

Multitouchkit: A Software Framework for Touch Input and Tangibles on Tabletops and Mobile Devices.

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



*by
René Linden*

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Ulrik Schroeder

Registration date: 13/08/2015
Submission date: 30/09/2015

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 2015
René Linden

Contents

Abstract	xi
Acknowledgements	xiii
Conventions	xv
1 Introduction	1
2 Related work	5
3 The Multitouchkit - MTK	11
3.1 General design decisions	11
3.2 Touch Processing	14
3.2.1 MTKTrace	16
3.2.2 MTKInputSource	19
3.2.3 Initialization of Touch Processing . . .	23
3.2.4 Recursive Touch Processing	29
3.3 Tangibles	33

3.3.1	PUCs: Passive Untouched Capacitive Widgets	34
3.3.2	PERCs: Persistently Trackable Tangibles on Capacitive Multi-Touch Displays	41
3.3.3	Tangible Simulator	51
3.4	Gestures	53
3.4.1	Standard Gestures	53
3.4.2	Custom Gesture Recognizer.	56
3.5	Visualization Support	58
3.6	TouchControlCenter	61
3.6.1	TCC in MacOS	61
3.6.2	TCC in iOS	62
3.6.3	MTKTangibleCreationScene	63
3.7	Sample Applications	66
4	Evaluation	69
4.1	Architecture	69
4.2	Scope	72
4.3	Features	73
4.4	Summary	75
5	Summary	77
5.1	Future work	77

Bibliography 79

Index 85

List of Figures

3.1	SpriteKit frame loop.	15
3.2	Different states of MTKTrace.	17
3.3	Sample scene.	25
3.4	Sample scene hierarchy.	26
3.5	Trace distribution decision tree.	27
3.6	Trace distribution shape problem.	28
3.7	Sample scene touch processing.	31
3.8	Illustration of PUC.	34
3.9	Scanline problem.	39
3.10	Two trace triples fit one tangible.	40
3.11	Illustration of PERC.	42
3.12	States of PERCs.	43
3.13	Two position problem.	44
3.14	Four position problem.	45
3.15	PERC decision tree.	52

3.16	Discrete gesture recognizer states.	54
3.17	States of continuous gesture recognizer.	55
3.18	Star Wars scene.	58
3.19	MTKPieMenu.	59
3.20	TCC for iOS.	63
3.21	MTKTangibleCreationScene.	64
3.22	AirHockey.	66
3.23	ColorFighter.	67
4.1	Possible layers of multi-touch frameworks.	70
4.2	Overview of existing multi-touch frameworks.	75

Abstract

In this thesis we present the Multitouchkit (MTK), a software framework for touch input and tangibles on tabletops and mobile devices, which is dedicated to ease the development of multi-touch applications on MacOS and iOS. In contrast to other frameworks, the MTK can be used in Apple's development environment. It is based on SpriteKit, a framework by Apple to develop applications with rich 2D graphics. The MTK is the first framework to support PUCs and PERCs introduced by Voelker et al. [2013, 2015]. The MTK enables developers to use any input hardware as source of touch information and use it in an MacOS and iOS application.

Acknowledgements

I want to thank everybody who supported me in my work, especially Simon Voelker for providing me with a lot of feedback and support in discussion about important design decisions. I am very grateful that Prof. Borchers made it possible for me to work on such an interesting project at his chair. I also want to thank Prof. Schroeder as my second examiner. I want to thank everyone else at the chair, who helped me when facing technical problems. Thanks to everyone who proofread my work. I want to thank my family and friends who supported me in my study and without whom I would not have been able to write this thesis. Thanks for all the support!

Conventions

Throughout this thesis we use the following conventions.

The thesis is written in American English.

MTK is short for Multitouchkit, which is the name of the described framework.

The thesis is written in first person plural. This was not used because several persons worked on this thesis but for esthetical reasons.

The MTK is based on a prototype version created by Simon Voelker. The prototype was fully revisited, refactored, documented, redesigned and extended.

Any term that is introduced for the first time is written in italic. Following appearances will not be italic. The exclusion of this convention are Objective-C methods, which are always written italic. Additionally are these functions reduced to a form without parameters. For example, *-(void)updateWithTimestamp:(NSTimeInterval)timestamp* will be reduced to *updateWithTimestamp:*.

Chapter 1

Introduction

Multi-touch interaction reached everyday life. Consumers use it in smartphones, tablets, laptops, tabletops, and more. An extension of multi-touch that is not commercially available yet is the use of tangible widgets which we will call tangibles.

Tangibles are mostly unknown to consumers.

Tangibles are objects that can be recognized by multi-touch surfaces. Researched for over a decade they are often referred to be useful in a large variety of application scenarios in combination with multi-touch surfaces [Rekimoto, 2002][Terrenghi et al., 2007]. Due to their shapes tangibles can give haptic feedback, which is mostly missing on multi-touch surfaces. This allows eyes free interaction with multi-touch surfaces, which is else very cumbersome or impossible [Weiss et al., 2009].

Tangibles are not new.

One of the reasons why tangibles did not reach everyday life may be that most of the commercially available multi-touch devices use capacitive touch technology. While tangibles on capacitive touch technology were already researched for over a decade [Rekimoto, 2002], PUCs - Passive Untouched Capacitive Widgets by Voelker et al. [2013] are the first tangibles that could be detected on commercially available and unmodified multi-touch displays. Other research relies on the user touching the tangible [Yu et al., 2011], uses modified touch displays [Liang et al., 2014], or used touch surfaces which use infrared light to

Tangibles are hard to detect on capacitive touch.

detect touches [Schöning et al., 2010].

PERCs are an improvement of PUCs.

To improve PUCs and increase the number of possible applications for tangibles on unmodified multi-touch surfaces Voelker et al. [2015] developed PERCs - Persistently Trackable Tangibles on Capacitive Multi-Touch Displays. Those new tangibles solve some problems that could arise when using PUCs. For example if a PUC does not move it will last only 5-30 seconds, depending on the filter mechanics of the recognition hardware. This arises the problem that the software can not distinguish between a tangible that was lifted and one that was filtered [Voelker et al., 2015]. Which other problems PERCs solve and how those and PUCs exactly work is described in Section 3.3.

A framework for development with tangibles is missing.

PERCs can be detected on many commercially available multi-touch surfaces and can be build with costs lower than 25 Dollar [Voelker et al., 2015]. Hereby they open a wide range of possible applications. A software framework that enables applications to receive multi-touch events and helps recognizing tangibles could ease the development process of such applications. We listed several frameworks and toolkits in Section 2 which have these properties. None of these is capable of detecting PERCs. This was no surprise since PERCs were developed only recently, but also none of the frameworks support the development of native iOS or MacOS applications.

Neither MacOS or iOS support tangibles.

Apple's MacOS and iOS are the most used operating systems after Windows and Android. The AppStore offers about 1.5 million applications. Unfortunately neither MacOS nor iOS support any tangibles. MacOS is not even capable of multi-touch other than the recognition of gestures via trackpad. Therefore the demand for such a framework exists.

We decided to develop the MTK.

To fulfill this demand we decided to develop a framework that enables developers to create rich 2D applications that are capable of multi-touch and tangible interaction. The framework should be written in Objective-C or Swift to be fully compatible with all Apple development tools and applications written for any Apple operating system. Additionally the developers should be able to use any commer-

cially available input source in combination with the framework, but focus should be towards multi-touch surfaces.

With these requirements in mind we implemented the *Multitouchkit*, short *MTK*, which we present in Chapter 3. We will evaluate the MTK with the feature list for multi-touch frameworks presented by Kammer et al. [2010] in Chapter 4. Before these chapters we will discuss existing frameworks and their fit for our requirements.

The MTK introduces tangibles to MacOS and iOS.

Chapter 2

Related work

In our search for software frameworks we set the requirements that it is written in either Objective-C or Swift and that it supports multi-touch and tangible input from many different input sources. Many of the found frameworks support different hardware as input for multi-touch and tangibles, but none was written in Objective-C or Swift. We now shortly explain each of the most relevant frameworks and toolkits.

GestureWorks is a SDK written in C++ to support the development of multi-touch applications written in different programming languages like C++, C#, Java, and Python. It is distributed and developed by Ideum. Unfortunately the framework does not allow the development of applications for MacOS or iOS. Additionally the framework does not support tangibles and is not open source, therefore an adaption of the framework was not possible for us [Ideum].

Breezemultitouch is a multi-touch framework that is targeted to show all internals of windows processes. Breezemultitouch allows in comparison to Windows 7 the developer to see how the rotate, move, etc. actions are interpreted, and allows to change this interpretation process. It targets Windows platforms by relying on Windows Present-

None of the found frameworks was written in Objective-C or Swift.

GestureWorks is a commercial framework developed by Ideum.

Breezemultitouch is targeted to Microsoft Windows.

<p>Miria is a multi-device input SDK for Silverlight and Moonlight.</p>	<p>tation Foundation (WPF)¹ and is completely oblivious of tangibles [Mindstorm Inc.].</p>
<p>jQMultitouch is a multi-touch web interface development framework and toolkit inspired by jQuery</p>	<p>Miria is a SDK to bring multi-device input UI controls into Silverlight² and Moonlight³. The SDK includes a set of multi-touch ready and gestures based user controls. Unfortunately it does not support MacOS or iOS, neither it does have any support for tangibles [CodePlex].</p>
<p>Midas is a Java framework that allows the developer to easily create complex gesture recognizer.</p>	<p>jQMultitouch is a lightweight toolkit and development framework for multi-touch web interface development [Nebeling and Norrie, 2012]. It is inspired by jQuery⁴ and allows the use of different browsers which support touch events. It unifies these touch events to allow the development of cross-browser applications. Additionally it includes default gesture recognizer, gesture templates, touch support as well as touch histories. Nebeling and Norrie [2012] reported that the implementation has performance issues.</p>
<p>TUIO is the de facto standard protocol for multi-touch communication via network.</p>	<p>Midas is a Java framework that is developed to provide adequate software engineering abstractions for developers to close the gap between the evolution in the multi-touch technology and software detection mechanisms. It mostly focuses on the declarative definition of gestures. This enables the developer to easily implement gestures without the need of processing a continuous stream of data from an input source. Midas allows the unification of input sources and has an easy way to allow developers the usage of gesture. The framework is not open source and has no support for tangibles [Scholliers et al., 2011].</p>
	<p>TUIO is a protocol defined to transport touch and tangible information between hardware and software layer via network [Kaltenbrunner et al., 2005]. It became the de facto standard protocol to use [Laufs and Ruff, 2010]. Kaltenbrunner and Bencina [2007] for example use it in their framework reacTIVision, an open source, cross-platform computer vision framework allowing the track-</p>

¹<https://msdn.microsoft.com/en-us/library/ms754130>

²<http://www.microsoft.com/silverlight>

³<http://www.mono-project.com/moonlight>

⁴<https://jquery.com>

ing of fiducial markers attached onto physical objects and multi-touch finger tracking. The framework is not listed in this chapter due to the fact that it is fully focused on visual tracking. Given its high popularity in research we added the support for receiving relevant TUIO data to the MTK.

TUIO AS3 is a toolkit that supports rapid prototyping of multi-touch user interfaces in combination with tangible. It was presented by Luderschmidt et al. [2010] and is based on TUIO and ActionScript⁵. The idea of TUIO AS3 was basically to enable applications written for Adobe Flash to use TUIO. This was an innovation since TUIO needs UDP or TCP connection which could not be easily achieved with Adobe Flash.

TUIO AS3 combines TUIO and ActionScript.

Argos is a graphical user interface builder for multi-touch applications, focused in musical performance and sound synthesis. One of the main goals of Argos is to provide a suite of C++ classes to facilitate the creation of innovative and experimental UI widgets. Argos is written in C++ and is thereby able to work on all major operating systems. It is also able to receive input data from many commercial multi-touch devices [Diakopoulos and Kapur, 2010].

Argos is a graphical UI builder with focus on musicians.

MT4j is a cross platform Java framework presented by Laufs and Ruff [2010]. It is focused to rapid and easy development of visually rich 2D and 3D applications. It is also able to support different kinds of input devices with a special focus on multi-touch support. The only support for tangibles is provided using TUIO, which is focused towards visually detected tangibles.

MT4j is a cross platform Java framework.

Squidy is a Java framework developed from 2007 to 2012 to ease the design of natural user interfaces by unifying various device drivers, frameworks and tracking toolkits in a common library and providing a central and easy-to-use visual design environment [König et al., 2009].

Squidy unifies received data from different input sources.

Sparsh UI is a project of the Iowa State University's Virtual Reality Applications Center (VRAC). It focuses on enabling users to easily create multi-touch applications on a variety of hardware platforms. Sparsh UI supports applications

Sparsh UI enable users to easily create multi-touch applications.

⁵<http://www.adobe.com/devnet/actionsript.htm>Actionscript

written in C/C++ and Java [Ramanahally et al., 2009].

PyMT is written in python and has some serious performance issues.

PyMT is a python module for developing multi-touch enabled and media rich applications [Hansen et al., 2009]. Laufs and Ruff [2010] complain about performance issues in some parts written in Python. The developers promised rebuilds in C and C++ to improve the performance, but no such updates were done.

Graffiti manages multi-touch interactions in tabletop interfaces.

Graffiti is a C# framework built on top of a TUIO client which manages multi-touch interactions in tabletop interfaces. It can be considered a similar approach to the MTK. Graffiti is able to support several hardware due to its TUIO client and may help the developer in his creation process of an application [De Nardi].

libTisch is built for cross-platform development of novel UI applications.

libTisch, also called TISCH framework, is a project targeted to cross-platform development of novel UI applications [Echtler and Klinker, 2008]. Echtler and Klinker [2008] describe it as a project to combine the common traits of existing frameworks. The project is still active and focused towards vision based touch sensing. The framework has support for multi-touch, tangible interfaces, and full-body interaction. It is cross platform compatible with Linux, MacOS X and Windows.

iGesture is a framework mostly focused on gestures.

iGesture is a Java-based gesture recognition framework focusing on extensibility and cross-application reusability. It includes tools for gesture recognition as well as the creation and management of gesture sets. iGesture focuses on single-touch and is meant as an extension of other software. It was not created to work as a framework that can gather different input sources of multi-touch hardware to help the developers creating multi-touch applications [Signer et al., 2007].

Touchlib is a library developed for creating multi-touch interaction surfaces.

TouchLib is a library for creating multi-touch interaction surfaces. It is written in C++ and works on Windows. It has capabilities to send recognized touches via TUIO, allowing the connection to other operating systems like MacOS. Additionally it includes a configuration app and a few demos to get started [NUI Group].

The result of the search is that none of the found frameworks is written in Objective-C or Swift. Most frameworks choose a language that allows cross platform compatibilities. Based on these results we decided to implement the MTK and not to improve any existing framework to our needs. In the next chapter we will discuss our framework, the Multitouchkit.

None of the found frameworks met the requirements.

Chapter 3

The Multitouchkit - MTK

In this chapter we explain the different features of the MTK and its concepts. We start with design decisions and will continue by explaining each of the features in more detail. In Chapter 4 we then categorize the MTK in the list of multi-touch frameworks analyzed by Kammer et al. [2010].

3.1 General design decisions

Support of MacOS and iOS. The MTK is written in Objective-C to provide native support for MacOS and iOS. This also allows the use of any of Apple's development tools. It would have been possible to use Swift instead but since the language is relatively new, still undergoing major changes and the prototype already written in Objective-C we decide to not use Swift.

The MTK is written in Objective-C.

SpriteKit. We decided to base the MTK on SpriteKit¹, a framework that provides a graphics rendering and animation infrastructure, which is often used to develop high-performance, battery efficient 2D games for iOS and

SpriteKit is used as base of the MTK.

¹<https://developer.apple.com/spritekit/>

	<p>MacOS. This should allow developers to build any UI or application required. SpriteKit and the MTK are focused towards 2D applications, but they can be easily extended using SceneKit² to allow 3D elements. We had the option to use other frameworks like Unity³ or Cocos2D⁴, but chose SpriteKit to continue our focus on developing for MacOS and iOS using Apple's development tools.</p>
<p>The rendered content is organized in a tree with a scene as root and nodes as elements.</p>	<p>SpriteKit uses a traditional rendering loop where the content of each frame is processed before the frame is rendered, see Figure 3.1. Animation and rendering is performed by an SKView object. The content of this view is organized into scenes, which are represented by SKScene objects. The SKScene class is a descendant of the SKNode class. When using SpriteKit, nodes are the fundamental building blocks for all content, with the scene object acting as the root node for a tree of node objects. The scene and its children determine which content is drawn and how it is rendered.</p>
<p>Using the MTK one has to subclass MTKScene.</p>	<p>To create a application using SpriteKit, one either subclasses the SKScene class or create a scene delegate. An application that uses the MTK has to use an active MTKScene instead, which is a subclass of SKScene. This is due to the fact that the touch processing, as described in Section 3.2, is performed in one of SpriteKits callbacks.</p>
<p>Apple like documentation is included in the MTK.</p>	<p>AppleDoc. To provide a rich documentation that matches the MTK's relation to Apple's operating systems and frameworks we decided to use AppleDoc⁵. AppleDoc is an open source project that generates Apple-like documentation based on the header file documentation, which we extensively included in the MTK. Additionally introductory tutorials and guides for common hardware setups are included in the documentation.</p>
<p>Not all input hardware is automatically supported by the MTK.</p>	<p>Support of several hardware types as input. One of our main requirements is that developers are able to easily use</p> <hr/> <p>²https://developer.apple.com/scenekit/ ³http://unity3d.com ⁴http://www.cocos2d-x.org ⁵https://github.com/tomaz/appledoc</p>

any input hardware. We implemented the MTK to develop 2D multi-touch applications targeted for the use with iPhones, iPads, multi-touch tabletops and similar input hardware. Due to this fact the limitation of supported input hardware is that their sent information is transformable to an object of class `MTKTrace`, explained in Section 3.2.1, which has all properties to represent a touch point that was recognized by a multi-touch surface. The only required information to create such an `MTKTrace` object is a position in a 2D coordinate. Hardware input that does not fulfill this requirement can not be processed in a normal way by the MTK. To still use this hardware input developers might implement their own input source, as in Section 3.2.2, to receive the input data and process it themselves.

