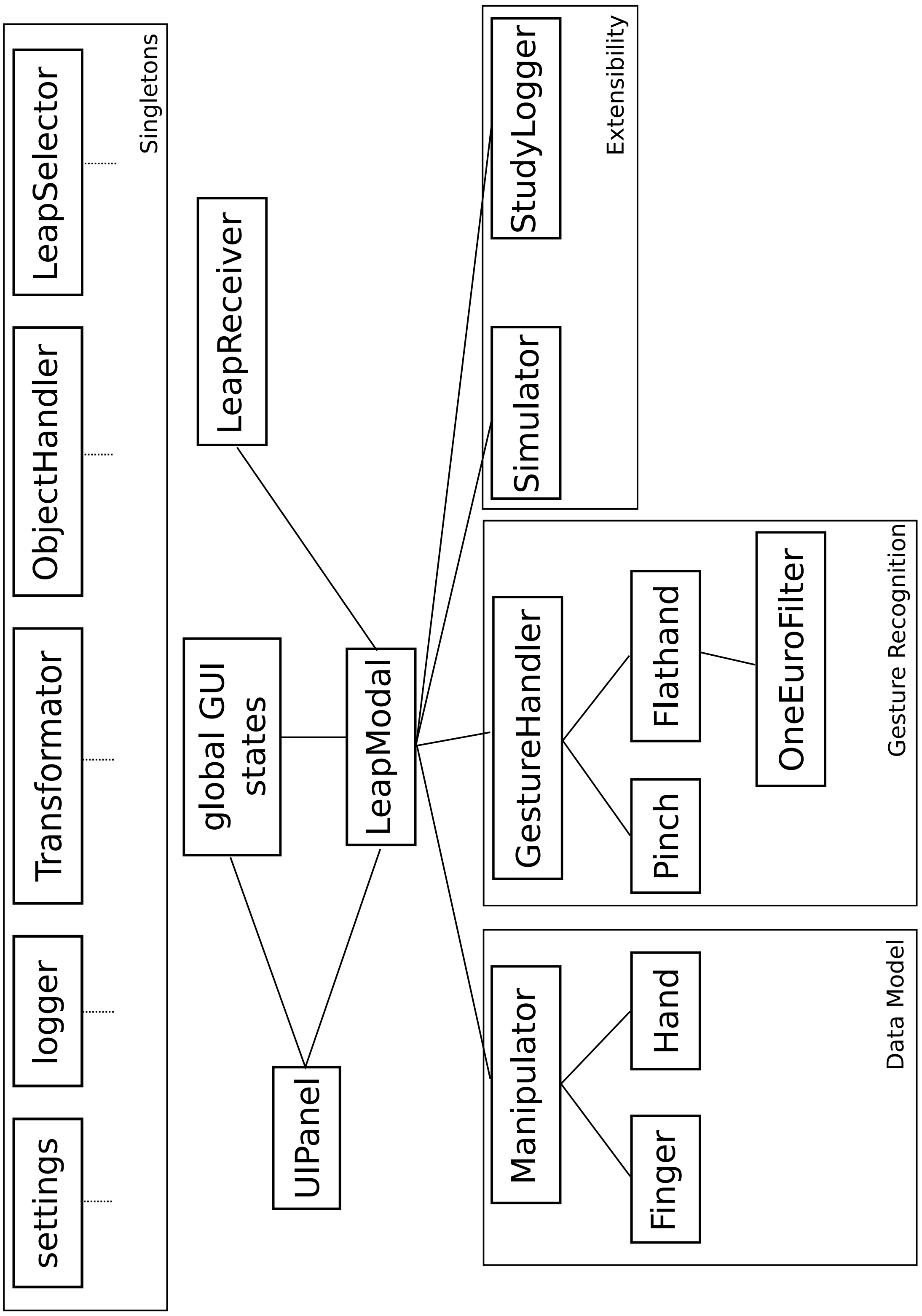


Figure 3.2: *Context of the different Python modules involved in the testbed and association in functionality.*

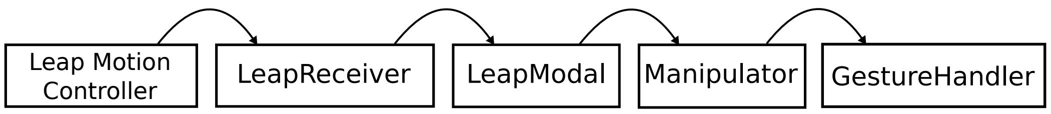


Figure 3.3: *Overview of the flow of data, detected by the Leap Motion Controller, in the program.*

3.3 Functionality

Besides a modularized structure, there are other basic functionalities provided by the testbed.

3.3.1 Displaying

Each `Finger` displays a sphere as cursor at the fingertip position.

To use a bare-hand input device, the user needs a point of reference in the program to know where he is manipulating at the moment, like a mouse cursor. In this testbed spheres are used to display the fingertips. For that, each `Finger` object has the logic to independently request a free displaying object from the `ObjectHandler` at creation and return it at deletion time. This object is used by the `Finger` to display the fingertip in the view of the camera. The position of the displaying object is adapted in each update step according to the data from the Leap Motion Controller. The distance of the cursors to the camera can be edited in the settings. To provide a sense of depth, additionally to the cursor spheres a torus is shown at the projection of each fingertip onto the ground plane:

Torus and cylinder provide a sense of depth.

$$\overrightarrow{cursorLoc_B} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

$$\overrightarrow{torusLoc_B} = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$$

A cylinder connects the sphere and the torus (see figure 3.4). The length of the cylinder is adapted to the distance between the cursor sphere and the torus and the location

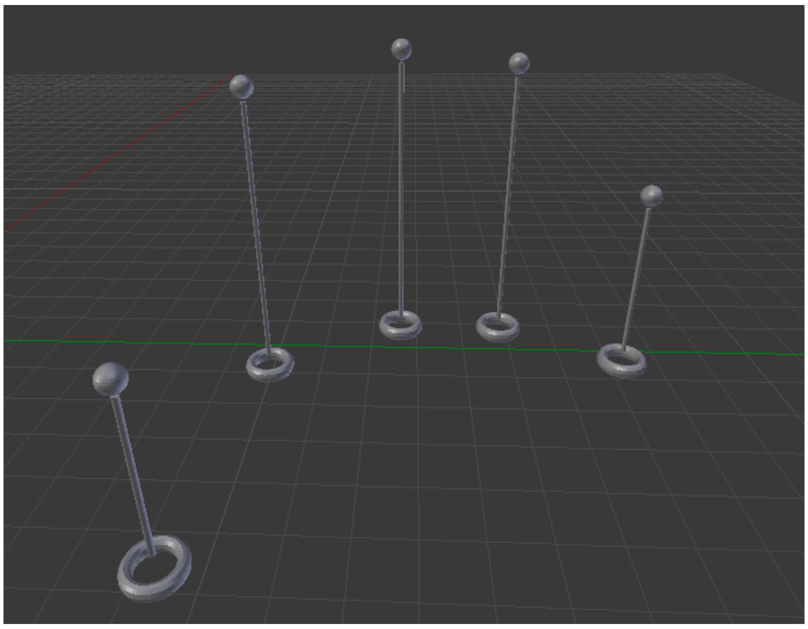


Figure 3.4: *Displaying a torus and a vertical cylinder as addition to the finger cursors in order to provide a sense of depth.*

of the cylinder is set to the point in the middle between the sphere and the torus:

$$\overrightarrow{cylinderLoc_B} = \begin{pmatrix} x \\ y \\ z/2 \end{pmatrix}$$

$$cylinderLen_B = |z|$$

These two objects are also managed by the `Finger` itself. The idea is taken from the system Wang et al. [2011] developed.

Selected objects are colored.

In order to provide feedback to the user if something is selected, selected objects are colored. For that, the `Pinch` gesture hides all `Finger` objects of the gesture if a grab is detected and calls the `ObjectHandler` to color the selected object. If the grab is released, the `Finger` objects are displayed again and the `ObjectHandler` requested to remove the color of the object.

3.3.2 Transformation

Transformations of the data between the coordinate systems are necessary.

The Leap Motion Controller and Blender have their own coordinate system what requires a coordinate transformation before displaying. Additionally, there is the coordinate system of the camera to provide displaying of the cursors according to the current view. For this transformations the `Transformator` singleton is responsible and provides functions to transform coordinates from one coordinate system to another. As described in section 3.2, the `Manipulator` passes the data structure of the Leap Motion Controller to the `Hand` and `Finger` objects at creation and at update time. Therefore, the objects get the data in the coordinate system of the Leap Motion Controller. Then, the necessary information are extracted by using the `LeapSelector` and transformed to the coordinate system of the camera by using the `Transformator` to have the right scaling and axis directions (see the differences between the coordinate systems in figure 3.5). Each object stores the data in the camera coordination system. For displaying the coordinates are transformed to the coordinate system of Blender to display in the view direction of the user - for example the fingertip positions to display the cursors always from the perspective of the user, like mouse pointers do. Also the gestures use the `Transformator` to convert calculated changes from the camera coordinate system to the coordinate system of Blender before application, like the Pinch gesture for moving an object into a certain direction from the point of view of the user.

Distilled representations save time intensive calculations.

To avoid repeated time expensive calculations each time a transformation is called during one update step, the `Transformator` is updated at the beginning of an update step and the transformation matrices are calculated according to the current view. Then, if a transformation function is called, there is only a multiplication of the vector with the transformation matrix. The matrices are calculated by the concatenation of different matrices for translation, rotation and scaling, what is described in the following.

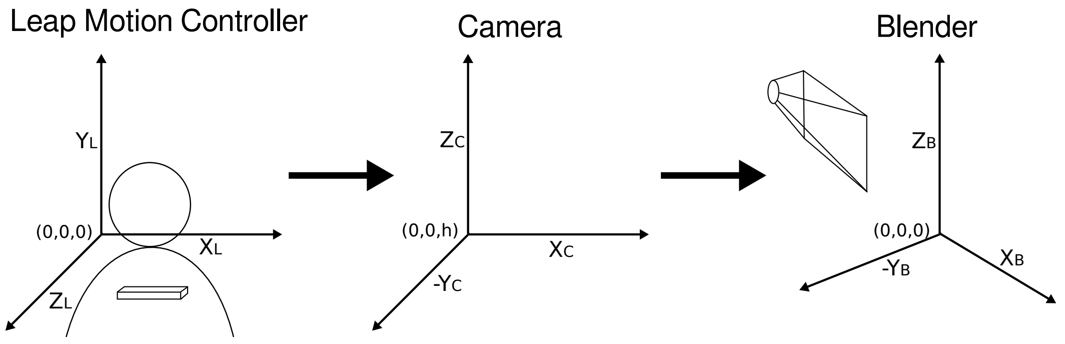


Figure 3.5: *Transformation of the Leap Motion coordinate system to the camera coordinate system for calculations and from the camera coordinate system to the Blender coordinate system for displaying.*

Leap Motion to Camera Transformation

Correct the origin and the axis direction for calculations.

This transformation is needed, because the Leap Motion Controller uses another axis direction then Blender. Besides, the origin of the coordinate system of the Leap Motion Controller is on the surface of the device what makes it impossible to make manipulations below the ground plane. To correct both, a translation of `LEAPOFFSET` steps upwards (editable in the settings) and a rotation of 90 degrees around the x-axis is done (see figure 3.5):

$$M_{LeapToCam} = Rotation_{(radians(90),\ 'X')} * Translation_{(-1\ *\ \texttt{LEAPOFFSET})}$$

The fact that the y-axis of the camera coordinate system is pointing backwards does not matter, because this is automatically taken into account. The matrix for transformations from the camera to the Blender coordinate system includes this in the rotation of the camera around the z-axis.

Used before saving data in internal model.

This transformation is used by the `Hand` and `Finger` objects to convert the extracted coordinates from the Leap Motion data structure to the camera coordinate system before saving.

Camera to Blender Transformation

The transformation from the camera coordinate system to Blender is for the displaying and the manipulation to be from the perspective of the user and independent of the current camera position and rotation in Blender (see figure 3.5). It is distinguished between a point, a rotation, and a direction transformation, because different calculations are necessary.

Take position and rotation of the camera into account for displaying.

