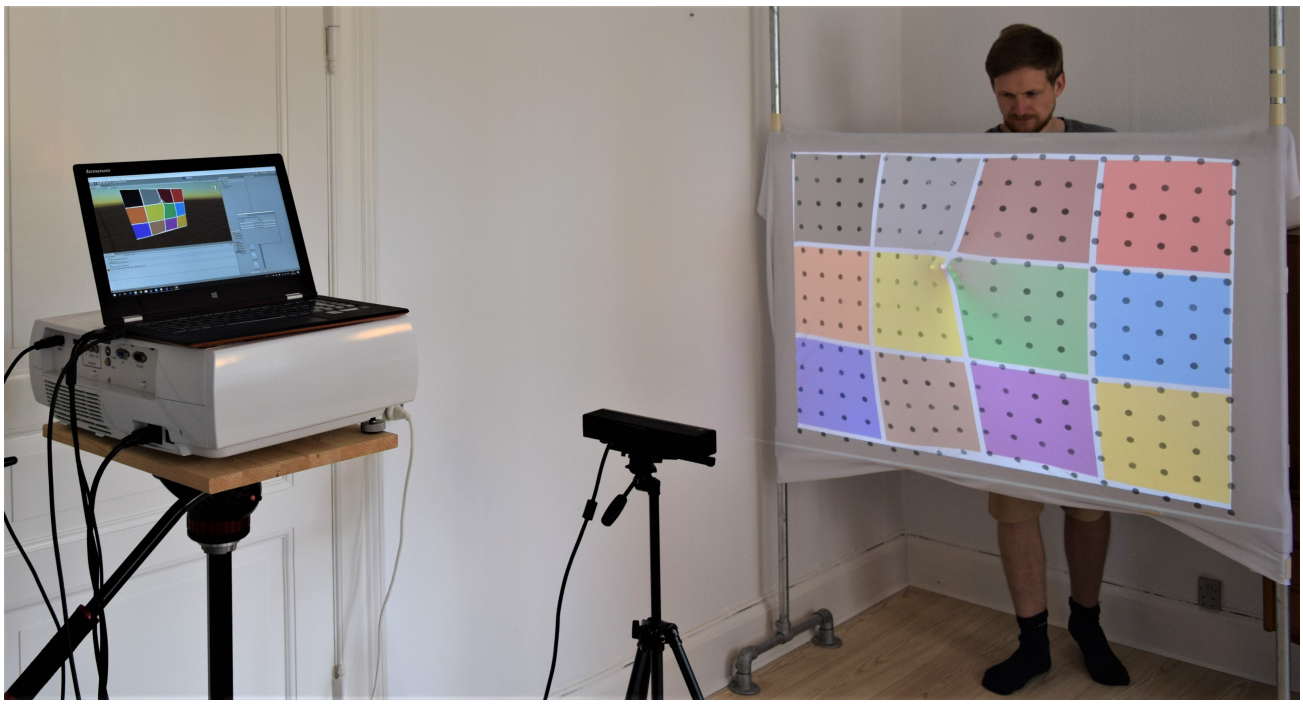
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Master Thesis

Peter Meyer & Mads Torhus

FlexIO

- A Deformable Screen

Advisor: Sebastian Boring

Handed in: May 22, 2018

1 Abstract

A fully organic deformable screen might be a thing of the future, but maybe we can simulate one with currently available commodity hardware. This thesis revolves around the idea of a fully deformable screen prototype and all the challenges of producing one. The prototype is realized using a so-called ProCam setup, and a deformable canvas for projection. The canvas is made of a stretchable piece of fabric attached to a frame. By tracking the surface of the physical canvas with a camera, we are able to create a virtual 3D model of it, which we can project back onto the physical canvas. The 3D model is deformed according to the deformations of the physical screen, allowing the user to directly deform the projected image.

We found that the project is divided into multiple fields of research that has to come together. First of all, we needed to be able to track the screen, which we do with the help of reflective markers placed in a grid structure. This, however, presents the problem of multi-object tracking, which becomes especially difficult when dealing with moving occluded objects and their reappearance. The thesis presents the use of the Hungarian algorithm which we chose to combat the issue, as well as our experience with other approaches. However, the matching of tracking points between frames of the camera remains an issue throughout the project.

The main focus in this project is to look for the best ways of estimating the location of occluded tracking points, as the screen should still be displayed properly even if we cannot visibly track its movement due to occlusion. We present three different approaches to estimate the position of occluded tracking markers which we compare and evaluate. The evaluation shows that an approach based on extrapolation on neighboring markers performs the best in our experiment.

Using the Unity game engine we are able to render the 3D model of the screen, on which we can place images or videos and deform according to the tracked markers(visible and estimated). For the ProCam setup to show the 3D model correctly on the physical screen, it needs to be calibrated. This is realized by adapting parts of an existing ProCam toolkit to fit our specific use.

Our tracking algorithm has some drawbacks and limitation that we expand upon throughout the thesis, however taking these into account, the prototype does allow for further investigation into the interaction space of a deformable screen. Lastly, we will present specific recommendations for how the prototype can be improved in future work.

Contents

List of Figures

List of Tables

2 Introduction

2.1 Motivation

In recent years, we have seen a lot of development and advances in research regarding the way we display and interact with information. Much of the research focuses on moving away from the 2D interaction of data, and towards natural 3D interactions.
This can be achieved in a number of different ways, including Virtual Reality (VR), Augmented Reality (AR) as well as deformable displays.
Many novel approaches have been created as well, often for a single purpose or test, projects like Levipath[33] which lets a small ball visualize a 3D path using ultrasonic sounds to move the ball, and Dynamic Pie Charts[48] which shows chart data in 3D, using movable LED illuminated pins.
However achieving a general purpose deformable interactable display is one field that is still in its early stages, mainly because of hardware limitations. Even though organic light-emitting diode (OLED) screens have allowed a certain amount of flex on what is considered a standard display, it is still far from being truly deformable.
While this is the case, we can easily expect that future advances will allow for complete deformability, in which case we will have a very different way of being able to interact with the screen.
For now, these types of interactions are an area in which not much research has been done due to the hardware limitations. However we hope to allow the investigation into how we can interact with a deformable display, that can not only bend but also stretch and provide direct haptic feedback[1] to the user.
As we can not at this point deform an active screen, we would like to simulate a deformable screen by projecting onto a deformable canvas made of stretchable fabric. By tracking the surface of the physical canvas with a camera, we would then be able to create a virtual 3D model of it, which we can project back onto the physical canvas. This will allow for the user to manipulate the 3D model directly by deforming the canvas, as well as provide haptic feedback in the form of resistance when the canvas is deformed.
Projection onto a stationary object can be challenging enough when having to deal with a non-flat surface. This is however achieved by Projection mapping, matching the outgoing projection to the position and shape of the target object. But allowing for the target object to deform and move, presents adds to the complexity. Additionally, a fully deformable screen should allow the user to grab the canvas into the hands, which will introduce self-occlusion of the canvas. Being able to correctly show the deformed screen while having to handle such occlusions are even more of a challenge.
While a few people have been working with the idea of the deformable displays, and multiple examples have shown up within the past years[45][54][39], we have yet to fully understand features and interactions, that the 3D capabilities and haptic feedback of a deformable screen provide. We are presented with new ways of interacting with information that can improve our workflow, or just give rise to fun applications. If implemented correctly, a deformable screen can allow direct manipulation of a virtual object in the physical space. It has taken years to understand the interactions made possible by each leap in technology, dating all the way back to the computer terminal, then to the mouse, and more recently with touch screen in 2D. Lately, Augmented Reality(AR) and Virtual Reality(VR) commodity products allow the user to get a 3D experience but does however mostly require the user to wear special headsets and handheld controllers for being able to manipulate objects in the 3D world. Also, haptic feedback for making the experience realistic in AR/VR is an entire field of research, in which no general satisfactory solution yet has been implemented.
Thus we would like to focus on creating a solution that does not require the user to wear any equipment and provide haptic feedback when the user interacts with it.
Existing solutions that meet these demands, and lets the user physically manipulate 3D models directly, shows that we do not yet fully understand the interaction space of working with such solution. This indicates that the translation between intuitive interaction and digital models might not be easy. Examples of use-cases used in existing projects are often picture, video and navigation interaction(e.g.[9][54][58][46]), where especially video interaction is not a natural occurrence in the physical world, and further research is needed to figure out what is

[1] `https://teslasuit.io/blog/haptic-feedback/haptic_feedback` accessed: 22/04-2018

a natural way of interacting with this.
Understanding interaction techniques are very important and a field of constant research. The work of Troiano et al.[50] investigates the interaction space of deformable screens by performing a guessability user study, where 17 participants were presented with 29 tasks inspired by[56][54][36] to solve. The task was solved using a screen made up of a frame with a flexible membrane. Instructions and stationary objects to manipulate were projected onto the membrane from the backside, so the participants had to imagine the actual manipulation of the objects, and solve the task using gestures they themselves found natural. The researchers identified 29 groups of interactions, indicating that the interaction space is rather complicated. Our work will not focus on the interactions themselves or applications. We will instead focus on developing a technical solution that enables researchers to investigate the interaction space of deformable screens further in a more realistic environment, allowing actual direct manipulation of the objects on the screen.

As mentioned, doing research on the subject requires a setup that does not yet truly exist, at least not with any standards for implementation. Each project we have encountered in our research has created their own setups from the ground up, and a big contributor to making these projects possible is the physical equipment used. Such as high-speed, high-resolution cameras, and high refresh rate projectors running from a high-grade computer to run the often custom tracking algorithms needed. These demands do slow down the development process, as not everyone can get a hold of this kind of equipment, and everyone has to develop their own backend.
We want to bring these demands down, by utilizing commodity hardware in the cheaper spectrum and working within the restrictions that they provide. From here we want to create a backbone tracking framework, which with minimal requirements will allow other developers or researchers to use our setup and implementation for further investigations into deformable screens.
For this to be feasible, we need a solution which can track the entire screen, while balancing both precision and processing cost, so that we can provide consistent tracking of deformations of the screen, and correctly show a 3D model deformed accordingly.
The project will require research into tracking algorithms, including algorithms for estimating the location of occluded parts of the screen.
Using both computer vision and physical models, to alter and show data in a real 3D space, while getting to work with physical aspects such as an actual screen, and a projector-camera set up is a personal motivation for us to tackle this project. If we can achieve a usable framework to improve future research, this only works to further the motivation. We aim for the outcome will be beneficial to others in the field.

2.2 Problem Description

A working prototype realizing a Deformable Screen requires a few different components to work. First and foremost we need to develop a robust tracking system for the screen. This has to be done according to the restrictions put in place by using only commodity hardware. For such hardware as cameras and projectors, this will often mean low framerates and high input lag. While it gives certain restrictions, it also sets a fairly high bar for processing time of the tracking algorithm.

The tracking system is required to handle any occlusions that might occur when deforming the cloth canvas of the screen. Thus in addition to tracking visible parts of the screen, we also need to provide estimates of occluded parts, to be able to reproduce a 3D model correctly. For these estimations to be reliable, they must closely model the actual behavior of such a fabric screen.
The tracking information needs to be translated into a 3D model on which we can show graphical content, and then project correctly back onto the canvas.
To get a projection that matches the screen, a calibration step is needed to compensate for the difference in the placement and perspectives of the projector and camera.
In the end, the project aims to be an easy to use, robust, and customizable framework for tracking a deformable screen, giving the users the benefit of direct haptic feedback. While we might not create an interface for every

camera, it should be possible for future researchers and developers to expand our project to work with the camera they intend to use. Also, it should be easy tweak parameters of the program to match their desired screen.
At deadline, a working prototype implementing the above mentioned should be available both in the physical form and with the programmed framework to support it.