For many other input hardware, the MTK allows to choose between several implemented input sources, which each represents one type of input hardware. The already implemented sources cover all hardware setups we had in our development environment. Additionally the JSON⁶ and TUIO input sources can receive data via network, which allows the connection of nearly any input hardware. Developers can implement their own input source subclass for specific input sources, if none of the given options fits. A list of supported input source types and a description of how the implementations work, can be found in Section 3.2.2.

All multi-touch hardware is supported by the MTK.

Support of multi-touch. The MTK supports an unrestricted number of touches of an unrestricted number of input sources at the same time. The only restriction we could think of is that the hardware may at some point be unable to render all cursors, which could cause a huge drop in the frame rate. All touches, independent from their input source, have an internal representation called `MTKTrace`, which we explain in Section 3.2.1. Additionally the MTK supports gestures by providing a set of standard recognizers and several customization options, described in Section 3.4.

The MTK supports any number of touches.

⁶<http://json.org>

PUCs and PERCs can be recognized by the MTK.

Support of tangibles. The second main requirement of the MTK is the support of tangibles. The MTK can recognize PUCs and PERCs. The detailed explanation of their recognition process can be found in Section 3.3. The current version of the MTK does not support other tangibles. As Kammer et al. [2010] reported many other frameworks support tangibles using TUIO. This could be a valuable extension of the MTK and could be implemented in the future.

To ease the development process the MTK includes standard UI elements.

Support of standard and custom UI. The use of standard UI components in combination with touch is a common use case that is supported by most of the existing frameworks listed by Kammer et al. [2010] and should therefore be supported by the MTK, too. We implemented several standard UI components and created the possibility for developers to add custom build UI elements based on any SpriteKit node. This is achieved due to the fact that the touch processing is performed in a category of SKNode, which is the parent class of all nodes in a SpriteKit application. In Objective-C a category defines additional functionality of an existing class without subclassing it, even if the source code is unavailable. Therefore any node is automatically able to process touches and gestures, which allows the creation of many different UI elements. Section 3.2 describes all details about the touch processing.

The framework includes the TCC to change standard settings.

Customizability. The MTK allows several settings in regard of input sources, output views, start scenes, tangibles and more. These configurations are saved in a XML file. We implemented the *TouchControlCenter*, short *TCC*, to provide a user interface to change these settings, as described in Section 3.6.

3.2 Touch Processing

In this section we explain the touch processing of the MTK, which spans from the collection of all input data to the pro-

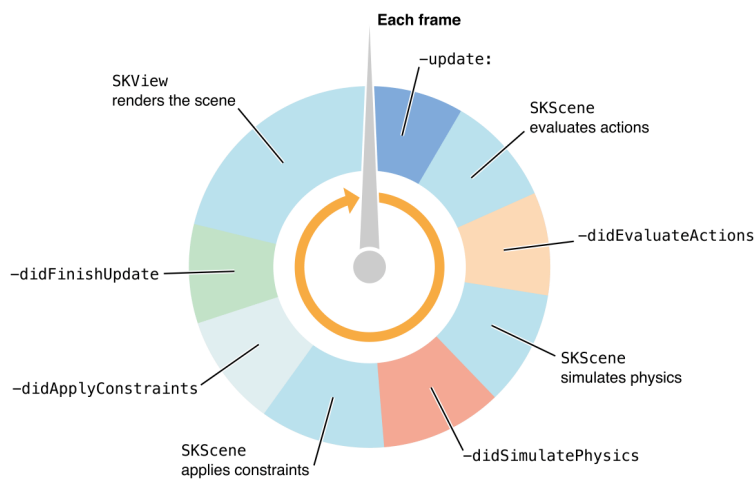


Figure 3.1: This image represents the calls executed by SpriteKit in each frame. This is a copy of the original on the Apple developer documentation website.

cessing of tangibles and gestures.

As mentioned earlier the MTK is based on SpriteKit. For that, we implemented the touch processing in the *update:* call of SpriteKit's processing loop, seen in Figure 3.1. This callback is the first in each frame and allows us to perform anything before any other processing of SpriteKit is done. It is called on the active SKScene. An application that uses the MTK has to use an active MTKScene instead, which is a subclass of SKScene. The *update:* method of MTKScene starts the touch processing. Any subclass of MTKScene need to call its parent's *update:* method in its own, to ensure the correct behavior of the MTK.

The touch processing in the MTK is split into two parts. The first part is the initialization, which starts with the updating of all MTKTrace objects. This is handled by each MTKInputSource. We will explain the classes MTKTrace and MTKInputSource the the following two sections. After this we will explain the initialization of the touch processing in Section 3.2.3 with the first part of the processing.

The touch processing is performed in the *update:* call of SpriteKit.

The touch processing is split in two parts.

3.2.1 MTKTrace

All information of one touch is saved in one MTKTrace object.

Each object of class MTKTrace, which we will call *trace*, represents the lifetime of one touch. A touch is normally anything that is recognized by the hardware as a human finger touching the multi-touch surface. For more specialized hardware it can be anything, but in the MTK it will be interpreted as a touch on a multi-touch surface.

The lifetime of a touch is saved in an entry for each frame.

The lifetime of a touch, which is saved by a trace, consists of information saved each frame while the touch was recognized by the hardware. In each frame a new MTKEntry object is created and added to the trace containing all current information of the touch. The object is called *entry*. Each input hardware might offer different information for each touch, but at least a position per recognized touch is required. Additional information can be saved in the properties offered by the MTKTrace. Therefore traces are the most important source of information for all analysis on touch events, in particular the tangible detection and gesture recognizers rely on these objects. In the following paragraphs we will discuss what kind of information is saved in each trace. Some values are saved per frame, others per trace.

Each trace has an unique identifier.

Identifier. The identifier is unique for each trace. While the application is running none of the traces will ever have the same identifier. The trace's identifier is determined by the MTK and is not related to any identifier provided by the input hardware.

The type of the origin helps to categorize touches.

Type of origin. The type of origin is defined by the input source that created the trace. It is used to identify from which kind of input source the trace was created. This is for example useful when implementing a gesture recognizer that only work for a specific kind of input source.

Identifies which input source created the trace.

Name of origin. The name of origin is set by the input source which created the trace, similar to the type of origin. It is used to identify the exact input source that created the

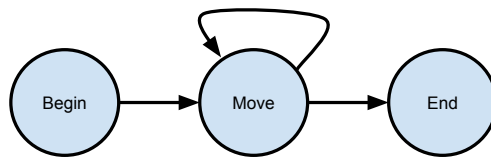


Figure 3.2: Different states of MTKTrace. Traces will start with a Begin state and finish in the End state. In between the state is always Move.

trace. This can be used for example to set a specific touch cursor for each input source.

State. In each frame the trace has one of three states, as seen in Figure 3.2. The state is saved in the entry created each frame. The first entry has the state Begin. It represents that the touch corresponding to the trace first appeared in this frame. The entry created in the frame in which the hardware signaled the end of the touch has the state End. This entry is also the last one, no further updates will reach the trace. The ended traces will be included in the processing for one last frame before they are moved to a set of old traces. All entries in between these will have the state Move, which implies that the touch is currently moving and thereby still updated.

Each trace can have one of three states.

Timestamp. All entries of the trace have a timestamp. The MTK determines one timestamp at the beginning of each frame. This timestamp is set in all entries created in that frame. This allows the comparison of events. For example all signals received from the tangibles bluetooth devices are also marked with the timestamp used in the frame and can thereby be compared to the timestamp of the trace's Begin state.

The timestamps of all entries are determined by the MTK.

A position per trace has to be provided by each input source.

Position. Each entry contains the position of the trace in the active MTKScene. The transformation from hardware to scene coordinates is done by the input source. The hardware coordinates are also saved in the entries, but are not directly accessible. This avoids the misuse of the hardware position, but makes them still available if needed for additional analysis. A position for each active touch is the minimum information a input source has to provide, every other information can either be determined by context or set to standard values.

The size can be influenced by minor and major axis.

Size. Major and minor axis. The size of the trace is used as size for its cursor. The major and minor axis can also be sent by the hardware and may help to determine a trace's size and shape.

Orientation is not a required value.

Orientation. The orientation is a vector defining in which direction the touch is pointing. If the hardware supports the orientation it will be set and can be taken in account in the cursor creation.

An MTKTrace contains all information collected in the lifetime of one touch.

Compression. MTKTrace is the representation of the complete lifetime of one touch on the multi-touch surface. This history of all information of all traces is especially interesting when recognizing gestures, but this permanent allocation of new objects caused some problems in performance. Therefore we changed the entry not to be an object but a C struct. We allocate about 300 entries, which are about 5 seconds of the applications runtime, for each trace at a time and fill them before new entries are allocated. This stopped the permanent allocation of objects and increased the overall performance of the MTK. The C programming in this part is hidden from the user and unit tests were added to the MTK to ensure the reliability.

MTKTraces will fill your memory.

While the fact that we save all traces with their full history is a nice feature used for example by gesture recognizer, it will on long application runs use all memory and will cause the system to crash. To solve this problem we added

two settings to the MTK to reduce the amount of memory traces occupy.

The first possibility for the user to reduce memory requirements is to set a *Compress Time*. Any trace that is in state End for a longer period than the Compress Time will have their entries deleted. All static information that are properties of the trace are saved separately and will still be available. Other properties that relied on the analysis of entries will be unavailable. This option frees a lot of memory if traces have a long lifetime, but our experience showed that this is not the case. Therefore does this compress option not save much memory. To free more memory we added the second compress option, a *Deletion Time*.

The Deletion Time is similar to the Compress Time. A trace that is in state End for a longer period than the Deletion Time will be deleted. All remaining strong references of the MTK are deleted, which allows the system to free the memory. This is no enforced memory deallocation. The memory may not be freed if any other object has a strong reference to the trace. This option will drastically free memory and allows to avoid memory issues at long application runtimes.

Compressing MTKTraces will remove most of the saved data.

To really ensure that the memory is not filled a Deletion Time was added.

3.2.2 MTKInputSource

Input sources are responsible for receiving data from input hardware and converting it into MTKTraces. We implemented several standard input sources to receive data from specific input hardware. Additionally we introduced a TUIO and a network (JSON) input source to provide support for any other possible input hardware. All these input sources are subclasses of MTKInputSource.

Each input source is a subclass of MTKInputSource.

The touch processing starts in each frame with the update of all traces, by calling *updateWithTime:* for each active input source. Input sources that were configured in the TouchControlCenter will be automatically loaded, others can be added manually to the list of all input sources, which is located in the MTKTable. In the *updateWithTime:* call in-

All input sources need to implement *updateWithTime:*.

put sources are responsible for converting all received input data to MTKTraces. Already existing traces have to be updated and new ones need to be created.

It would improve the MTK to extract the transformation code.

Input sources have to transfer each touch position given in hardware coordinates to scene coordinates. The current implementation transfers them relatively using a given input and output resolution. In this part the MTK could be improved to allow a more modular transformation that can be customized for each input source. A solution we have planned for future work is the use of a delegate in each input source. The delegate could implement methods that allow an arbitrary transformation from any input to any output. This is for example interesting to show traces of a specific input source in only a specific area of the output.

The MTK is still able to achieve an arbitrary transformation.

This was not yet implemented due to time limitations and the fact that the MTK is still able to achieve this using the global delegate. As we will discuss later in this chapter exists a global delegate exists that is called right after the processing of all input sources. At this point all traces could still be modified before they are used for any other processing.

In the following paragraphs we will present all already implemented input sources, followed by the explanation on how to create a new one.

The mouse input source can simulate touches and can be used for testing without actual touch input.

Mouse. One of the input source classes we implemented is able to process input data coming from an ordinary computer mouse under MacOS. A main purpose is to enable development without having a multi-touch interface connected. The mouse input source will emulate touch points from mouse events that are located in the scene. Any left mouse button press will generate a touch at the cursors position. The touch will follow all movements of the cursor while the left mouse button is held. The touch will disappear if the button is released. A right click creates a permanent touch at the cursors position. This touch can be dragged with the left mouse button and will disappear if clicked with the right mouse button.

UITouch. This input source is our standard input source for iOS. UITouch is the default format in iOS to handle touch input. Therefore we implemented an input source that receives all UITouch events from the UIView, which is opened in any iOS application created with the MTK, and transforms them into MTKTraces.

UITouch input source is based on the native iOS touch events.

TUIO. TUIO is a protocol to send touch and tangible events via network. It is used in several research publications and custom touch table setups, as already discussed in Chapter 2. TUIO supports different types of touches and tangible objects ranging from 2D to 3D. Due to the fact that it is the de facto standard protocol to send multi-touch events via network we decided to implement an input source for it. The input source is based on the MacOS client software provided on the TUIO homepage⁷. It transfers all received TUIO 2D touch events to traces. In doing so the MTK does support any hardware that is able to send TUIO elements to the MTK.

TUIO is widely used and is therefore addressed by the MTK.

While the protocol has build-in tangible support, the MTK does not support these. The MTK is designed to work with capacitive multi-touch surfaces, but TUIO was originally created to work with light based sensing techniques, as for example in reacTIVision⁸. Therefore the parameters received via TUIO protocol are meant for tangibles recognized via light sensing technology. The MTK is targeted to tangibles recognizes via capacitive touch technology, but it might still be interesting for future versions to support tangibles that are sensed with other technologies and sent via TUIO.

The MTK does not support TUIO tangibles.

JSON. The JSON input source can receive JSON objects from a given IP and port. The received data needs to be formatted as expected by the input source and at least contain a position and id per touch. In terms of compatibility this input source is the most important since it is created to work via network with nearly any other hardware.

The JSON input source is constructed as general network input source.

⁷<http://www.tuio.org/?software>

⁸<http://reactivision.sourceforge.net>

Any other hardware can be added to the support of MTK using JSON input source.

We use this input source for example with our PPI multi-touch surface. We implemented a windows application that collects all data from the PPI driver and sends them as JSON via network to the MTK. The input source reads this data and converts it to MTKTraces. The same procedure could be done for any other hardware.

Delegate methods increase the customizability of the input source.

We introduced two delegate methods in this input source, *didReceiveNewData:* and *didAddNewEntryToTrace:basedOnReceivedData:*. The first one allows to modify any data received by the input source, before it is used to update traces. In case this method is implemented by the delegate, the received data does not need to be formatted in the specified way, except that it needs to be a JSON object. In the call the delegate is then responsible to format the data in the specified format. The second method is called after each trace that was updated. It allows developers to modify the given trace after the input source processed its input data.

It is possible to simply add custom subclasses of MTKInputSource.

Custom input. The MTK allows the implementation of new input sources. To create a custom input source one has to subclass MTKInputsource and implement the *update-WithTime:* callback. An initialized object of the new class can then be added at runtime to the input sources in the MTKTable. After this step the input source is part of the call for trace collection in each frame.

A general Windows client would reduce the overhead.

We presented several options to connect input hardware to the MTK, but we have some improvements left for future work. Most of the the available multi-touch hardware is probably compatible with Windows and therefore can be used to generate Windows touch events. A future version of the MTK could include a client for Windows that collects all Windows touch events and sends it via network to the TUIO or JSON input source. This would ease the connection of new input hardware to the MTK.

We explained traces and input sources, which are responsible for the update of traces. This transformation of input data to updates for traces is the first step of the initialization of the touch processing. We will now continue with

the other steps.

3.2.3 Initialization of Touch Processing

As mentioned earlier, the touch processing of the MTK is split into two parts. The first one is the initialization. It is performed at the beginning of each frame by the active scene. It consists of eight steps which we will discuss in this section. The second part is the processing itself, which is performed by each node in the currently active scene graph after the initialization process. The initialization consists of the following eight steps:

The initialization of touch processing has eight steps.

