2007:30 HIP

BACHELOR'S THESIS

Gesture Analyzing for Multi-Touch Screen Interfaces

Michael Thörnlund

Luleå University of Technology

BSc Programmes in Engineering
BSc programme in Computer Engineering
Department of Skellefteå Campus
Division of Leisure and Entertainment

2007:30 HIP - ISSN: 1404-5494 - ISRN: LTU-HIP-EX--07/30--SE

Luleå University of Technology, Campus Skellefteå
Bachelor of Science degree in Games Programming (GP), 180 ECTS

GESTURE ANALYZING FOR MULTI-TOUCH SCREEN INTERFACES

Michael Thörnlund

Abstract

The way we use computers today will soon change. The technology of the future will allow us to interact with the computer on a whole different level from what we are used to. The tools we use to communicate with the computer - such as the mouse and the keyboard, will slowly disappear and be replaced with tools more comfortable and more natural for the human being to use. That future is already here.
The increase rate of how touch screen hardware and applications are used is growing rapidly and will break new grounds in years to come. This new technology requires new ways of detecting inputs from the user - inputs which will be made out of on-screen gestures rather than by the pressing of buttons or rolling mouse wheels.
This report describes the gestures defined, the methods used to detect them and how they are passed on to an application.

Sammanfattning

Sättet som vi använder datorer på idag kommer snart förändras. Framtidens teknik kommer att ge oss möjligheten att interagera med datorn på ett helt annat sätt än vad vi har blivit vana med. De verktyg som vi idag använder för att kommunicera med datorn, i de flesta fall musen och tangentbordet, kommer sakta men säkert att försvinna och ersättas med alternativ som känns mer naturliga för människan. Den framtiden är redan här.
Graden med hur användningsområden för applikationer kan representeras i ett touchscreengränssnitt ökar markant just nu. Det gäller även för antalet nya typer av hårdvaror. Denna nya teknik kräver nya sätt att interagera med datorn och nya sätt för datorn att förstå inmatningar - som inte längre kommer att bestå av knapptryckningar eller hjulrullningar, utan snarare av gester direkt på skärmen.
Den här rapporten beskriver de definierade gesterna, metoderna som används för att tolka dem och hur de förs vidare till applikationen.

Preface

This thesis project was carried out at gsCEPT, Luleå University of Technology, campus Skellefteå, during the period of 10 weeks in the spring of 2007. The university had recently looked into the FTIR technology and was in the act of acquiring the equipment needed for it.
We were a total of 4 students who got the assignment to develop different parts for a touch screen interface with belonging proof-of-concept software. My part was to create a software library to sort the multiple finger inputs on the screen and interpret the gestures made out of them. These gestures are then passed on as inputs to an application.

The library is meant to be used in future development for the multi-touch interface.

I have been working closely with Niklas Brolin, whose part in the project was in the area of gesture recognition. Since we both were into gestures, it came natural to share advices and ideas between us.

I would like to thank Johannes Hirche at LTU who has been my mentor during this project. The work he did on the TouchLib (an open source finger input detection library) configuration made it easy for me to focus on the gesture library alone.
Thank you to Patrik Holmlund (LTU) and Arash Vahdat (LTU) too, for the opportunity to work in this exciting field.

Figure 1: Game application

1 Introduction

1.1 Background and Purpose

The goal of this project was to create a gesture analyzing software library, which was to work as an intermediate layer between the hardware screen and the application(s) projected on it. The application will receive the gesture data as input, along with information on which objects are affected. The obvious things anyone would expect as doable to objects shown on a touch screen are to *select* (multi- and single selection), *rotate*, *scale* and *move* them, so those are the gestures focused on.

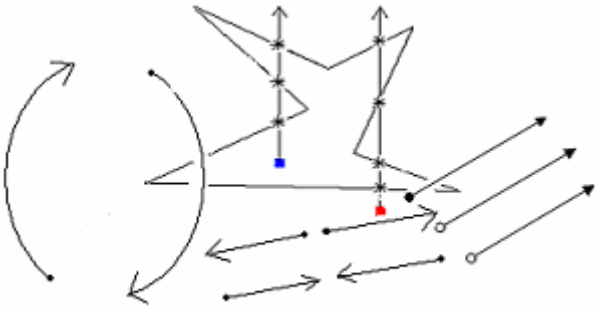

Figure 2: Gesture symbols

1.1.1 FTIR

FTIR stands for *Frustrated Total Internal Reflection* and is the most widely spread representation of the multi-touch technology today. There are occurrences of other methods [1, 6] as well, but FTIR seems to be the method most frequently referred to. It is also considered among the cheapest [2] setups.

The principles of the technology are quite simple:
Infra Red light is shined into the side of a pane of a somewhat transparent higher-index medium (acrylic, glass, plastic) and is trapped inside this medium by the refraction index of the material. The light sources in the FTIR-case are a number of infra-red diodes attached to the side of the pane, while TouchLight [1] uses one infra-red illuminant shining on to the surface. Total Internal Reflection occurs when the light ray is traveling inside a higher-index medium and strikes a surface boundary into a lower-index medium. When a finger touches the surface of the pane, the light is frustrated, causing the light to scatter downwards where it is picked up by an IR camera. It is really an example of the laws of geometric optics.