The point transformation is used to get the `Hand` and `Finger` positions relative to the Blender coordinate system for displaying. For that, additionally to the rotation and position of the camera in the Blender coordinate system, a certain distance of the point to the camera position is taken into account, because the center of manipulation is in the focus field of the camera. Therefore, the point is translated along the camera view direction to have it at a certain `distance` from the camera:

Point transformation approach.

$$\overrightarrow{cursorLoc_B} = \overrightarrow{camLoc_B} + \texttt{distance}_B * \overrightarrow{camDir_B}$$

The `distance` from the camera is different in each mode of the testbed (see section 3.3.6) and can be edited in the `settings` file. Besides, the point is rotated around the z-axis so that the point is in the view of the camera relative to Blender. For that, the Python API is used to extract the rotation of the camera r_z around the z-axis from the rotation matrix M_{camRot} of the camera to apply the same rotation to the point. Finally, the location of the point is scaled by the `DIRECTNESS` factor, because the Leap Motion Controller has a more fine coordinate system. The `DIRECTNESS` factor can also be edited in the settings. Then, the transformation matrix is calculated to have a compact representation which can be easily applied to the point by a multiplication:

$$M_{CamToBlender} = Translation_{(cursorLoc_B)} * Rotation_{(r_z, \, 'Z')} * Scale_{(\texttt{DIRECTNESS})}$$

No rotation of the camera around the x-axis and y-axis is taken into account to display the point relative to the ground plane and give the user a point of reference and avoid confusion (see figure 3.5).

Rotation transformation details.

The rotation transformation is needed to get angles from the camera coordinate system relative to the coordinate system of Blender. For example, if the camera is rotated 90 degrees around the z-axis, a rotation around the x-axis of the camera is a rotation around the y-axis of Blender. This transformation is used by the `Flathand` gesture to get an angle difference of the hand relative to the current camera position (see section 3.3.3 for information about gesture recognition). As result a rotation matrix is calculated for a given rotation in the camera coordinate system, so that the result can be directly applied. First, the axes of the camera relative to Blender are calculated. The current y-axis of the camera relative to Blender is the projection of the camera direction onto the ground plane (xy-plane), because the y-axis always points into the view direction of the camera (see figure 3.5):

$$\overrightarrow{curYaxis_B} = \overrightarrow{camDir_B} - \left(\frac{\overrightarrow{camDir_B} \cdot \overrightarrow{planeNormal_B}}{\sqrt{\overrightarrow{camDir_B} \cdot \overrightarrow{camDir_B}}}\right) * \overrightarrow{planeNormal_B}$$

Then, the current x-axis of the camera relative to Blender is calculated by using the Python API to rotate the current y-axis by 90 degrees. The z-axis always points upwards, because rotations around the x-axis and y-axis are not taken into account for displaying:

$$\overrightarrow{curZaxis_B} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

These axes are calculated while the `Transformator` is updated so that it is not necessary to calculate them each time they are needed. Finally, to transform a rotation r_C from the camera coordinate system to the coordinate system of Blender, the Python API is used to calculate a rotation matrix around a current axis $\overrightarrow{curAxis_B}$ of the camera relative to Blender:

$$M_{rot_B} = Rotation_{(r_C,\ \overrightarrow{curAxis_B})}$$

Direction transformation procedure.

There also is the need of transforming directions, because it is not necessary to apply a translation to a given direction, like what is done with point transformations. This kind of transformation is used by the gestures to transform the difference of two hand positions from the camera coordinate system to the Blender coordinate system, because the movement direction of an object depends on the current camera angle. A direction can be divided into part vectors along each axis:

$$\overrightarrow{dir_C} = \begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ y \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ z \end{pmatrix}$$

What means, to get the direction it is necessary to go x units along the x-axis, y units along the y-axis and z-units along the z-axis. For direction transformations these lengths along each axis of the camera have to be applied to the current axes of the camera relative to Blender:

$$\overrightarrow{dir_B} = x * \overrightarrow{curXaxis_B} + y * \overrightarrow{curYaxis_B} + z * \overrightarrow{curZaxis_B}$$

3.3.3 Gestures

Pinch and flathand gesture for manipulations.

As gestures, a pinching gesture (see figure 3.6(a)) for manipulating objects and a flat hand gesture (see figure 3.6(b)) for manipulating the camera are chosen. With this two gestures the main manipulations for study tasks - manipulating objects and the camera - are given. Special gestures can be added easily (see section 3.4).

3.3.4 Pinch

Pinch is natural grab gesture.

The pinch gesture is the natural expectation of how to grab an object and manipulate it, as one would do in the real world. Since there was no build in pinch gesture when the testbed was developed, an own recognition is written.

Activated by connecting two fingers.

To enable a pinch, two fingers have to be brought together and for releasing they have to be separated again. For the gesture recognition it is distinguished between creating a `Pinch` object, set a `Pinch` object active (object grabbed), and set a `Pinch` object released (let object go). In each

Details of pinch recognition.

update step the `GestureHandler` calculates the distance between any two `Fingers` of a `Hand`. A `Finger` has the logic to calculate the euclidean distance between itself and a given other `Finger`:

$$\overrightarrow{fingersDiff}_C = \overrightarrow{ownLoc}_C - \overrightarrow{otherLoc}_C = \begin{pmatrix} dx \\ dy \\ dz \end{pmatrix}$$

$$distance_C = \sqrt{(dx)^2 + (dy)^2 + (dz)^2}$$

If there are two `Fingers` of a `Hand` which are within the distance of releasing (`PINCHEND`) and creating the gesture (`PINCHSTART`), the GestureHandler creates a `Pinch` object. These distances can be set in the `settings` module. From then on the gesture is updated in each update step and independently checks the state of itself. By this, only a small part of the logic has to be in the `GestureHandler` and the main part is organized in the gesture itself. If there are more than the two fingers of the gesture or the distance between the two gesture fingers is larger than the `PINCHEND` parameter, the gesture is released. But if there is only one finger detected, the pinch gesture is set active (object grabbed). We chose this method for indicating an active pinch, because the Leap Motion Controller can't distinguish between the thumb and the index finger (most natural fingers of a pinch gesture) if they are to close. If the `Pinch` gesture is active, it manipulates the nearest object. For that, the gesture requests the nearest object or vertex (dependent of the current mode; see section 3.3.6) in a certain range (range modifiable in the `settings` module) from the `ObjectHandler`. The `ObjectHandler` iterates through the vertexes of each manipulation object to find the closest one to the `Pinch` position and returns either the vertex or the object of the vertex. It is distinguished between manipulation objects and non-manipulation objects so that there are objects which can not be manipulated by a gesture and can be used for creating the appearance of a task. This differentiation is done by a keyword in the name of the manipulation objects (key is editable in the settings). Afterwards, the `Pinch` displays the grab as described in section 3.3.1. It still is updated to detect if the two `Fingers` are separated enough to abort the gesture and to move the grabbed object.

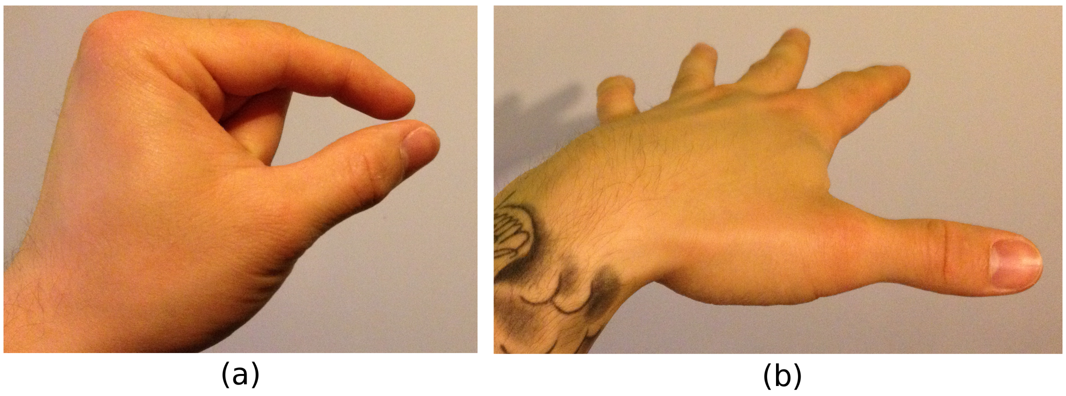


Figure 3.6: *Gestures: (a) Pinch, (b) Flathand.*

Hand position change used for location update of object.

The movement of the object is calculated by measuring the difference between two successive hand positions:

$$\overrightarrow{handsDiff_C} = \overrightarrow{oldLoc_C} - \overrightarrow{curLoc_C}$$

Using the hand for the moving step of an object is much more stable than using the fingertip position, because the Leap Motion Controller can detect the hand better and more accurate than the fingers. Then, the distance is multiplied with a smoothness factor to scale down the movements of the user, transformed to the Blender coordinate system, and applied to the object:

$$\overrightarrow{objectLoc_B} = \overrightarrow{objectLoc_B} + \overrightarrow{handsDiff_B}$$

Drop range to hide inaccuracy of detection.

If the object is moved to far away (`OBJECTDROPLINE`) from the camera, the pinch is aborted:

$$\overrightarrow{objCamDiff_B} = \overrightarrow{cameraLoc_B} - \overrightarrow{handLoc_B}$$

This drop range is used, because the accuracy of the Leap Motion Controller decreases the farer away the fingers are from the origin of the controller. To hide that fact from the user and avoid lack of assurance, the object is dropped if it is out of a certain range (range editable in the `settings` module). In order to abort a `Pinch` gesture, it is set released, the object is deselected, and the gesture is deleted by the `GestureHandler` in the next update step.

3.3.5 Flathand

Flathand gesture for camera manipulation provides enough degrees of freedom.

For the camera manipulation a flathand gesture suits the best, because it provides translation into each direction and rotation around each axis what is necessary to manipulate the 6 degrees of freedom of a camera. Furthermore, the movements can be controlled more accurate than, for example, with a fist, because the surface is bigger.

Activated by connecting and releasing two fingers while hand is horizontally.

This gesture is enabled by holding the hand horizontally and then bringing any two fingers together and separate them again. It can be aborted by showing less than five fingers, so by bringing two fingers together. Here again it is distinguished between the creation of a `Flathand` object and an active gesture. The `GestureHandler` just detects a possible `Flathand` gesture, the rest is handled by the gesture object to hide the functionality independently in the gesture itself. If the `GestureHandler` detects a `Hand` with five `Fingers`, it creates a `Flathand` object which then is updated in each update step and checks the starting and the aborting of the gesture. The gesture counts the visible `Fingers` of the `Hand`. If there is an update step with four detected `Fingers`, followed by an update step with 5 detected `Fingers`, the gesture additionally checks the torque of the `Hand`. For that, the angle between the `Hand` normal[5] and the z-axis is calculated by using the Python API. If the torque of the `Hand` is less than the `FLATHANDANGLE` parameter, the gesture is started. In the `settings` module the offset of holding the hand horizontally to start the gesture can be adapted. The gesture is aborted if there are less than four `Fingers` while the gesture is not yet active and while the gesture is active if less than five fingers are visible.

Details of flathand recognition.

Hand position change used for location update of the camera.

If the gesture is active, the camera can be rotated by rotating the hand around the z-axis and tilted downwards and upwards by tilting the hand downwards and upwards. Hand translation is used for zooming (translation along the view direction; y-axis) and translation of the camera along the x-axis and z-axis (relative to the user). This is done, because the hand only provides 3 degrees of freedom for transla-

[5]Vertical vector to the hand

tion. Zooming is more important than a translation along the y-axis, because the center of manipulation is at the origin most of the time and zooming is used more often. It can be circumvented by rotating the camera so that the translation direction is possible by a x-axis translation. This four possibilities are the same provided by each 3D editing tool, but adapted to the hand. Adding more possible manipulations, like tilting to the right and left, would make the camera manipulation more difficult, because more degrees of freedom are manipulated at the same time. Also, this kinds of manipulation are not essential, because no professional 3D editor implements them. Camera translation and the zooming are detected by using the difference of two positions of the `Hand` between two update steps as relative change. Here also the position change of the hand is used, because the detection of the hand is much more stable than the detection of the fingers:

$$\overrightarrow{handsDiff}_C = \overrightarrow{oldLoc}_C - \overrightarrow{curLoc}_C = \begin{pmatrix} dx \\ dy \\ dz \end{pmatrix}$$

The difference is multiplied with a smoothness factor to smooth the movements of the user. Since the zooming value of the camera is always along the view direction, it is independent from the current camera angle in the Blender coordinate system. The y-component of the difference in the camera coordinate system can be used directly:

$$cameraZoom_B = cameraZoom_B - y$$

For the camera translation only the x-axis and the z-axis are taken into account:

$$\overrightarrow{changeLoc}_C = \begin{pmatrix} dx \\ 0 \\ dz \end{pmatrix}$$

Because the x-axis and the z-axis of the difference are relative to the camera, it is necessary to use the `Transformator` to convert the changes to the Blender coordinate system (see section 3.3.2). Then the changes can be applied to the camera location:

$$\overrightarrow{cameraLoc}_B = \overrightarrow{cameraLoc}_B + \overrightarrow{changeLoc}_B$$

Hand direction change used for rotation update around the z-axis.