1. Update all MTKTraces
2. Call *preProcess:* of global delegate
3. Update cursors
4. Update tangibles
5. Update global gesture recognizer
6. Distribute traces to SKNodes in scene
7. Call *postProcess:* of global delegate
8. Start recursive scene processing

Update all MTKTraces. The first step is the transformation of all input data to traces. Each input source is responsible for updating its traces, depending on the data it received from its input hardware. This transformation step was explained in the last two sections in the description of the classes MTKTrace and MTKInputSource. The resulting array of active traces is then used in all of the following steps. In each of the steps traces can be added, removed or modified.

The first step was already explained.

The global delegate allows to manipulate traces before any processing started.

Global delegate *preProcess*: call. The second step is the *preProcess*: call of the global delegate. The global delegate is a delegate developers can set application wide, it is therefore independent from the currently active content. It is called directly after the update of all traces, which allows for manipulation of traces before the actual processing starts. One example would be the transformation of the traces' positions depending on their input source.

Each trace has one cursor.

Update cursors. The next step is updating the cursor for each of the traces. A cursor is the visualization of a touch point. It indicates the position, size and rotation of active touches. The MTK provides a standard cursor. It is possible to customize the standard cursor and to set individual cursors for each of the traces. In case the input source provided a size or orientation the MTK is able to change the cursors accordingly.

In future versions of the MTK cursors should be more customizable.

The MTK does not allow to add any rules that for example specify that traces from one input source may always have a special cursor. It would be a useful feature for future versions. The MTK is currently able to apply such transformation using the *postProcess*: call of the global delegate or the scene delegate to make this transformation depending on the currently active scene.

Tangibles always work with all available traces.

Update tangibles. The next step is the processing of tangibles. In the MTK tangibles are part of the active scene. Their processing is part of the initialization of the touch processing since it is explicitly performed by the active scene. How the tangible processing works, which includes the recognition and recovery of all tangibles, will be discussed in Section 3.3. All traces that are used by tangibles will be removed from the set of traces available for the processing of the following steps.

Global gestures are performed on all traces.

Update global gesture recognizer. After the tangibles all global gesture recognizers are updated. Gesture recognizers work on a list of traces and look for certain patterns in

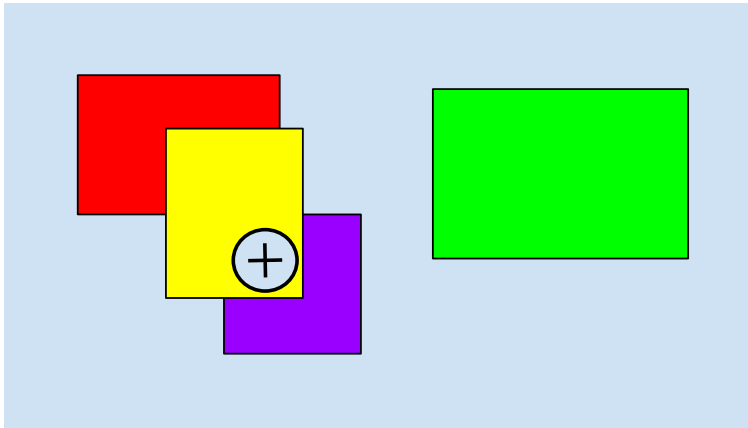


Figure 3.3: A sample scene (blue background) containing four nodes (differently coloured rectangles) and one trace cursor (circle with cross in it).

their movement. How they exactly work and which are available in the MTK is discussed in Section 3.4. Usually gesture recognizers are attached to a specific node and will perform their analysis on the node's traces. Global gesture recognizers however are not attached to a specific node. Instead they are processed at this point of the initialization of touch processing to allow application wide gestures. They are processed on all traces that are still available after the update of tangibles.

Trace distribution. The next step in the initialization of the touch processing is the distribution of new traces. In each frame are traces either new or already bound to a node in the scene. Those traces that are new and not used by the update of the tangibles or the global gestures need to be bound to one of the nodes in the scene.

We will use the example scene in Figure 3.4 to illustrate this distribution. In this scene are several rectangular nodes and a trace's cursor. In SpriteKit it is possible to generate this sample scene in different child parent relations, therefore we illustrated their relation in Figure 3.4. Parents are above their children and nodes further to the left are earlier in the children array of the parent and therefore rendered

The next step in touch processing is the trace distribution.

We use a sample scene to illustrate the situation.

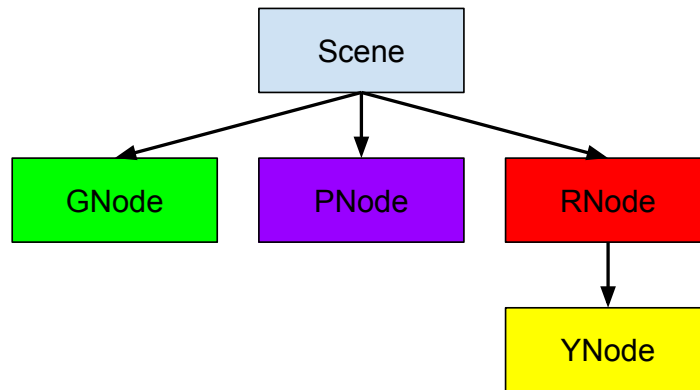


Figure 3.4: The hierarchy of the sample scene. Parents are above their children.

earlier. Therefore the rendering order is Scene, GNode, PNode, RNode, YNode, Cursor.

The trace distribution is based on a *nodesAtPoint:* call.

We now discuss to which node the trace is bound to in which case. The distribution is dependent on the hit test of SpriteKit. We use the function *nodesAtPoint:* of SpriteKit to determine which SKNodes are at the position of the trace.

The hit test returns nodes that we do not want.

The result of the function is an array of all SKNodes at the given position. Unfortunately this includes some nodes that we like to ignore in the trace distribution. Which nodes we will exclude will be discussed in the following paragraphs. We illustrated the decision process in Figure 3.5.

Hidden nodes can not receive traces.

The result array includes nodes that are marked as hidden. It might be an irritating fact for the user to be unable to touch a node, because an invisible node is blocking the touch, therefore all hidden nodes are removed. In case a developer explicitly wants an invisible area that can receive traces he may set the color of a node to clear color. In this case it will not get removed from the result array and will receive touches, but is invisible to the user.

Only touchable nodes will receive traces.

We added the property *isTouchable* to SpriteKit's SKNode using a category. It defines if a node is able to receive and process traces. SpriteKit does not know about this

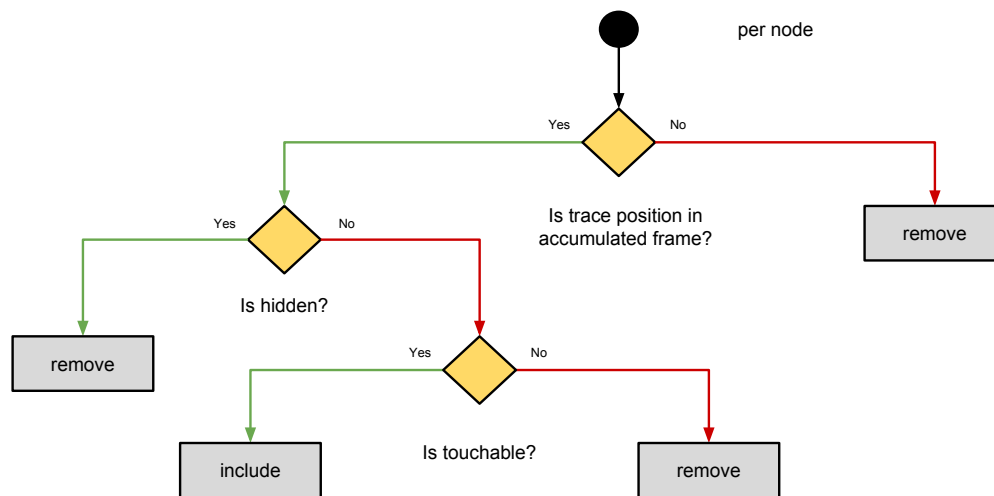


Figure 3.5: The decisions made for each node, to find the one that receives a new trace. The node which is still included and has the highest absolute zPosition will be chosen.

extension and will include untouchable nodes in the result of *nodesAtPoint*;, therefore we also remove untouchable nodes.

Additionally the nodes are not ordered as they are visible to the user, but in their appearance of the scene structure. Therefore we reorder the nodes to fit the order in which the user will see visible nodes.

We order the received node in the reverse draw order.

After the transformation of the result array is the first node in the array the node that will receive the trace. In our scenario this could be one of four nodes (RNode, YNode, PNode, Scene), which one it is depends on their type and setting. The GNode will never be in the returned array of *nodesAtPoint*;, because the trace's position is not in the area of the GNode, its bounding box or its accumulated frame. Therefore it can not receive the trace.

GNode will never receive the trace.

If YNode is touchable it will get the trace, independent from any other node being touchable or not. This is due to the fact that YNode is the top node and the trace is in its visible area.

YNode is likely to receive the trace.

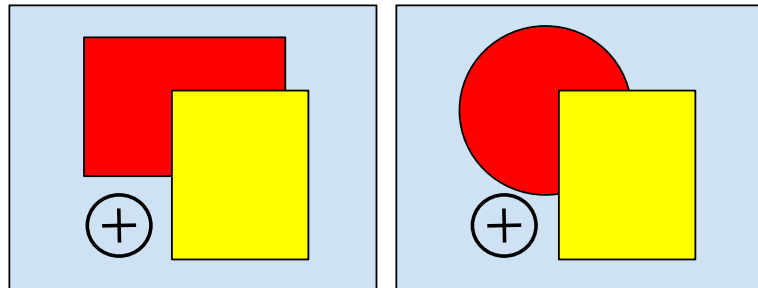


Figure 3.6: In the left scene the array returned by *nodesAtPoint:* called with the trace's position will return RNode. In the right scene, where RNode is a SKShapeNode instead of SKSpriteNode, it will not be included in the result array.

The scene may also receive the trace.

SpriteKits documentation is clear with what *nodesAtPoint:* should return.

SpriteKit is not consistent with the result of *nodesAtPoint:*.

In case YNode is not touchable it depends on the settings and types of the other nodes. If the RNode is also not touchable but PNode is, PNode will get the touch. Again because the trace is in PNodes visible area. If neither YNode, RNode nor PNode is touchable the trace is captured by the scene.

The one case that is special and causes some inconsistencies is the following: If YNode is not touchable, but RNode is. Apple's documentation of *nodesAtPoint:* states the following as return value: "An array of all SKNode objects in the subtree that intersect the point. If no nodes intersect the point, an empty array is returned.". The discussion then clarifies this with the following sentence: "A point is considered to be in a node if it lies inside the rectangle returned by the *calculateAccumulatedFrame:* method.". If we now follow the description of *calculateAccumulatedFrame:* we get the following statement by the documentation: "Calculates a rectangle in the parent's coordinate system that contains the content of the node and all of its descendants." [Apple Inc.].

We found one case in which the SpriteKit does not seem to work as documented. While it should return all nodes which accumulated frames contain the trace's position, this seems to be untrue for SKShapeNodes. We illustrated two scenes in Figure 3.6 where in both cases the calculated accumulated frame of RNode should contain the trace's position and therefore in both cases return the RNode. Unfor-

tunately is this not the case. If RNode is a SKSpriteNode, seen in the left sample scene, then the result array contains RNode, which is the behavior described in the documentation. But is the RNode a SKShapeNode, seen in the right scene, then it will not be contained in the result array. The trace will then be bound to the scene node.

In the current version of the MTK is the inconsistency still present, due to the fact that we assumed that the *nodesAtPoint:* is an performance optimized function. A custom hit test may have a huge performance impact and as long as developers are aware of the inconsistency can they avoid any problems. It is also possible that future versions of SpriteKit may fix this problem.

A custom *nodesAtPoint:* implementation may fix the problem.

Call global delegates *postProcess:*. The final step in the initialization of the touch processing is the call to the global delegate. As the *preProcess:* call, that allowed to manipulate for example traces before they are used for cursors, tangibles, global gestures, and the distribution in the scene, the *postProcess:* call is to manipulate any of these changes made. At this point the delegate may evaluate any expected results or change them.

The delegate can perform last changes.

All the explained steps are performed at the beginning of each frame. They are initialized by the currently active scene. After the initialization of touch processing will the actual processing start, which is a function call of the active scene, which will be recursively called on each of its children and their children. This call is discussed in the next section.

The active scene starts all initialization steps of the touch processing.

3.2.4 Recursive Touch Processing

Immediately after the initialization of the touch processing the active scene will continue with the recursive processing call. The processing is performed by each node in the scene graph, by executing the function *processTraceSetWithTimestamp:* which will return a set of traces. We implemented this call in a category on SKNode to achieve this processing

Each node in the scene will perform the same steps.

on each node in any given SpriteKit scene. All of the nodes in the scene and the scene itself will perform the following steps in this function call:

1. Call *preProcess:* of delegate
2. Call *preProcess:*
3. Call *processTraceSetWithTimestamp:* of all child nodes
4. Update gesture recognizers
5. Call *postProcess:*
6. Call *postProcess:* of delegate
7. Propagate traces

We continue with the given sample for further explanation.

We will stick with the example scene in Figure 3.3 and the parent-child relation of Figure 3.4 to discuss those steps in more detail. All of the nodes are touchable and not hidden. The active scene will start the whole process by calling its own *processTraceSetWithTimestamp:* method after the initialization process.

The delegate allows to make any changes before the processing of the node starts.

Call *preProcess:* of delegate. Each node in the scene can have a trace delegate. This delegate is called at the beginning and the ending of the processing. As in the initialization of the touch processing the delegate has the chance to manipulate the node or any bounded trace, to influence the processing.

The *preProcess:* method is abstract in SKNode.

Call *preProcess:* of node. The next step is the call of *preProcess:* in the node itself. The method is abstract in SKNode and can be overwritten by any subclass. Any processing is possible in the implementation of the method. This distribution throughout the scene is a paradigm change to SpriteKit. In SpriteKit the scene will get an update call, in which the scene will update anything in it as some kind of controller. Nodes in the scene will not get an update call in SpriteKit. This pre- and later the *postProcess* call change this by giving each node two update calls per frame.

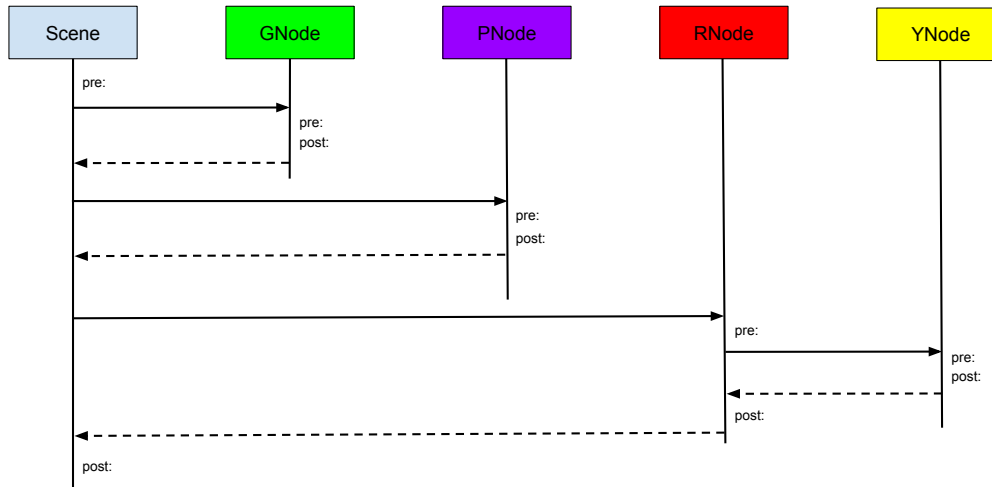


Figure 3.7: The call chain for the processing the the sample scene.

Call *processTraceSetWithTimestamp*: of all child nodes.

At this point of the *processTraceSetWithTimestamp*: call of the active scene will the scene call the *processTraceSetWithTimestamp*: method of all its children. The children will then perform the steps we discussed so far and also recursively call its children, as illustrated in Figure 3.7.

After the own *preProcess*: all children will start their processing.

Update gesture recognizers. At this step in the processing all traces that could possibly reach the node are collected. All children had the chance to propagate touches. The next step is to update the gesture recognizers. Each node can have a set of gesture recognizers. Which recognizers exist in the MTK and how they work is discussed in Section 3.4. Additionally, each node can have a set of gesture recognizers for standard transformations like rotation, translation, or scaling. These, like normal gesture recognizers, work on the nodes traces and are updated in this step, too. Which of those gesture recognizers are enabled can be set in the property *transformationConstraints* of the node.

The update of the gestures may or may not consume touches.

Call *postProcess*: of node. After these steps the *postProcess*: of the node is called. Similar to the *preProcess*: it is an

The *postProcess*: call is similar to the *preProcess*: call.

abstract method and can be used by subclasses to implement any custom behaviors. In contrast to the *preProcess:* call, all children already completed their processing here. It may also be that the set of active traces for the *postProcess:* includes more or other traces than in the *preProcess:*, since the children could have propagated traces to their parents.

The delegate can do final changes.