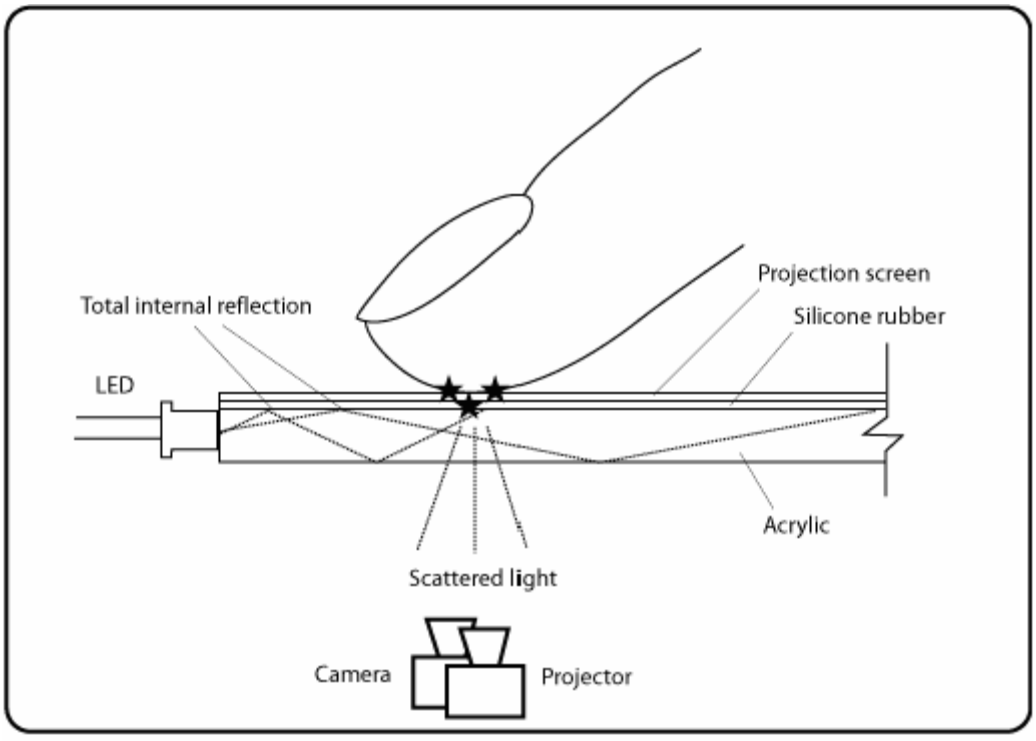

Figure 3: FTIR technique overview

On the side opposite from the user is a camera with visible light filter that registers the scattered light. There is also a projector which projects the image on to the projection screen.

1.1.2 Hardware

During the project there was direct access to a simple prototype, which was crucial during the development phase. The need for instant testing during implementing is always a necessity. Particularly in this case, since the technology developed for is very much considered cutting edge at this time. In many cases unforeseen problems occurred as the project proceeded. Instant testing is decisive in those cases.

The pane of the prototype was made out of glass. Infra-Red LED's were attached to the side of the glass, giving the result described in 1.1.1. On the side opposite from the user was a back projection material, which provided the user with a clear image from the projector. To register the frustrated light, we used a web-camera with a filter to block out the visible light. In our case the filter was made out of a couple of fluorescent-light exposed photo negatives, which were attached to the camera lens. With a little calibration and configuration, the setup worked nicely for a testing environment.

1.1.3 Multi-Touch

The traditional way of interacting with a computer is by using a mouse and/or a keyboard. We provide the computer with inputs more or less by the use of buttons. Regardless of the input type, the computer can more or less only handle one input at the time which makes the input handling and sorting very easy.

However, multi-touch is as far from single input handling as one can come. The amount of concurrent events in this interface is limited only by the data type holding the number of finger inputs. The amount of simultaneous users is pretty much unlimited in the same way, which of course comes in handy for larger scale display systems. That amount of potential synchronous inputs requires new ways to detect the inputs. Since they aren't the kind of ”on/off” inputs we are used to in the traditional sense, there are needs for new ways to interpret and analyze the input type and the gesture(s) they make out.

The multi-touch screen interface makes the user handling very intuitive. It is in some ways comparable to paper sheets laid upon a table, in the sense that there isn't really any need for a manual in how to interact with a piece of paper or move it around - the simplicity of it makes it all very natural for us. Software interaction through a multi-touch interface has the potential of becoming as intuitive and natural for us as moving papers around on a table.