For the rotation around the z-axis the direction *dir* of the hand is used[6]. The current direction and the direction of the last update step are projected onto the xy-plane:

$$\overrightarrow{planeProjection_C} = \overrightarrow{dir_C} - \left(\frac{\overrightarrow{dir_C} \cdot \overrightarrow{planeNormal_C}}{\sqrt{\overrightarrow{dir_C} \cdot \overrightarrow{dir_C}}}\right) * \overrightarrow{planeNormal_C}$$

Then, the Python API is used to calculate the angles between the y-axis and the projections. The relative rotation change then is the difference between the last `Hand` rotation and the current `Hand` rotation:

$$changeRot_C = oldRot_C - curRot_C$$

Here again the value is multiplied with a smoothness factor to smooth the hand motions. Because the rotation difference is relative to the camera and not to the current camera rotation in Blender, the `Transformator` is used to transform the rotation into a rotation matrix of the Blender coordinate system (see section 3.3.2). Finally, the rotation change is applied to the camera by using the Python API to rotate the camera rotation matrix through the calculated rotation matrix.

Hand normal change used for rotation update around the x-axis.

For the rotation around x-axis the same math is used. Here just the hand normals of two successive update steps are used and the projection onto the yz-plane. Then again the rotation of the last and current `Hand` normal compared to the y-axis are calculated. Finally, the angle difference between the last rotation and the current rotation of the `Hand` is smoothed by a smoothness factor, transformed to the Blender coordinate system, and applied to the camera rotation matrix.

Filter remove jitter of the hand.

The explained smoothness factors can be adjusted in the `settings` file. Bigger factors result in a more direct manipulation, whereas smaller values are used for a more smooth manipulation. Besides these factors, the by Casiez et al. [2012] presented OneEuroFilter is used to remove jitter of the hand. This filter takes a value and returns a filtered value with eliminated fluctuations. There are filters for the x, y, and z axis of the `Hand` normal, `Hand` direction, and

[6]In which direction the fingers are pointing on average

Hand position. The filters are applied to the vectors of the Hand before they are used for the above calculations. To find the parameters of the OneEuroFilter, the standard parameters are used as a starting point. Then, the `mincutoff` parameter is adjusted to remove jitter while moving the hand at a very low speed. Afterwards, the `beta` parameter is increased to minimize lag while moving the hand quickly. It showed that a value of 1.0 for the `beta` parameter, a value of 300.0 for the `mincutoff` parameter of the filter for the `Hand` normal, and 100.0 for the `mincutoff` parameter of the translation filter and the filter for the `Hand` direction works. Also the settings for the OneEuroFilter can be edited in the `settings` file in order to change the directness of the gestures.

3.3.6 Modes

Object mode for translating objects and *edit mode* for changing their form.

There are two possible modes in this testbed, which are analogue to modes of Blender: the *object mode* and the *edit mode* which enable the user to translate objects and change the form of an object. In the *edit mode* objects can be selected and moved with the `Pinch` gesture, as described in section 3.3.3. To edit an object, it has to be in *edit mode*. For that, first an object has to be selected and then the "d" key pressed. The mode of Blender is changed to *edit mode* so that the wire frame of the selected object is shown. Furthermore, all other objects which were not selected are hidden. Now, the `Pinch` gesture manipulates vertexes and no longer object origins. The difference to the *object mode* is that the `ObjectHandler` returns the nearest vertex and not the object of the nearest vertex. In this mode the form of an object can be changed. Because more accurate movements are needed to edit an object, the fingertip spheres (see section 3.3.1) are nearer to the camera in the *edit mode* than in the *object mode*. To get back to the *object mode*, the "d" key has to be pressed again. Then, all objects are shown and the edited object is deselected. This two modes allow the functionality of moving and editing objects so that studies with coarse objects can be done as well as studies with precise manipulations with vertices.

"d" key switches mode of selected object.

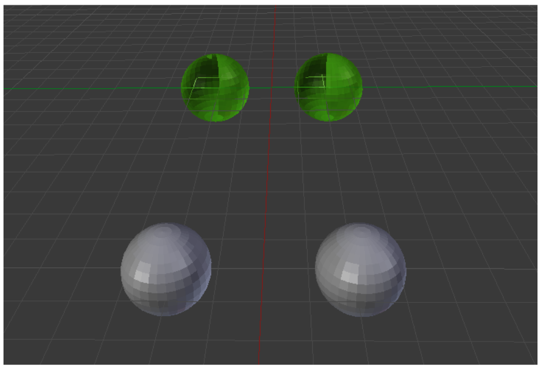


Figure 3.7: Example for a customized scene to design the appearance of a task.

3.4 Extensibility

The advantages of having a testbed which provides a basic structure to remove the time expensive step of implementing everything from scratch were discussed in section 1.3. This section shows the interface to customize the testbed and how easy it is to create a specific task for a study.

Design the appearance of a task in the scene of Blender.

First a scene has to be created to design the appearance of the task. Figure 3.7 shows a scene with two cursors and two targets which was created for our study. The functionality of Blender is used to add objects and to design and position them according to the needed structure for the task. Additionally, the camera has to be set and fixed if a specific camera position is needed.

Modularized structure can be easily extended to add more base functionality.

Sometimes the base functionality has to be extended. In order to provide an interface for quick task development, interchangeability and extensibility are very important. For that, the different modules with the base functionality can be extended. Additional task specific functionality can be added easily by implementing new submodules and calling them from the central responsible module. For example, for adding a new gesture only a new one similar to the structure of the existing ones, but with new logic has to be created. Then, it has to be called from the initiating part, the `GestureHandler`, like the other ones. This simple process is possible, because each module has its own logic, is managed by a central station, and is independent from the others. For simply changing the input device the receiver thread has to be adapted in a way that it always provides the newest data from the new device. Also, the `LeapSelector` has to be replaced by a module which returns the correct parts of the new data structure.

Two interfaces are provided to add logic and logging.

Besides extending the base functionality further customization has to be done to add task related logic to the scene and to log specific information for evaluation of the study. This process is simplified by a base structure which encapsulates the structure of the program from the task related code. There are two interfaces provided which are integrated into the existing program and are called automatically. Therefore, one can concentrate more on what to do in a task and not on implementing the structure. The two interfaces are explained in the following.

3.4.1 StudyLogger

Customize the `StudyLogger` for task specific logging.

The `StudyLogger` module provides the functionality to gather study related information to evaluate the study afterwards. During the sequential processing of the main operator the `update` method of the `StudyLogger` is called and thus every time something changed. This function handles the start and stop of the logging and the logging itself. It can be customized to log information which are not captured by the `RawLogger` integrated in the Blender source code. The Python API of Blender is used to get states of the scene, like the position of an object. Then, the logger of the `StudyLogger` module, which creates a file on the desktop, can be used to log the needed states in each update step. To manage the stopping of the module, return True to stop or False to proceed the updating of the `StudyLogger`. Because the state of the `Simulator` (see section 3.4.2) is passed, one can stop if the simulation stopped (can be enabled in the settings).

The following code snippet is an example of how the `StudyLogger` module was used in our study:

```
def update(self,runState, simulationFinished):
  cursor = bpy.context.scene.objects["Cursor"]
  target = bpy.context.scene.objects["Target"]
  diff = cursor.location - target.location
  self.logger.info("Distance: {0}".format(diff.length))
```

We customized the `update` function of the `StudyLogger` to log the euclidean distance between a cursor and a target. This enabled us to evaluate the users performance when tracking a target with a cursor.

Available structure allows to focus on the task.

The structure of a frequently called module and the provided logging functionality enables to start directly with the implementation of the task specific information without the need of setting up a logger and a structure for continuous calling.

3.4.2 Simulator

Customize the `Simulator` to add task specific logic to the scene.

This module is to add logic to the task. It is similar to the `StudyLogger` - there is an `updateAlways` and an `update` function which handle the start and stop and can be customized to do task related logic. Return True to stop the simulation and False to proceed. The `updateAlways` function is called every time the main operator is called and the `update` function every time the main operator is called and the simulation is set active by the GUI. Because of this, one can distinguish between always active logic and logic which can be started. These two functions offer the possibility of manipulating the appearance of a scene easily and changing attributes of objects by using the Python API of Blender.

Distinguish between startable and always active logic.

For example, the `update` function can be used for moving an object a step further in every update step to animate the object. The `updateAlways` function can be used to draw general things of the scene, like a connection between two moving spheres. In the following a code snippet is shown, which is an example of how we used the `updateAlways` function to animate the scene:

```
def updateAlways(self):
  cursor = bpy.context.scene.objects["Cursor"]
  target = bpy.context.scene.objects["Target"]
  material = bpy.data.materials["Matching"]
  diff = cursor.location - target.location

  if diff.length <= ERRORTHRESHOLD: # color object
    if len(cursor.data.materials) == 0:
      cursor.data.materials.append(material)
    else:
      pass # already colored
  else: # delete color
    if len(cursor.data.materials) != 0:
      cursor.data.materials.pop(0,True)
    else:
      pass # already deleted color
```

There is a target and a cursor which is manipulated by the user. By this code we achieve that the cursor is colored if it is close enough to the target.

The animation rate can be assumed as the same than the update rate of the `LeapModal` operator (50 Hz, editable in the settings), because the Simulator is called during the sequential processing of the `LeapModal` operator and the Blender internal update rate for displaying is faster than the update rate of the testbed. As maximum animation rate 222 Hz were measured. This was achieved by setting the update rate parameter of the testbed to an infinite small value and measuring the average time between two animation steps. Thus, smooth animations are possible.

Animation rate equals `LeapModal` update rate (max 222 Hz).

Also in this module a supporting structure is given so that one can concentrate on the logic for a task and does not have to spend time on implementing a base structure.

Provided base structure saves time.

3.4.3 Settings

As described in section 3.2, the `settings` module is integrated into every module and manages all parameters. This

Find all adjustable parameters in the `settings` module.

is useful, because there is a central place where the program can be adjusted and it is not necessary to search through each file. The standard parameters should work good, but if some problems occur or special settings are needed, the parameters can be changed there. For example, if a more direct manipulation is necessary, the relevant smoothness factor can be adjusted. Each parameter is documented in this `settings` file.

In this chapter the structure and base functionalities of the testbed were described. Also, the way how to design a specific task with this testbed and how easy it is compared to implementing everything from scratch was shown. To prove the functionality of the testbed, it is used to conduct a case study. In the following chapter this study is presented.

Chapter 4

Case Study: Replicating Symmetric Bimanual Interaction Study

The previous chapter presented a testbed for 3D editing with bare-hand input devices. To test the testbed, we replicated the study of Balakrishnan et al. [2000]. As mentioned in section 2.3, he showed useful facts for bimanual interaction with digitizing tablets. We added midair bare-hand input as a condition to the original study. This chapter describes the experimental setup and the next chapter describes the results.

Study was conducted to test the testbed.

4.1 Experiment

4.1.1 Tasks

The task was a bimanual object tracking task, adapted from Balakrishnan et al. [2000], in which the user had to track moving targets on the screen with both hands (see figure 4.1). This kind of task was chosen in order to provide a constant difficulty over the whole task and to investigate factors of bimanual performance in general. There were four

Symmetric bimanual object tracking task was used to investigate influencing factors.