Call *postProcess:* of delegate. Now all steps of the touch processing are performed and the trace delegate of the node has the chance to do any changes to adjust the results.

Each node can propagate traces to its parent.

Propagate traces. As mentioned earlier, nodes are able to propagate touches to their parents. In this case the traces are still bound to the node, but they are given to the parent for this processing call so it might use it for processing. Usually all unused traces are propagated to the parent node, but each node can decide individually for each trace.

After the touch processing the normal SpriteKit frame loop in Figure 3.1 on page 15 will continue. No future calls are explicitly used by the MTK for standardized behaviors or procedures and can be used by developers without interference of the MTK. We will now continue by explaining the touch processing, tangible detection and gesture recognizer, which were previously left out.

3.3 Tangibles

The following explanation of tangibles, especially of PUCs and PERCs, is a summary of information found in [Voelker et al., 2013] and [Voelker et al., 2015].

Tangibles are physical objects that can be detected by touch surfaces. One big advantage of tangibles is their rich haptic feedback in comparison to touch surfaces. Applications can not take advantage of the human touch senses, which would be helpful for example in eyes free interaction, because the user always only feels the screen. In contrast tangibles can have many different forms and materials, providing the user with rich haptic experience [Voelker et al., 2013].

Tangibles improve touch interaction.

PUCs were the first tangibles that could be detected on an unmodified commercial capacitive touch surface. Earlier tangibles were detected using light based touch sensing technologies [Kaltenbrunner and Bencina, 2007], modified capacitive touch surfaces [Liang et al., 2014] or relied on the user's touch [Rekimoto, 2002].

Tangible recognition is already implemented.

In tangible detection the problem with commercially available touch sensing hardware is often that it is specialized to sense human fingers. Any other input like tangibles is filtered out. One approach to deal with this is to have the contact areas of the tangibles imitate the human touch.

Capacitive touch tables will not recognize normal tangibles.

Simply shaping the contact areas like touches will not force touch surfaces to recognize touches or tangibles. As described in [Voelker et al., 2013] capacitive touch surfaces recognize touches by using transparent electrodes located above the display panel. If the impact of the tangible's contact areas on the electromagnet field emitted by these electrodes is not bigger than the thresholds implemented by the hardware's filter mechanics, they will not be recognized as touches. Unfortunately tangibles have typically no connection to ground and not enough mass to have an impact on the detection field. Therefore Voelker et al. [2013] introduced PUCs, which use a trick to overcome this filter mechanics.

PUC and PERC can be detected without a user touching them.



Figure 3.8: The original PUC illustration by Voelker et al. [2013]

3.3.1 PUCs: Passive Untouched Capacitive Widgets

PUCs introduced
tangibles to
capacitive touch.

One type of tangibles supported by the MTK and working on unmodified capacitive touch surfaces are PUCs by Voelker et al. [2013]. One typical PUC is shown in Figure 3.8. PUCs are made of conductive material that contacts with the multi-touch surface at different positions. The conductive areas touching the surface, we will call them *marker*, are in a specific size that is similar to the touch of a human finger. The markers are connected to each other and arranged in a special pattern. The pattern will trick the multi-touch surface to recognize touches without needing a user to touch the tangible. As mentioned earlier capacitive multi-touch surfaces have a grid of electrodes to detect touch input. In the scanning process, there is only one electrode active at a time, the others are grounded. The pattern of the tangible ensures that in most cases only one of the markers is on an active electrode and the others are on grounded ones. Therefore the marker on the active electrode is grounded by the other two and will be recognized as a touch.

However some problems remain when using PUCs. The first one is that if the tangible stays long enough in one place the filter mechanics in most recognition hardware will remove the generated touches after around 5-30 seconds [Voelker et al., 2015]. This causes the problem that the software can not be sure if the tangible was lifted off the table or if it is still in place, but the touches were filtered. The second problem is the identification of the tangibles. The pattern for PUCs that will generate the most reliable touches is a circular marker pattern. Other possibilities exist, but are not that numerous, especially if one tries to distinguish different tangibles from each other. The only characteristic property of a PUC marker pattern is the distance between the different markers. To distinguish the tangibles the differences must be big enough to be discernable from small errors that occur when reading marker positions, which makes the identification of different tangibles very hard [Voelker et al., 2015]. It is also important that one of the distances between the markers is significantly different to the others to determine the rotation of the tangible. If all distances are not distinguishable then the rotation of the PUC at placement time is arbitrary.

PUCs had some issues that were addressed by PERCs.

PUCs and PERCs have to have at least three markers, but could also have more. Three is the minimum amount required to reliably track the position and rotation. They could have more than three markers, but it does not add more functionality. In the MTK we decided to fix the number of markers to be three.

In the MTK PUCs and PERCs have three markers.

Definition

PUCs basically consist of three conductive markers arranged in a fixed pattern. Their distances and relative angles to each other are constant, except for possible flickering of the touches generated by the hardware. The tangible creation scene, see Section 3.6.3, can be used to create the description for a PUC. The tangible has to generate three touches. The MTK will then read all distances and angles of the touch to each other and save them. This data is required to identify a tangible in the recognition process.

PUCs need three touches to be described.

The number of PUCs is limited.

The number of PUCs the MTK can distinguish is limited, since the distances between the markers need to be different enough to identify a PUC. The minimum distance between each marker depends on the hardware. Markers that are too close to each other may interfere each other by not generating touches, merging to one touch or generate heavily flickering touches. The maximum distance between the markers is limited by the maximum size of the touch surface. Often surfaces are very small, like an iPhone or iPad. Therefore placing more than one tangible at a time already reduces the maximum distance drastically.

Detection

We now explain step by step how the MTK will recognize PUCs. There are several special cases and problems we tackled in the implementation, which will be explained later on. We will now explain the optimal case.

The recognition process for PUCs is rather simple compared to PERCs.

We first have a look at the two recognition states a PUC can have: Recognized and NotRecognized. A PUC changes to the Recognized state if three traces were found that match the tangible's description. It will stay in this state until all of the traces are lost again. As long as two traces are active it will update position and rotation. With only one trace the rotation and position changes can not be calculated correctly and are therefore ignored. Is the tangible's state Recognized its digital representative is visible, else it is hidden. A PUC is in state NotRecognized if it has no active trace left. At the beginning of the recognition process all PUCs are in the state NotRecognized. The MTK will scan all available traces to find a triple of traces that fits the distances and angles saved in the tangible's description. If a matching triple is found its traces will be set as the active traces of the PUC and the state will be changed to Recognized.

Recovery

Touch points generated by tangibles may flicker.

After the recognition of the tangible its recovery functionality is used to improve its recognition. Our observations

showed that touches generated by markers are not as consistent as those of a human finger. Touches may disappear and then immediately or after a while reappear. This flickering is handled by the MTK. The MTK will immediately try to use new traces to recover a tangible in case one or two of its traces end. It will use the remaining moving traces of the tangible in combination with new ones to form triples. If a triple fits it will replace the ended traces with the new ones.

This process is faster than a new recognition of the tangible, since the MTK still knows at least an approximate position and rotation of the tangible and can filter the possible traces to those that are near to the tangible's last position. If only one trace is active then the tangible is not updating the position and rotation. Just searching for new triples around the last position of the tangible may cause problems, if the tangible still moves, but does not generate touches. This case is not a common case, since moving PUCs should generate touches [Voelker et al., 2013]. But to avoid problems in such situations the last trace is removed from the tangible if it continues moving away from the tangible's old position. In case no more trace of the tangible is active it is returned to the NotRecognized state and starts searching for traces in the whole scene.

It is possible that PUCs use traces that are not really part of the tangible, but were wrongly assigned. For example three finger touches that by accident matched the tangible's description. In these cases the tangible has the chance to detect this using the deformation check.

In each frame the tangible checks if the active traces still match the tangible's description. If this is not the case the PUC assumes that one or more of its traces are not correctly assigned. If the tangible has only one trace left the deformation check can not check any distances. The tangible can react in two different ways, in case two traces are remaining and a deformation is detected. Both traces are removed from the tangible, if they are both moving. In case only one is moving, the stationary one is removed. This is due to the assumption that this one is probably a Ghost Touch. For more information have a look in Section 3.3.1.

Local search will increase the performance.

Tangibles could have wrongly assigned active traces.

Deformation check helps from recovering of wrongly assigned traces.

Special Cases

The described recognition and recovery features may work reliable in most cases, but has still some issues that could not be solved by the software using the given hardware or that interfere with the normal processes. We will now discuss these cases.

Filtering of PUCs is a problem.

Filtering of stationary touches. In case the user does not touch or move the tangible for a longer period of time the touches beneath it will be filtered by most of the available hardware after around 5-30 seconds [Voelker et al., 2015]. The MTK can in this case not distinguish between a tangible that was lifted off the table and one which touches were filtered, but remained on the table. PUCs therefore just return to their NotRecognized state, which will cause the tangible to be invisible again.

It is possible that not all marker of the tangible will generate touches.

Too few traces at placement. Depending on the layout of the hardware scanlines and the size of the tangible, it is possible that not all three marker will generate touches when placed on the touch surface (see Figure 3.9) [Voelker et al., 2013, 2015]. Unfortunately this problem of PUCs is not solvable by the software. A PUC can not be recognized before not all of the three traces are active at the same time. Moving or touching the tangible will in most cases cause all three markers to generate touches, therefore this case is rare.

Ghost Touches are not distinguishable by software.

Ghost Touches. The input hardware we used did sometimes wrongly detect touches. Moving touches generated from tangible markers stopped moving and remained for several seconds before disappearing. We are not sure how and why this is happening. The hardware will still send updates and therefore we did not find a way to distinguish those touches from normal ones. We called those touches Ghost Touches. But since this case is very rare we expected that it is more likely that the tangible has just one trace left while being stationary. Additionally we could not observe

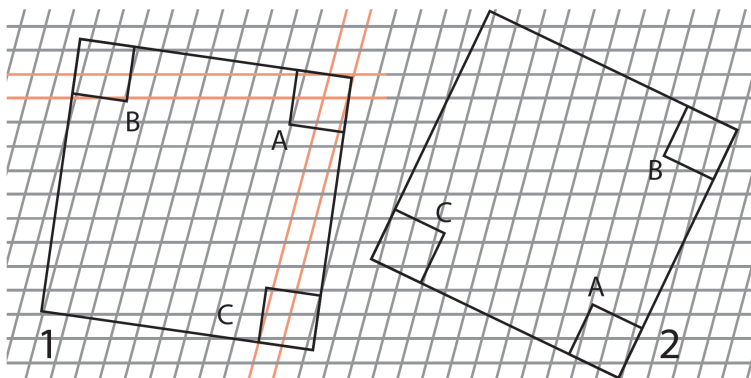


Figure 3.9: This is the original illustration of the scanline problem by Voelker et al. [2015]. In (1) B and C will generate touches, but A will not. In (2) all markers will generate touches.

a situation in which it happened with more than one trace at a time. We assume that this phenomenon is caused due to some problems in the filter mechanics of the hardware.

It is possible that such a Ghost Touch is used by a tangible. In this case the tangible has the chance to detect this using the deformation check. If the tangible is moved the Ghost Touch will remain at its position and the distances between other traces and the trace of the Ghost Touch will not be correct any longer. In this case the wrong trace can be identified as the stationary one and it will be freed from use with the tangible.

One case the deformation check can not resolve is if the last trace of a tangible uses a Ghost Touch. While PERCs have the chance to check if the tangible is still at the expected position, PUCs do not have this opportunity. Luckily the trace will disappear after a while. We could just free the last trace and hide the tangible. But since this case is very rare we expected that it is more likely that the tangible has just one trace left while being stationary.

More than one option. Another big problem of PUCs is that the software can not decide which set of traces it

PUCs may accidentally use Ghost Touches.

Ghost Touches are still a problem.

More than one trace triple forces the MTK to guess.

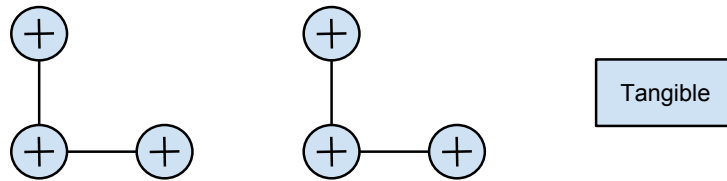


Figure 3.10: Two trace triplets, which match the description of the tangible arise the question which one the correct one is.

should use if more than one option is available, as seen in Figure 3.10. The MTK is forced to guess. A wrongly used triplet can be detected by the deformation check, if touches change their relation position to each other. Nevertheless is this a mayor flaw, since this causes the problem that some randomly placed fingers can be recognized as tangible, which is annoying for users. The frequency of this problem increases with the amount of patterns saved as tangibles.

Using the origin of traces reduces the options.

We gathered several ideas to improve the recognition process in such a case. Currently all traces are handled equally and independent from the input source. This could lead to the fact that a tangible consists of traces that came from more than one input source. We can think of specific tangibles where this could be possible, but in our current setup and use of tangibles it is not possible. A tangible is only placed on one input source, all used traces should therefore be from the same hardware. By filtering this it could reduce the number of possible traces fitting for a tangible.

Dynamical thresholds would be an important improvement.

One of the most promising solutions is probably the adjustment of thresholds. How far each of the traces jitters heavily depends on the input hardware and tangible. The MTK could use filter and learning algorithms to dynamically adjust the thresholds for each tangible and input source. In doing so the number of fitting possibilities for each tangible could be reduced.

An analysis of the traces histories may be interesting.

Additionally an analysis of the previous positions of each trace could filter unfitting traces. Tangibles are static

in their form, therefore the distances of their generated touches are constant, except for some minor flickering. A triple that is used for tangible detection should in all previous frames also fit the tangible's description. In the current recognition process only the position in the current frame is considered. If for example a user moves around with three or more fingers it may happen that they will fit a tangible's description in some frames. The current implementation will then assign a tangible to these touches. Future versions could avoid this by analyzing their previous positions.

The current MTK version does not consider such an analysis due to time restrictions. For such a method it is important to have a reliable algorithm that takes into account that it is possible that the distances of the traces change if a user touches a tangible and that their distances may be different when moving. Therefore it would be important to use dynamical thresholds in combination with this technique. Due to the fact that most of the problem can be solved using PERCs we did not investigate in this direction. It is still an important improvement since the reliability of the PERCs detection could also be improved using such techniques.

These improvements are important for future MTK versions.

The second kind of tangibles recognized by the MTK are PERCs. PERCs solve several of the listed problems. We will now explain PERCs and how the MTK will recognize them.

3.3.2 PERCs: Persistently Trackable Tangibles on Capacitive Multi-Touch Displays

PERCs are an improvement of PUCs introduced by Voelker et al. [2015]. PERCs consist of a marker set similar to PUCs, but have additional active hardware. The version presented by Voelker et al. [2015] can be seen in Figure 3.11.

PERCs are improved PUCs.

The recognition of PUCs is done only via touches generated by the tangible's markers. PERCs extend this by using the new hardware to actively send information about their state to the MTK. The bluetooth chip will connect with the computer and send every update of the surface and light

The three main components are the bluetooth chip, the surface sensor and the light sensor.

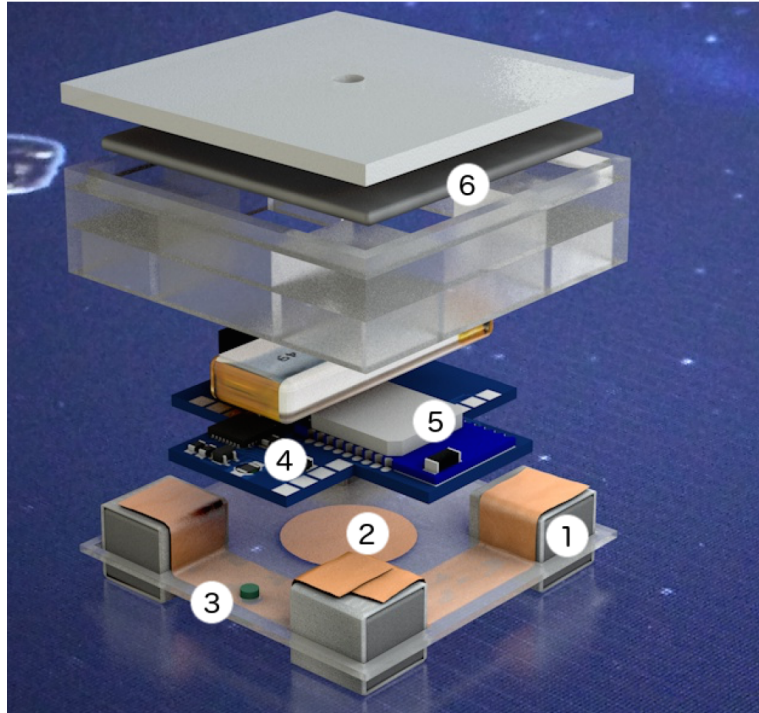


Figure 3.11: The original PERC illustration by Voelker et al. [2015], showing the six main components: (1) marker pattern, (2) field sensor, (3) light sensor, (4) micro controller, (5) Bluetooth element, and (6) lead plate.

sensor. The surface sensor can detect whether the tangible is on the surface. It searches for the signal a capacitive touch surface emits when scanning for touches. It will send the state `OnSurface` in case the sensor receives the signal and `OffSurface` else. The light sensor is measuring the brightness beneath the tangible. Its state can either be `White`, if the brightness is high enough, or `Black` in all other cases.