1.1.4 Gestures

In order to achieve the intuitive interactive means mentioned above, there is need to define a number of gestures. The ones focused here were as stated earlier; *Move*, *Rotate*, *Select*/*MultiSelect* and *Scale*. Once these gestures are translated into class objects, the simplicity of interaction will come clear. The cases of *Move* and *Rotate* probably don't need any further introduction, they are pretty self explanatory. Just think of the gesture movements needed to move or rotate the piece of paper on the table - quite obvious every day interaction really. *MultiSelect* and *Scale* however, are things you don't generally expect as doable to a paper sheet.

Selection/Multi-Selection

A number of objects are projected onto the screen. The objects within the area of the "line" drawn by the finger(s) are selected, the ones outside aren't. An event is sent to the application, which will decide what action to take.

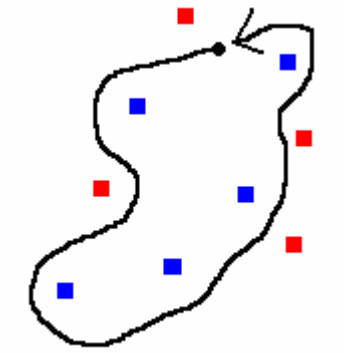

Figure 4: One finger selection

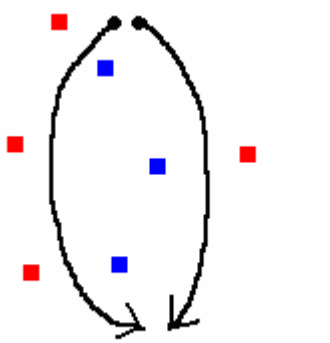

Figure 5: Multi finger selection

The red and blue dots represents on-screen objects. The blue ones become selected.

Scale

This gesture requires at least two fingers and is an example of simultaneous inputs. To scale an on screen object, grab it with the fingers and bring them together to scale down, or separate them to scale up.

Figure 6: Scale

The scale event is sent to the application that holds the object, with information of the grade of the scaling as well as the object it affects. Quicker finger motion means faster scaling.

1.1.5 External Software

The gesture library will depend on an external library for the finger input detection. The gesture library will gather the input data, then store the data and analyze what kind of gesture it generates. For the purpose of the finger detection it was decided early on to use TouchLib - an open source library for infra red blob detection. The reason for the choice was easy; it filled all the needs for us.

The gesture library will register itself as a listener to TouchLib and thereby get to implement the virtual functions for '*finger down*', '*finger moved*', and '*finger released*'. For each of those events the gesture library will gather 'TouchData', an object consisting of *PositionX*, *PositionY*, *DeltaX*, *DeltaY*, *Area*, *DeltaArea* and a unique *ID*. When a finger is pressed, it receives an ID, and the rest of the data gets updated every frame until the finger is released. Of course, the area data has the nice consequence of the TouchLib library being pressure sensitive.

TouchLib relies on the use of other libraries as well:

OpenCV	Open Computer Vision Library, a collection of algorithms and sample code for various computer vision problems.
DSVideoLib	A DirectShow wrapper supporting concurrent access to framebuffers from multiple threads.
VideoWrapper	A library that provides a single abstract API for interfacing video camera libraries.
OSCpack	A set of C++ classes for packing and unpacking OSC packets.

2 Methods

The work was divided into three major sections: Research, Design and Implementation. This is a reliable model which has been used in a countless amount of projects and seems to be the most obvious. Out of the 10 weeks period the project lasted, roughly 3 weeks was planned for each part and one week for documentation and wrapping up.

2.1 Research

Since the area of multi-touch still is quite new, there aren't too many well documented projects out there yet. The research phase became a time where we tried to grasp the concept of it. We spent quite some time to get the configuration of our prototype right and to get the TouchLib up and running. Once that was done, we were focusing on getting an understanding of TouchLib, and what we could expect out of it. It comes with a couple of demo applications, so there are some help in getting started.

There is an online open source multi-touch community called the NUI Group, which holds a lot of information on both hardware and software within the FTIR technology. A lot of time was spent browsing their forums.

After a short while we got the hang of it and could easily write some test applications. Each finger is stored into a `std::map`, where the key element is the ID of the finger currently active. The object type holds a `std::map` of the trail the finger draws, where the object type is of *Vector2* which connects the points for every new position. This model means Ordo(n2) for each iteration. Since the principals of multi-touch are what they are, they seem to imply a lot of dimensional list iterations and the risk of poorly optimized code is apparent.

Figure 7: Drawing application

It is apparent that the library will have to keep track of the object currently being affected by gestures. Since the application is the only one who has knowledge of its objects, we had to come up with some object mutual to both the application objects and the gesture library. We quickly decided on an idea involving axis aligned bounding boxes. Positions in an on screen application are mutual for everything represented on the screen, and the positions and sizes of the bounding boxes has to be updated by the application and aligned to the objects they represent. Since the number of application windows isn't limited in any way, and since this *is* multi-touch we're talking, there has to be some way to identify the current application too.