2.3 Approach

We have already hinted how we will realize parts of the system, this section aims the provide a better overview as well as allowing us to elaborate on our approach for creating a Deformable Screen.
We will realize a prototype of a deformable screen using a so-called ProCam setup and a deformable canvas for projection. The ProCam setup consists of a camera for tracking the surface of the screen, which we will translate into a virtual 3D model, and project it back onto the screen using the projector. For us to be able to track the surface, we will use a 3D camera providing us with depth information.
The canvas will consist of a stretchable and retractable piece of fabric attached to a rigid frame. In order to make the surface of the canvas trackable, we will add a grid of retro-reflective markers onto the surface. Besides giving us depth information, by choosing a 3D camera that uses Infrared(IR) light for obtaining the depth information, we can use the reflection from the markers in an IR-image to locate the markers 2D coordinates. Combining this to the 3D position of the markers, we will create a computer graphics 3D mesh, on which we can show various graphical content. The mesh will represent our virtual model which, as explained, is projected back onto the screen.
We will now go into further details about the software we need to develop for being able to realize the deformable screen. We have divided this into three individual tasks:

2.3.1 Tracking

We need a tracking algorithm allowing us to track the screen by using a combined infrared(IR)- and depth-camera for tracking the mentioned reflective markers. To keep track of the individual markers we will assign them ID's, and match the markers between successive frames based on the markers 3D position. Matching the ID's between frames is a challenge, especially when handling many tracked objects that are all more or less identical. From now on, we will call this task of matching the markers between frames "Point Matching". Furthermore, we will use the words "marker" and "point" interchangeably when referring the markers.

2.3.2 Occlusion

When having occluded markers, we need to do two things; figure out which markers are no longer visible, and estimating the position of them. We will do that by defining estimation models that utilize knowledge about behavior specific to a fabric screen. As there exist no such models we will have to define them ourselves, and we will try our three different approaches.

2.3.3 Projection

With the 3D positions of the grid of markers at hand, we will perform a triangulation on them to construct a computer graphics 3D mesh forming the 3D model of the screen. Additionally will make it possible to add various graphical content to the screen by changing the texture of the mesh. When the 3D model is completed, we can project it onto the screen, needing only to make sure, that the relationship between camera and projector is accounted for. This will be obtained through the process of ProCam calibration.
Bringing all of this together represents our FlexIO solution.

2.4 Contributions

This project contributes the FlexIO framework for realizing a deformable screen. We can divide the deliverables of the project into:

- The physical screen consisting of the frame and the trackable canvas.
- A program for tracking the surface of the deformable screen.
- A program translating the tracked information into a computer graphics 3D model that can be projected onto the screen.

Also, the report presents the development process and our considerations for the designing these components. Most importantly for the physical aspects, this includes a discussion of the design of markers for tracking, the screen canvas, and the density of markers on the canvas.
An important factor in the project is that we have realized a deformable screen for real-time use utilizing only commodity hardware. Likewise, the report presents our considerations to meet these restrictions.
Furthermore, the report presents three different models for the estimation of occluded markers in the tracking program. We have implemented all the three estimation models, developed a testing framework, and evaluated the performance of these models.
Throughout the project, we have focused on making the framework easy for others to expand upon, as to further drive research in the field.

3 Overview of the Thesis

In this thesis, we will aim to explain the different parts that make up the FlexIO framework. Going through theory and implementation needed to create the final prototype as well as evaluate the capabilities and weak points of the product.

- **Section 4** Describes related work, exploring both the problems and the possible solutions that other researchers already have investigated, which will set the scene for the project considerations we have been through.
- **Section 5** Will go through the physical design needed to make the project work, mainly the physical aspects of the screen, but also the camera and projector setup.
- **Section 6** In System Design, we go through the most important aspects of theory used for implementing the FlexIO system.
- **Section 7** The implementation section will go into details about the actual implementation and tools used to create the framework.
- **Section 8** Shows photos of the Deformable Screen in use.
- **Section 9** Breaking Point Analysis, addresses boundary testing to investigate the limitations of our tracking system.
- **Section 10** In the Estimation Evaluation section we will evaluate our three estimation models for handling occlusions.
- **Section 11** Evaluation of Efficiency, will go through the run times for different parts of the code, revealing things to improve, as well as the theoretical capabilities for real-time operations.
- **Section 12** Will go through the results and observations that we have found throughout the project and discussing how they relate to the theory we have used.
- **Section 13** Finally we will conclude the project, by talking about the end result, as well as propose future work that might be done to improve our prototype.

4 Related Work

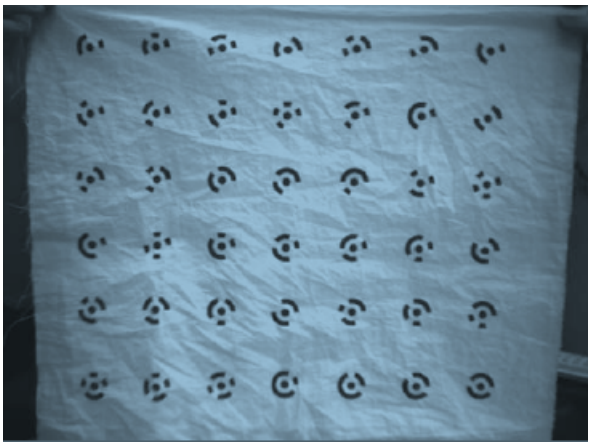
Other researchers have explored projects that in some way resembles parts of ours, and drawing from their experiences is the easiest way to avoid trying things, others have already proven to be less effective while getting an idea of the possibilities that are available for solving each problem.

4.1 Tracking the Cloth

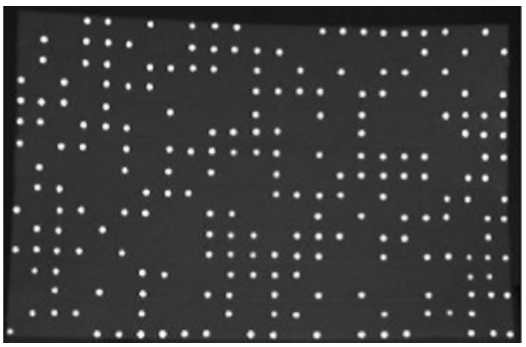
First of all, we needed to know how we can track the screen canvas. The act of tracking an object in real-time is challenging an presents multiple problems to handle. These problems are often defined by the specifics of the case. In our case, we want to track a cloth screen. The cloth makes for a very uniform and featureless item to track making the task rather challenging.

4.1.1 Unique Markers

In most cases when tracking a single large or uniform object, defining features are added to the object to ease the tracking. This is often done by adding certain markers to help define the surface. There are however many different types of markers and different ways of utilizing them. One approach is to let the markers store and give as much information as possible[5][31]. In these cases, markers are either completely unique as seen in figure 1a, or if enough unique markers cannot be created, then markers with the same design will be placed far enough away from each other to make them distinguishable just by location.



(a) Image of unique markers used by Bradly et al.[5].



(b) Image of unique grid structure used by Fujimoto et al. [31].

Figure 1: Examples of different unique marker setups for tracking the uniform plane.

The idea behind these markers is the same as with Quick Response (QR) codes. Using their unique shape, they can store information relevant for their trackings, such as relative placement to other such markers or the dimensions of the tracked surface. While the markers are great for tracking reliably, and at high speed, they have definite drawbacks when it comes to occlusion. Occluding a point just by a bit will most likely cause all information in that marker to be lost, and moving two markers very close, or even making them overlap, could give misinformation.
The work of Fujimoto et al.[12] takes the idea a bit further though, creating a nonuniform but preset array of markers that allow them to find features in the marker array as seen on figure 1b. If they miss certain markers or sections of the screen, a feature search will allow them to find the most likely candidate for each visible section, allowing for dynamic distinct markers.
These projects, however, share a weakness that makes them practically unusable for us in their current form, in that they do not allow stretching, as this will alter the distinct markers or marker arrays too much to be read properly. Stretching is either a feature they ignore, or mark down as future work. We want, however, our screen

to be as deformable as possible. In practice, this means that most information should not come from specific markers, but from the information, we can obtain, without their help.

4.1.2 Random Markers

Some projects try to take it to the other end of the spectrum, not using any sort of specific markers or patterns[39][51], and they just add the markers randomly onto the surface they want to track. The methods utilized in these examples both rely on creating a mesh matching the size of the surface of the object they want to track, before using the movement of the markers to determine deformations. Both of the articles are very brief, and sparsely documented, but both use the visible features for tracking while ignoring what is occluded. This means that they either cannot handle occlusion at all, or they rely heavily upon estimating the areas for which they have no markers, areas where the defining features are missing due to occlusion. The Deforme project[39], which augments a deformable mass with projection, does, however, try to handle this in an interesting way. They create a computer graphics mesh that they can deform, and do so not by the position of the features in a new frame, but only using the movement vectors between two frames. By only tracking the movement of the features between two single frames, and then applying this movement to the predefined virtual mesh via interpolation, they can locate a completely new set of features for the next couple of frames, so that they do not rely on features that might have been occluded. While this gives an easy way to avoid some of the problems that occlusion might bring, the interpolation on the mesh does however also mean that they lose a large amount of precision, as occlusion happening during deformation, obscures the feature points that are actually moved. This results in a fluid deformation, lacking any provable precision. As the article does not hold any information on performed tests, our evaluation of the project must be based on the proposed method and the supporting video [2].
A related approach which has a large mass of random points, that in this case is represented by the features of a picture, is presented by Hilsmann and Eisert [15]. In this project, they track part of a t-shirt, finding the shirt by its color, and then finding a heavily textured region, such as a printed picture on the shirt. From here they utilize an optical-flow method looking only at the movements, filtering out changes in lighting and color, so that changes between frames are mostly movements of the cloth. The movements are transferred to an initialized mesh, that can then be re-textured virtually. The approach does not estimate occluded parts, but does however still handle self-occlusion by shrinking the occluded areas as they get occluded, not removing them, but letting them become small enough to be irrelevant, so that they never disappear from the mesh, and holds the same relative position. The approach seems robust and holds up, but it is stated that large deformations are hard to handle. the product is only virtual, and the results only present at the computer monitor, so it is hard to say how it would work when adding a projection onto the tracked surface.
Many of the methods utilizing feature point matching, image deformations, and optical-flow constraints[37] [13][21], are generally not very robust, and have issues with the computational cost, not allowing for real-time use which would be a big issue for us, as a non-real-time screen is not very useful for a deformable screen. Even the ones that can run real-time relies too much on estimations as they have no known control points to match up against which makes the screen less precise, and leaves visible deformations to be handled by estimations instead of direct information.

4.1.3 Matrix Grids

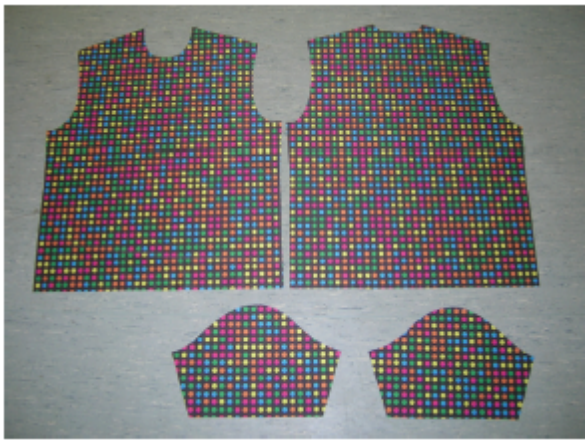
The last marker structure we have observed is the matrix grids. The markers themselves carry no information, but their position is initially set in ordered rows and columns with equal distances between each marker[54][52]. One of the benefits is that we have a known initial state, which can be used as a starting point. While the markers do not carry information about their relative position, the fact that markers are placed at certain intervals still allow us to know the offset they must have moved from the original position. The tracking of these markers are straightforward for the initial state, however, the main concern is when deformations and occlusions occur. Having to match each marker in a new frame to one in the previous frame is a difficult problem when the markers

[2] `https://www.youtube.com/watch?v=HLABh1jZnDQ` accessed: 10/04-2018

are not distinct. Multi-Object Tracking (MOT) in general is an area of research, that has a lot of focus, and one for which people constantly try to improve[24]. When tracking using matrix grids, the logical choice is to look at two successive frames and identify the markers in the new frame, by their closest marker in the old frame, however with large movements in the tracked surface, this becomes unreliable.
Often a filter will be applied to try and foresee where markers are moving. Particle models and filters such as the Kalman filter is widely used[10][28][25], however, they expect tracked objects to move in a certain trajectory. When a ball is thrown through the air, these methods can estimate that it will keep going in the same direction, with a certain drop off in velocity and height due to resistance and gravity. Tracking markers on a deformable screen when people use the screen, however, presents a very different behavior as the movements are not so easily predicted, and can change suddenly between frames. One solution to counteract large movements and deformations is by having less time in between frames by using high-speed cameras. However such cameras are expensive and not something everybody has access to.