PERCs solve problems of PUCs.

With these new components the MTK is able to recognize five different states of PERCs, as seen in Figure 3.12. This solves some problems that occurred when detecting PUCs. The first one is the distinction between a lifted and a filtered tangible. If the tangible's touches were filtered the surface sensor will still sense the signal emitted by the multi-touch surface. The second one is the identification of different tangibles. Every bluetooth chip has its own unique iden-

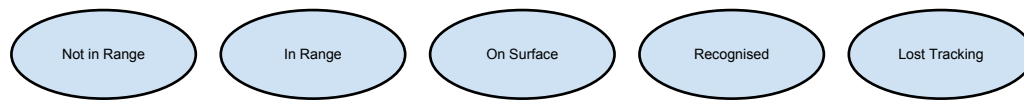


Figure 3.12: All states in which PERCs can be.

tifier, the MTK is therefore always certain which tangible is placed onto the surface. A distinction via the distances between markers like it is done in PUCs is not necessary.

We will now discuss the recognition and recovery process in more detail, to better understand how exactly PERCs work.

Definition

The description of PERCs is very similar to PUCs, since PERCs are basically PUCs with extra hardware. They need to save all information that were already saved for PUCs. Additionally they require a bluetooth identifier of the tangible's bluetooth chip which is required for the bluetooth communication and an offset from the tangibles position that defines its light sensor location.

The description of PERCs is nearly the same as for PUCs.

With this information the MTK is able to recognize the tangible. The PERC's recognition process is based on the process used for PUCs, but is much more complex, since the number of states and possibilities is extended. We will start with one of the new recognition feature of PERCs, the light sensor check. Afterwards we present the actual detection process.

Light Sensor Check

Since the light sensor request is an important step in the recognition of PERCs, we first explain how it works and which problems can arise before explaining the recognition process. As already mentioned the light sensor does know only two states, White or Black, which represent if the area

Light sensor knows two states: White and Black.

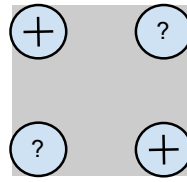


Figure 3.13: Illustration of problem arising when the hypotenuse is recognized. Two possible positions for the third trace. Both positions are beneath the tangible.

beneath the light sensor is bright or not. Each change in that value will be sent via bluetooth to the MTK.

The light sensor check is very simple.

The light sensor check works in the following way. The MTK will read the last sent light sensor value of the PERC. Based on this information it determines the color, that needs to be shown beneath the light sensor to change its state. The MTK will place a shape with this color beneath the position where it expects the light sensor to be. This expected position is determined using a triple of traces or a position and rotation, plus the light sensor offset saved for each PERC. The MTK is sure that the tangible is at the calculated position, if the light sensor of the tangible sends a color change of its light sensor.

Having only two touches may be enough to recognize the tangible.

While PUCs could only be recognized if all three traces are active at the same time, it is possible to recognize PERCs with two traces. With two traces and the tangible's description the options for the light sensor's position are limited. The software can start a light sensor request for each of the positions and determine the correct one.

A limited number of options is important.

The number of possible light sensor positions increases if the distances between the tangible's marker are not distinguishable. Our standard tangible, seen in Figure 3.11, for example has markers forming a right triangle with two similar legs and one hypotenuse. Two touches that appear and fit the tangible's description then have one of two distances. If the distance between the two touches is similar to the length of the tangible's hypotenuse, two possible positions for the last touch exist, as illustrated in Figure 3.13.

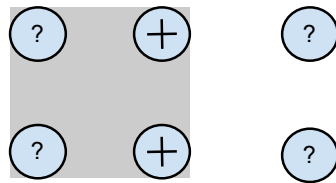


Figure 3.14: Illustration of the problem arising if only the leg of the triangle pattern is recognized. Four possible positions for the third one. Some are not beneath the tangible.

Is the distance between the two traces equal to a leg, the number of options increases to four, as illustrated in Figure 3.14. Checking one option after the other is still possible, but will cause a delay. The color change of the light sensor in most cases is received only a few frames after the change, as long as the light sensor check is testing the correct position. Is the position incorrect the check will last for a much longer period of time to ensure that it is the incorrect position and not a delayed response. Therefore, the MTK does only use the light sensor to recognize a tangible with two traces if they are the hypotenuse.

As already mentioned, the MTK was designed to work especially well with the standard PERCs, therefore it would be advantageous to only test the hypotenuse because the user may not see the light sensor shape. If the two touches are the hypotenuse we can check both positions without the threat that the user may notice this process, since all possible positions are below the tangible's area, as seen in Figure 3.13. If the possibilities increase to four, as seen in Figure 3.14, some of the tested areas are outside of the tangible and may irritate the user.

If the MTK is used with non standard PERCs this process is flawed. Using the hypotenuse does then not guarantee that the light sensor shape is still covered by the tangible. Additionally it is then possible that the calculated hypotenuse is not unique, since the pattern does not have to be a right triangle. In future versions the MTK could adapt to the situation that the saved pattern is different from the standard PERC and change its behavior.

Checking two positions is much faster than four positions.

The light sensor shape should not be visible to the user.

Future MTK versions should adapt to different tangible shapes.

Moving triples can not be checked by the light sensor.

Unaffected of the size and shape of the tangible the light sensor has the problem that it can not check moving tangibles. The recognition of traces by the hardware, receiving the data via network and showing the shape takes some time. If a triple of traces is moving and needs to be checked the shape that is used for checking will not be beneath the light sensor. Therefore the result may be corrupted. It is possible to use prediction algorithms to guess where the position will be to allow a light sensor check in these cases, but the current version of the MTK does not have such strategies. The recognition process implemented in the MTK avoids the test of any moving trace triple.

The light sensor might be checking the wrong position.

Another problem when using the light sensor is that if the light sensor is checking at an incorrect position, it is possible that the scene beneath the actual light sensor position changes and causes the light sensor to send a state change. Therefore the check will confirm the wrong position and the MTK will recognize the tangible with the wrong touches. To avoid this situation all other possible positions also show a shape, but in the color the light sensor currently recognizes, which then will not cause a change.

The PERCs hardware is in a prototype state.

A problem that is caused by the hardware of PERC itself is the low reliability of the light sensor. Due to the experimental hardware a light sensor request is not guaranteed to give the correct results. Sometimes it happens that the light sensor state is changed due to light condition changes around the setup. It is also possible that the light sensor did not change its state, because the screen was not bright enough to trigger a change. These problems can not be addressed by the MTK, but can be resolved by improving the hardware of PERCs. The result of these issues is that the MTK does not rely on the light sensor results. Some checks are done more often, for example are triples tested regularly if no other option is available.

Detection

The recognition of PERCs starts with a bluetooth connection.

We now explain step by step how the MTK is able to recognize PERCs. The first step of the PERC's recognition pro-

cess is that the MTK connects to the bluetooth chip of the tangible. The state of the tangible changes after this process from its initial NotInRange to the InRange state.

When the tangible is placed on a touch surface the tangible's surface sensor detects this and will send this information via the bluetooth module to the MTK. The MTK receives the information and changes the tangible's state to OnSurface. The process described so far is exclusive to PUCs, PUCs are oblivious to any of these state changes.

Voelker et al. [2015] reported that the tangible's touches are recognized in 99% of the cases within a time window of 150 ms around the bluetooth signal. The MTK therefore prefers traces from within this time interval. This interval is hardware dependent and future versions of the MTK should allow to set this for each hardware. In the current implementation a time window of 200ms is set.

The MTK will start scanning for the tangibles after it received the OnSurface signal. So the first thing the MTK does is to search for all traces that began 100ms before the signal and start searching for triples that fit the tangibles description. If any of the triples do fit the description it will use them to recognize the tangible. As discussed in PUCs this can cause problems if more than one triple fits. Therefore the tangible that is recognized via such a guess will be saved until 100 ms after the OnSurface signal was received. The tangible stays recognized with the given traces, if not more than one triple is available after 100ms and none of the used traces conflict with any other PERC that needs to be recognized. We illustrated a rough decision tree that is done for each tangible in each frame in Figure 3.15.

This guessing of a triple is fast, but not guaranteed to be correct. It is for example possible that the tangible generates less than three traces at placement and the used triple does not consist of traces belonging to the tangible. Voelker et al. [2015] reported that this is caused due to the alignment of the scanlines, as we explained in Section 3.3.1. In the used hardware it appeared in four angles. One may notice that the reported issues are in the worst case situation for PERCs, where no human is touching the tangibles and

PERCs will send a notification about their placement on the touch surface.

Touches generated by the tangible will be received by the MTK in a time window of 150 ms.

Traces that began in the time window are likely to be part of the tangible.

The MTK is guessing, which could cause problems.

they are not moved at all. Since it is very likely that the triple is the placed tangible, we decided to guess instead of verifying the triple with a light sensor check. In doing so we improve the detection speed and allow the detection of a single moving tangible, but reduce the reliability.

Light sensor checks will determine if trace triples are correct.

The tangible will use the light sensor to identify which of the triples to use if at the end of the time window more than one triple is available. At the end of the window the MTK will save all triples that are available and test each possible position. If a triple is moving the tangible detection will wait for it to be stationary before testing it. If a stationary triple is available it will be checked. If the check is positive the tangible is detected with this triple, if not it will be removed from the set of remaining possible triples. If only one possibility is left, the MTK will use the leftover triple to recognize the tangible without a light sensor check. It is very likely that the last option is the tangible. Therefore we use the triple without additional verification, which allows the recognition of the tangible, even if it is still moving.

The MTK will maintain all possible triples.

In each frame the MTK checks if the triples still fit the tangible's description and if the traces are still active to maintain a limited set of correct possible triples for the tangible. A triple is removed from the set, if it does not fit the description any longer. In case some of the triple's traces end successors are searched to replace them. The triple will be removed if no successor is found. At this the number of possible options will be reduced to find the correct one faster.

The implementation could cause wrongly detected tangibles.

In two situations this implementation could wrongly detect a tangible. The first one is the same as in PUCs. In rare cases it is possible that none of the checked triples is the correct one, since the tangible only generated one or two traces. As already explained, we assume that this case is rare so we allow the MTK the guess.

Moving trace triples can not be checked.

The other problem is that the light sensor can not control moving triples. To be faster and to allow the recognition of a moving triple we continue the evaluation of triples that are stationary while others are moving. It is possible that, while one of the stationary triples is checked, the light sen-

sensor beneath the moving triple will send a state change. In this case the checked stationary triple instead of the moving one will be used for recognition, instead of the moving one. We could wait until all triples stopped moving, but this would cause the whole process to be delayed for an unknown amount of time. The currently implemented solution is our trade off between fast and responsive detection on one hand and the number of possible wrongly detected tangibles on the other. A future MTK version in combination with improved PERC hardware may be able to create a better solution.

A tangible reaches the last recognition step in case the MTK could not find a fitting trace triple in the time window, which did not conflict with other tangibles. In this phase the MTK searches for fitting trace triples and tuples. As already mentioned, a PERC can also be detected using only two traces. The MTK will check each of the possibilities with the light sensor as described before. The difference is that the MTK will only recognize the tangible if a set of traces was confirmed by the light sensor, not if only one is left. In this state the guessing is not allowed anymore, since the chances to use a wrong set are increased.

This behavior tries to ensure that tangibles in this phase will only be detected if the process is sure that the traces can be associated with the tangible. It is important to note that the correctness is still not guaranteed. As already explained in the light sensor description it is not reliable enough to be completely sure that the confirmed triple is correct.

Does the user at any time in the process lift the tangible off the table the surface sensor will stop measuring the signal of the multi-touch surface and send this information to the MTK. The MTK knows that the tangible left the table and changes its state to InRange. The recognition process is stopped and any already guessed traces are removed from the tangible.

If no other method worked, the tangible requires a light sensor confirmed triple.

Wrongly detected tangibles are in this state also possible.

Lifting PERCs of surface will set the state back to InRange.

Recovery

PERCs have mostly the same recovery process as PUCs.

The recovery process of PERCs is mainly the same as in PUCs. If one or two traces are lost PERCs will use the same functionality as PUCs to recover the lost traces. The difference is that if all of the tangible's traces end, PERCs are able to know that they are still on the multi-touch surface, due to their surface sensor. Thereby the MTK can filter possible trace triples for those that consist of traces that are near the tangibles last position.

The stationary check allows to prove that the tangible did not move.

The MTK will use the light sensor to perform a stationary check to ensure that the tangible is not moved from its last known position. This check is a light sensor check using the last known position and rotation of the tangible. If the tangible is still in the same place it will respond and the tangible will be sure that the position is still correct, else the MTK will change the tangible's state back to OnSurface. This will hide the tangible and allow the recognition process to search for traces in the whole scene.

The unreliable light sensor causes us to check several times.

As already discussed, the light sensor is not fully reliable. The MTK changes the tangibles state after three failed stationary checks to reduce the cases in which a tangible is wrongly hidden again.

Special Cases

Many of the special cases and remaining problems are already mentioned in the recognition process, which was designed to tackle those. Therefore they are not listed again in this section.

In few cases PERCs generate only one touch.

Too few traces at placement. It is possible that PERCs generate less than three traces. The MTK has no chance to recognize the tangible if only one trace is generated by the tangible, but Voelker et al. [2015] reported that this happened in less than 3.2% of the cases, as long as the tangible is not touched by the user or moved.

Ghost Touches. The problem of using Ghost Touches to recognize or recover tangibles is still present, but PERCs, similar to PUCs, use the deformation check to recover from this situation. In the case that the last remaining trace is a Ghost Touch, where PUCs could not identify the difference, PERCs use their surface sensor and stationary check to make sure that the tangible is still in place.

Ghost Touches are no problem for PERCs.

3.3.3 Tangible Simulator

While developing a tangible application one may not always have a working touch surface and tangible at hand, which is why we created a tangible simulator. The simulator is based on the mouse input source described in Section 3.2.2. By emulating touch points and bluetooth signals the tangible will go through the normal process of tangible recognition.

The fake tangibles are created using generated touches and bluetooth signals.

The tangible simulator can be controlled with the following commands. By pressing, *Command + T*, the scene gets an overlay showing all available tangibles. Pressing *Number Key + Command* selects the available tangible with the pressed number. *0 + Command* selects a new tangible, which will be created and placed in the scene. Pressing *Command + Left Mouse Button* will create an OnSurface signal and create three traces that fit the tangible's description. Hereby does the MTK recognize the tangible at the clicked position. If the user presses *Command + Left Mouse Button* on an already existing tangible he is able to drag the tangible. Again not the actual tangible is modified, but the traces used to recognize the tangible. Pressing *Option + Left Mouse Button* on an existing tangible allows to rotate the tangible. *Right Mouse Button + Command* on an existing tangible will remove the traces of the tangible and send an OffSurface bluetooth signal, which will result in a removal of the tangible.

The simulator has different fixed commands.