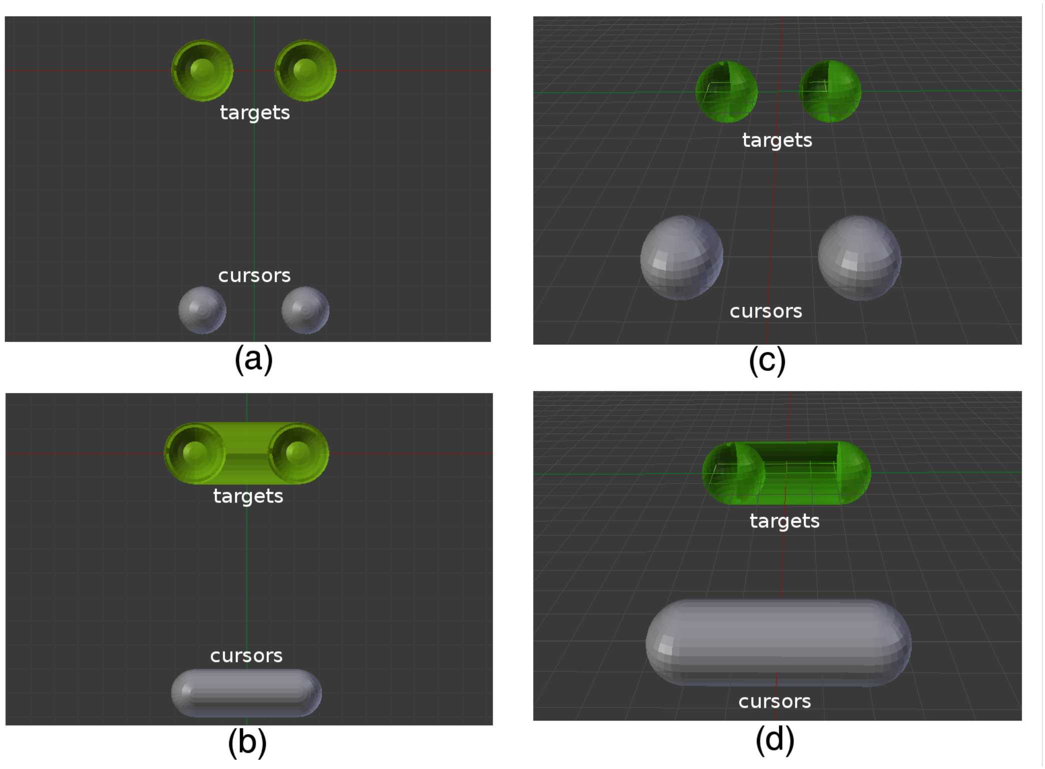


Figure 4.1: *Tracking task overview with green targets and grey cursors: (a) separated 2D, (b) integrated 2D, (c) separated 3D, and (d) integrated 3D.*

independent variables: *attention*, *visual integration*, *difficulty*, and *setup*.

Factor one: *visual integration*.

To change *visual integration*, there was a *separated* and an *integrated* variant. In the *separated* variant (see Figure 4.1 (a) and (c)) the user manipulated two *cursor spheres*, one by each hand, to follow two *target spheres*. The *cursor spheres* were of 1.8 cm diameter and in grey, the *target spheres* were of 2.4 cm diameter and in green. The participant controlled the left *cursor sphere* with his left index fingertip and the right *cursor sphere* with his right index fingertip. For each task the path of the *target spheres* were different, but predetermined (see section 4.5). The movements were symmetric - both targets moved the same distance in a given time. Over the task the distance between the targets and the speed of the targets was kept constant. In the *integrated* variant (see Figure 4.1 (b) and (d)) the *target spheres* and the *cursor spheres* were connected by a solid cylinder. Participants had to match the *cursor rectangle* exactly with the *target rectangle*. The *target rectangle* moved around in the same way the *target spheres* did in the corresponding *separated* condition. Also here the speed and length of the rectangle was kept constant. Both, the *separated* and the *integrated* condition, are the same in a motor action point of view, but they differ in a visual point of view. The *integrated* variant is perceived as a single target, whereas the *separated* variant is perceived as two separated targets.

Factor two: *attention*.

The *attention* was varied by the distance between the moving targets. For the *singular attention* variant a distance of 4cm was used so that the targets were visible in the focus field of the user and he had to attend only to a single area. For the *divided attention* variant 21 cm were used so that the user had to divide attention and switch with his focus between the two targets.

Factor three: *difficulty*.

The *difficulty* of the task was manipulated by using different speeds of the moving targets. For the *slow* variant a speed of 2 cm/s was used and for the *fast* variant a speed of 4 cm/s .

4.1.2 Setups

Factor four: *setup*.

There were three setups: touchscreen with 2D tasks (*T2D*), Leap Motion with 2D tasks (*L2D*), and Leap Motion with 3D tasks (*L3D*). These were chosen to see whether the effects explored in the study of Balakrishnan et al. [2000] hold also for bare-hand input devices. The *T2D* condition represented the digitizing tablet stylus condition in the original study. The effect of midair manipulation was investigated by comparing the *T2D* and *L2D* condition. *L2D* and *L3D* were compared to illustrate the effect of adding the third dimension. In the touchscreen setup the user sat in front of a horizontal touchscreen (see figure 4.2(a)), in the Leap Motion Controller setup he sat in front of a monitor with the Leap Motion Controller under his hands which was 24 cm away from the screen (see Figure 4.2(b)). The controller was placed in a height such that when the user manipulated a sphere at the top most position of the screen, the

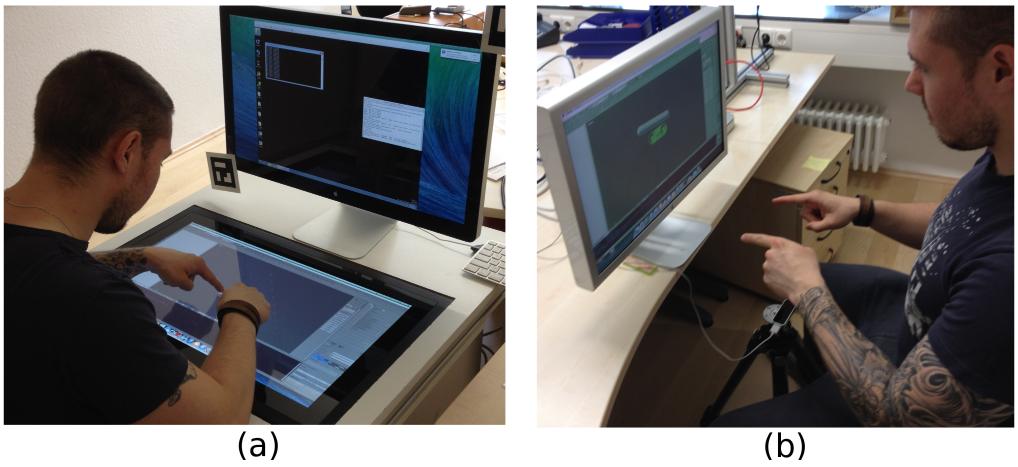


Figure 4.2: *Setup overview: (a) touchscreen, (b) Leap Motion Controller.*

angle between his torso and his arm was 80 degrees. In the 2D variants the user was presented a scene which was orthogonally projected onto the ground plane (see Figure 4.1(a) and (b)), in the 3D variant he was presented a 3D scene with the camera looking from the front downwards to the origin (see Figure 4.1(c) and (d)).

4.2 Metrics

In this study three different measures to evaluate the behavior of the participant were used.

Average root mean square error was used as measure for the performance.

One important measure to compare the accuracy is the *overall tracking performance*. The used metric was the average root mean square error, since it is independent from the dimension and enables to compare the different setups. For this, in each update step the root mean square error (RMS) between each cursor sphere and the corresponding target was calculated:

$$\overrightarrow{diff}_{l/r} = \overrightarrow{target}_{l/r} - \overrightarrow{cursor}_{l/r}$$

$$RMS_{lh/rh} = \sqrt{\frac{1}{dim\left(\overrightarrow{diff}_{l/r}\right)}\overrightarrow{diff}_{l/r} * \overrightarrow{diff}_{l/r}}$$

This lead to an error for the left and right hand (RMS_{lh} and RMS_{rh}). The total RMS error for a task was calculated by adding up the error for the left and right hand:

$$RMS_{tot} = RMS_{lh} + RMS_{rh}$$

Symmetry compared the average root mean square error of both hands.

In an symmetrically assigned task[1] the performance can be distinguished between *symmetric* and *parallel* performance. *Symmetric* means that the performance of the hands is equal in terms of error rate. So the average RMS errors of the left and right hand was compared. If statistically significant, the performance was not symmetric.

Level of parallelism was measured by using the m-metric for tracking tasks.

Parallelism describes the equality of the movement of the two hands, so if the two hands perform the same movements at the same time. To measure parallelism, Masliah et al. [1999] introduced the m-metric for docking tasks. Here, the adapted m-metric of Balakrishnan et al. [2000] was used to measure parallelism in a tracking task. This metric takes magnitude and direction into consideration and not only if movements of the two hands are performed just at the same time. First, in each update step the error reduction between each cursor and the target position was calculated (see figure 4.3) :

$$\%ER = \frac{\text{magnitude of movement towards target}}{\text{movement required to reduce error to 0}}$$

This value is 1 if the cursor matches perfectly the target and 0 if the cursor isn't following the target. In the case that the denominator is zero $\%ER$ is 0.

The parallelism was calculated in each update step by comparing the error reduction of each hand:

$$Parallelism = \frac{min\{\%ER_{lh}, \%ER_{rh}\}}{max\{\%ER_{lh}, \%ER_{rh}\}}$$

[1]Identical assignment of roles to the hands

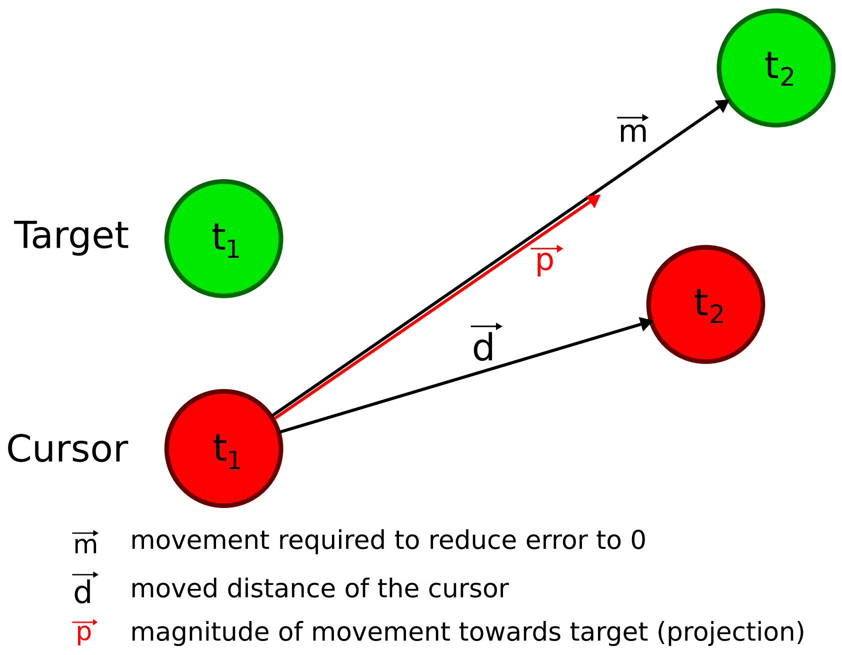

Target
t_1
t_2
$\vec{m}$
$\vec{p}$
$\vec{d}$
t_2
Cursor
t_1
$\vec{m}$ movement required to reduce error to 0
$\vec{d}$ moved distance of the cursor
$\vec{p}$ magnitude of movement towards target (projection)

Figure 4.3: *M-metric for measuring parallelism.*

The average parallelism of a task was used to measure the level of parallelism. It results in 1 if the hands are reducing their errors identically and in 0 if the hands are moving sequentially. In the case that the denominator is zero the parallelism is 1.

4.3 Apparatus

Detailed values for the testbed and tasks.

The background was always black, a grid on the xy-plane was shown, and the program was updated every 0.02 seconds for the Leap Motion Controller setups. For the touchscreen setup the rate had to be adapted to a faster rate of one update per 0.008 seconds, because of delays of the cursor following the finger position. There was no delay in the Leap Motion setup so the update rate was kept. In order to see if the cursor is inside the corresponding target, it had an opacity of 0.7. The camera view was fixed and 20.46 cm away from the origin, the eyes were 40cm away from the screen.

We had two different study setups: one for the T2D setup and another for the L2D and L3D setup.

4.3.1 Touchscreen Setup

This experiment was conducted on a graphics accelerated 2 x 2.26 GHz Quad-Core Intel Xeon Mac Pro with OS X. The screen was a 27-inch Perceptive Pixel LCD capacitive display with a resolution of 2560x1440, a touch input sample rate of 120 Hz, and an display update rate of 60 Hz. The fingers were used for the input.