2.2 Design

2.2.1 Engine

The goal of the gesture library was for it to be easy to maintain, as well as easily scalable in terms of adding new gestures to it. The applications (i.e. the listeners) should also be separated from TouchLib, so that they just will have to listen to gestures, apart from listening to gesture - and finger events. The gesture library listens to finger events and provides the application with the gestures it detects.
The design went through some changes a couple of times before it finally ended up with the following model:

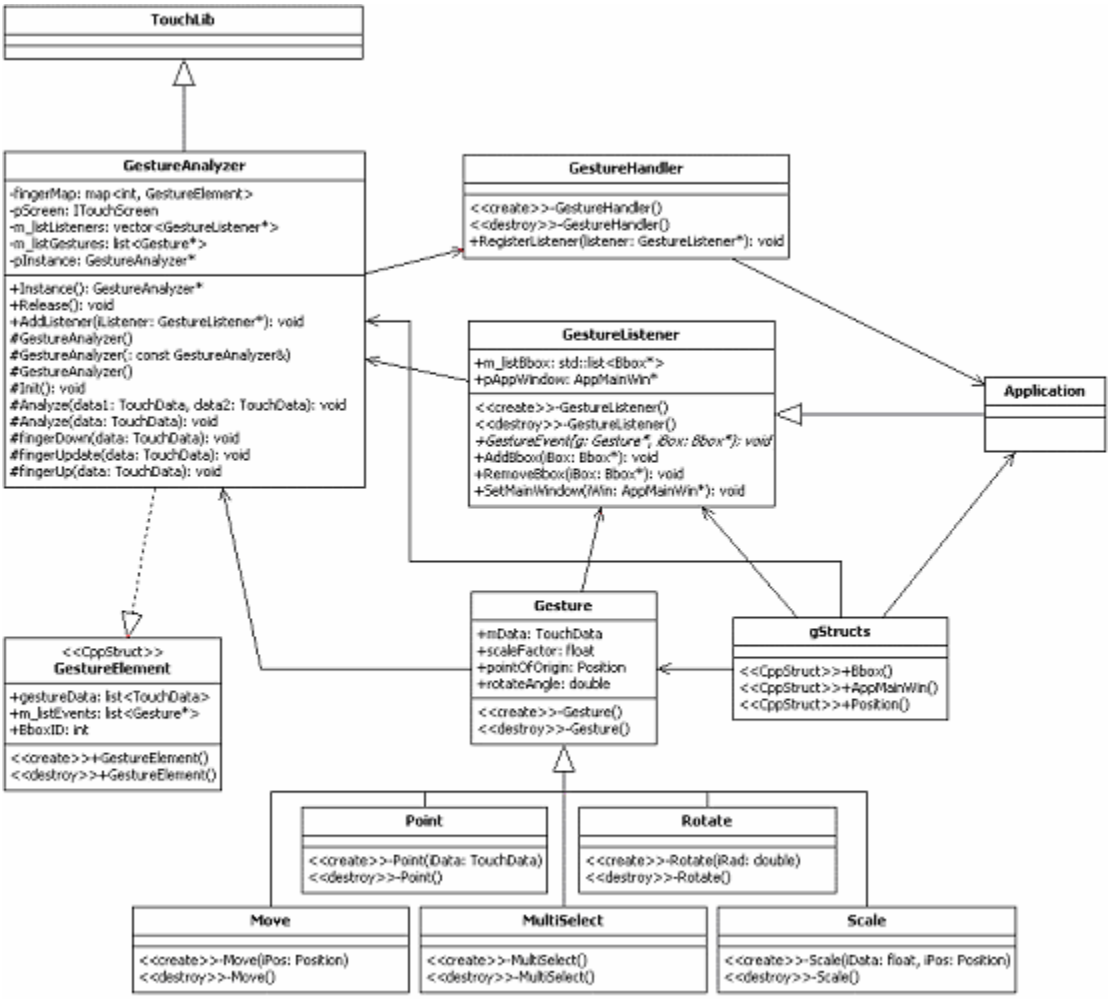

Figure 8: Design model overview

The application is separated from the TouchLib and the inheritance structure of the gesture classes provides for the scalability. Every new type of gesture is just added as a new sub class to the *Gesture* base.

The application inherits from the *GestureListener* class and registers itself as a listener at the *GestureHandler*. The application will thereby receive gesture events from the pure virtual function `GestureEvent(Gesture*, Bbox*)`.