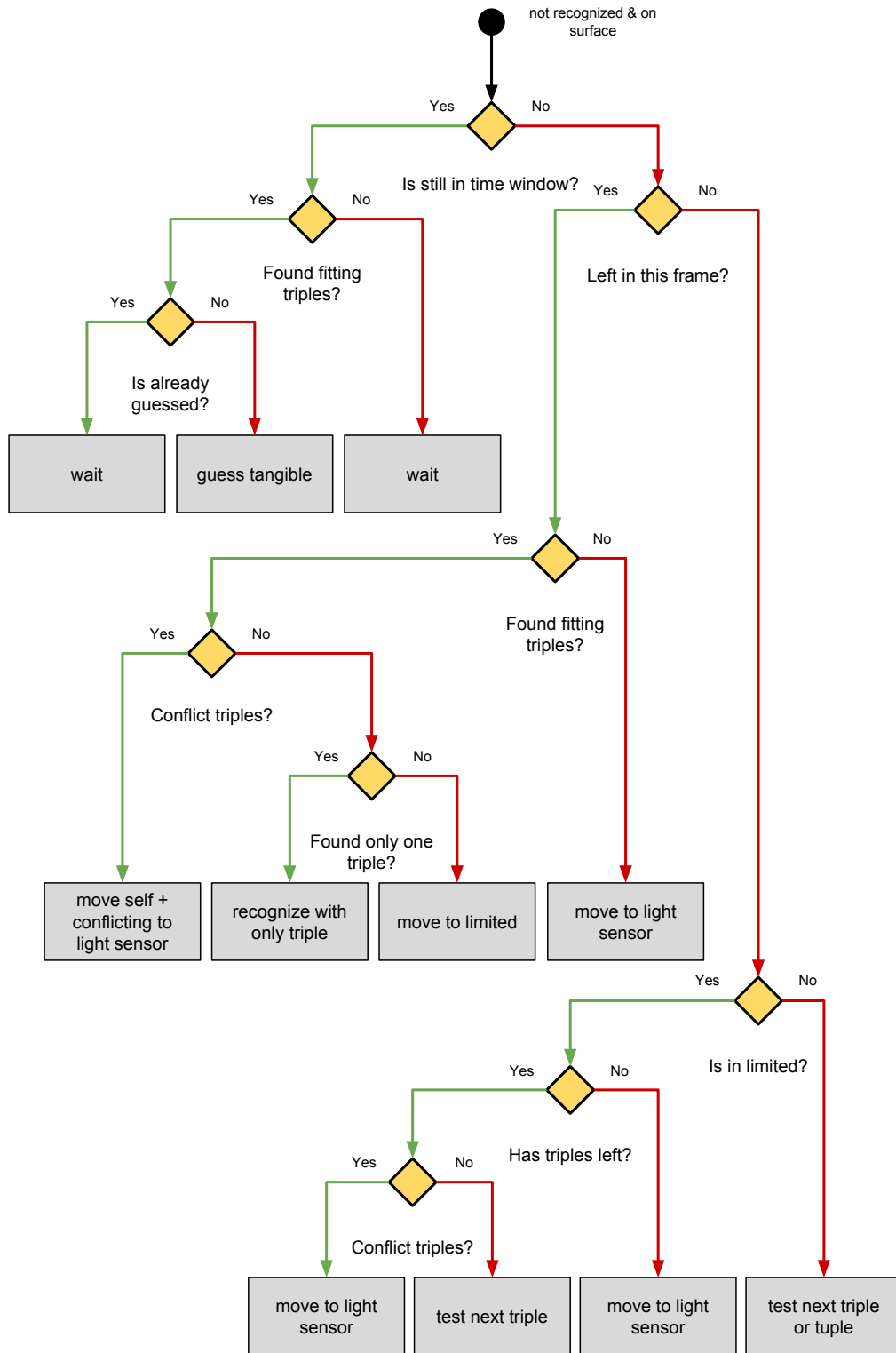


Figure 3.15: A decision tree roughly illustrating the MTK's decision for each tangible in each frame. Moving to limited means only using the triples of the time window, moving to light sensor uses all available triples and tuples that fit the description.

3.4 Gestures

Gestures are a common way to enrich the user's interaction possibilities with multi-touch surfaces and Kammer et al. [2010] mentioned them as an important part of multi-touch frameworks. Therefore we decided to include gesture recognizer into the MTK. In the MTK gesture recognizers work on a list of traces and look for certain patterns in their movement. Except for global gesture recognizers they are attached to a specific node and will perform their analysis on the node's traces.

Gestures are widely used in multi-touch applications.

The focus of this thesis was to implement the MTK's hardware independence and tangible detection. We therefore did not investigate in finding the perfect realization of gestures, but implemented a basic set which is similar to the set provided by Apple in iOS. This is used to reinforce the familiarity of Apple developers with the MTK. Additionally we added some custom made recognizers and possibilities for developers to create new ones.

Gesture recognizer in the MTK are rudimentary.

3.4.1 Standard Gestures

Based on the Apple gesture recognizer we implemented press, release, tap, hold, swipe, pan, rotate and pinch gesture recognizer. Additionally we implemented MoveIn and MoveOut.

We implemented ten standard gesture recognizer.

All gesture recognizer are subclasses of `MTKGestureRecognizer` and implement the function `processTraces:forNode:withTimestamp:`. This method gets a set of `MTKTraces` as input, processes them and sets a new state for the gesture recognizer. It is also possible that the recognizer uses all traces of the scene, which is done in the implementation of MoveIn and MoveOut. When changing states all subclasses have to follow specific state changes to ensure the correct behavior of the MTK. Which possible state changes exist depends on the type of gesture. We classified our gestures in two different sets like Apple did. We differentiate between discrete gestures, for example a

The recognizer will process traces and cause state changes.

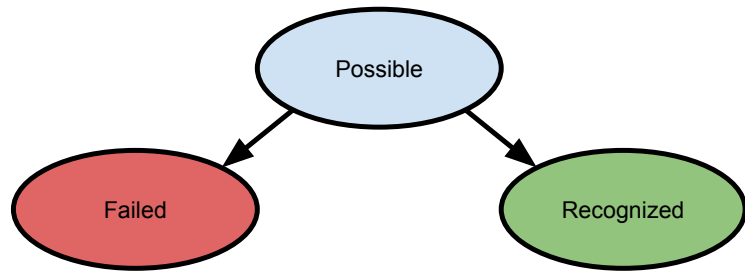


Figure 3.16: State graph for discrete gesture recognizer. This is very similar to the states Apple’s gesture recognizer can have.

tap gesture, that is recognized and immediately finished, and continuous gestures, for example pan, which are recognized and then change until they are finished.

Discrete gestures have three states.

Discrete gestures have three different states, as seen in Figure 3.16. The first state is the Possible state. This is the normal state in which the gesture is neither recognized nor failed. The other two states are Failed and Recognized. A gesture changes to Recognized if some of the given traces fit the gesture. For example a single release gesture checks if a MTKTrace exists that has the state End. A gesture is Failed if the given set of traces may not fit the recognizing process and it is not possible with these traces to recognize the gesture. For example the recognition of a right swipe is Failed if the set of traces include only one trace which is moving from right to left.

Continuous gesture recognizer have six states.

The second set is the continuous gesture recognizer. These recognizers have six different states, as seen in Figure 3.17. The states Possible and Failed are similar to discrete recognizer. If the continuous recognizer identifies that the given traces form the beginning of the gesture the state is set to Began. This is for example the case if a pan gesture recognizer received a trace which moved a minimum distance. After the gesture recognizer state changed to Began, it will send updates with the state Changed for each frame as long as the gesture is not canceled or recognized. From this state the recognizer may change to Recognized or Canceled. The Recognized state represents a correctly ended gesture. Can-

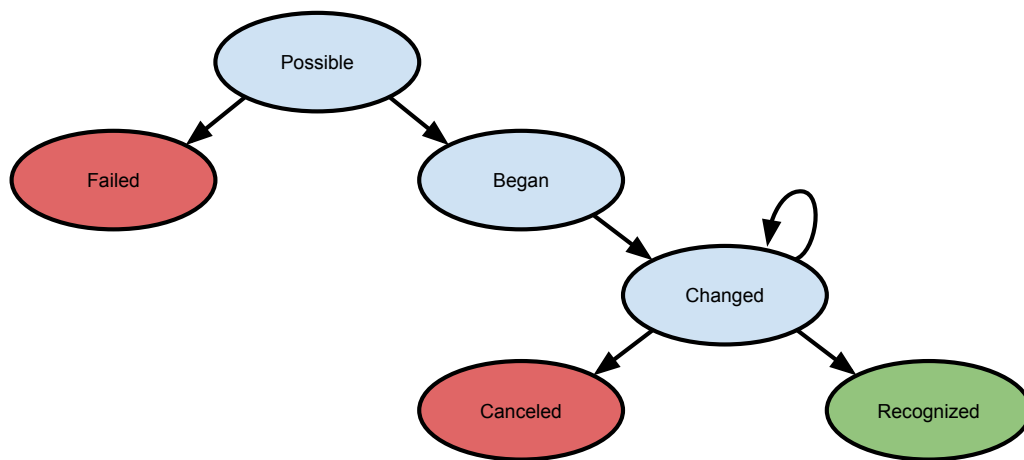


Figure 3.17: States of continuous gesture recognizer. The states are very similar to those of Apple’s gesture recognizer.

celed is the opposite state, showing that the gesture did not behave as expected and thereby rendering all previous Began and Changed states to invalid. In case of a pan gesture recognizer that only recognizes a right pan, it is possible that the recognizer changes to Began and Changed after a trace moved some distance to the right, but when the trace changes its direction back to the left the gesture is not pan anymore and the recognizer will change to Canceled.

Kammer et al. [2010] mentioned that it is important for frameworks to provide information about touches to ease the creation of gesture recognizers. Our MTKTrace implementation provided us with exactly those required information. Gestures get all bounded traces of the node they are bound to and can access all previous states of these traces directly via the given MTKTrace object. This way we had no problems with missing information for recognizers.

Some trace information are important for gesture recognizer.

Additionally, the recognizer can access other traces via the MTKTable and the current scene. That was for example used in the implementation of the MoveIn and MoveOut gesture recognizers which check if a trace exists that moved in or out of the nodes area. Such traces are not bound to the node and will therefore not reach the recognizer within the normal processing, but can be accessed via MTKTable.

The MTKTable enables the access to all traces.

3.4.2 Custom Gesture Recognizer.

Different options exist in the MTK to create new gestures.

The second important part including gestures that Kammer et al. [2010] mentioned is the customizability of gesture recognizers. A developer using the MTK has different options to get the recognizers he wants to, action blocks, recognizer blocks, delegate and implementing a new gesture recognizer.

Several action blocks can be added for each state.

Action blocks. The first option to customize a recognizer are action blocks. These can be added to a gesture recognizer for each state. In these action blocks the developer may do whatever he desires. We used it for example in combination with the pan gesture recognizer to create a Drag recognizer. An action block is added to the Changed state of the pan gesture recognizer, in which the position of the attached node is modified. If the user touches a node that has a Drag recognizer attached and moves his finger, the pan recognizer will change its state to Began and each following frame will send the state Changed. In each Changed state update the action block is triggered and updates the position of the node accordingly to the traces position.

Recognizer blocks can change the conditions for the Recognized state.

Recognizer blocks. Recognizer blocks are performed if the gesture recognizer will change its state to Recognized. Each block returns a boolean value. If all of these are True, the recognizer will change its state to Recognized, otherwise it will stay in its current state. This is for example used in the gesture recognizer released in node, which is a combination of the release recognizer and the a recognizer block. Normally a release recognizer will change its state to Recognized if a trace ended. The release in node recognizer will perform the added recognizer block before the change. The block checks if the trace's last position was in the node, if not the recognizer will not change its state.

A delegate is an alternative to block.

Delegate. The MTK allows to use a delegate instead of blocks. The delegate is called after each state change and

may alter whatever it likes while the recognizer is working normally. Like this more complex changes can be made.

Subclass MTKGestureRecognizer. The last option to implement a gesture recognizer is by implementing a new subclass. To do this a developer has to subclass MTKGestureRecognizer and implement *processTraces:forNode:withTimestamp:*. As already mentioned, the states should be changed like we discussed.

These subclass objects are handled as any other gesture recognizer. By adding them to any node using *addGestureRecognizer:* the processing will automatically call them. The developer should follow the explained state changes, but apart from that nothing more is required to make the MTK work with custom gesture recognizers. Additionally the action block, recognizer block and delegate handling is fully implemented in the MTKGestureRecognizer class and is therefore already available for any new gesture recognizer class.

Subclassing is the final option when creating custom recognizer.

All new recognizer are used in the same way as existing ones.

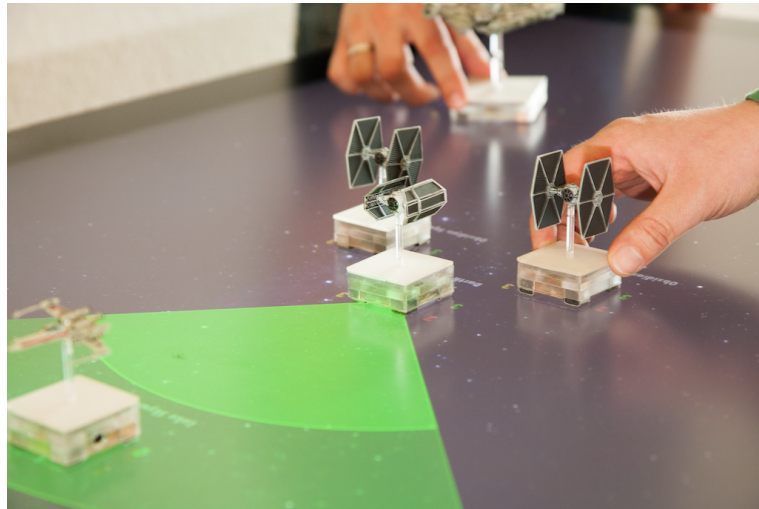


Figure 3.18: A sample SpriteKit scene of a Star Wars game that uses PERCs and multi-touch as input. This image is taken from Voelker et al. [2015].

3.5 Visualization Support

Figure 4.1 by Kammer et al. [2010] shows the diagram of features for multi-touch frameworks. The topmost part in this figure is the visualization support. This includes all possibilities and support the framework offers to create visual output.

SpriteKit offers many visualization features.

SpriteKit. Due to the fact that we based the whole MTK on SpriteKit, developers can use any functionality provided by SpriteKit. SpriteKit is designed to enable developers to create rich 2D applications, which includes animations, sparkle effects, physics and more. One sample application can be seen in Figure 3.18, in which SpriteKit and PERCs are used to create the basic gameplay of a Star Wars tabletop game.

SpriteKit can be extended to use 3D scenes.

SpriteKit can use SceneKit to be extended by 3D elements. Apple designed SpriteKit to easily include SceneKit elements, which are with small adjustments able to process MTKTraces, too.

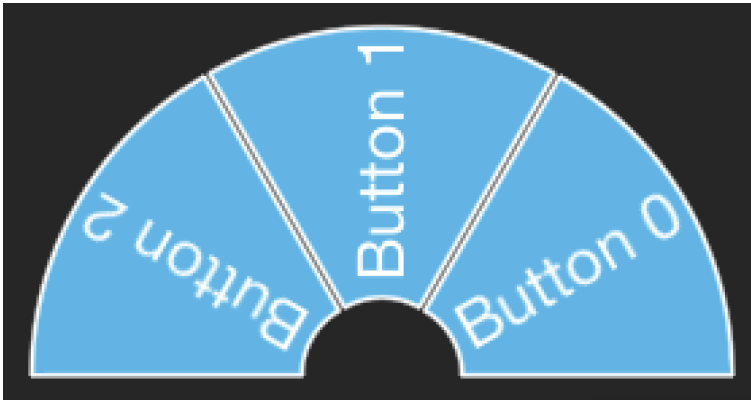


Figure 3.19: This is the MTKPieMenu. It can be set with different numbers of buttons and angles in which the buttons will be placed.

GUI Elements. When developing applications with SpriteKit developers are usually able to use Apple's standard UI elements. Unfortunately these elements work with UITouches or NSEvents which can not be easily generated by us. We could therefore not integrate the use of standard Apple UI elements in the MTK. To partly compensate this lack of UI elements we implemented our own set of controls. The number of elements in the MTK and their functionality is very limited, due to the fact that this was not our main requirement. Future MTK versions should increase the amount of available elements.

The elements we implemented are button, switch button, slider, rotary slider, grid view, list view, scroll view and drawing area. How they work and which properties they offer can be found in the documentation. It is important to note that most of the functionality is implemented in a way that allows developers to modify and customize the GUI elements, even if explicit properties or functions are not offered by the API. All GUI elements are build out of base types of SpriteKit and therefore all elements are part of the scene tree and can be easily accessed.

In future versions this part of the MTK needs improvements. The elements are very basic and sometimes cumbersome to use. One of the first improvements would be to

The MTK is not able to use Apple UI elements.

GUI elements can be modified even if the specific properties and functions are not explicitly included.

Still many options left for future work.

add a full digital keyboard and a text box. The possibility to set a style for all GUI elements centrally would also be a nice feature. Another improvement would be a general event system which is standardized for all GUI elements and that may also include gesture events, which do not exist in the MTK so far.

3.6 TouchControlCenter

The MTK offers a variety of options that can be configured, like different output windows, different input sources, tangibles and more. All these settings are saved in a file which is loaded at each start of the MTK. This file may be altered by developers to customize the MTK without the need of source code. To ease the process of setting these options we developed the TouchControlCenter, short *TCC*. We provide a MacOS and an iOS Version. The TCC has the goal to offer a UI for most of the MTK's configurations.

It is important to note that the TCC does not cover all possible settings, neither in the MacOS nor the iOS version. We implemented a basic version that helps setting the most common information. For future work an improved and extended version of both the MacOS and the iOS TCC are important.

3.6.1 TCC in MacOS

The MacOS version is a standalone application based on Apple's UI components. It is divided in three segments: General settings, Application settings and Tangible settings.

General Settings. In the General settings area one can define different profiles. Each profile defines the size of the scene and other settings, like size and visibility of the cursors. Profiles also have a list of Viewports. Each Viewport is defined by two rectangles. The first one is the area in the scene which is visible in the Viewport and the other one is the position and size of the window in which the content of the Viewport is shown. A profile also includes several input sources. Each input source offers a different set of options. This part is not fully implemented yet and will need a much more general approach to work with unknown input sources in the future.

The TCC allows the customization of the MTK without source code.

The current version of the TCC is not covering all settings.

General settings include input sources and views.

In the application settings one may set the active profile and loaded scenes per application as well as all added tangibles per scene.

Application Settings. The Application settings allow to set the active profile, the starting scene and the loaded scenes per application. For each of the loaded scenes it can be defined which tangibles will be available in the scene. The list of tangibles is filled with all tangibles that were defined in the Tangible settings. The list of applications and available scenes is automatically filled. Every started application on the machine using the MTK as well as any MTKScene class included in the started application is added to the TCC lists.