4.3.2 Leap Motion Controller Setup

In this setup a graphics accelerated 2.4 GHz Intel Core 2 Duo Mac mini with OS X was used. The screen was a 23-inch Apple Cinema Display with 1920x1200 resolution and an update rate of 60 Hz.

4.4 Participants

Eight right-handed participants at the mean age of 23.88 years participated voluntarily in the study, six male and two female.

4.5 Design

Mixed design was chosen to reduce the time for each participant.

For the *attention*, *setup*, and *difficulty* factor a within-group design was used. To reduce the duration of the experiment for each participant and avoid fatigue, for the *visual integration* factor a between-group design was chosen. The presentation order of the setups for Leap2D and Leap3D were counterbalanced across the participants of a group, because between these two a learning effect could have been occurred, since the Leap Motion Controller was used for two conditions. We expected no learning effect between the touchscreen and the Leap Motion Controller.

Setups were counterbalanced.

Each participant performed the tasks with either the *integrated* or the *separated* condition and the *setups* in a for the group counterbalanced order. To measure a stable performance after an initial learning curve, for each *setup* condition the participant performed practice trials until the standard deviation of average root mean square of three successive trials was less than 0.3 (this threshold was determined by a pilot study). At most six practice trials per setup were performed to avoid fatigue. Each participant stabilized before this limit and performed on average 3.17 practice trails before the actual study. Afterwards, four testing trails were performed, one for each of the combination of *speed* and *attention*. The order was fixed for each *setup* and participant: *slow* and *singular*, *slow* and *divided*, *fast* and *singular*, *fast* and *divided*. Each trial lasted 45 seconds on average. If a task failed during the execution because of detection problems of the Leap Motion Controller (disappearing cursors), the task was repeated at the end of the trials for this *setup* to avoid an adulterated RMS error. Between each trial there was a break of about 20 seconds and after 15 minutes a break of about 3 minutes.

Initial learning curve was excluded.

Fixed order of tasks was used.

Failed tasks were repeated at the end.

For the experiment a total of 96 non-practice trials were performed:
8 participants x
1 visual integration condition (*Integrated*, *Separated*) x
2 attention conditions (*Singular*, *Divided*) x
2 speed conditions (*Slow*, *Fast*) x
3 setup conditions (*T2D*, *L2D*, *L3D*)
= 96 trials of 45 seconds each

The experiment for each participant lasted 40 minutes on average.

Experimenter started the tasks, they finished automatically.

Participants started a trial by matching the cursors to the targets, the targets changed color to signal best fit. The experimenter then started the test by counting from 3 backwards and pressing a start button. Then, the targets began to move for 45 seconds. Afterwards, they stopped moving and the test was finished.

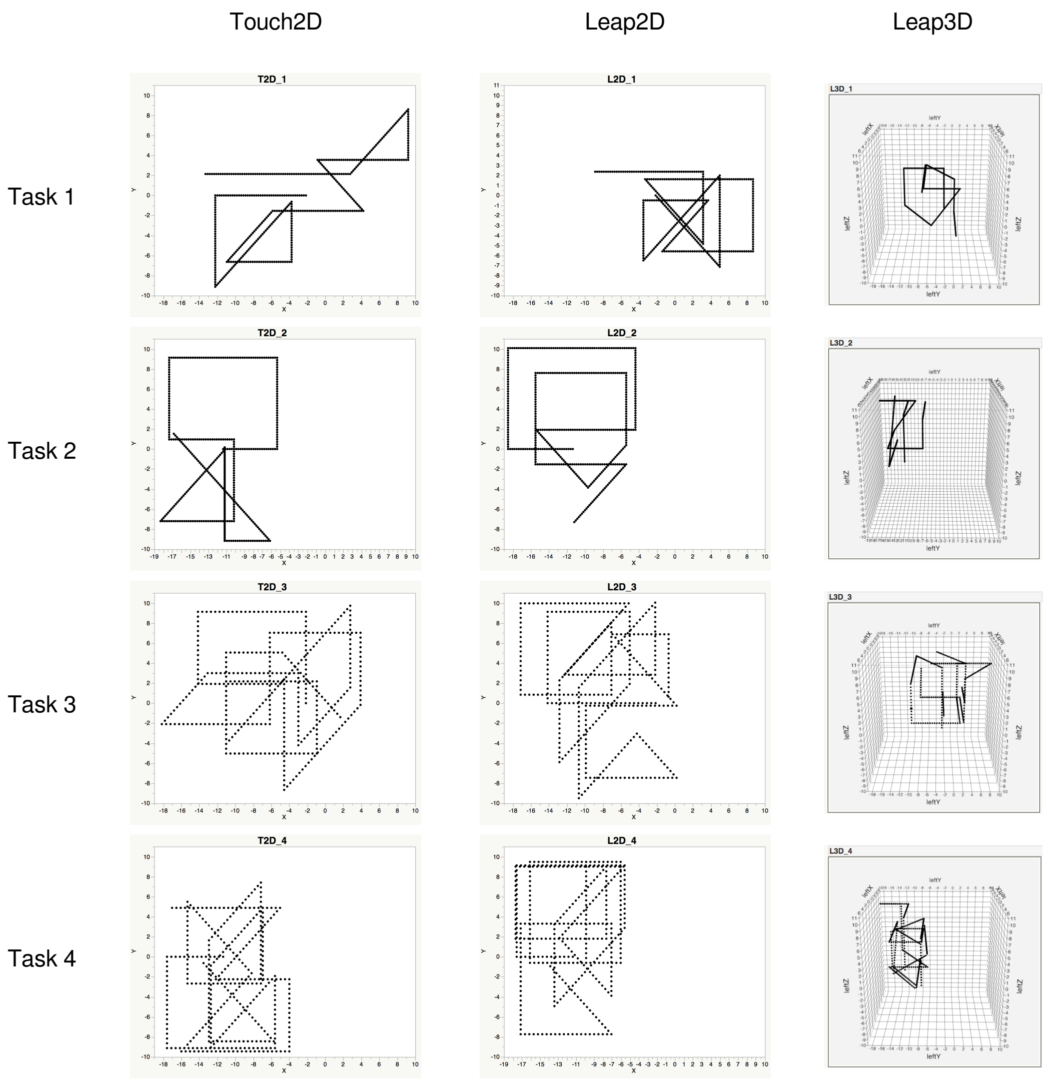


Figure 4.4: *Overview of the paths used for each setup and task.*

The paths were different for each of the four tasks and each setup, but predetermined. Because of that, they seemed to be random for each participant, but the conditions stayed the same. In figure 4.4 the used paths can be seen. 12 direction changes were used for the *slow* and 24 direction changes for the *fast* variant.

Predetermined paths were used.

4.6 Hypotheses

In the following the expected effects to occur are listed.

4.6.1 Each Setup

Hypotheses: *Accuracy*

H1:
The *integrated* visual stimuli condition will result in more accurate tracking than the *separated* condition.

H2:
The *singular attention* condition will result in more accurate tracking than the *divided attention* condition.

H3:
The *slow* speed condition will result in more accurate tracking than the *fast* speed condition.

Hypotheses: *Symmetry*

H4:
The *integrated* visual stimuli condition will be performed more symmetrically than the *separated* condition.

H5:
The *singular attention* condition will be performed more symmetrically than the *divided attention* condition.

H6:
The *slow* speed condition will be performed more symmetrically than the *fast* speed condition.

H7:
The *integrated* visual stimuli condition will be performed with greater parallelism than the *separated* condition.

H8:
The *singular attention* condition will be performed with greater parallelism than the *divided attention* condition.

Hypotheses: *Parallelism*

H9:
The *slow* speed condition will be performed with greater parallelism than the *fast* speed condition.

4.6.2 Overall

H10:
The *T2D* setup will result in the same accurate tracking than the *L2D* setup. The *L3D* setup will be performed less accurate.

Hypothesis: *Accuracy*

H11:
The *T2D* setup will be performed equally to the *L2D* setup, according to the symmetry. The *L3D* setup will be performed less symmetrically.

Hypothesis: *Symmetry*

H12:
The *T2D* setup will be performed with greater parallelism than the *L2D* setup, which will be performed with greater parallelism than the *L3D* setup.

Hypothesis: *Parallelism*

H13:
The hypotheses H1-H9 are the same for each setup.

Hypothesis: *Generality*

In this chapter a detailed description of the study and expected effects were given. The next step is to evaluate the gathered data by the study and check if the hypotheses are true.

Chapter 5

Results and Discussion

This chapter deals with the evaluation of the conducted user study described in the previous chapter. Only statistical results are given, whereas the next chapter contains interpretations. In order to analyze the hypotheses (see section 4.6), it is highlighted which conditions influenced the investigated measures and graphs are given. In the end limitations of the study are discussed.

5.1 Results

5.1.1 Performance

For the overall tracking performance the average total root mean square error ($AvgRMS_{tot}$) is computed (see section 4.2). The $AvgRMS_{tot}$ for all conditions can be seen in figure 5.1.

Integration had no effect on the performance, but *attention* and *speed* had.

Users made similar errors regardless of the *integration* (F(1,6)=0.1221, p=0.7387). Thus, H1 is not confirmed. As expected, users made more error when the *attention* is divided (M=1.5266 cm) than in the *singular attention* condition (M=1.2970 cm)(F(1,77)=27.3684, $p<0.0001$). So, hypothesis H2 is confirmed. H3 also is confirmed, because users performed better in the *slow* condition (M=1.0599 cm)

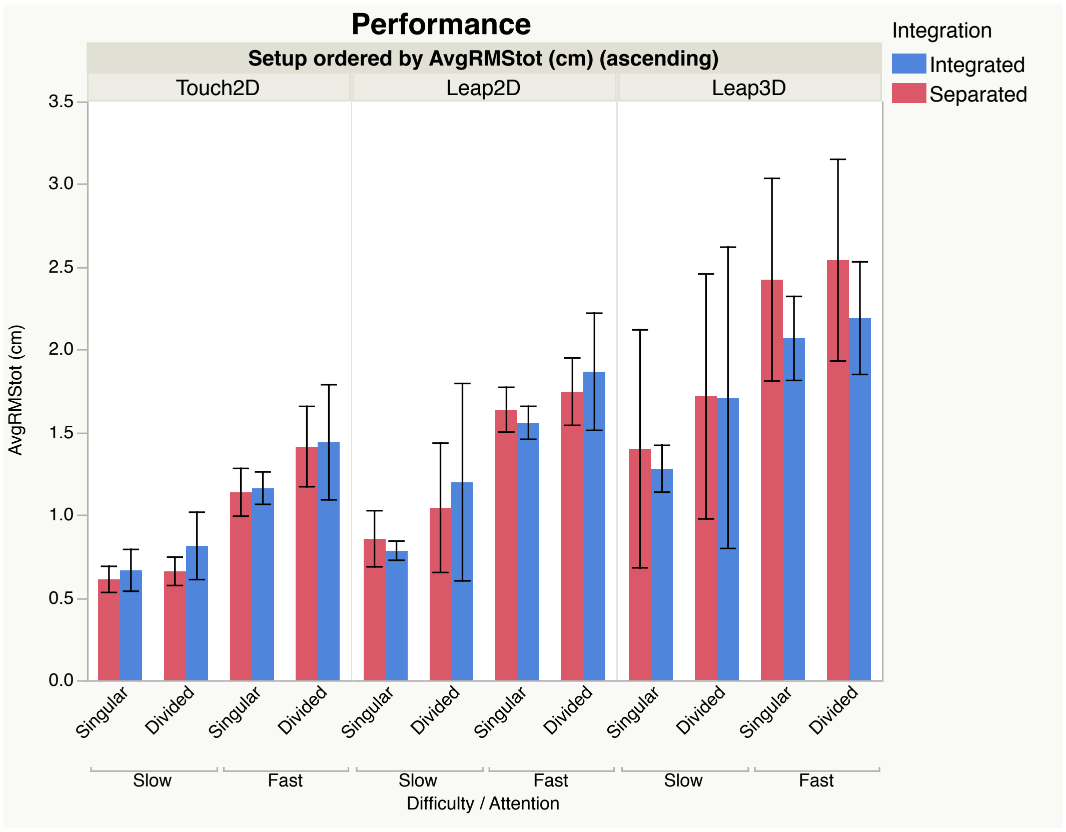


Figure 5.1: *Overall tracking performance for all conditions, measured by AvgRMS$_{tot}$. Constructed using mean and 95% confidence interval.*[a]

[a]Used for every graph from now on

Performance differed among the setups.

than in the *fast* condition (M=1.7637 cm)(F(1,77)=256.9912, p<0.0001). The only other significant effect is found for the *setup* factor (F(2,77)=152.0937, p<0.0001), showing that performance varied between the three setups. The least errors were made in the *T2D* setup (M=0.9863 cm), followed by the *L2D* setup (M=1.3347 cm). In the *L3D* setup the most errors were made (M=1.9145 cm). Thus, H10 is also not confirmed. One conspicuous difference between the setups is that performance differed much more among the users in the *L3D* setup (SD=0.5497 cm) than in the *L2D* (SD=0.4319 cm) and the *T2D* setup (SD=0.3421 cm)(Levene(2,93)=4.4963, p=0.0137). There are no further significant interaction effects.