4.1.4 Multi-Cam Setups

As well as using expensive high-speed cameras to improve the tracking, a lot of attempts has been done, trying to track or motion capture clothing using multi-camera setups[38][42] with specialized cameras. This is another expensive method that is not easily accessible, and while it provides certain benefits in that we can remove some occlusions due to we see the object from several viewpoints, and get more information on the cloth's actual movements, it often requires very costly computations. The heavy algorithms used for creating animations of cloth are not able to run real-time, without sacrificing accuracy, which is why some researchers have experimented with motion capture. The computations for the motion capture setup can achieve real-time speeds, with the right equipment, and with marker or feature point detection such as used by Scholz et al.[42] seen on figure 2a, it can be done without the standard physical markers or attachable markers used for motion capture as seen on figure 2b[3], which would otherwise alter the movement of the fabric.



(a) Image of color markers used by scholz et al.[42].

(b) Image of standard motion capture marker [3].

Figure 2: Examples of markers used for motion capture.

The motion capture techniques differ widely according to the methods used but share some similarities. For the motion capture to be robust, a great deal of work is needed in the setup and calibration phase. A known feature set such as color-coded markers, gives a more stable outcome, while using the existing features of the cloth makes it less stable, but more agile as it can use any well-textured fabric.
While the multi-cam approach has many benefits, in the amount of data they provide, they are generally costly

[3] `http://mocapsolutions.com/wp-content/uploads/2012/11/19mm-SBXV-600.jpg` accessed: 10/04-2018

both in terms of computation and money, which differs from our initial idea of creating an easy to duplicate, cheap, and real-time solution.

4.2 Dealing with Occlusion

When having to deal with occlusion, we need to be able to estimate the position of any marker that is not visible. There are many ways of achieving this, but mainly it comes down to modeling the fabric with the information we have, can gather, or assume. In related work, the projects that deal with tracking and modeling like ours, have many different approaches. Including the projects that handle only virtually modeling of fabric, there are even more ways of tackling the problem. When looking at the overall problem of occlusion handling, and not the physical aspects of using cloth, a linear approach estimation is a good starting point.

4.2.1 Linear Estimation

Deformable screens can be many things, and often the screen consists of a flexible but relatively rigid material such as paper or something close to it[45][51][32][12][19]. This means that we can bend it, fold and move it, but deformations requiring stretching, such as shearing or local single axis deformations, are not possible. The conclusion being, that the relative distance between markers along the surface will stay the same throughout deformations leaving us a linear aspect to work from.
One linear model that can be used for estimating occluded points is a simple interpolation[12]. In this approach, they take the visible markers around an occluded one, and as the distance between each neighboring point is known, the occluded point is simply estimated to be positioned in between the visible points. The method is not completely accurate due to the unknown curvature of the material between markers, and the error it might cause accumulates when estimating markers using already estimated neighbors. Because of this, it is important to minimize the error of each estimation by taking the most reliable estimations first, these being the markers with all eight neighbors visible, before estimating points with less visible neighbors.
In the work by Narita et. al.[32] they use extrapolation on the neighboring points in the cardinal directions(North, East, South, West), by normalizing the position given by taking the sum of each cardinal points times two, minus the cardinal points neighbor in the same direction. This method works better than interpolation when not all cardinals are visible, but does still produce some error, leaving it with the same drawbacks for accumulated errors. Their solution for this being that occluded markers without visible neighbors is discarded until they become visible again. One thing to note when using extrapolation, however, is that we actually see stretching play a part, as we use cardinal points further out, however depending on where the stretch is happening, this will be a good or bad thing for the estimation. A stretch directly between the first and second cardinal point will estimate a stretch on the occluded point which is not there. In this project, they evaluate the extrapolation method for tracking a non-stretchable paper-screen. They also try the same approach on a stretchable material in the form of a t-shirt, but unfortunately without any official evaluation. We have found a video [4] where the researchers show the method applied to the t-shirt which looks rather promising. However, they do only try a light stretch deformation, while making sure not to occlude the tracked area with their hands. Furthermore, we cannot assess the precision of the system from a short video alone.
Linear estimations in general, are easy to use and their estimations are rather robust, the problem, however, is that cloth is hardly linear in its movements, and this can lead to poor estimations especially for larger deformations, as the error can build up if the material is not behaving linearly.

4.2.2 Physical Models

While we have not seen many projects with the goal of a deformable surface detection that utilizes physical models for the estimation of occlusions, there is an area of research devoted to cloth simulation from which point estimation is, of course, a large part, as it is the main aspect of simulation in general.
The models try to use as much information about cloth as possible, how it is woven, the stretchability of the

[4] `https://www.youtube.com/watch?v=-bh1MHuA5jU` accessed: 10/04-2018

thread along with the physical laws that all things must abide by.
The most commonly seen approach is variations on the Mass Spring model[16][21], which constraints particles in the cloth between each other using springs for which force can be added in various form. Using twelve neighboring points, it provides a balance between structure and local deformation.
There have been other attempts at creating physical models from actually observing cloth [53], using a machine learning approach, however, this will be very dependent on the fabric used, and not so much a general model for cloth. Each created data-driven model, will, of course, be highly customized for a single piece of fabric, adding the need for calibration of each new piece.
Physical models have the benefits of being flexible and allow for a wide array of movements being estimated closer to the real world, given the variables are set correctly. They are however harder to handle and requires tuning to behave lifelike. However, the models are meant for estimating all points in a mesh for each frame, and not just a few occluded points in a pattern, but altering the models for our purpose might still prove to retain the positive qualities of the approach.

4.3 Summarization

While looking through the related work, we had to make some initial choices based on our premise.
We already knew that we wanted something that was inexpensive and easy for others to replicate. Using IR reflective markers would allow us to have points on the fabric which were easily trackable when visible. However, with our project being centered closely around deformation, the downsides of both the unique markers and feature-based tracking such as the inability to reliable track deformations especially during occlusion these were ruled out. Our choice of markers and their position fell upon non-unique markers in a matrix grid. Using this approach we can visibly track each marker without occlusions, as they will fit into the predefined grid, and we can use the markers to directly match up against a mesh created for our 3D output. The difficulties, however, comes with the addition of occlusion and large deformations as multi-object tracking of indistinct markers is a tough problem without a universal solution, one for which a robust approach might be difficult to achieve. However with the right estimation of occluded markers, we might be able to use a less robust approach, but give it the best chances of success. This means, that we were not going to go with a single estimation method, but rather with multiple, to find the best results.

5 Physical Design

This section describes the considerations and design choices for the physical implementation of FlexIO. The physical setup of the system consists of a combined Depth/IR-camera, a projector and a canvas working as a screen seen in figure 3 and 4. Figure 3 shows the screen from the back side where it has a grid of dot markers. The screen itself is made of a frame with a large cloth canvas.
The camera captures the 3D coordinates of the entire frame which is sent to a computer that uses the dot markers to create a graphical 3D model of the screen's surface. The 3D-model can be overlaid with various graphics e.g. a User-Interface(UI) and is projected back onto the screen.

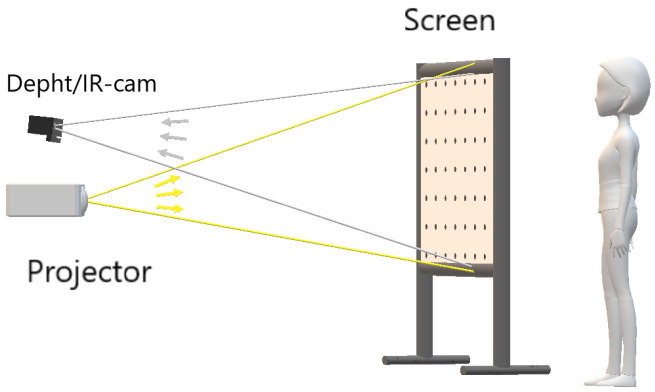


Figure 3: The FlexIO setup seen from the backside of the screen.

Figure 4 shows a slightly rotated view of the setup where we can now see what the user, in front of the screen, sees. The 3D-model is projected onto and through the canvas for the user to see. The user can manipulate the screen with her hands, causing the 3D-model to deform, and thus also the projected image.

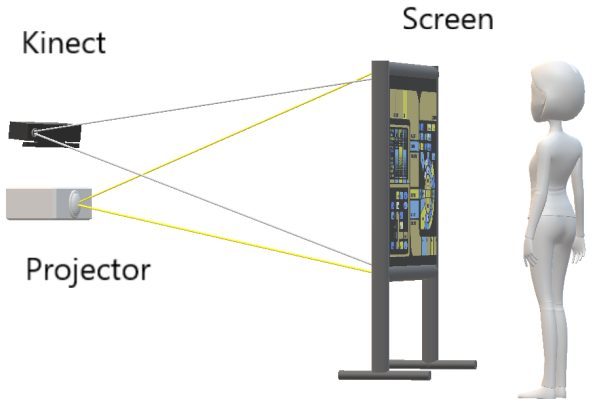


Figure 4: The FlexIO setup seen from the front of the screen.

Besides the elements seen in the figures, the camera and projector are connected to a PC for processing. The computer used in our project has the following specifications:

CPU	i7-4790K @ 4 GHz
GPU	NVIDIA GeForce GTX 960 MSI Twin Frozr
RAM	24 GB DDR3
OS	Windows 10 Home
Storage	Samsung 840 PRO SSD 512 GB

Table 1: Specifications of the PC used in our project.

5.1 The Screen Frame

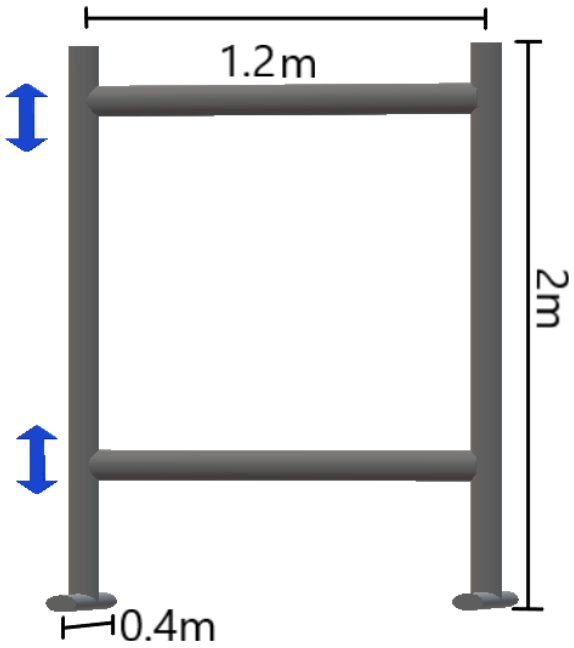
The first, very rough, prototype of the screen consisted of bedding sheet, working as the canvas, nailed onto a wooden frame as seen in figure 5. This screen was exclusively made to get started on the project, allowing us to work on the tracking early on.

(a) The front of the screen. *(b) Backside of the screen.*

Figure 5: Prototype of our deformable screen with width 90 cm and height 80 cm.

Once we had a good understanding of what we needed from a screen and had acquired materials, we constructed a better prototype using a large piece of elastic fabric strapped around a much thinner rigid rectangular frame. To provide rigidity of the frame, we constructed it using plumbing pipes connected by pipe clamps as seen in figure 6a

(a) The actual frame for our final prototype.



(b) Dimensions of the frame.

Figure 6: The frame for our deformable screen.