In the tangible settings tangibles can be defined.

Tangible Settings. The tangible settings show all defined tangibles and recognized bluetooth devices. New tangibles can be created using the MTKTangibleCreationScene, which can be opened from the Tangible settings. Read Section 3.6.3 for further information.

3.6.2 TCC in iOS

Many unnecessary settings were removed from the iOS version.

The iOS version of the TCC is much smaller. So far we did not test how to add other input sources than the UITouch input source to iOS applications. Therefore all of the settings for the input sources are removed. iOS allows only one full screen window per application. This is why we decided to allow only one scene size for MTK applications started in iOS, which is full screen. Any Viewport or scene related settings are therefore removed. What is still included is the start scene and which tangibles will be included in this scene, as well as the definition of new tangibles and the deletion of old ones.

Few settings are available in the iOS TCC.

The current version can be seen in Figure 3.20. In the left column are all available scenes. The selected one is the one that will be used as start scene. In the other column are all tangibles which are defined. The selected one will be added to the starting scene. By pressing delete tangibles the TCC will switch in deletion mode. If one then selects tangibles they will be deleted if the delete tangibles button is pressed again. If one presses the config tangibles button it will switch to the MTKTangibleCreationScene. The start

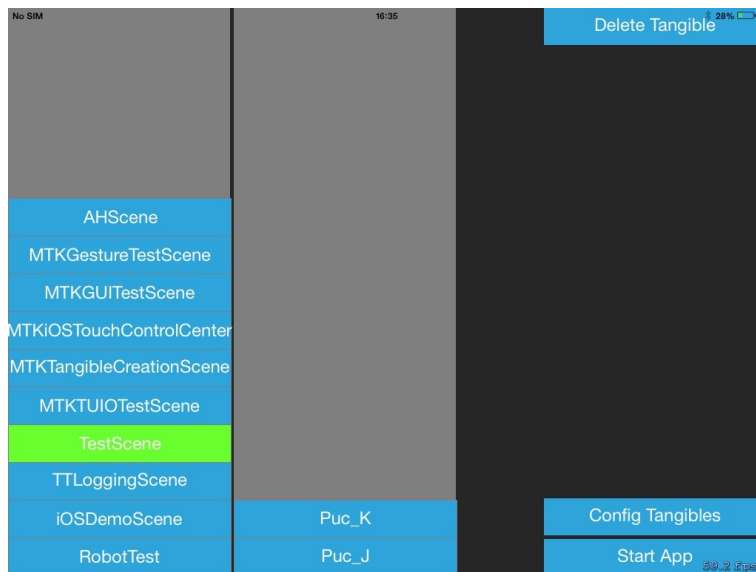


Figure 3.20: The current version of the TCC for iOS.

app button will start the selected scene.

The current implementation of the iOS MTK is very limited. Applications built with the MTK have more possible settings, which can not be set with the current TCC in iOS. Additionally this TCC for iOS is no standalone application, but is loaded at each start of the application in iOS. A better solution would be standalone application that changes the configuration file, which is then loaded by the MTK.

A new TCC for future MTK versions is required.

3.6.3 MTKTangibleCreationScene

The MTKTangibleCreationScene is used by the TCC in MacOS as well as in iOS. It allows to create and define new tangibles. The difference in functionality only concerns the Close button. In iOS it will return to the TCC, while in MacOS the close will close the window in which the MTKTangibleCreationScene was opened.

The WidgetCreationScene can define new tangibles.

The first thing to do when configuring a new tangible, is to decide if the tangible is a PUC or PERC. If any bluetooth

First step is to decide if PUC or PERC.

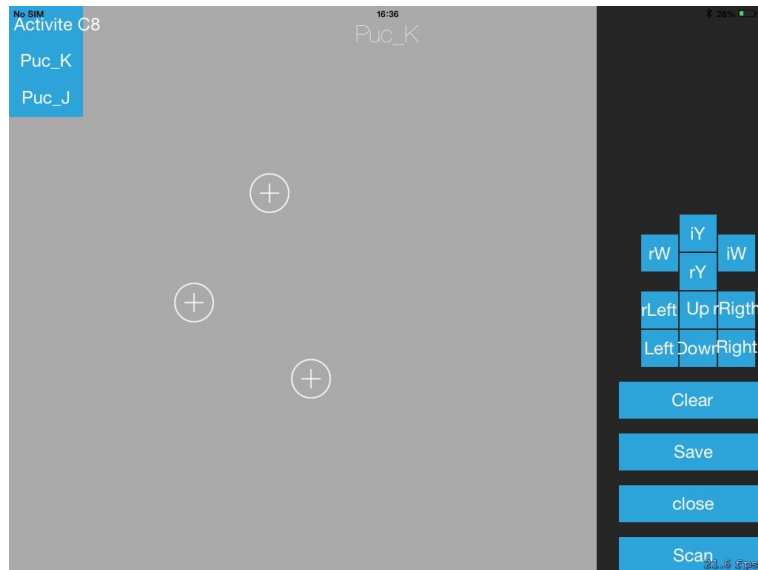


Figure 3.21: A sample SpriteKit scene of a Star Wars game that uses PERCs and multi-touch as input. This image is taken from Voelker et al. [2015].

module is in range and not used for another tangible description a button with the name is added in the top left corner of the scene. With these buttons one can select an bluetooth identifier for the creation of a PERC, if none is selected the created tangible will be a PUC. The currently selected identifier is shown in the label at the top of the scene.

The scene will automatically get the description of the three markers.

After this the tangible should be placed in the gray area. If all three markers generate touches the Scan button should be pressed. If there are exactly three touches in the gray area, a new tangible will be generated using those traces. Pressing Save will then add the new tangible description to the existing ones.

Most values are not directly configurable.

Using the arrows on the right one may change the size and offset of the tangible's area. All other settings are set to standard values and can be accessed via the configuration file. The Clear button will delete all of the existing tangible descriptions.

This scene is a good helper to define standard tangibles, but if a non standard pattern is used some settings will only be accessible through direct manipulation of the configuration file. This is impossible in iOS since the access of data from other apps is not allowed and the scene itself does not offer any access to the file. In MacOS this is possible, but often cumbersome. Setting for example the offset of the light sensor is important but not that easy in non standard pattern, since it is not clear where the actual position of the tangible is and how the x- and y-axis is placed. Especially this setting could be done automatically by the MTK. Turning one half of the scene white and the other black should allow to identify on which side the tangible is. Doing the same thing with the correct half of the scene will result in some kind of binary search and will ultimately find the light sensors position and thereby the offset of the light sensor.

The MTKTangible-CreationScene is in its current version insufficient to configure tangibles.



Figure 3.22: Two players using PERCs to play AirHockey. This picture is taken from the video by Voelker et al. [2015].

3.7 Sample Applications

We implemented several sample application using the MTK. Three of the most prominent are listed in this section: *Tangible Demo*, *Airhockey* and *ColorFighter*.

Tangible Demo shows the recognition of tangibles.

Tangible Demo. When using the MTK or configuring tangibles it is always important to know if the detection of tangibles is correct and that the application is running correctly. The tangible demo is included in the MTK and is a scene where all tangibles are replaced using a holo indicating the current position and rotation of the tangible. Additionally the name of the tangible is listed.

The airhockey scene is the scene mostly used for demos.

AirHockey. We implemented a small airhockey game. Thanks to Florian Busch we got two tangibles that look like air hockey mallets. We used the physics of SpriteKit to add digital objects that follow each tangible mallet. These digital objects will collide with the digital airhockey puc. By

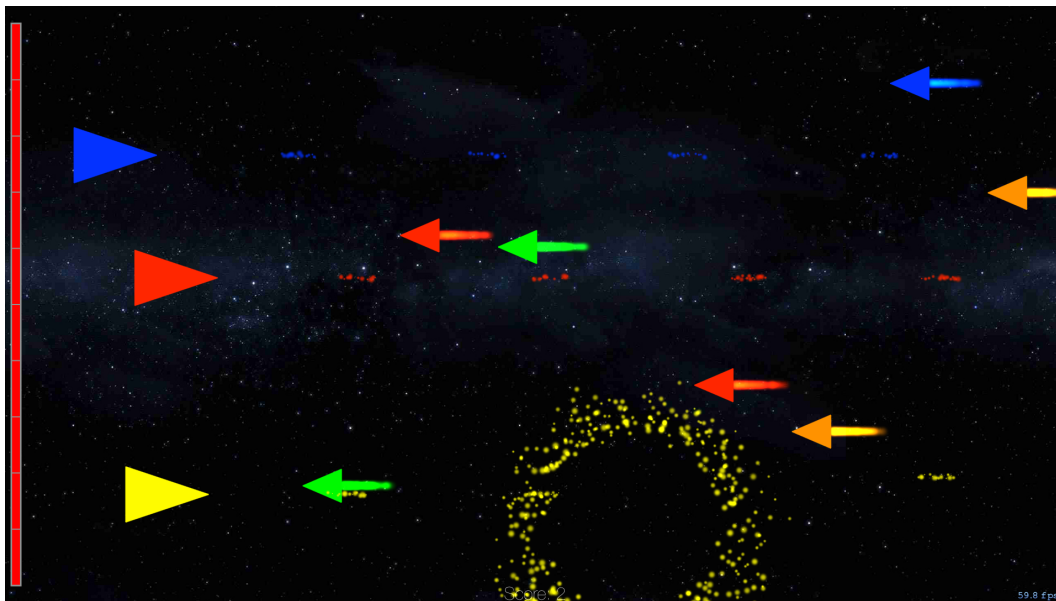


Figure 3.23: ColorFighter is a game similar to Space Invaders. The player has to hit enemy ships with a shot colored as the enemy's ship.

adjusting friction and bounciness we achieved a relative realistic feeling. The game works like a normal airhockey game. If the airhockey puc will hit the edges it will bounce back and if it goes in one of the goals the other player will get a point and eventually win. To show that this combination of real and digital world is capable of more than the physical tables, we added a power up that will add another puc to the game if collected.

ColorFighter. Another implementation is the game ColorFighter. The player gets three ships that will automatically shoot. At the opposite side of the screen smaller enemy ships will spawn. They try to reach the player's side before getting shot. If they archive this the player will lose a live, which is indicated by the health bar at the left side. After the player loses 10 lives the game is over. To make the game more interesting enemy ships have different colors. Each ship can only be destroyed if it is hit by a shot with the same color. The color of the shot is defined by the color of the ship that fired the shot. The player has three differently colored ships. If two ships are close to each other

ColorFighter is another demo application for the MTK.

their colors will mix and create a new one. This effect is lost if they are moved away from each other. If both of the other ships are in reach of the third it will turn white. Like this the player can achieve 7 different colors. Enemy ships will have one of these colors. In the current implementation the players ship are movable using drag and rotate gestures. In a future version we plan to use tangibles for this game, too.

Chapter 4

Evaluation

In this chapter we take a look into the work of Kammer et al. [2010], who analyzed different existing multi-touch frameworks. He identified several components with which he can distinct multi-touch frameworks. In Figure 4.1 one can see an overview of the features Kammer et al. [2010] identified. We will now proceed from bottom to top in this diagram and evaluate how the MTK addresses the different parts which are divided into features, scope, and architecture.

Kammer et al. [2010] did an analysis of existing multi-touch frameworks.

4.1 Architecture

The lowest area in Figure 4.1 is the architecture. It is divided in two layers. The lowest layer consists of two parts: Platform and hardware independence. The layer above consists of the event system. We will discuss all three parts below.

Platform independence. The fact that the framework is not bound to one operating system but can be used on any platform is called platform independence. A platform independent framework can reach a bigger number of users, since it can be used on more systems. Frameworks often

Platform independence allows frameworks to run on different systems.

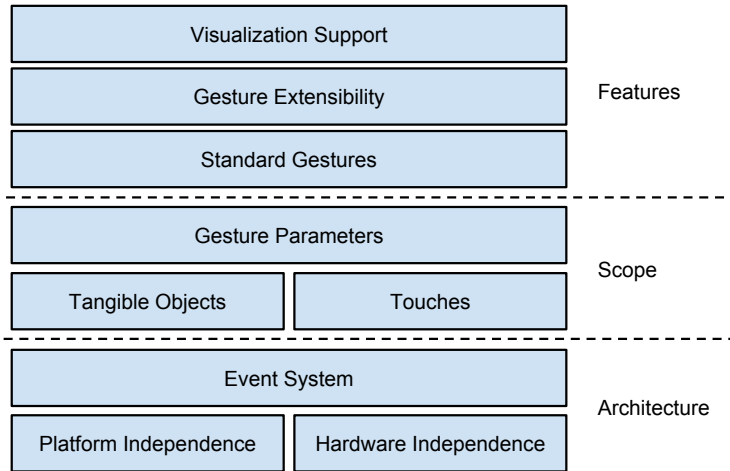


Figure 4.1: Replication of the diagram in Kammer et al. [2010]. It shows the different layers and features a multi-touch framework can have.

accomplish this by using programming languages like Java (Laufs and Ruff [2010]), Python (Hansen et al. [2009]) or JavaScript (Nebeling and Norrie [2012]), which are platform independence.

Platform independent frameworks may be slower than native ones.

The disadvantages of platform independence is for example the loss of performance and the lack of platform specific features. Performance often suffers when using cross platform languages like Java [Hansen et al., 2009, Nebeling and Norrie, 2012], since they may need additional overhead to run on all systems. Other languages may offer cross compiling to overcome this performance issue. Nevertheless will developers of cross compiled applications lack the ability to use the native IDEs and frameworks developed for a specific operating system or programming language.

We want a framework with native support for MacOS and iOS.

Since our main focus is to develop a framework for MacOS and iOS applications we decided that we will use Objective-C. Using this the MTK will be a native framework for MacOS and iOS, but not run on any other operating system. This allows us to use software development features for iOS and MacOS provided by Apple, but limits the operating systems the framework supports.

Hardware independence. The second part in the lowest layer of Figure 4.1 is the hardware independence. Hardware independence means that the framework may support touch input from many input sources. The usage of several input sources should be eased by the framework. Most of the existing frameworks achieve this by using TUIO [Kammer et al., 2010], but some also support Windows 7 Touch and custom device adapters. An overview can be found in Figure 4.2.

Multi-touch frameworks often support several hardware via TUIO.

We introduced several implementations of input sources in Section 3.2.2 to ensure the hardware independence of the MTK. We implemented a TUIO input source and custom implementations for other sources. Also Windows 7 Touch and any other device is supported using the JSON input source and its possibilities to modify received data via delegate.

The MTK supports TUIO, Windows 7 Touch and any other device adapter.

Event system. The third part in the architecture layer is the event system. In this part Kammer et al. [2010] identifies two variants. The first one is that the framework has some kind of gesture server which may process all gestures and sends all gesture events for example via network. This achieves some kind of loose coupling, allowing several clients to listen to the same gestures. The second alternative is that the framework's gesture recognizers raise gesture events which will be processed by the listeners.

Kammer et al. [2010] does two distinctions in the event system.

In the MTK we implemented the second variants. Gestures as well as all touch input are processed locally on each node. The delegate of the recognizers or the set action blocks may react on state changes.

The second distinction done by Kammer et al. [2010] in the event system is also related to gestures, but part of the actual gesture recognition instead of the distribution of the gesture events. In some frameworks the gesture processing is done central using a gesture registry and an abstraction of the UI. The registry then queries the application about its visual components to associate gestures with UI elements. The decentral approach is that each UI element allowing gestures processes them by itself.

A framework's gesture recognition can be done either decentral or central.

In this case the argumentation is the same: The MTK is decentral, since the touch and gesture processing is completely done on each node.

4.2 Scope

The second area in Figure 4.1 is the scope. Scope has three parts: gesture parameters, tangible objects, and Touches. As in the architecture section we will now discuss each part.

Most frameworks support tangibles via TUIO.

Tangible objects. Kammer et al. [2010] state that most of the frameworks support tangibles simply by implementing the TUIO protocol. Like this the framework can either process tangibles or send recognized tangibles to clients via network. Only a few of the frameworks actually focus on tangibles.

We focused on PUCs and PERCs.

As already explained in Section 3.3, the MTK has a focus on recognizing PUCs and PERCs. It is not capable of recognizing TUIO tangibles. This is a feature that could be integrated in future versions.

Which touch information are available depends on the framework.

Touches. Kammer et al. [2010] stated no clear distinction that can be made in this area, but discussed what touch information each framework provided. While some frameworks provide only a position per touch, many other frameworks have additional information like direction, size, and more.

The MTK provides access to any information ever received for a touch.

In Section 3.2.1 we presented MTKTrace, which contains all information the MTK provides for a single touch. It includes a history of all information ever received for a single touch, including identifier, position, size, and more. Several analysis are also already implemented, like the check if the trace is stationary or how old it is. Additionally can gestures and other objects perform any needed analysis using the trace's entries, which serve as the touch's history.

Gesture parameters. As in the previous paragraph, Kammer et al. [2010] did not provide a clear definition of what gesture parameters are and how they judged if a framework provides them or not. In general this point means that the framework provides parameters of recognized gestures, which specific parameters are provided depends on the gesture.