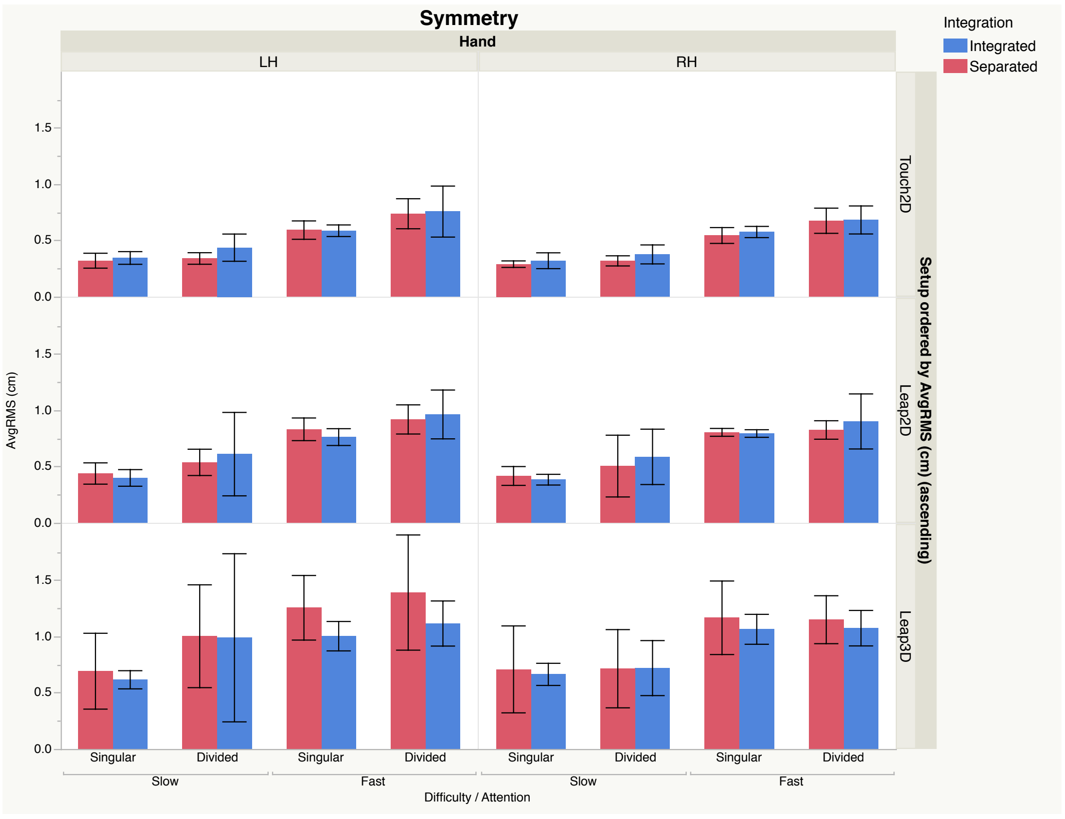


Figure 5.2: *Average tracking performance of the left and right hand, measured by AvgRMS, for all experimental conditions.*

5.1.2 Symmetry

The results for each hand, broken down by the different conditions, can be seen in figure 5.2. Overall, we find a asymmetry between the two hands (F(1,172)=9.9752, p=0.0019). Users made more errors with the left hand (M=0.7347 cm) than with the right hand (M=0.6771 cm) (see figure 5.3).

Asymmetry is found between both hands.

Further analysis is done with the difference between the performance of the right and left hand ($AvgRMS_{lh} - AvgRMS_{rh}$) as dependent variable. H4 is not confirmed, because the tasks were performed with a similar symmetry regardless of the *integration* (F(1,6)=0.0928, p=0.7709) (see figure 5.4(a)). There is a significant difference (F(1,77)=14.3371, p=0.0003) for the *attention* factor, what

Integration and *difficulty* had no effect on the symmetry, *attention* only in the *L3D* setup.

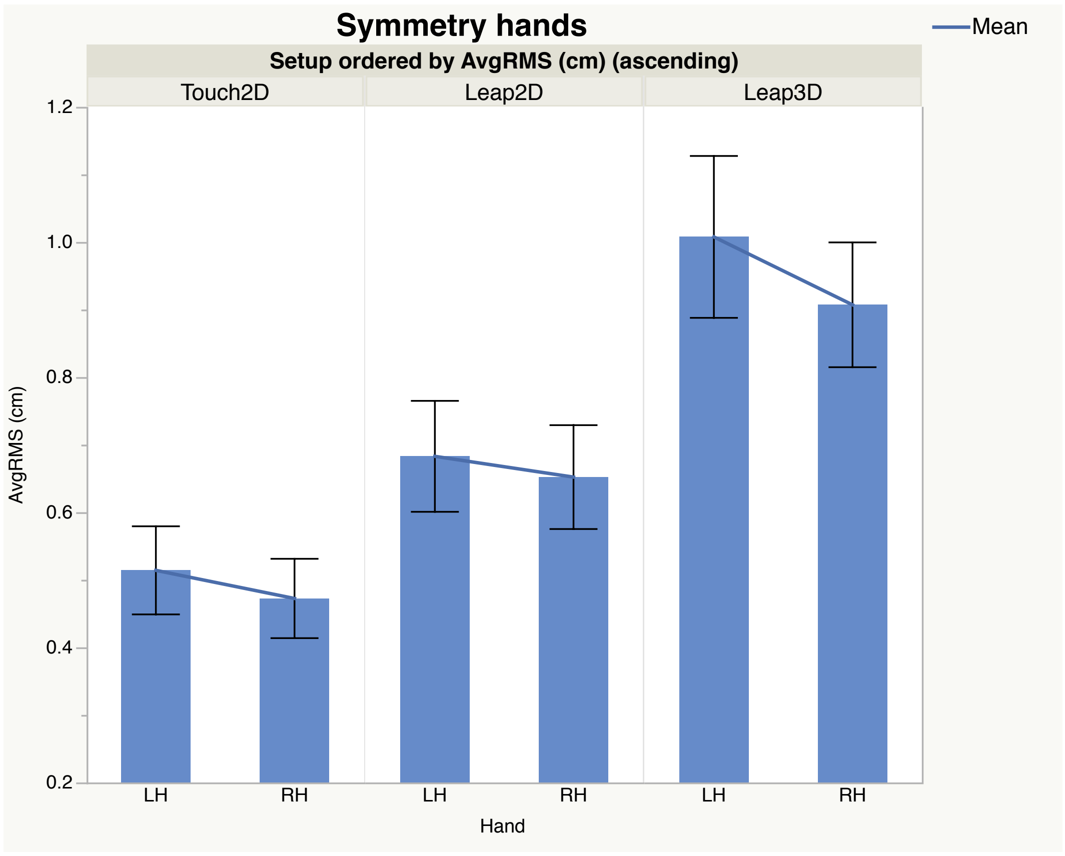


Figure 5.3: *Mean of the average tracking performance of the left and right hand, measured by AvgRMS, showing a slight asymmetry.*

together with a significant difference for *setup * attention* (F(2,77)=3.3188, p=0.0414) indicates that the *attention* factor had a different effect in each setup. Figure 5.4(b) shows that users performed the tasks with similar errors for both hands in the *singular attention* condition of the *L3D* setup, while they made more errors with the left hand (M=0.8163 cm) than with the right hand (M=0.7103 cm) in the *divided attention* condition. No significant different errors were made with both hands for the *attention* factor in the *L2D* and *T2D* setup. Thus, H5 is confirmed for the *L3D* setup, but not for the *T2D* and the *L2D* setup. Also, errors were similar for both hands in the two *difficulty* conditions (F(1,77)=0.0832, p=0.7738), so H6 is not confirmed (see figure 5.4(c)). Finally, the three setups were performed with a different symmetry (F(2,77)=7.6056, p=0.0010). Therefore, H11 is not con-

Symmetry differed among the setups.

firmed. The *T2D* setup was performed most symmetrically with a mean difference between the hands of 0.0432 cm, followed by the *L2D* setup with a mean difference of 0.0725 cm. The *L3D* setup was performed with a mean difference of 0.1444 cm between the hands least symmetrically. Additionally, the difference between both hands differed more among the users in the *L3D* setup (SD=0.1969 cm) than in the *L2D* (SD=0.0607 cm) and the *T2D* setup (SD=0.0363 cm)(Levene(2,93)=12.9670, p<0.0001). No more significant interaction effects are found.

5.1.3 Parallelism

To quantify the level of parallelism of the two hands, the m-metric (see section 4.2) is used. The results can be seen in figure 5.5.

Integration had no effect on the parallelism, but *attention* and *difficulty* had.

Since users performed the tasks with similar parallelism regardless of the *integration* (F(1,6)=0.4264, p=0.5380), H7 is not confirmed. The amount of parallelism was higher in the *singular attention* condition (M=0.4810) than in the *divided attention* condition (M=0.4236) (F(1,77)=91.3177, p<0.0001). Thus, H8 is confirmed. Also, with regard to the *difficulty* factor the tasks were performed with different parallelism (F(1,77)=71.4184, p<0.0001), but the average parallelism was higher in the *fast* condition (M=0.4776) than in the *slow* condition (M=0.4269). H9 is not confirmed. At last, users performed with a different parallelism in the three setups (F(2,77)=3.2568, p=0.0439), what confirms H12. Movements were performed most parallel in the *T2D* setup (M=0.4619), followed by the *L2D* setup (M=0.4517). The least parallelism was reached in the *L3D* setup (M=0.4432). There are no further interaction effects influencing the parallelism. Also the difference of parallelism among the users stayed the same in the three setups (Levene(2,93)=1.4915, p=0.2304).

Parallelism differed among the setups.