The metal-clamps of the frame can easily be loosened to adjust the frame's height to accommodate users of varying heights. Figure 6b and 6 shows the frame where the screen is lowered differently. Furthermore, this design enables us to experiment with different screen sizes and aspect ratios.
The stretchable fabric is wrapped around the frame as seen in figure 7

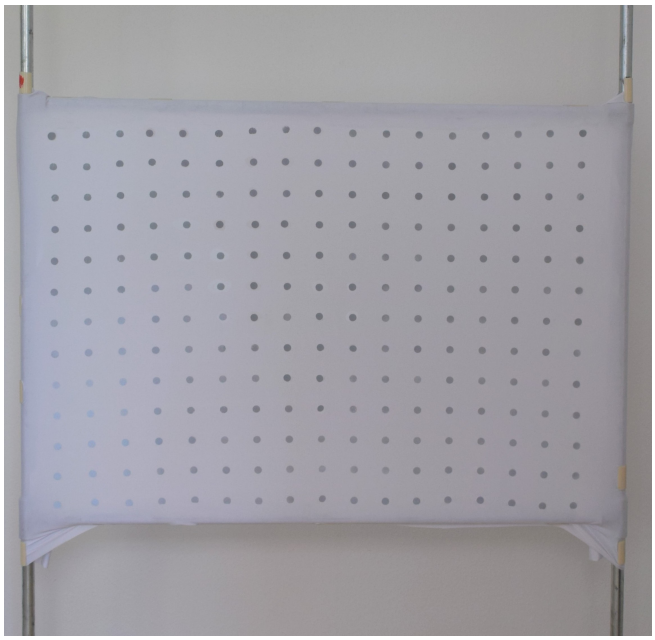
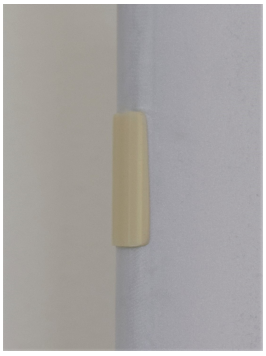


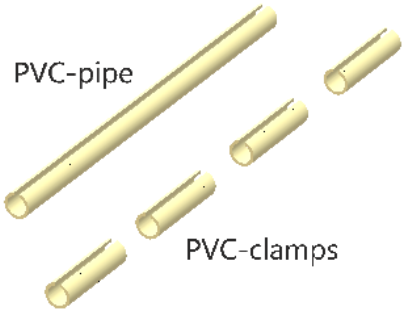
Figure 7: The fabric that makes up the projection canvas strapped onto the frame. The fabric measures 120cm X 85cm.

The fabric is fastened to the frame using plastic clamps constructed from PVC-pipes as seen in figure 8. These

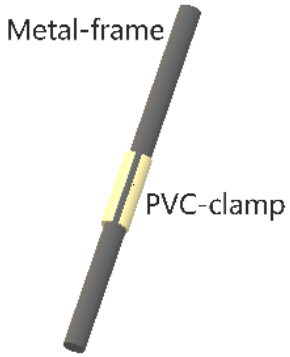
plastic clamps are made by 12 cm cuts of a ø 25 mm PVC-pipe, and then cutting a slit of 5 mm as seen in figure 8b. The plastic clamp can then be snapped onto the fabric on top of the frame-pipe to hold the fabric in place. Multiple plastic clamps are used to loosely stretch the fabric, as to keep it flat, and attach it to the frame as seen in figure 7:



(a) Close-up of a clamp on the screen from figure 7.



(b) A PVC-pipe(left) is cut up to clamps(right).



(c) Illustrates how the clamp snaps on to the frame , here seen without the canvas.

Figure 8: Plastic clamps used to fasten the fabric to the frame.

This clamp-design allows us to easily change the canvas.

5.1.1 The Canvas

As mentioned, we would like the canvas of the deformable screen to be a flexible membrane that allows deformations but retracts fairly quickly to its original shape. In addition, the material should preferably be white as it is a canvas for projection.
Early in the project, we were given a piece of elastic fabric (20% elastane 80% polyamide) by our supervisor to get started with our second prototype. This fabric has the desired characteristics of being deformable and retracting. However, it can be hard to grab because of the very smooth surface. As we would like the canvas to be easy to grab, we investigated what other researchers have previously done. In the work of Troiano [50] et al. they performed a user-study for investigating user-defined interactions on a deformable screen. Additionally, they also performed a pre-study where they presented deformable screens of different canvas-materials to a group of 10 participants.
They tried out the following materials:

1. A rubber sheet made of latex.
2. A mixture of cotton and elastane (95% cotton, 5% elastane).
3. A mixture of cotton and spandex (90% cotton, 10% spandex).
4. A mixture of polyester and spandex (92% polyester, 8% spandex).
5. A mixture of lycra and elastane (90% lycra, 10% elastane).

The study reports that the participants chose the mixture of Lycra and elastane (90% lycra, 10% elastane) as the preferred material for the deformable canvas based on its high resistance, stretchability, and smoothness. We took a closer look at this material, and to our understanding, it seems like lycra and elastane is one and same thing. This is supported by the quote here from the brand Lycra's home page[5]:

[5] `https://www.lycra.com/en/About/LYCRA-SPANDEX-FAQ` Accessed: 19/04-2018

"LYCRA® fiber is the brand name of the original spandex (elastane) fiber invented in 1958."

Nevertheless, we must assume that the fabric was mainly made of elastane, and thus quite different from our material with only 20% elastane. Since our main focus is tracking the screen, and the fabric we were handed suffices just fine for that purpose, we have not tried out different materials for the canvas. However, if we were to evaluate the deformable screen with user involvement, the material could have an important role in the users' perception of the screen.

5.1.2 The Dot markers

To be able to track the screen in the x- and y-coordinates we needed some form of fiducial markers. Finding the best markers for our the setup, that was both easily trackable and non-intrusive, turned out to be rather difficult, and this section describes the iterative process of designing the markers in chronological order.
Preferably we would like the markers to be completely transparent to avoid visible dots in the projected image, but to get started we were just looking for a cheap, fast, and easy way to create the dot grid. Early on in the project, before we had the frame, we tried putting a vest with a reflective strip in front of the IR-camera to see if it would stand out in the image. The reflective strip was significantly brighter than the rest of the scene, meaning it is very applicable for tracking because it is easy to segment the image. We used this reflective piece to successfully implement basic tracking of a single dot marker while we considered how to build the screen.

5.1.2.1 Reflective Paint 1st Attempt

When the rough prototype from figure 5 was constructed, we needed to add some markers. The markers seen in the figure was painted on using Albedo 100 light metallic spray paint [6]. We did not aim to make a perfect grid with equal distances between the dots on the prototype, so we made a simple cardboard template and painted a 5x5 dot grid(also seen in figure 5), without measuring the spacing between each dot. Figure 9a shows how the dots were painted using a single-dot movable template.

(a) Painting a reflective marker using a cardboard template.

(b) A very unevenly painted dot marker.

Figure 9: Dot-markers on the first rough prototype from figure 5.

At first glance, the template worked fine, but it felt like the paint kept changing viscosity and sometimes almost clogged the nozzle of the can even though we did shake the paint for more than a minute(as the instruction prescribes). Moreover, it seemed that after painting the dots, the wet paint spread a bit in the fabric. This

[6] `http://www.albedo100.co.uk/albedo100-reflective-spray-light-metallic/` Accessed: 19/04-2018

resulted in very uneven markers e.g. seen in figure 9b.

Another issue, that only got apparent after some time, was that when the paint was dry, and we manipulated the screen multiple times, the paint seemed to fade in their reflectivity. We experimented with how much paint we applied and found out that if the layer was to thin the marker faded fast, and if the layer was thick the paint cracked and simply sprinkled off the fabric.

5.1.2.2 Reflective Thread

Preferably we would like the markers to be rather uniform and to stay reflective so we looked for alternative solutions. Our next approach was creating the reflective markers by embroidering reflective thread into a circular shape as seen in figure 10. The markers reflected the IR-light very well, but as seen in figure 10b and 10a especially, it pulled the fabric together compromising the uniform surface, and the flexibility of the fabric. Figure 10c shows a dot where the reflective thread is only embroidered loosely in a single direction after a circular template in an attempt to leave the fabric relaxed. Moreover, the gaps between the stitches allow for the light from the projector to somewhat penetrate the marker. This design, however, leaves the marker as an oval shape. The difficulties of using the thread, combined with our lacking skills related to embroidery, unfortunately, ruled the approach out.

(a) Embroidered reflective marker with overlapping stitching in two directions, ϕ 10 mm.

(b) Embroidered reflective marker with overlapping stitching in two directions, ϕ 5 mm.

(c) Embroidered reflective marker with stitching in a single direction ϕ 15 mm.

Figure 10: The best three attempts to create a reflective marker by embroidering reflective thread.

5.1.2.3 Reflective Patch

Our next attempt to create reflective markers was to cut a reflective strip into small rectangular patches and sow them onto the fabric screen as seen in figure 11:

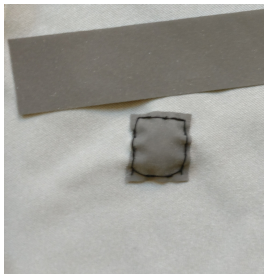


Figure 11: A strip of reflective material in the top of the image, and below a small patch of the strip sown onto the fabric screen.

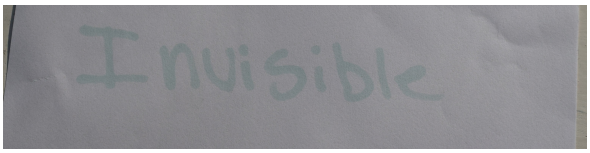
The patches are highly reflective and work very well with the IR-camera, but unfortunately, it represented other

problems. Like the embroidery-approach, the rigidity of the reflective patch sewn onto the fabric compromises the stretchability of the screen. Also, it is completely impenetrable for the projected light, causing very apparent dots in the projected image.

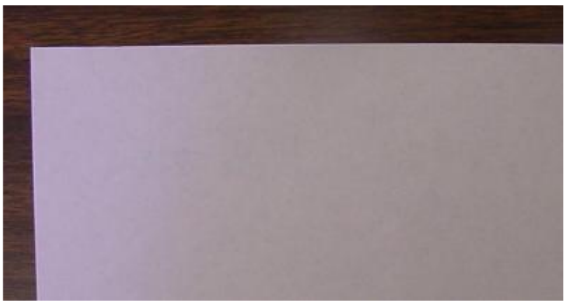
5.1.2.4 IR Invisible Ink

Early on in the project, while researching related work, we encountered an IR Invisible Ink Writing Pen which was used to paint markers for the tracking of a canvas surface[32]. The pen is called **IR1PenSm**[7], and it contains ink that has a stimulation (absorption) wavelength in the spectrum around 793 nanometers(nm) and then emits the light with an emission around 840 nm. The vendor advertises that the ink can fluoresce with the right setup. This means that it is important that both the IR-emitter and the IR-camera(a camera with an IR-filter) have the correct wavelengths for the light to be absorbed by the ink, and for the emitted light to be captured by the camera.
In this project, we are using a Microsoft Kinect camera with IR-projector emitting light at a wavelength in the spectrum around 860nm (described in further details in section 5.2). Although the absorption wavelength of the pen is lower, wavelengths are often not limited to the extract listed value, but rather just spanning the surrounding spectrum. Information including the exact spectrum is, to our knowledge, not available(for the ink, the Kinect's IR emitter, and the Kinect IR camera), so we tried ordering the IR-ink hoping that some of the light from the Kinect or just natural light would stimulate the ink.

The first thing we noticed when applying the ink was that is was not as invisible to the eye as advertised as seen in figure 12. In figure 12a we have written an "invisible" word, and the vendor has done the same in figure 12b.