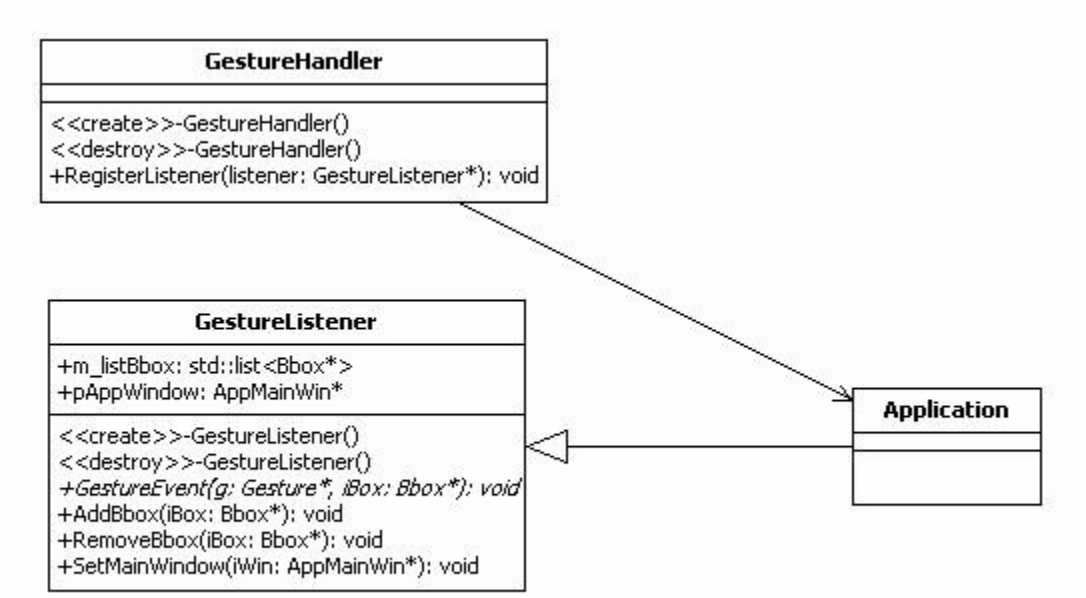

Figure 9: The application registers as a listener for gestures

The data types of *Bbox* and *AppMainWin* comes from a collection of structs, *gStructs*, which holds the different objects necessary to communicate with an arbitrary application, of which the gesture library knows nothing about.

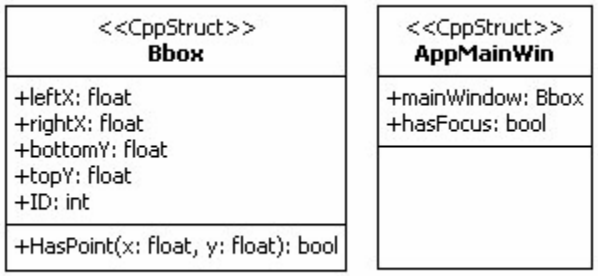

Figure 10: Structs

The application will have to let the gesture library know what objects it will take into account when it comes to analyzing the gestures being performed on them, as well as updated and oriented extreme points of the same objects. One thing all the visible objects in every application have in common is a screen position. That is why the *Bbox* model was chosen. The *leftX*, *rightX*, *bottomY* and *topY* are the extreme points of the object, and are also the axis of the object oriented bounding box enclosing it. Adding and removal of Bboxes to/from the system is made through the functions `AddBbox(Bbox*)` and `RemoveBbox(Bbox*)` in *GestureListener*. Every *Bbox* of the application has a unique ID, which is important for the communication between the gesture library and its listeners. Every application also has a main window, which has a representation in form of the *AppMainWin* object. It is really a *Bbox*, but treated

separately in the gesture library. If the variable *hasFocus* is true, the application will be taken into account for gestures being performed within its area.

GestureAnalyzer is the class located closest to the hardware. It inherits the TouchLib interface and gets to implement the pure virtual functions that are important for further analysis: `fingerDown`, `fingerUpdate` and `fingerUp`. The *TouchData* they provide are sent to the `Analyze(TouchData)` function for single finger analysis and to the `Analyze(TouchData, TouchData)` for the analysis of multiple fingers. Multiple finger gestures aren't necessarily done by two fingers, but they are analyzed in pairs.