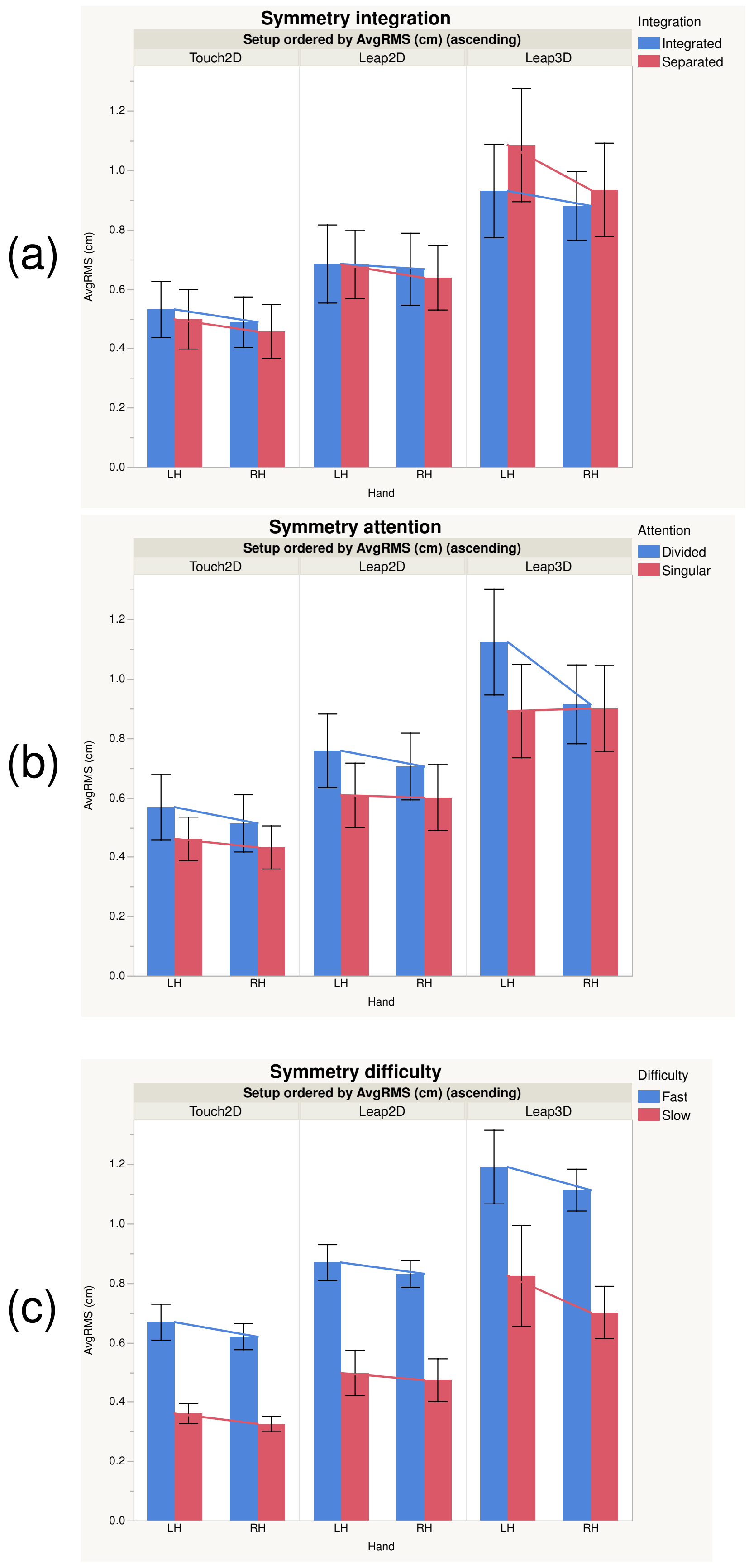


Figure 5.4: *Tracking performance of the left and right hand, measured by AvgRMS: (a) visual integration, (b) attention, and (c) difficulty.*

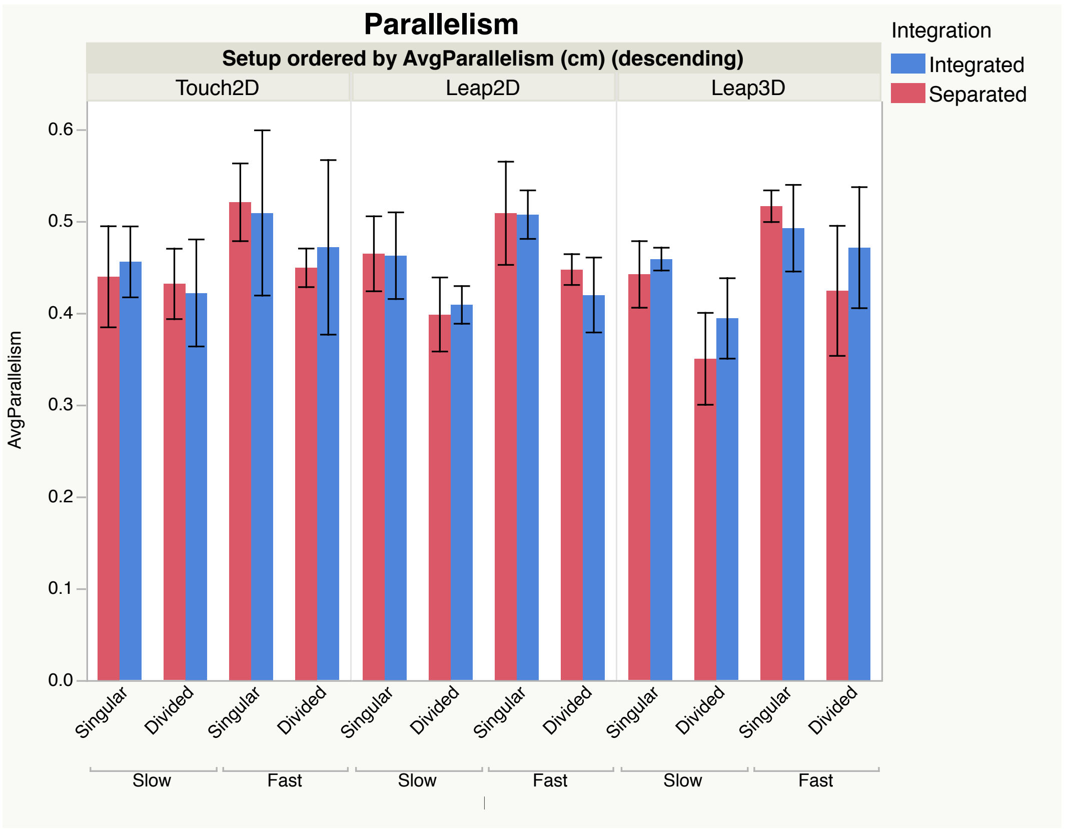


Figure 5.5: *Average parallelism, measured by the m-metric, for all experimental conditions.*

5.1.4 Comparison of Setups

In order to see whether the hypotheses hold for each setup, they are checked independently. Table 5.1 compares the three setups and shows that the results of the different hypotheses, except H5, are shared across the setups. Thus, H13 is not confirmed.

One hypothesis differs among the setups.

	H1	H2	H3	H4	H5	H6	H7	H8	H9
T2D	✗	✓	✓	✗	✗	✗	✗	✓	✗
L2D	✗	✓	✓	✗	✗	✗	✗	✓	✗
L3D	✗	✓	✓	✗	✓	✗	✗	✓	✗

Table 5.1: Comparison of the hypotheses among the three setups.

5.1.5 Comparison to Original Study

Only little differences to the original study.

Since we replicated the study of Balakrishnan et al. [2000], a comparison to the original study is essential. Table 5.2 compares the results of both studies and highlights differences. Most of the results are the same, but in opposition to the original study we found no significant effect of *visual integration* on the users motions at all. The original study showed a better symmetry and parallelism for the *integrated* condition. Also, we found better parallelism for the *fast* condition and not for the *slow* condition like Balakrishnan et al. [2000] did.

	H1	H2	H3	H4	H5	H6	H7	H8	H9
Original	✗	✓	✓	✓	✗	✗	✓	✓	✓
T2D	✗	✓	✓	✗	✗	✗	✗	✓	✗

Table 5.2: Comparison of the results between our study and the original study.

5.1.6 Limitations

Generalizability of the results is limited.

Since the user study only explored effects on symmetric bimanual interaction and not also on asymmetric bimanual interaction, it is not possible to compare both interaction types whether they differ among the investigated factors. Furthermore, an object tracking task is no usual standard task when using a bare-hand input device. Therefore, the generalizability of the results is limited. We manipulated the *difficulty* of the task by varying the speed, but there might be other factors which influence the difficulty of a task.

This chapter gave the statistical results and connections of the conducted user study described in the previous chapter. But what is the contribution of this study? The last thing is to round off the study by giving interpretations and resulting further research questions.

Chapter 6

Summary and Future Work

In this thesis the software testbed Leap Blender was presented. The diversity of different bare-hand input devices and several studies with them show that there is the need of investigating the users behavior with this new kind of input. To avoid reimplementation and the drawbacks of implementing everything from scratch, we developed the software testbed Leap Blender. This testbed provides some base functionalities and an extendable and customizable structure to ease the preparation of studies for bare-hand input. The concepts and the structure of the program and how to use it to prepare a study task was explained. In order to test the testbed and show the advantages over a reimplementation, the study of Balakrishnan et al. [2000] was replicated. The study conducted what potential factors may influence symmetric bimanual interaction with a touchscreen, the Leap Motion Controller in 2D tasks, and the Leap Motion Controller in 3D tasks. For that, we let participants perform an object tracking task with different conditions and setups.

Testbed Leap Blender was presented.

Study tested the testbed and showed its benefits.

This chapter summarizes the study and draws conclusions from the results. Also, the usefulness of the testbed for the preparation of our study and some possible future work is highlighted.

6.1 Summary and Contributions

6.1.1 Case Study

Summary of the results of the study.

From the user study a set of results can be extracted. First, an integrated visual stimuli has no effect on the users behavior. Neither the performance, nor the level of symmetry and parallelism are influenced by this factor. Secondly, the attention has an effect on manipulating objects, with a better performance and parallelism of singular attention conditions. The symmetry is influenced just in a 3D setup with the Leap Motion Controller. In 2D setups the symmetry is not influenced by attention. Thirdly, slow movements are conducted with better performance, but fast movements are conducted more parallelly. There is no effect on the symmetry of motions by changing the difficulty. Fourthly, there is a slight difference between the performance of both hands. Lastly, performance and level of symmetry and parallelism of the motions of the user differ in setups for the touchscreen, the Leap Motion Controller with 2D manipulations and the Leap Motion Controller with 3D manipulations. Best results are achieved with the touchscreen, with which the results of the different users are close together. With a Leap Motion setup for 3D manipulations the worst results are achieved. The results of a setup with the Leap Motion Controller for 2D manipulations are located in between the two other setups.

Shared phenomenon among the setups explainable with fundamental factors.

The shared phenomenon of the motions among the three setups can be explained with the factors *integration*, *attention* and *speed* being fundamental factors of manipulation and independent from the input device. But this is not covered by this study. The fact that divided attention causes the motions of the user to be more asymmetric in the L3D setup, but not in the other two setups, might come from the problems of the user to orient in a 3D environment. This is supported by overall worse symmetrical motions in the 3D setup.

Differences might come from orientation problems in 3D.

Differences among the studies have to be investigated.

The differences to the original study do not exclude a validity of both results, because in the original study a digitizing tablet with two pens was used and not a touchscreen. Users might perform motions differently when using objects for input. This was not investigated by our study. But most of the results are the same, what proves the credibility of the testbed.

Same results prove the credibility of the testbed.

Overall, there is a slight asymmetry between the two hands what supports the kinematic chain model of Guiard [1987], because one hand follows the other. Therefore, total symmetry is not possible at all what has to be considered in GUI design. Providing supporting constraints might be a solution. Even if the three setups differ in performance, symmetry, and parallelism of the motions of the user, each of them is influenced by factors in nearly the same way. So, finding the best input device is no matter of finding the best device for the prevalent factors which might influence the results, but finding an input device for the needed performance of the user. If the motions of the user should be accurate, symmetrical, and parallel, the touchscreen suits the best. With it the expected results are best and stable among the users. The advantage of the Leap Motion Controller is to control three dimensions directly. This is not possible with the touchscreen which only can manipulate two dimensions at the same time. Using the Leap Motion Controller as 2D input device does not have benefits compared to the touchscreen.

Total symmetry is not possible.

Considering needed accuracy, symmetry, and parallelism is enough for finding a device.

For symmetric bimanual interaction difficult tasks and divided attention should be avoided. Best results are expected for this kind of interaction if the task is concentrated in the focal visual field of the user. Otherwise, additional feedback is needed. Furthermore, divided attention and difficulty cause the interaction to become sequential. Our results show that no visual integration is needed between the two hands. Thus, it is not necessary to connect the cursors of both hands or show the edges of a manipulating object. Since there is only a slight difference between the two hands and there is no effect influencing the symmetry, one does not have to care about a leading or main hand for a task. Comparing the symmetry and parallelism, we found that worse parallelism does not lead to worse symmetry. Therefore, it is not necessary to choose a parallel task to achieve symmetrical motions.

Best performance expected in slow tasks with singular attention.

No visual integration is needed.

No parallelism is needed for symmetry.

6.1.2 Software Testbed

Testbed eased the preparation of the study.

The testbed eased the implementation of the user study. After the decision for a tracking task we could start directly with the implementation and did not waste time for implementing base functionalities for logging, communication with the Leap Motion Controller, and displaying. First, the scene was created by the tools of Blender, then the logging module was customized to log necessary information for the evaluation of the study. Because there is already a logger set up, the only thing to do was to get the scene information with the Python API of Blender. At last, the needed logic was implemented in the simulation class. Much effort was saved, because the structure of a frequently update is already given. The Simulator was just extended by movements for the target spheres, a function to draw big spheres at the position of the index finger, and a function to connect the two cursors for the integrated condition. So, just task specific work had to be done.

Benefits of the testbed.