*(a) The word "Invisible" written with the IR Ink Pen **IR1PenSm** on a sheet of white a4 paper by us.*



*(b) The code "ID#353-985" written with the IR Ink Pen **IR1PenSm** on a sheet of white a4 paper by the vendor (From maxmax [7]).*

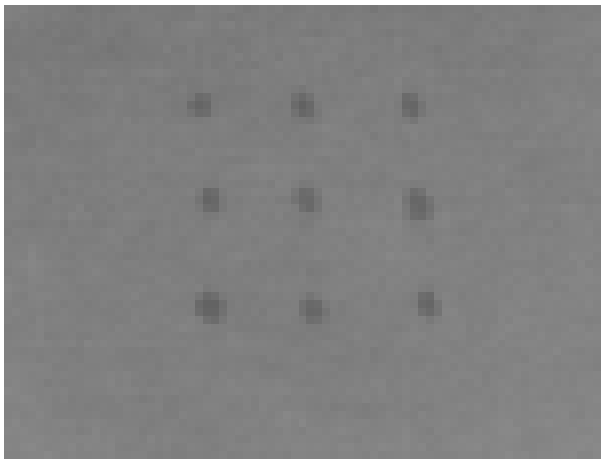
Figure 12: Ir ink applied to white a4 paper.

Figure 13a shows a small grid of dots painted with the IR-ink on fabric, and again it is rather easy for the human eye to spot the ink. The ink leaves a faint green stain on the white fabric which will leave a slightly discoloration in the projected image in our setup. However, the slightly green stains are still much more transparent than the other markers we have considered.
We tested the markers using the Kinect's IR-sensor in a dark room, and realized that we are able to see the ink absorbing the light of the Kinect's emitter, as seen in figure 13b where the ink is seen as dark spots. The dark spots can be tracked just like bright spots, but unfortunately, they are not distinguishable from shadows from folds in the fabric, especially when the camera is further than 1m from the screen, which it needs to be in our setup.

[7] `https://maxmax.com/shopper/product/cid-15272` Accessed: 19/04-2018

[7] `https://maxmax.com/phosphorsdyesandinks/infrared-phosphors-dyes-and-inks/infrared-down-conversion-powder/ir-ink-down-conversion` Accessed: 19/04-2018

*(a) A grid of dots painted the IR Ink Pen **IR1PenSm** on white fabric.*



(b) The grid of IR Ink dots from figure 13a captured by the Kinect's IR camera in a close-up.

Figure 13: IR-ink on fabric.

The vendor of the IR ink has an example on the website[7] showing both absorption(figure 14a) and fluorescence(figure 14b) of the IR ink. However, the fluorescence in figure 14b is obtained by direct and focused light by a 16 red led flashlight. As we cannot have visible light in our scene because of the projection, having a powerful red light-source is not an option. The vendor does however also inform that illuminating the ink using an infrared laser or narrow band IR source in the 715nm to 800nm range, combined with an IR camera with an 830nm filter should make the ink fluoresce. We have not tried this out since we do not have such equipment available, we do not know if it will work without very concentrated light as in figure 14b so lighting up the entire screen may not be feasible, and because it presumably will not work with standard Kinect with a filter around 860 nm.

(a) Viewing IR Ink Absorption using a modified Fuji F30 camera and XNite715 filter. (From maxmax [7]).

(b) Viewing IR Ink Fluorescence using a modified Fuji F30, XNite780 filter, 16LEDRed Flashlight with Concentration Lens(From maxmax [7]).

Figure 14: IR-ink on fabric.

Other projects have however managed to use the IR ink for tracking, but in slightly different contexts or with more sophisticated equipment.
In the work of Park et al.[34] they use the dark spots, caused by the absorption of IR light in the ink, as a marker used for inserting AR elements into a scene. Here they paint the marker on a piece of paper placed near the camera so that the ink is highly illuminated by the IR-light. Another project by Joele et al. [17] just mentions that the IR-ink exist and can be used for tracking. The work most analogous to ours, using the IR-ink, is the

before mentioned work of Narita et al. [31], where they use IR ink dot markers for tracking cloth. In their setup, as seen in figure 15, they use a much more powerful IR-light source (CCS HLDL2- 450X45IR-DF-W, peak wavelength 860 nm) and a filter for blocking visible light (LDP LLC XNite71537, cut wavelength 715 nm) also leaving the ink as dark spots. However, by using two of these very powerful IR light sources, they manage to make the IR ink distinguishable from wrinkles and folds in the fabric. Unfortunately, we do not have such equipment available, it would interfere with the Kinect's depth estimation, and would make us compromise our ambition of using only commodity products.

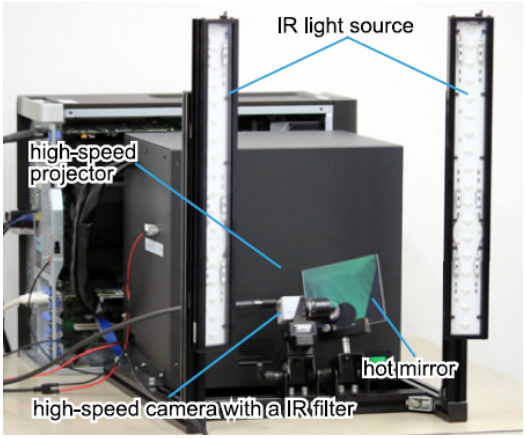


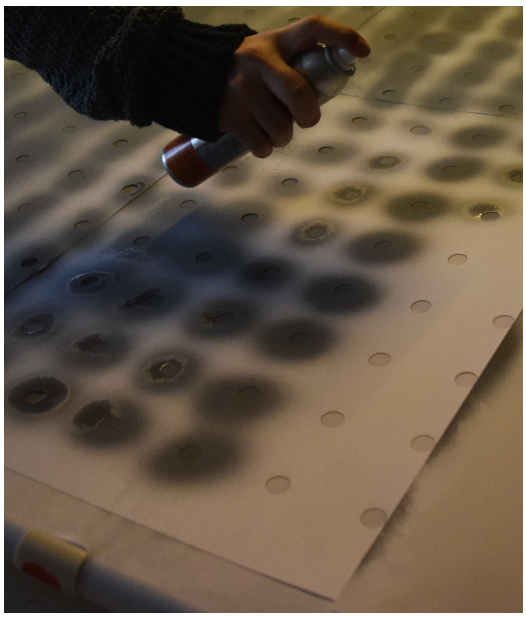
Figure 15: The setup from the work of Yoshihiro et al. [31].

5.1.2.5 Reflective Paint 2nd Attempt

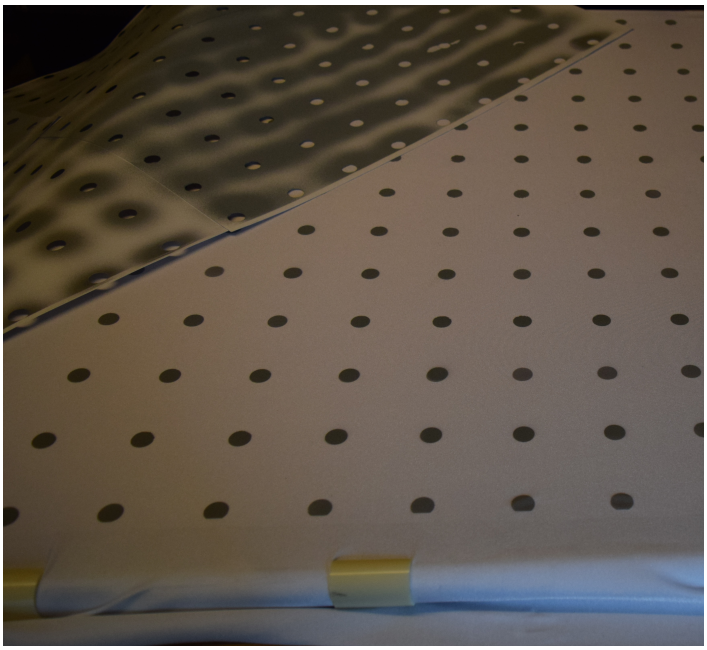
The issues with the IR ink caused us to return to our working solution with retro-reflective spray paint. This time, we ordered from another vendor MEON[8] as our supervisor had good experience with applying this paint to fabric. Moreover, this paint is more translucent which is very good for our setup where we want the projected light to penetrate markers. Unfortunately, the paint did not show up because the vendor does no longer deliver to Denmark, so we had to use the Albedo 100 paint again.

Learning from our former issues with applying the paint evenly in a perfect grid, where markers are evenly spaced, we had carton stencils(A2 sized) for covering the entire screen cut out by a laser cutter. Figure 16a shows the stencils glued onto the fabric screen, and how the screen was painted with the reflective spray paint. In figure 16b the stencil is removed, leaving a nice grid of round reflective markers.

[8] `http://shopping.meonuk.com/reflectlight-spray-glow-in-the-dark` Accessed: 19/04-2018



(a) Painting a grid using carton stencils with pattern of a perfect grid cut out by a laser cutter and glued onto the screen.



(b) The stencil is removed, leaving a nice evenly spaced grid of round reflective dot markers.

Figure 16: Painting the grid of dot markers using laser-cut stencils.

The paint is clearly visible gray on the fabric and not the invisible markers we had envisioned, which we will elaborate on in the later discussion section. However, the markers suffice for us to test the setup, develop tracking methods, and perform experiments with a deformable screen fulfilling our objective.
The dot markers used in our prototype has a diameter of 2 cm, and we have created two trackable canvasses with different spacing between the reflective markers. Both of the canvasses have the dimensions 120cm X 85cm like also seen in figure 7. One of the canvases has a 9x7 (width x height) grid of the reflective dots with a spacing between the dots of 12cm, and the other has 17x13 with 5cm between the markers.

5.2 Camera

Our setup needs an IR-camera and an IR-emitter to track our reflective markers in the x- and y-axes, as well as a depth camera, to track in the z-axis. Most of the cameras in the related work section have high-speed sensors, but this project was based on the premise of using cheaper consumer grade products, as we want others with similar financial restrictions to be able to use our research themselves.
We have investigated what cameras that are on the market, and there are three main approaches for obtaining depth information:

- **Stereo Camera Triangulation**, where triangulation between two frames taken at the same time, from to cameras placed differently, are used to obtain depth information. (e.g. PlayStation Camera [9] and also seen in mobile phones e.g. Huawei P9 [10]).
- **Structured Light**, where a special pattern of IR light is projected onto the scene, and deformations of this pattern enables algorithms to compute depth information. (e.g. Kinect v1 for Xbox 360 and PC [11], and the newly released Intel RealSense D415 and D435 stereo 3D cameras [12].

[9] `https://www.playstation.com/en-us/explore/accessories/vr-accessories/playstation-camera/` Accessed: 19/04-2018
[10] `https://consumer.huawei.com/dk/phones/p9/` Accessed: 19/04-2018
[11] `https://support.xbox.com/en-US/browse/xbox-360` Accessed: 19/04-2018
[12] `https://realsense.intel.com/stereo/` Accessed: 19/04-2018

- **Time-Of-Flight(ToF)**, where the depth is determined by the time it takes for IR light to travel from an emitter to a scene, and back again. (e.g Kinect v2 for Xbox One and PC[13] and Creative Senz3D [14])

The Stereo Camera Triangulation technique does not offer the precision we are looking for, and neither does it have a build-in IR-camera ruling this option out.
Especially the RealSense D435 has nice specifications of stereo 3D cameras and a frame-rate of 90 Frames Per Second(FPS) which is the maximum FPS we have encountered on a commodity depth camera. Unfortunately, we cannot use the structured light cameras because we need the entire scene to be lit up by the IR light, and not just a pattern of light, for being able to track our markers. If we were to use a Structured Light camera, it would require that we had an external IR light source and camera, using a spectrum of infrared light falling outside the range of what is used for the depth camera. Such a setup would be expensive and cumbersome, thus we had to use a ToF-camera that illuminates the entire scene, and hence all visible reflective markers. We decided to use the Microsoft Kinect for this purpose as it is a consumer product which is easy for everyone to acquire, and comes with a SDK for developers to use[15]. In section 6 however, it is described that the software implementation is independent of the Kinect and that other ToF cameras can be used in our framework as well.
The Kinect records color(1920x1080), infrared(512x424) and depth(512x424) frames simultaneously at 30 Frames Per Second(FPS).
The depth-frame is acquired using the Kinect's ToF camera which in simple terms emits Infrared(IR) light signals and then measures how long it takes for them to return [23]. In the same fashion as bat's way of navigating in the dark by emitting high-frequency sounds and sensing its surroundings by listening to the echo.
Most ToF cameras are based on what is called pulse modulation where a light-source pulses its light signal(toggling the light on/off, at a fixed wavelength) at a specific frequency, a sensor captures the light's reflection from a scene, and calculates the distance to an object using the phase-shift seen in figure 17.