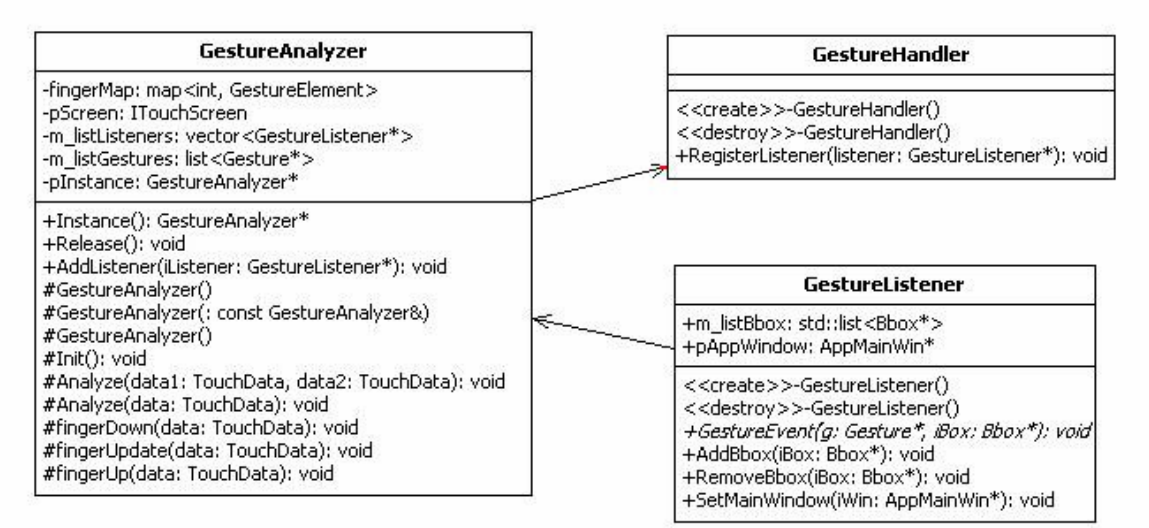

Figure 11: The GestureAnalyzer provides the listener with gesture inputs

GestureAnalyzer is a Singleton class, meaning that a new instance of the class is created if one does not exist. Every application registers as a listener in *GestureHandler*, from where an instance of *GestureAnalyzer* is being created. The *GestureHandler* serves as an interface between the application and the TouchLib, separating the two for the reasons mentioned above.

The data type of *GestureElement* seen as the object type in the `std::map fingerMap` in *GestureAnalyzer* is a data type used only within the class and is represented by a local struct which contains the data of what a gesture is made.

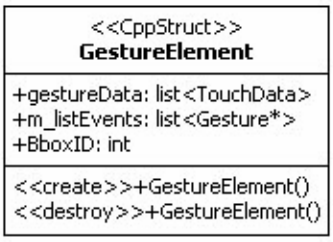

Figure 12: GestureElement

The std::list gestureData holds the *TouchData* of every finger currently active. All the gestures being made is added to the std::list m_listEvents. The *BboxID* is used to check which *Bbox* of the application is affected. It has a default value of -1, which means that a gesture is being made but not affecting the particular *Bbox*.

2.2.2 Gestures

The gesture design was more or less a matter of turning hand motions into objects, even though the hand made gestures themselves are hard to define [3]. The interaction should feel as natural as possible, and in no way limited to any pre-defined pattern of how to do this and that. There aren't two people that interact with objects in exactly the same way; in fact, the chance of two gesture movements ever been identical is probably close to zero.

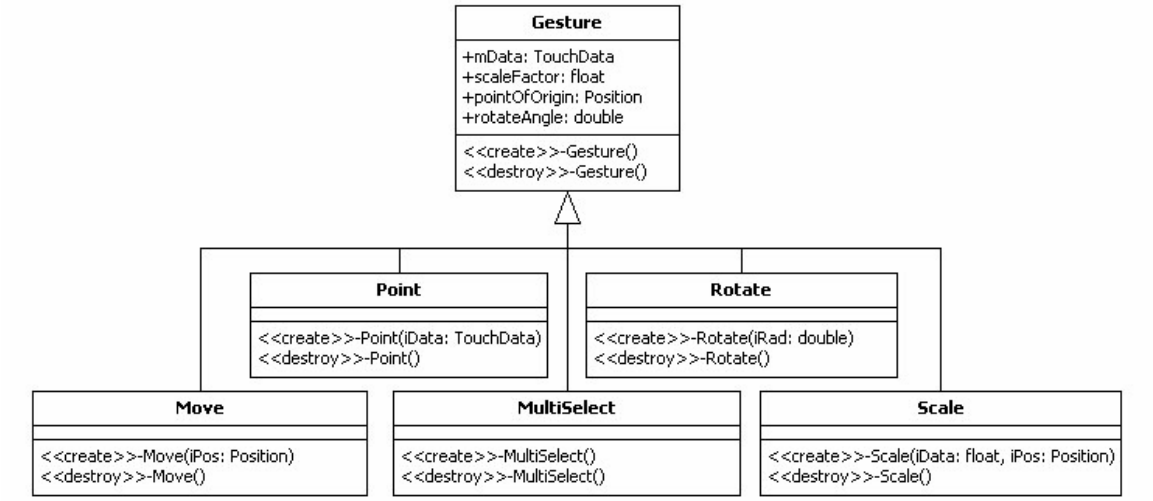

Figure 13: The Gesture base class and the sub classes

The gesture classes have an inheriting structure, with the base class *Gesture* representing the gesture family. Regardless of what type of gesture that emerges in the analyzing functions of *GestureAnalyzer*, the *Gesture* pointer sent to the listener will have the type of the corresponding sub class.

Point is the simplest to determine of all the gestures. It holds a *TouchData*, which simply is all the data from TouchLib forwarded in one big package. Since it contains all the information of the light blobs' position and size, the listener can decide what to do with it. Single selection of an object is an apparent use of it. Every finger pressed onto the screen will initially compose a gesture of this type.

Rotate requires a minimum of two fingers to perform. The fingers are placed on the screen and moved in a rotating manner.