Most multi-touch frameworks provide gesture parameters.

In case of the MTK this is highly dependent on the gesture recognizer itself. All gesture recognizers provide the traces they used for recognition, but no other gesture parameters that are the same for all recognizer are provided. Each gesture can define its own set of parameters. This may be a limitation of the MTK if such parameters are required, since developers may have to extend the gestures to be provided with these parameters.

4.3 Features

The top area in Figure 4.1 by Kammer et al. [2010] is named features. It is separated in three layers: standard gestures, gesture extensibility, and visualization support.

Standard gestures. The first layer, standard gestures, contains everything related to the support of standard gestures. All frameworks evaluated by Kammer et al. [2010] support gestures. The distinction which is made is if the framework supports online and offline gestures. Online gestures are gestures like rotation and scale, which are processed while the user still interacts with the system. Offline gestures are evaluated after an interaction, like the analysis of touches forming a circle to activate a menu.

All evaluated multi-touch frameworks support standard gestures.

Both of these gesture types are supported by the MTK. As described in Section 3.4, gestures are processed on each node with the nodes bounded traces. In doing so the implementation of online gestures like scale and rotation are easily performed. Additionally can gesture recognizers receive more touches by accessing the node's old traces or

We support online and offline gestures.

all traces in the scene. This allows the processing of offline gestures.

The MTK supports global gestures.

Another gesture type Kammer et al. [2010] defined are global gestures, which some frameworks support. These gestures are available application wide and can be performed independently from the content. As we already described in Section 3.2.3, they are processed in the fifth step of the initialization of the touch processing. Here they are processed independently from the active scene.

Most frameworks extend gestures by subclassing existing ones.

Gesture extensibility. The second layer is the Gesture extensibility. Each framework should provide a simple way to implement new gestures. Kammer et al. [2010] reported that the evaluated frameworks range from providing just raw data to subclassing.

The MTK provides all gesture extensibilities Kammer et al. [2010] listed.

The gesture recognizers and their extensibility are explained in Section 3.4. Each existing gesture can be modified using a delegate, action blocks and a recognition block. Additionally any new gesture can be implemented by subclassing `MTKGestureRecognizer`, which already provides functionalities like the block management. All gesture recognizers provide all traces related to their recognition process, as well as access to all other existing traces, which equivalents an access to all sent raw data. Therefore the MTK offers all possibilities for gesture extensibility Kammer et al. [2010] described.

Kammer et al. [2010] state the need of visualization support for any multi-touch framework.

Visualization support. The last layer in Figure 4.1 is the visualization support. Kammer et al. [2010] state that any framework needs visualization support, either in having an own set of extensible UI components, or the possibility to create own UI elements.

The MTK provides visualization support due to the support of any SpriteKit scene.

The MTK provides built-in UI elements as well as the creation of own ones. As described in 3.5, SpriteKit is a framework to render nearly any possible 2D scene. The base of all scenes are `SKNodes`, which we extended with categories to be touchable. Because of this any scene build with SpriteKit




	<small>cross-platform single platform</small> <small>integrated library gesture server</small>	  	Architecture				Scope			Features		
			TUIO	Win7	Device Adapter	Gesture Events	Tangibles	Touch Params	Gesture Params	Standard Gestures	Gesture Extensibility	Visualization support
MT4j		✓	-	✓	decentral	supported	✓	✓	online	super class	custom widgets	
SparshUI	>img alt="integrated library icon" data-bbox="251 224 266 239"/>	✓	-	✓	central	supported	-	✓	online	super class	-	
Surface SDK		-	✓	-	decentral	focus	✓	✓	online	raw data	WPF multitouch controls	
Breezemultitouch		✓	-	-	decentral	not supported	✓	-	online	wrapper class	WPF multitouch controls	
Miria		✓	✓	✓	decentral	not supported	✓	✓	online	raw data and recognition support	WPF multitouch controls	
Graffiti		✓	-	-	central	focus	✓	✓	online	super class	-	
libTISCH	>img alt="integrated library icon" data-bbox="251 324 266 339"/>	✓	-	✓	central	supported	-	✓	online	super class	custom widgets	
PyMT		✓	✓	-	decentral	supported	✓	-	online offline	raw data and recognition support	custom widgets	
GestureWorks		✓	-	-	central	not supported	-	✓	online	super class	custom widgets	

Figure 4.2: This overview of all features of existing multi-touch frameworks is taken from Kammer et al. [2010].

can be combined with the MTK to process touches, gestures, tangibles, and all other things the MTK provides. Some frameworks also have the possibility to provide 3D scenes. 3D is not explicitly included into the MTK, but can be achieved due to SceneKit which supports 3D and can be easily integrated into SpriteKit scenes.

4.4 Summary

Kammer et al. [2010] identified several aspects to distinct existing frameworks. We evaluated each part for the MTK. To sum up this evaluation we revisit Figure 4.1 in which Kammer et al. [2010] listed all evaluated frameworks and their features.

The MTK can now also be placed in this figure. Going from left to right in the figure, and top to bottom in this chapter:

The MTK ...

- ... is partly cross-platform, since it supports MacOS and iOS, but no other platform.
- ... has no gesture server, but has an integrated library.
- ... supports parts of TUIO.
- ... can support Windows 7.
- ... can support any device adapter.
- ... has decentral gesture events.
- ... has a focus on tangibles (PUCs and PERCs).
- ... provides touch parameters.
- ... can provide gesture parameters.
- ... supports online and offline gestures.
- ... supports gesture extensibility via super class, raw data, delegate and blocks.
- ... includes visualization support due to the support of any SpriteKit scene and custom UI elements.

Chapter 5

Summary

We introduced the MultiTouchKit, a framework to integrate multi-touch, PUCs and PERCs into MacOS and iOS. We described all current features, ongoing developments and possible improvements of the MTK. Additionally we compared those features against those Kammer et al. [2010] identified in their evaluation of existing multi-touch frameworks. To summarize we can state that the MTK addressed all features presented by Kammer et al. [2010].

5.1 Future work

Since the development time of the framework was limited, we concentrated on a reliable basis and had to ignore some features that may be very interesting for future versions. Some of those were already discussed at different points in the work. In this section we list some essential features we would like to see in future versions.

Enhance UI and Gestures. We already explained that our focus was to integrate different input sources and to recognize tangibles, therefore gestures and standard UI elements are in a very basic state. Especially the UI elements would benefit from a common event system, which could be ex-

Enhance the
standard UI
Elements

tended to gesture events. A loadable GUI configuration file which may include a color template would also be a nice extension to customize UI elements without recompiling the MTK or manipulating them in the source code.

Improve the
TouchControlCenter

The TCC is a helpful tool to configure the MTK. Unfortunately it is missing several settings due to the complexity of the settings the MTK offers. It needs a complete redo to be a perfect configuration tool for the MTK. Additionally, since iOS now allows the access of data from other applications, it would be much more helpful if the TCC is a standalone application that configures a shared settings file. The current implementation is a scene that is loaded before the actual application.

Support TUIO
tangibles.

The MTK is able to receive touch information via TUIO, but none of TUIO's tangible information are used. We could support any kind of light sensing technology if we address TUIO input in more detail. The MTK could easily support tangibles here, which are different from PUCs and PERCs.

Support of better and
other PERCs.

The development of the MTK was done mostly for standard PERCs. This was a given requirement by the supervisor since it would probably result in the most reliable tangible detection for the MTK. We think that it is possible to implement a similar reliable implementation that works on any form of PERCs. In many cases we already implemented the detection in a general manner, but in some situations it is still focused on the standard design.

PERCs are not fully
reliable so far.

Additionally are some cases in the detection not reliable because PERCs still have some flaws. For example is the light sensor not very reliable. We could improve the recognition process if the hardware is more reliable. It may also be possible that PERCs get an updated hardware that may allow new interaction possibilities, which may allow a much better recognition. For example Florian Busch works in his thesis on movable PERCs, which is another connection between digital and real world using tangibles that can react to changes in the digital world. We already included very basic support to sent data to tangibles, but this could be extended to allow easy access to the tangible's data.

Bibliography

Apple Inc. Sknode class reference. URL https://developer.apple.com/library/ios/documentation/UIKit/Reference/SKNode_Ref/. Accessed: 2015-09-17.

CodePlex. Miria sdk - multi device input ui controls for silverlight and moonlight. URL <http://miria.codeplex.com>. Accessed: 2015-09-17.

Alessandro De Nardi. Grafiti - gesture recognition management framework for interactive tabletop interfaces. URL <https://grafitiproject.wordpress.com>. Accessed: 2015-09-17.

Dimitri Diakopoulos and Ajay Kapur. Argos: An open source application for building multi-touch musical interfaces. In *International Computer Music Conference*, 2010.

Florian Echtler and Gudrun Klinker. A multitouch software architecture. In *Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges*, pages 463–466. ACM, 2008.

Thomas E. Hansen, Juan Pablo Hourcade, Mathieu Virbel, Sharath Patali, and Tiago Serra. Pymt: A post-wimp multi-touch user interface toolkit. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces, ITS '09*, pages 17–24, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-733-2. doi: 10.1145/1731903.1731907. URL <http://doi.acm.org/10.1145/1731903.1731907>.

Ideum. Gestureworks - multitouch & hci software framework. URL <http://gestureworks.com/>. Accessed: 2015-09-17.

- Martin Kaltenbrunner and Ross Bencina. reactivation: A computer-vision framework for table-based tangible interaction. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction, TEI '07*, pages 69–74, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-619-6. doi: 10.1145/1226969.1226983. URL <http://doi.acm.org/10.1145/1226969.1226983>.
- Martin Kaltenbrunner, Till Bovermann, Ross Bencina, and Enrico Costanza. Tuio: A protocol for table-top tangible user interfaces. In *Proc. of the The 6th Int'l Workshop on Gesture in Human-Computer Interaction and Simulation*, pages 1–5, 2005.
- Dietrich Kammer, Mandy Keck, Georg Freitag, and Markus Wacker. Taxonomy and overview of multi-touch frameworks: Architecture, scope and features. In *Workshop on Engineering Patterns for Multitouch Interfaces*, 2010.
- Werner A. König, Roman Rädle, and Harald Reiterer. Squidy: A zoomable design environment for natural user interfaces. In *CHI '09 Extended Abstracts on Human Factors in Computing Systems, CHI EA '09*, pages 4561–4566, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-247-4. doi: 10.1145/1520340.1520700. URL <http://doi.acm.org/10.1145/1520340.1520700>.
- Uwe Laufs and Christopher Ruff. Mt4j - a cross-platform multi-touch development framework. In *Workshop of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 2010.
- Rong-Hao Liang, Liwei Chan, Hung-Yu Tseng, Han-Chih Kuo, Da-Yuan Huang, De-Nian Yang, and Bing-Yu Chen. Gaussbricks: Magnetic building blocks for constructive tangible interactions on portable displays. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems, CHI '14*, pages 3153–3162, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557105. URL <http://doi.acm.org/10.1145/2556288.2557105>.
- Johannes Luderschmidt, Immanuel Bauer, Nadia Haubner, Simon Lehmann, Ralf Dörner, and Ulrich Schwanecke. Tuio as3: A multi-touch and tangible user interface rapid

- prototyping toolkit for tabletop interaction. In *Self Integrating Systems for Better Living Environments: First Workshop, Sensyble*, pages 21–28, 2010.
- Mindstorm Inc. Breezemultitouch - multi-touch framework for wpf 3.5. URL <https://code.google.com/p/breezemultitouch/>. Accessed: 2015-09-17.
- Michael Nebeling and Moira Norrie. jqmultitouch: Lightweight toolkit and development framework for multi-touch/multi-device web interfaces. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '12*, pages 61–70, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1168-7. doi: 10.1145/2305484.2305497. URL <http://doi.acm.org/10.1145/2305484.2305497>.
- NUI Group. Touchlib - a multi-touch development kit. URL <http://nuigroup.com/touchlib/>. Accessed: 2015-09-17.
- Prasad Ramanahally, Stephen Gilbert, Thomas Niedzielski, Desirée Velázquez, and Cole Anagnost. Sparsh ui: A multi-touch framework for collaboration and modular gesture recognition. In *ASME-AFM 2009 World Conference on Innovative Virtual Reality*, pages 137–142. American Society of Mechanical Engineers, 2009.
- Jun Rekimoto. Smartskin: An infrastructure for free-hand manipulation on interactive surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '02*, pages 113–120, New York, NY, USA, 2002. ACM. ISBN 1-58113-453-3. doi: 10.1145/503376.503397. URL <http://doi.acm.org/10.1145/503376.503397>.
- Christophe Scholliers, Lode Hoste, Beat Signer, and Wolfgang De Meuter. Midas: A declarative multi-touch interaction framework. In *Proceedings of the Fifth International Conference on Tangible, Embedded, and Embodied Interaction, TEI '11*, pages 49–56, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0478-8. doi: 10.1145/1935701.1935712. URL <http://doi.acm.org/10.1145/1935701.1935712>.

Johannes Schöning, Jonathan Hook, Tom Bartindale, Dominik Schmidt, Patrick Oliver, Florian Echtler, Nima Motamedi, Peter Brandl, and Ulrich von Zadow. Building interactive multi-touch surfaces. In Christian Müller-Tomfelde, editor, *Tabletops - Horizontal Interactive Displays*, Human-Computer Interaction Series, pages 27–49. Springer London, 2010. ISBN 978-1-84996-112-7. doi: 10.1007/978-1-84996-113-4_2. URL http://dx.doi.org/10.1007/978-1-84996-113-4_2.

B. Signer, U. Kurmann, and M.C. Norrie. igesture: A general gesture recognition framework. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, volume 2, pages 954–958, Sept 2007. doi: 10.1109/ICDAR.2007.4377056.

Lucia Terrenghi, David Kirk, Abigail Sellen, and Shahram Izadi. Affordances for manipulation of physical versus digital media on interactive surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '07*, pages 1157–1166, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-593-9. doi: 10.1145/1240624.1240799. URL <http://doi.acm.org/10.1145/1240624.1240799>.

Simon Voelker, Kosuke Nakajima, Christian Thoresen, Yuichi Itoh, Kjell Ivar Øvergård, and Jan Borchers. Pucs: Detecting transparent, passive untouched capacitive widgets on unmodified multi-touch displays. In *Proceedings of the 2013 ACM International Conference on Interactive Tabletops and Surfaces, ITS '13*, pages 101–104, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2271-3. doi: 10.1145/2512349.2512791. URL <http://doi.acm.org/10.1145/2512349.2512791>.

Simon Voelker, Christian Cherek, Jan Thar, Thorsten Karrer, Christian Thoresen, Kjell Ivar Øvergård, and Jan Borchers. Percs: Persistently trackable tangibles on capacitive multi-touch displays. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology (to appear)*, UIST '15, New York, NY, USA, November 2015. ACM. doi: 10.1145/2807442.2807466. URL <http://dx.doi.org/10.1145/2807442.2807466>.

Malte Weiss, Julie Wagner, Yvonne Jansen, Roger Jennings, Ramsin Khoshabeh, James D. Hollan, and Jan Borchers. Slap widgets: Bridging the gap between virtual and physical controls on tabletops. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '09*, pages 481–490, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-246-7. doi: 10.1145/1518701.1518779. URL <http://doi.acm.org/10.1145/1518701.1518779>.

Neng-Hao Yu, Li-Wei Chan, Seng Yong Lau, Sung-Sheng Tsai, I-Chun Hsiao, Dian-Je Tsai, Fang-I Hsiao, Lung-Pan Cheng, Mike Chen, Polly Huang, and Yi-Ping Hung. Tuic: Enabling tangible interaction on capacitive multi-touch displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11*, pages 2995–3004, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0228-9. doi: 10.1145/1978942.1979386. URL <http://doi.acm.org/10.1145/1978942.1979386>.

Index

AirHockey	66
AppleDoc	12
Argos	7
Breezemultitouch	5–6
ColorFighter	67–68
entry	15–16
evaluation	69–76
future work	77–79
gestures	53–57
GestureWorks	5
Ghost Touch	38–39
Grafiti	8
iGesture	8
input source	19–23
introduction	1–3
jQueryMultitouch	6
JSON input source	21–22
libTisch	8
marker	33–34
Midas	6
Miria	6
mouse input source	20–21
MT4j	7
MTKEntry	15–16
MTKInputSource	19–23
MTKTangibleCreationScene	63–65
MTKTrace	15–19
Multitouchkit	11–68
node	12

PERC	41–51
PUC	33–41
PyMT	8
related work	5–9
scene	12
SKNode	12
SKScene	12
Sparsh UI	7, 8
SpriteKit	11–12
Squidy	7
Tangible Demo	66
Tangible Simulator	51
Tangibles	33–51
TISCH framework	<i>see</i> libTisch
touch processing	14–32
TouchControlCenter	61–65
Touchlib	8
trace	15–19
TUIO	6–7
TUIO AS3	7
TUIO input source	21
UITouch input source	21
visualization support	58–60