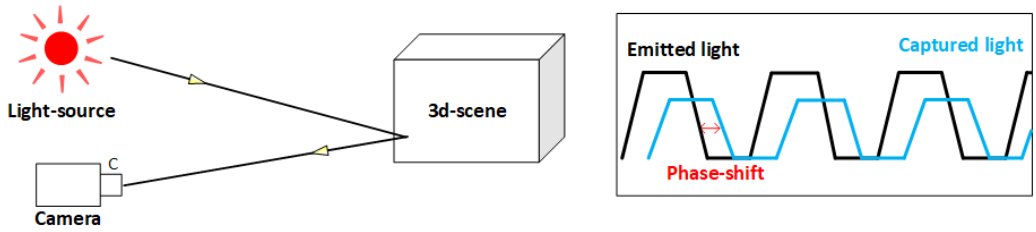


Figure 17: ToF-system, emits pulse modulated light out(with a fixed wavelength, for the Kinect @860nm) which returns to the sensor lens with some delay (phase shift) and attenuation. The phase shift describes the distance to the object that reflected the light.

As we know the speed of light[16] and the phase-shift gives us the Round Trip Time (RTT[17]) we also know the distance the light has traveled. A problem arises when experiencing phase-wraps, meaning that the phase shift is so large that captured light is an entire period delayed in relation to the emitted light. For example, we will see a phase wrap with a depth of 1.87 m at an 80-MHz modulation frequency, meaning that things that are a bit further away(than 1.87 m), will appear to be right in front of the camera[43]. Modern ToF cameras solve this issue using a technique called correlation-based time-of-flight (C-ToF), where a single depth image is created from multiple pulses of IR light(with a fixed wavelength) with various modulation frequencies and phases. Specialized phase unwrapping algorithms are used on the set of recorded frames to eliminate the ambiguities caused by phase wraps[14]. The Kinect is a C-ToF camera sending pulses on the modulation frequencies of approx. 120 MHz, 80 MHz, and 16MHz[43] all at a fixed wavelength of 860nm. It actually uses 10 frames to generate each depth frame, meaning that the depth sensor actually records @ 300 Hz but only delivers the

[13] `https://www.xbox.com/da-DK/xbox-one/accessories/kinect` Accessed: 19/04-2018
[14] `https://us.creative.com/p/web-cameras/creative-senz3d` Accessed: 19/04-2018
[15] `https://developer.microsoft.com/en-us/windows/kinect` Accessed: 19/04-2018
[16] Speed of light 299.792.458 m / s
[17] RRT - Here meaning the time it takes for the light to travel from the emitter to the scene, and back again

optimized depth frame for every 10'th frame via the SDK running @ 30 HZ.

A group of researchers accompanied by Microsoft, and with access to the source code for the Kinect, have modified the code to deliver every frame @ 300 Hz, and used it as a high-speed camera for the purpose of tracking a ball[47]. We were very interested in boosting the frame rate, as many papers in the related work use high-speed cameras. However, the Kinect records the sequence of 10 frames followed by a short break, before it records again, limiting the benefit of the extra frames. In the study, they spread out the frames so that they were equally spaced in time, but this undermines the Kinect's algorithm for creating the depth image. Instead of using the default depth algorithm, they performed model-based tracking, which is unfortunately limited to the purpose of tracking a ball. Thus we had to settle with the default 30 Hz of the Kinect camera.
The specifications of the Kinect's depth sensor is seen in figure 18:

Process Technology	TSMC 0.13 1P5M
Pixel Pitch	10u*10u
Pixel Array	512*424Pixels
Chip size	8.2mm*14.2mm
System Dynamic Range	> 2500 = 68db
Modulation Contrast	68% @ 860nm @50Mhz
Modulation Frequency	10-130Mhz
Average Modulation Frequency	80Mhz
FOV	70 (H) X 60 (V) degrees
Depth Uncertainty	< 0.5% of range
Distance Range	0.8-4.2m
Operating Wavelength	860nm
Frame Rate	max 60fps (typical 30fps)
ADC	2GS/s
Effective Fill Factor	60%
Reflectivity	15%-95%
Chip Power	2.1W
Responsivity @ 860nm	0.144 A/W
Readout Noise	320 uV differential
F#	1.07
ADC Resolution	10

Figure 18: Specifications of the Kinect v2's depth sensor. In the final product, the frame rate has been limited to 30 FPS and is not able to achieve 60 FPS (from Payne [35]).

Unfortunately, TOF and C-TOF depth measurements are also dependent on the surface, and e.g (semi-)translucent (glass, plastic, etc.), highly reflective (mirrors, steel, etc.) or non-reflective surfaces(dark black) are problematic. This presents a challenge for us because we want to track a surface with highly reflective areas, for which the Kinect returns a distance close to zero independent of how far the reflective area is away from the camera. This issue is, however, something we will present a solution to later in the report.

5.3 Projector

A projector is used to project the 3D model from the computer onto the fabric screen. We are using a standard consumer projector Epson EH-TW3200 1920x1080 @60HZ projector[18] which is positioned in the setup such that the projected image covers the entire fabric screen.
The projector has projection ratio 1.34 - 2.87:1 meaning that if we want a 1-meter wide image, the projector has to be at least 1.34 meters away, and at most 2.87 meters away from the projection canvas. This rather high projection ratio, characterizing standard lenses, makes the setup rather space consuming because the projector needs to be placed far away from the screen, compared to short throw projectors with a ratio at about 0.5.

[18] `https://www.epson.de/en/products/projectors/home-cinema/epson-eh-tw3200-with-hc-lamp-warranty#specifications` Accessed: 19/04-2018

6 System Design

In this section, we will cover the design of our system for realizing the deformable screen. The section is divided into three main parts:

- **ScreenTracker**, describing the system for tracking the surface of the deformable screen, achieved by using an IR-camera for tracking the reflective markers in 2D, and a depth camera to add an extra dimension enabling 3D tracking.
- **Visualization**, Here we cover how the tracking data from the ScreenTracker is translated into a 3D computer graphics model.
- **Calibration of the Projector and the Depth Camera**, Finally we will cover how we calibrate the system to correctly project the 3D model back onto the deformable screen.

The design is explained using a high abstraction level aiming for a broad theoretical understanding of the design fully independent of implementation specifics.

6.1 ScreenTracker

We start by covering the ScreenTracker in the following sections referring to the pipeline shown here in figure 19:

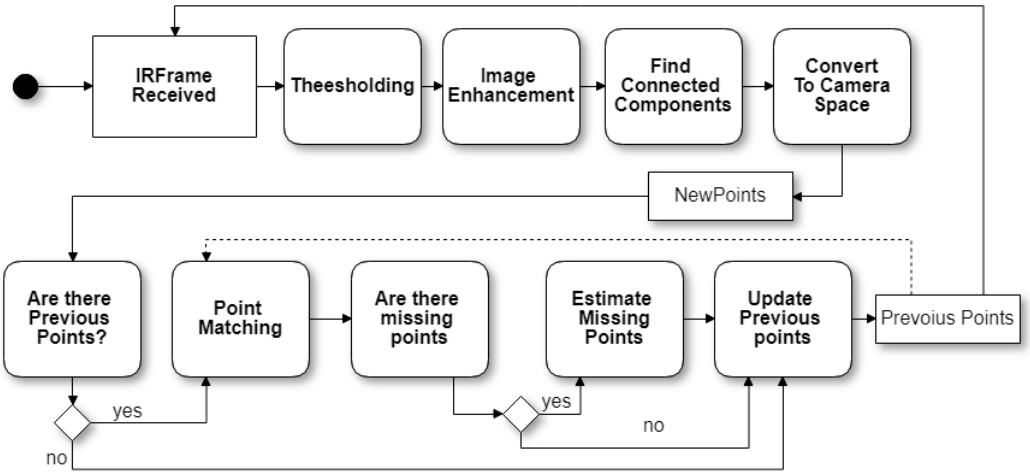
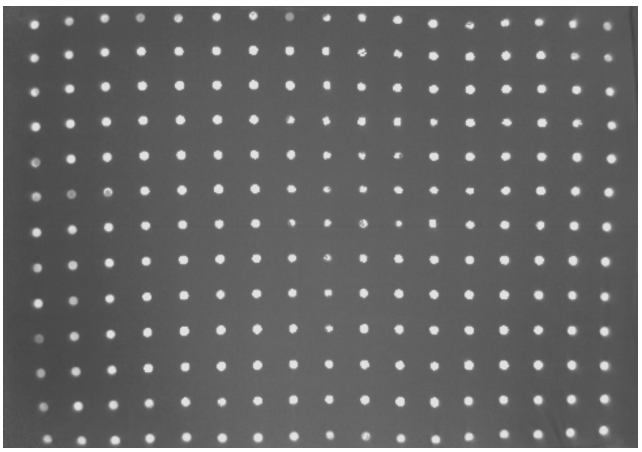


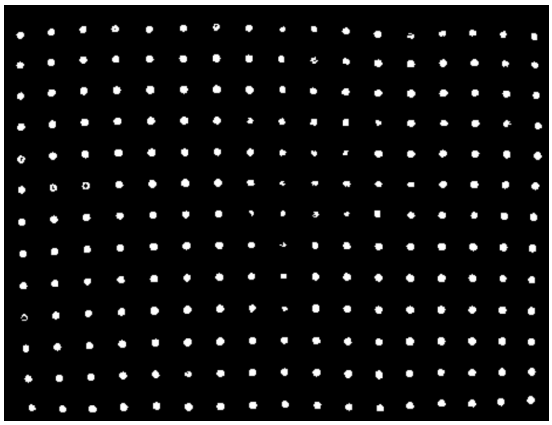
Figure 19: Pipeline for the ScreenTracker.

6.1.1 IRFrame Received & Thresholding

When an infrared frame arrives from the camera, we want to segment the reflective markers. As the reflective markers reflect the IR-light, they have a distinguishable high light intensity as seen in figure 20a. Thus we can segment the markers simply by thresholding the image[44, chpt. 10.3] with a high threshold, leaving only the markers in the resulting binary image as seen in figure 20b.
Fully adaptive thresholds such as ”Adaptive Mean”-, ”Adaptive-Gaussian”-, and Otzu’s-threshold[44, chpt. 10.3] can also be used for the purpose, and have the advantage that they require less tuning in changing lighting conditions. For those reasons, we will use a fully adaptive threshold for our tracking pipeline.



(a) Infrared image showing the flexible screen.

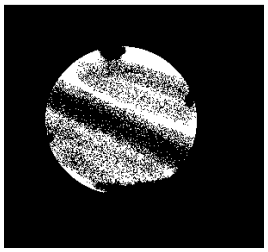


(b) Binary image produced by thresholding the image to the left with a high threshold.

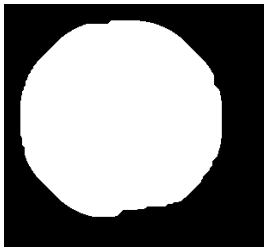
Figure 20: Frame from the IR-camera showing the flexible screen.

6.1.2 Image Enhancement

After the infrared frame has been thresholded the individual reflective markers rarely appears as completely uniform round shapes, but often has some noise(or 0 intensity) looking somewhat like the dot in figure 21a. This can result in that the face of a single marker is disconnected, and in later steps in the pipeline, tracked as multiple markers. Morphological operations [44, chpt. 8.8] can be used for removing noise in images. First morphological dilation of the image removes the ”black holes” in faces of the markers so that the white pixels all are connected as seen in figure 21b. The dilation leaves the marker unrealistically large in the image, and morphological erosion can shrink it down again to the original proportions as seen in figure 21c. Performing the morphological operations with a circular kernel can additionally enhance the circular shape of the dot marker.



(a) Example of marker seen in a thresholded IR-image of the screen.



(b) Figure 21a exposed to dilation.

(c) Figure 21b exposed to erosion.