There was no time intensive testing phase, because only the extended parts had to be tested. The concentration was on adapting the task specific functions. Additionally, for this study it was not necessary to understand the whole testbed, because no extensions of the base functionality were needed and the study specific parts are fully independent of the functionality of the testbed.

Focus was on the task.

Because of this testbed, more time was available for setting up a task, for conducting the study, and for evaluation. Therefore, we could focus on the study and not on the system.

6.2 Future Work

Necessary to improve the detection of hands and fingers.

This thesis introduced the new software testbed Leap Blender. Because of that, there is the possibility for further improvements and enhancements. At some points we struggled with the detection of the hands by the Leap Motion Controller. If motions are performed where one hand covers the other, the controller loses the covered hand. Also, at the corners of the detection range there were some problems. To solve this, the testbed has to be upgraded to the Leap Motion v2 software. This new version of the Leap Motion Controller software provides an always visible and stable skeleton, based on assumptions, and would solve the problems of detection and increase the smoothness of motions. In order to provide a larger choice for interaction techniques, more gestures could be implemented. For that, the already provided gestures of the Leap Motion Controller could be used and integrated into the testbed.

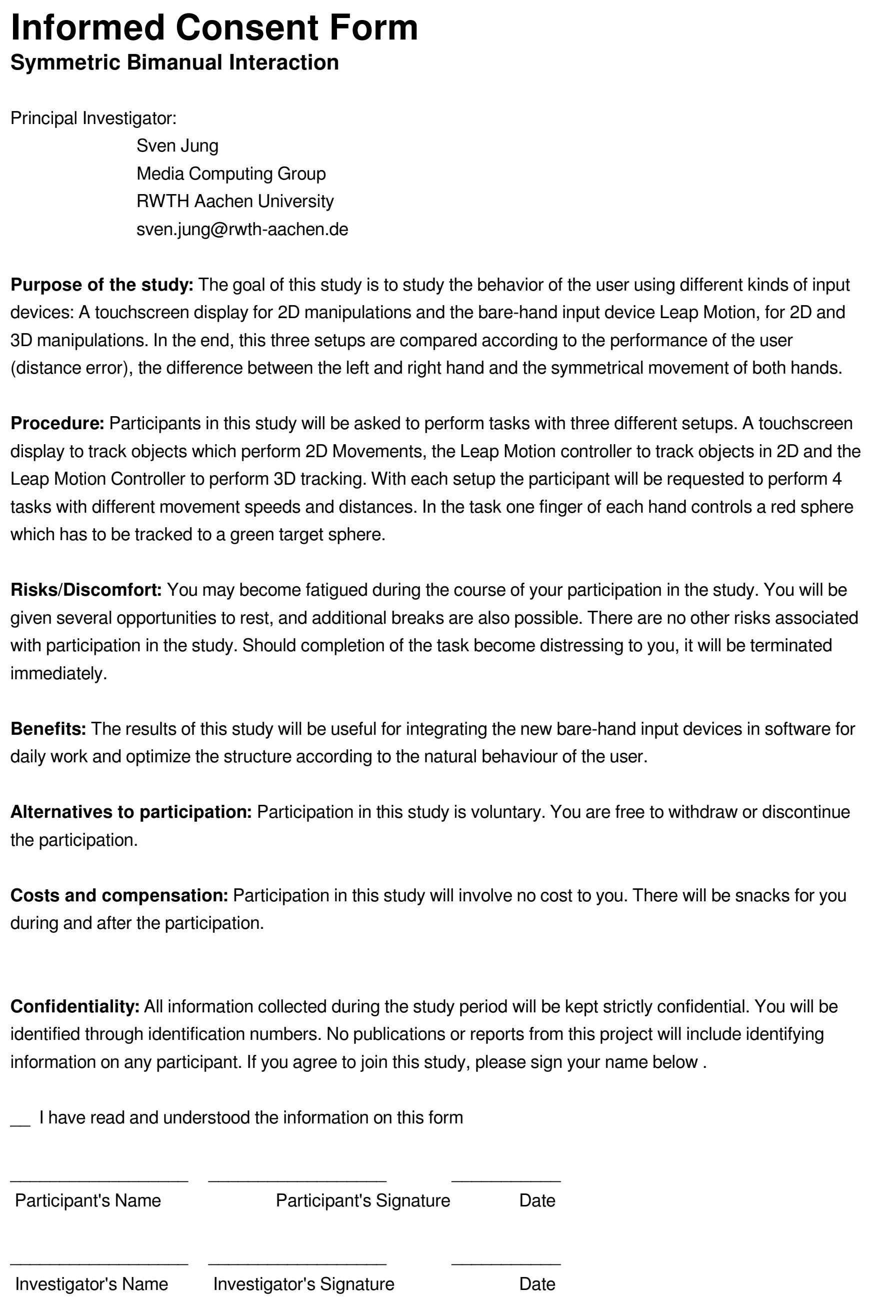
More studies are needed to make general statements.

In order to make general statements about natural motions of the user in mid-air manipulation, it is necessary to make further studies about more natural tasks of interacting with the computer and explore more influencing factors. Furthermore, additional studies about asymmetric bimanual interaction would make it possible to compare symmetric and asymmetric interaction. Maybe there are fundamental factors which are influencing the human performance, no matter of the input type. Our study revealed a slight asymmetry between both hands. To find out which hand follows the other and if this differs between right and left handed people, also further studies are necessary.

There are many open research questions for the topic of midair manipulation. The introduced testbed is a way to more and easier studies of bare-hand input. Further studies can improve the knowledge about the human behavior in free space manipulation tasks and lead towards new and easier interactions with the computer.

Appendix A

Study user information



Informed Consent Form

Symmetric Bimanual Interaction

Principal Investigator:

Sven Jung
Media Computing Group
RWTH Aachen University
sven.jung@rwth-aachen.de

Purpose of the study: The goal of this study is to study the behavior of the user using different kinds of input devices: A touchscreen display for 2D manipulations and the bare-hand input device Leap Motion, for 2D and 3D manipulations. In the end, this three setups are compared according to the performance of the user (distance error), the difference between the left and right hand and the symmetrical movement of both hands.

Procedure: Participants in this study will be asked to perform tasks with three different setups. A touchscreen display to track objects which perform 2D Movements, the Leap Motion controller to track objects in 2D and the Leap Motion Controller to perform 3D tracking. With each setup the participant will be requested to perform 4 tasks with different movement speeds and distances. In the task one finger of each hand controls a red sphere which has to be tracked to a green target sphere.

Risks/Discomfort: You may become fatigued during the course of your participation in the study. You will be given several opportunities to rest, and additional breaks are also possible. There are no other risks associated with participation in the study. Should completion of the task become distressing to you, it will be terminated immediately.

Benefits: The results of this study will be useful for integrating the new bare-hand input devices in software for daily work and optimize the structure according to the natural behaviour of the user.

Alternatives to participation: Participation in this study is voluntary. You are free to withdraw or discontinue the participation.

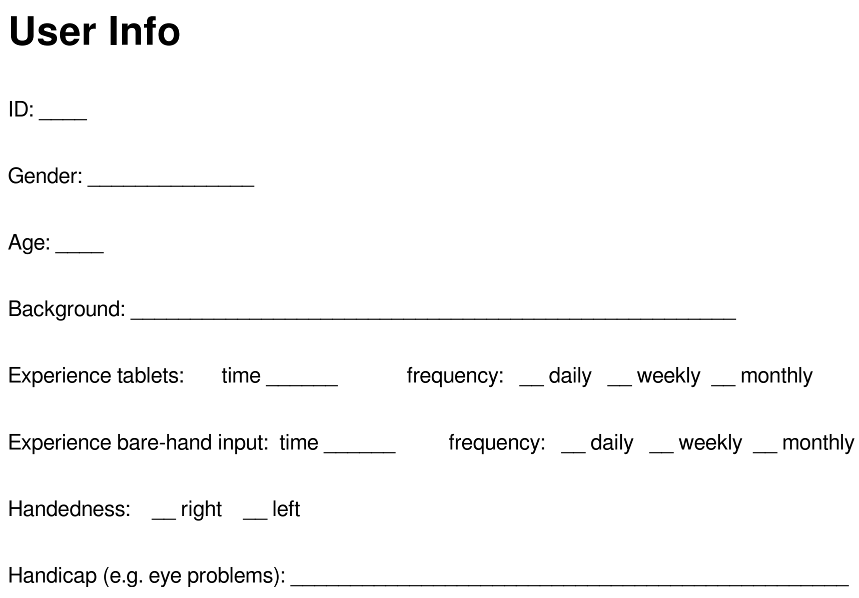
Costs and compensation: Participation in this study will involve no cost to you. There will be snacks for you during and after the participation.

Confidentiality: All information collected during the study period will be kept strictly confidential. You will be identified through identification numbers. No publications or reports from this project will include identifying information on any participant. If you agree to join this study, please sign your name below .

__ I have read and understood the information on this form

Participant's Name	Participant's Signature	Date
Investigator's Name	Investigator's Signature	Date

Figure A.1: Consent form used for user study.



User Info

ID: ____

Gender: ______________

Age: ____

Background: ____

Experience tablets: time ______ frequency: __ daily __ weekly __ monthly

Experience bare-hand input: time ______ frequency: __ daily __ weekly __ monthly

Handedness: __ right __ left

Handicap (e.g. eye problems): ____

Figure A.2: User information form used for user study.

Bibliography

Bruno R. De Ara´jo et al. Modeling on and above a stereoscopic multitouch display. 2012.

Ravin Balakrishnan et al. Exploring bimanual camera control and object manipulation in 3d graphics interfaces. 1999 ACM Conference on Human Factors in Computing Systems (CHI'99), 1999.

Ravin Balakrishnan et al. Symmetric bimanual interaction. CHI '2000, 2000.

William Buxton et al. A study in two-handed input. CHI '86 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 1986.

Didier Casalta et al. Evaluating two-handed input techniques: Rectangle editing and navigation. ACM CHI'99, 1999.

Géry Casiez et al. 1€filter: A simple speed-based low-pass filter for noisy input in interactive systems. CHI '2012, 2012.

Yves Guiard. Asymmetric division of labor in human skilled bimanual action: The kinematic chain as a model. 1987.

Maurice R. Masliah et al. Measuring the allocation of control across degrees-of-freedom. Graphics Interface (GI) '99, 1999.

Fabrizio Nunnari et al. Hand tracking for 3d editing, Last visited: Juli 2014. URL http://slsi.dfki.de/software-and-resources/hand-tracking-for-3d-editing/.

Leigh Ellen Potter et al. The leap motion controller: A view on sign language. OzChi 13, 2013.

Jeremy Sutton. Air painting with corel painter freestyle and the leap motion controller: A revolutionary new way to paint! SIGGRAPH 13, 2013.

Radu-Daniel Vatavu et al. Leap gestures for tv: Insights from an elicitation study. TVX 2014, 2014.

Robert Y. Wang et al. 6d hands: Markerless hand tracking for computer aided design. UIST '11, 2011.

Frank Weichert et al. Analysis of the accuracy and robustness of the leap motion controller. May 2013.

Igor Zubrycki et al. *Recent Advances in Automation, Robotics and Measuring Techniques*. Springer International Publishing, 2014.

Index

3Gear, 5

asymmetric interaction, 4

Balakrishnan et al., 7
bare hand input device, 2
bimanual interaction, 4
Blender, 9

contributions
- abstract, 4
- detailed, 54–56

Data Gloves, 5

evaluation, *see* results
extensibility, 28–32

future work, 56–57

gestures, 21

hypotheses, 42

implementation, 9–32
independent variables, 34–35

Kinect, 5
Kinematic Chain Model, 7

L2D, *see* Leap Motion Controller 2D condition
L3D, *see* Leap Motion Controller 3D condition
Leap Blender, xiii
Leap Motion Controller
- advantages, 5
- appearance, 2

m-metric, 37
mode
- edit mode, 27

- object mode, 27

module
- GestureHandler, 12
- LeapModal, 11
- LeapReceiver, 11
- LeapSelector, 13
- logging, 12
- Manipulator, 12
- ObjectHandler, 13
- Settings file, 13
- Simulator, 12
- StudyLogger, 12
- Transformator, 12

OneEuroFilter, 26

Python, 10

results, 45
RMS, *see* Root Mean Square Error
Root Mean Square Error, 36

study, 33–43
summary, 53
Symmetric Bimanual Interaction, 7
symmetric interaction, 4

T2D, *see* Touchscreen 2D condition
two-handed input, 4

Typeset September 24, 2014