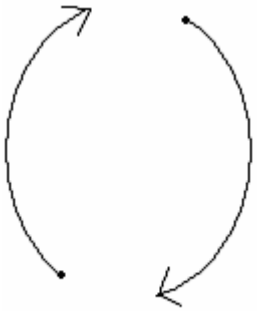

Figure 14: Rotate

The *rotateAngle* variable in the *Gesture* base class will hold the amount of radian rotation from the previous frame. It will have a positive value for clockwise rotations and a negative value during anticlockwise rotations. The listener will get an event of a rotation occurring, with information on the affected *Bbox* along with the updated amount of rotation.

Move is a simple example of a natural interaction with the interface. It is performed by grabbing and dragging the object(s) across the screen. The number of fingers isn't important since the average delta position of the fingers is calculated for as long as the finger(s) are active.

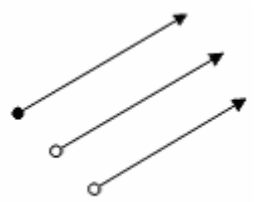

Figure 15: Move

The listener gets the current *Position* - a struct that holds nothing but the x - and y - screen coordinates. These are a part of the *TouchData* object, they are easy to detect.
The *pointOfOrigin* variable gets updated with every new position.

MultiSelect is a gesture that makes it possible to select a number of objects in one stroke. Just as we are used to from traditional interfaces, only that the multi-touch interface invites for new ways to do it. A line is drawn around the objects that are to be selected. One definition of whether an object is inside a given area is the Jordan Curve Theorem. It claims that a particular point is inside a line if, for any ray from this point's position, there is an odd number of crossings of the ray with the line along one of the x - or y - axis.

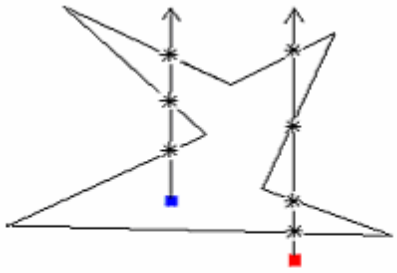

Figure 16: MultiSelect

From the blue object to the left in Figure 16, three crossings are found, i.e. the object is inside. From the red object on the right, there are four occurrences of line crossings, which mean that it is outside of the area by definition. The listener will receive a list of *Bbox*'es representing the objects that are affected by the gesture.

Scale is the last of the defined gestures. It is easily detected by analyzing the *DeltaX* and *DeltaY* of the *TouchData*. If the deltas of the finger movement have different signs, they are heading in different directions. If at least two fingers are affecting a *Bbox* in this manner, a *Scale* gesture is being performed on the corresponding object. A *Scale* gesture can as well be treated as a zoom-effect.

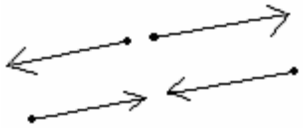

Figure 17: Scale

The listener receives a multiplying factor, with which the object will be scaled. The *scaleFactor* variable is calculated for as long as the finger is active. If the *scaleFactor* > 0 the fingers are moving apart, else if *scaleFactor* < 0, they are moving together.

Note that the *Bbox* representations in every one of the gestures as well might correspond to the main window of the application. This mean that the *Scale* gesture might be treated as a zoom effect and the *Move* gesture can be used as a pan effect.
The gestures themselves aren't predefined in any way. It is up to the listener what to do with them.

2.3 Implementation

The design model described in 2.2 is the final implementation. Here are the pre-conditions for the different gesture events listed.

On *Rotate*

```
if( (data1.dX * data2.dX < 0) && (data1.dY * data2.dY < 0) )
```

To perform this gesture, the fingers will have to have different headings, both in the x - and y - direction. The *dX* and *dY* of the *TouchData* have either a positive or negative value, depending on the current direction.

On *Scale*

```
if( (data1.dX * data2.dX < 0) || (data1.dY * data2.dY < 0) )
```

The fingers have to be heading in separate directions in either direction.

On *Move*

```
if( (data1.dX * data2.dX > 0) || (data1.dY * data2.dY > 0) )
```

Every finger involved in the gesture has to be aligned to roughly the same heading.

On *MultiSelect*, analyzing is performed in `fingerUp`

```
for( std::vector<GestureListener*>::iterator it =
m_listListeners.begin(); it != m_listListeners.end(); it++ )
    {
          if( (*it)->hasFocus )
          {
                for(std::list<Bbox*>::iterator it2 = (*it)->
                m_listBbox.begin(); it2 != (*it)->m_listBbox.end();
                it2++ )
                {
                      if( IsInside(*it2) )
```

`IsInside` performs a bitwise check to see if the first bit of the number of crosses is equal to 1, which is a condition that has to be true for an odd number of line crossings along the axis out of the current *Bbox*.

The gesture of *Point* is performed immediately in the `fingerDown` function. There isn't really a precondition for it, other than the occurrence of a touch event. The finger will most probably take part in the creation of other gestures during the `fingerUpdate` function though. As long as a finger is active, it is a part of the objects under observation.