Figure 21: Image enhancement using Morphological operations [44, chpt. 8.8].

The combination of first applying dilation and then erosion is also called morphological closing, but keeping the two operations separate allows us to run them each for a distinct number of iterations. Even though we can enhance the image using the morphological operations, we should be careful as too much dilation can ”melt” two closely spaced markers into one. Figure 22 shows the binary image of the screen enhanced by the morphological operations:

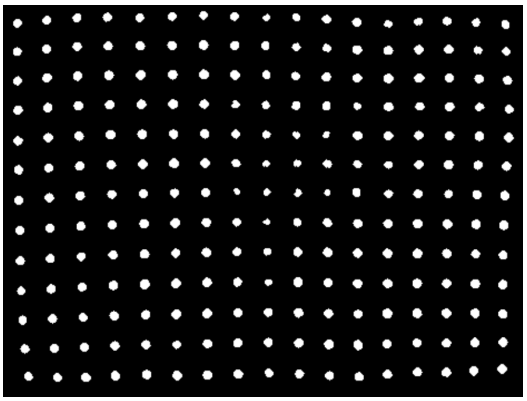


Figure 22: The binary image from figure 20b exposed to first dilation, and then erosion.

6.1.3 Find Connected Components

We now have a binary image showing uniform reflective markers and are interested in the location of each of them. Here we define the position of a marker as the centroid of the marker in the IR-image. Since we know that all the pixels making up a marker are connected, we can locate all the markers in an image simply by finding all its connected components. Finding connected components in binary images has been done since 1966[40], and most image processing and computer vision libraries include methods for finding the connected components and their centroids.

6.1.4 Convert to Camera Space

We would like to track the markers in a real-world metric 3D space, so we need to convert the positions of our tracked markers from the 2D pixel coordinates of the IR-image to the 3D real-world coordinates. Such a conversion requires that we know the depth of the markers, which we can obtain using the depth camera. However, as described in 5.2 the reflectivity of the markers enabling us to track them in 2D(x,y), ruins the depth information, so we cannot measure the z-coordinate directly in the region of a marker. We overcome this limitation of the TOF-camera by taking depth samples around the reflective markers, and use them to estimate the depth of the markers.
Having the 2D pixel coordinates and the z-coordinate unfortunately is not adequate for computing the 3D position of the marker as it requires knowledge about the camera used for recording the frames. The later section 6.3 about the setup's projector and camera calibration also covers how we can convert between the 2D pixel coordinates and real-world 3D metric coordinates, and exactly what we need to know about the camera. However, most frameworks for 3D cameras include methods for converting coordinates between the 2D and 3D space.
When the coordinates have been converted, the positions of the detected markers are passed on in the pipeline as the variable newPoints(also seen in the pipeline from figure 19).

6.1.5 Are There Previous Points

In the first frame we do not have any previous points, and thus the points found in the previous step defines the screen. Each detected point is assigned an ID used to identify the marker, and we continue on to "Update Previous Points".
In the later frames, we will have information about the points detected in the previous frame, which needs to be matched to the points found in the current frame, identified by their id, in order to perform the tracking.

6.1.6 Point Matching

From the second frame and onwards the points from the previous frame has to be matched to the points detected in the newly arrived frame.

Our first approach to this was simply finding the nearest neighbor from the new frame to the previous using a KD-tree [4] ($O(\log n)$ for a nearest neighbor search) for efficiency. However, using this method, we can get a different result based on which way we search for the nearest neighbor(from new frame to the previous or vice versa). The straight-up nearest neighbor approach also has the issue of not assigning all points if multiple are closest to the same one. To counter this we solved conflicts by taking the closest one, and then try to reassign the rest.
It turned out, however, that this can cause major problems for the algorithm not being deterministic and conflicts leading to points getting swapped. Looking at figure 23 we see the points from the previous frame as black circles, and the points from the new frame that have changed, as orange circles. Going through them from the lowest to the highest index, we end up with **N5** and **N8** being matched to **P5** and **N9** to **P8** after first run. As **N5** is closest to **P5** this will win the conflict, leaving only **N8** without a match, and **P9** being the only open position we will end up having switched **N8** and **N9**.

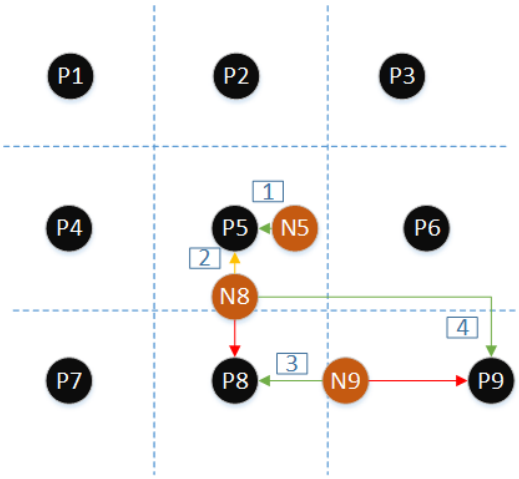


Figure 23: KD-Tree Nearest neighbor algorithm process. Points P from the previous frame are shown as black circles, and the points N from the new frame as orange circles.

To try and counteract the fact that distance is the only variable to evaluate, we also tried a dead reckoning approach which is often used in navigation, and is defined by Merriam Webster dictionary as

> *"the determination without the aid of celestial observations of the position of a ship or aircraft from the record of the courses sailed or flown, the distance made, and the known or estimated drift"*

or simply as "guesswork". The approach uses previous locations and looks at direction as well as speed, drift and any other factors one might have, to determine an area where we expect our new point would show up, from here we could then again use our Nearest Neighbor to find the correct point. This approach, however, is only effective if we can be somewhat certain that a moving object will continue in the same general direction, with somewhat the same speed. For deformations of the deformable screen, this is not the case, however, as it is more than likely that the stretchable cloth will be released and retract backward along its path at some point. The approach might still be valuable, but would require further research as our attempts were not successful.

Another effective approach we have seen in related work [32] is to define a small window around every point in the previous frame and perform a local search for the point in the new frame. However, as also mentioned in their work, this does only work when the dot-marker moves very little between successive frames, which is why they use a 500fps camera. Unfortunately, the Kinect only provides 30 fps ruling this approach out. Moreover, they allow for markers to get completely lost in their tracking, which we cannot because we would like to keep

[18] `https://www.merriam-webster.com/dictionary/dead%20reckoning` Accessed: 19/04-2018

track of the entire screen to allow gesture detection in future work.

Thus the above approaches were discarded, and we looked for other algorithms. Instead of just finding the nearest neighbor for each point, we can treat it like an assignment problem which is one of the fundamental combinatorial optimization problems presented as:

> *"The personnel-assignment problem is the problem of choosing an optimal assignment off n men to n jobs, assuming that numerical ratings are given for each man's performance on each job."*[30]

Where an optimal assignment gives, either the highest or the lowest sum of ratings, depending on if we want to look at it as a minimization or maximization problem.
We can convert our case to such a problem by looking at the numerical distance from n points in the current frame, to n points in the previous frame, and find an optimal assignment which gives the lowest sum of distances. The Hungarian-algorithm also called the Munkres-algorithm(named after James Munkres [30]), solves the assignment problem optimally and deterministic in $O(n^3)$. Besides the running time of the algorithm itself, the distances between all combinations of previous and new points have to be computed, making this approach rather computationally costly. But as we only have a low frame rate of 30 FPS in our current setup, it should not present any issues running in real-time.
Our approach of point matching using the Hungarian algorithm does however still present major issues in tracking the reflective markers, as this rather simple method is very vulnerable to swapping ID's of markers when they pass each other directly in the same location. This is illustrated in figure 24, showing a cut-out of the deformable screen seen from the back(where the camera is placed), here we see that the screen is grabbed into a hand forcing the markers together. In such cases, when the markers pass each other, the Hungarian Algorithm is likely to swap their ID's because it only considers the distance from the points detected in the previous and current frame. We will return to this issue later in this section.

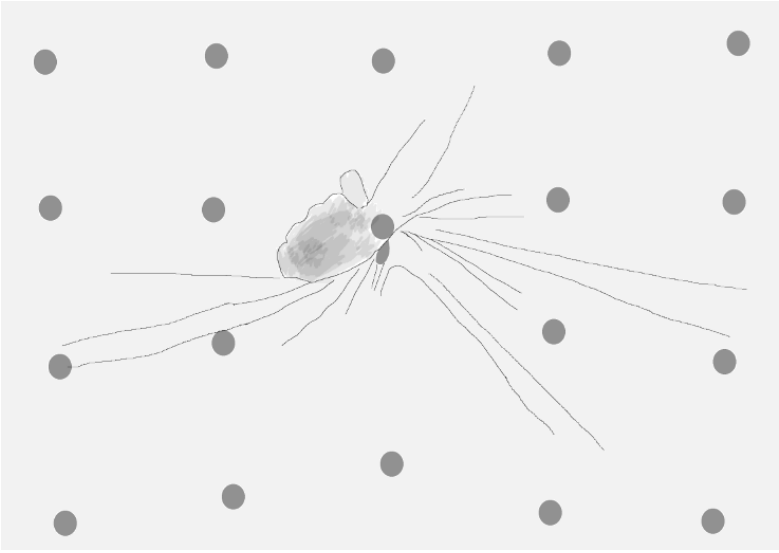


Figure 24: Illustration of a cut-out of the deformable screen being grabbed into a hand, seen from the backside of the screen where the camera is placed.

6.1.7 Are there Missing Points & Estimate Missing Points

If all points are visible and could be matched to a previous point we can continue to "Update Previous Points". Otherwise, if there are not as many points detected in the current frame, as there were in the first frame, we need to estimate the positions of the missing points. Estimating occluded points represent a major part of this project, and we have tried out three different estimation methods which we call:

- Extrapolation model.
- Vector Displacement model.
- Mass-Spring model.

6.1.7.1 Extrapolation

Our first approach for estimating missing points was inspired by the paper "Dynamic Projection Mapping onto a Deformable Object with Occlusion Based on High-speed Tracking of Dot Marker Array" [32] where they perform extrapolation on the visible points to estimate the position of missing ones. In the paper, they rely on a perfect grid with equal distance between dot markers, and they keep a simple parameter for each tracked dot to indicate whether it is visible or not. They present the property seen below, where $s_i(t)$ means the status of the i'th tracked dot marker at time t:

$$s_i(t) = \begin{cases} s^0\text{: visible from camera and tracked correctly,} \\ s^1\text{: occluded or tracked incorrectly} \end{cases}$$

Likewise, $u_i(t)$ is the position of the marker at time t, and for each frame of their high-speed camera(500 fps) they update the markers information. If a marker is not visible due to occlusion, the status is updated to s^1, and they calculate an estimate $\hat{u}_i(t)$ of the markers position using the neighboring markers as seen in figure 25. Here $N_i(t)$ is the set of visible markers(with status s^0) adjacent to an occluded marker at time t. $k(j)$ is a marker which is adjacent to the markers $j \in N_i(t)$ in the direction from i to j.

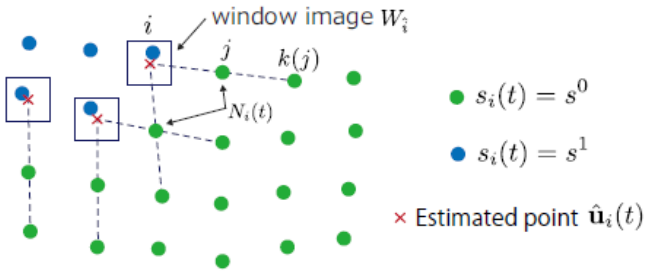


Figure 25: Illustration of the extrapolation scheme (from [32]).