With this model it is necessary to perform a 4 dimensional iteration to locate the correct *Bbox* and provide the listener who owns the belonging object with information on the current gesture event and which object it affects. The *hasFocus* variable of the *pAppWindow* makes sure that the application that gets an event also should have it. If `pAppWindow->hasFocus` is false, it doesn't matter if the application's `mainWindow->HasPoint( data.X, data.Y )` is true. This is to prevent that overlapping applications (where both `mainWindow->HasPoint` is true), will get the gesture event. The if statement also provides for optimization in the way that the library only checks for gestures made on the application that *hasFocus* and *HasPoint*.

```
// every listener
for( std::vector<GestureListener*>::iterator it =
m_listListeners.begin(); it != m_listListeners.end(); it++ )
{
     if( (*it)->pAppWindow->hasFocus &&
         (*it)->pAppWindow->mainWindow->HasPoint( data.X, data.Y ) )
     {
           // every finger
           for( std::map<int, GestureElement>::iterator it2 =
           fingerMap.begin(); it2 != fingerMap.end(); it2++ )
           {
                 // every Bbox
                 for( std::list<Bbox*>::iterator it3 = (*it)->
                 m_listBbox.begin(); it3 != (*it)->m_listBbox.end();
                 it3++ )
                 {
                       // every Gesture
                       for( std::list<Gesture*>::iterator it4 =
                       fingerMap[data.ID].m_listEvents.begin(); it4 !=
                       fingerMap[data.ID].m_listEvents.end(); it4++ )
                       {
                             ....
                       }
                 }
           }
     }
}
```

3 Results and Future Work

The gesture library is working and the gestures are being registered correctly. I haven't had the opportunity to test the library in a situation outside of the testing environment at this time. I set up a test application for the development phase, so I know that everything works as intended. The listener gets the gesture events and the gesture analyzing part works as planned. These coming days we will combine all the parts into a unity - The hardware interface, the applications and this gesture library, with the addition of Niklas Brolin's gesture recognition part.

The project group has come up with some great ideas of applications taking advantage of the FTIR technology, and I am sure there are some exciting applications in the pipeline that will be realized in a near time. This has been the first project in this area within the university and we have gained some valuable experience and drawn some conclusions along the way. Hopefully coming projects will benefit from that.

4 Discussion

Gesture movement definition will definitely become more and more important for several of the future development within the areas of touch screen and multi-touch. There have been attempts on standardizing gestures - attempts which aren't bad at all. However, since the possibility is there, I think we should leave the kind of "general" gesturing (by general I mean the gesture types presented here) as open as possible and not predetermined in any way - at least not in terms of how many fingers should be pressed or which way to rotate to generate gestures that has a real physical representation, with all the intuitiveness that comes with it. The gestures of *Move* and *Rotate* do come to mind. There are of course areas where some kind of definition of the gesture input becomes necessary [4, 5], but the general gesture approach should be kept on as a natural way as possible.

The implementation and the design sometimes went hand in hand. Finally it landed in the design model as described in 2.2. Since I had to perform frequent testing of the library, I needed to be on location for all the implementation and have access to the hardware prototype. Most of the time went to implementing/designing the library itself. As I got further into the implementation of a certain piece, I sometimes realized that the design model didn't stand and had to go back to the drawing table for re-designing. That's the story of development.

The area of multi-touch is definitely an area which will expand in the future. Since the technology is here, I can't see why we should remain at the traditional single input interfaces for long.

5 Abbreviations

FTIR	-	Frustrated Total Internal Reflection
LED	-	Light Emitting Diode
gsCEPT	-	gaming studies, Computer Entertainment and Production Techniques
ECTS	-	European Credit Transfer and Accumulation System

6 References

[1] Andrew D Wilson — *TouchLight: An Imaging Touch Screen and Display for Gesture-Based Interaction.* International Conference on Multimodal Interfaces, October 13–15, 2004, State College, Pennsylvania, USA

[2] Jefferson Y. Han — *Low-Cost Multi-Touch Sensing through Frustrated Total Internal Reflection.* Symposium on User Interface Software and Technology, October 23-27, 2005, Seattle, Washington, USA

[3] Vladimir I. Pavlovic, Rajeev Sharma and Thomas S. Huang — *Visual Interpretation of Hand Gestures for Human-Computer Interaction: A Review* Pattern Analysis and Machine Intelligence, IEEE Transactions on, Publication Date: Jul 1997

[4] Hrvoje Benko, Andrew D. Wilson and Patrick Baudisch — *Precise Selection Techniques for Multi-Touch Screens* Conference on Human Factors in Computing Systems, Proceedings of the SIGCHI conference on Human Factors in computing systems, Montréal, Québec, Canada, Year of Publication: 2006

[5] Dean Rubine — *Specifying Gestures by Example* International Conference on Computer Graphics and Interactive Techniques, Proceedings of the 18th annual conference on Computer graphics and interactive techniques, Year of Publication: 1991

[6] Paul Dietz and Darren Leigh — *DiamondTouch: A Multi-User Touch Technology* Symposium on User Interface Software and Technology, Proceedings of the 14th annual ACM symposium on User interface software and technology, Orlando, Florida, Year of Publication: 2001