They estimate $\hat{u}_i(t)$ using the formula below, which is the normalized sum of two times the position of adjacent visible markers, minus the position the next point in that same direction:

$$\hat{u}_i(t) = \frac{1}{|N_i(t)|} \sum_{j \in N_i} 2u_j(t) - u_{k(j)}(t)$$

On the other hand, if the marker is detected in the frame and thus is visible, its position $u_i(t)$ is updated, and the status is set to s^0. Additionally, the original study had an extra step that estimated if the markers moved in an unrealistic path, and if so the marker's status was set to not visible to avoid tracking errors. In this study the simply ignored markers with the invisible state in their visualization and accepted not having tracking information for all markers. As we envision our system to be the tracking foundation for later work to be able to recognize and distinguish different gesture interactions, we need to at least guess where we expect the markers to be a given time. Thus contrary to the original study, we allow the model to extrapolate on its neighborhood even if it is occluded if it cannot extrapolate on visible markers. We call this for allowing "estimates of estimates".

6.1.7.2 Vector Displacement

The second method for estimating the position of missing markers is based on how much surrounding markers have moved from their initial position in the first frame. We expand on the notation from the previous section,

meaning that the initial position of the dot markers in the first frame at time t_0 is denoted as $u_i(t_0)$. We define the set of surrounding markers at time t as $M(t)$ given by a missing marker's cardinal points(north, east, south, west), second cardinal-points meaning the second markers in each cardinal direction, and the inter-cardinal points(north-east, south-east, south-west, and north-west) as seen in figure 26 :

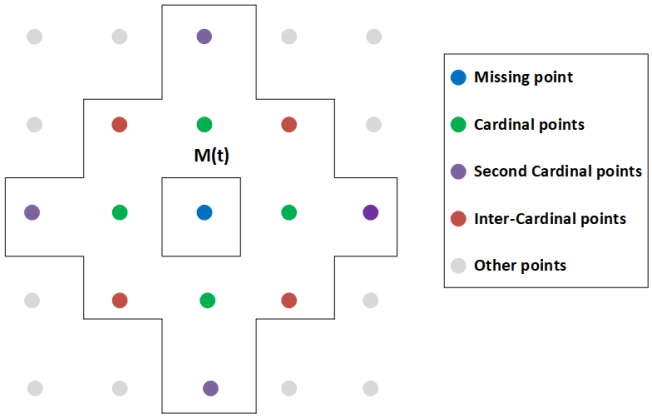


Figure 26: Illustration of a marker's connection to its neighborhood.

Furthermore it is a requirement that a marker is visible $s_i(t) = s^0$ to be in the set $M(t)$. Figure 27 shows how the position of a missing marker with s^1 is found for a single point(the point with the red arrow). The marker's estimated position is given by its original position plus its estimated displacement vector $\hat{v}_i(t)$(the red arrow). $\hat{v}_i(t)$ which is estimated using the normalized sum of the surrounding markers displacement vectors $v_i(t)$

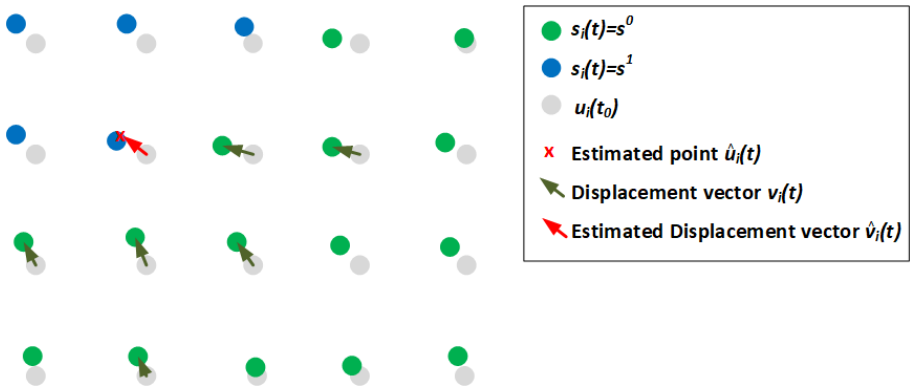


Figure 27: Illustration of the Vector Displacement estimation scheme.

The formula below shows how the estimated displacement-vector of a missing point is calculated:

$$\hat{v}_i(t) = \frac{1}{|M_i(t)|} \sum_{j \in M_i} v_j(t)$$

Where the displacement vector $v_j(t)$ is given by the vector between the initial position of a marker $u_j(t_0)$ and its current position $u_j(t)$, so we can write the estimate of $\hat{v}_i(t)$ as:

$$\hat{v}_i(t) = \frac{1}{|M_i(t)|} \sum_{j \in M_i} u_j(t) - u_j(t_0)$$

Finally the estimated position of a marker $\hat{u}_i(t)$ is simply the estimated displacement vector added to the marker's original position:

$$\hat{u}_i(t) = u_i(t_0) + \hat{v}_i(t)$$

Or seen with all the variables:

$$\hat{u}_i(t) = u_i(t_0) + \left(\frac{1}{|M_i(t)|} \sum_{j \in M_i} u_j(t) - u_j(t_0) \right)$$

As this model has a larger neighborhood than the extrapolation model, and we do not calculate the estimate using the neighbors in pairs, we do not allow estimates of estimates. If none of the 12 neighbors are visible, we simply do not update the position and keeps the last estimated or visibly tracked position.

6.1.7.3 Mass-Spring model

Simulating cloth and other stretchable and deformable materials is an entire field of research in computer graphics, and as we are trying to represent a cloth screen as a 3D computer graphics model, we have looked into existing methods. We would like to experiment with using methods for cloth simulation for estimating the position of occluded markers.
Research in cloth simulation was initially started in the 80's with the first simple geometrical models [55] [11] in 1986 and a physical model based on elasticity theory [49] from 1987 later expanded on in 1992 [7].
Another physical model based on springs is the Mass-Spring model by Provot[16], which has the ability to simulate the structure of cloth, and has a low computational cost.
Numerous models improving the realism of cloth simulation has been proposed e.g. the Particle model [6] generating more sophisticated results when attempting to reproduce folds and buckles, and even a Data-driven elastic model for cloth[53] where the data is recorded by deforming actual cloth.

As we are interested in testing if cloth models can be used for the purpose of estimating occluded markers, we would like to try using the Mass-Spring model. This choice is based on that it is a physical model that captures the basic structure of the cloth while requiring limited computational resources allowing real-time simulation.

The Structure of the Mass Spring Model

The Mass Spring model gets its name from that it models a stretchable surface(e.g. cloth) as a mesh of small particles with masses connected by springs, adding force to the particles based on their elongation. In this section, we will use the terms particle and point interchangeably.
The work of Provot suggests to create an elastic model consisting of a mesh of $m \times n$ virtual masses connected to their neighbors by three types of springs, assigned as seen below:

- **Structural Springs**, linking masses [i,j] and [i+1,j], and [i,j] and [i,j+1].
- **Shear Springs**, linking masses [i,j] and [i+1,j+1], and [i+1,j] and [i,j+1].
- **Flexion Springs**, linking masses [i,j] and [i+2,j], and [i,j] and [i,j+2].

Figure 28 shows the springs defined above assigned to a point $P_{i,j}$ in a $m \times n$ mesh, here $P_{i,j} = P_{1,1}$.

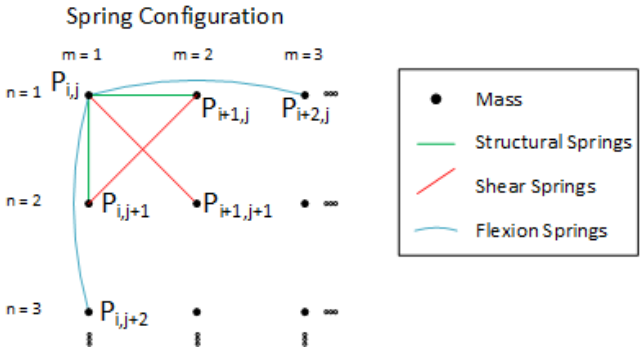


Figure 28: The figure shows how springs are assigned in the $m \times n$ mesh of masses in Provot's Mass Spring model.

It is rather hard to see how the springs connect all the points in a mesh in figure 28, but when the springs have been assigned to the entire mesh, then a single point Pi, j is actually connected to its neighbors like seen in figure 29 to the left. This view also gives the naming of the spring-type more sense, because it is now more obvious that under pure shear stress, mostly the sheer springs are constrained; under pure flexion stress, occurring when bending the mesh, mostly the flexion springs are constrained; and under pure compression or traction stresses, e.g. occurring when stretching vertically and horizontally, mostly the structural springs are constrained.

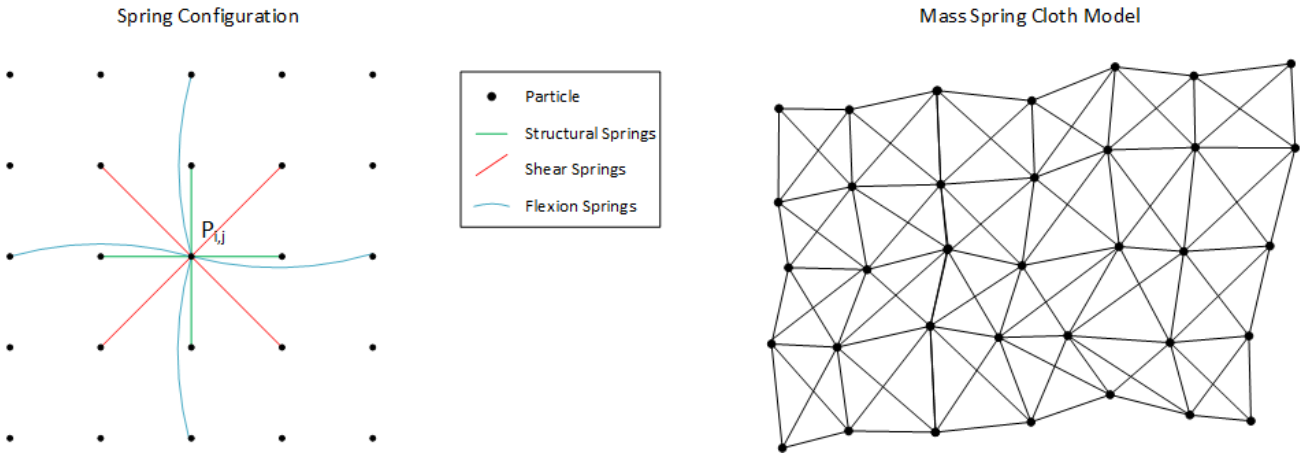


Figure 29: The spring configuration for a single point $P_{i,j}$ in the mesh(left). A cloth modeled by the mesh of masses and springs, here the flexion spring has been left out to make the structure more clear (right).

Forces

The position of a particle can be derived from second law of motions:

$$\mathbf{f} = \mathbf{ma} \tag{1}$$

Where **f** is the force applied to a particle with mass **m** and acceleration **a**. Acceleration is defined as the rate of change in velocity, and velocity is defined as the rate of change in position. Thus velocity is the 1st derivative of the position and acceleration is the 2nd derivative with respect to time. So for a single particle $P_{i,j}$ the position, which we will call $u_{i,j}$, is determined by:

$$\mathbf{m}_{i,j}\ddot{\mathbf{u}}_{i,j} = \mathbf{f}_{i,j} \tag{2}$$

Provot divided the force term into two, the external forces $\mathbf{f}_{ext}$ and internal forces $\mathbf{f}_{int}$ of the fabric.

Internal Forces

The internal forces $\mathbf{f}_{int}$ are expressed by the springs interconnecting the particles. Most commonly, linear springs governed by Hookes law are used between the particles. Hooke's law states that the force(**f**) needed to extend or compress a spring by some distance x scales linearly with respect to that distance, also expressed by the equation below:

$$\mathbf{f} = -\mathbf{k}\mathbf{x} \tag{3}$$

Where k is called the spring-constant describing the stiffness of the spring, and the negative sign indicates that the spring is a restoring force. We define the forces of a spring connecting two points $P_{i,j}$ and $P_{k,l}$, and expands on the above expression:

$$\mathbf{f}_{int}(P_{i,j}, P_{k,l}) = -\mathbf{k}_{i,j,k,l}(\overrightarrow{P_{i,j}P_{k,l}} - l^0_{i,j,k,l})\frac{\overrightarrow{P_{i,j}P_{k,l}}}{||\overrightarrow{P_{i,j}P_{k,l}}||} \tag{4}$$

Where $\overrightarrow{P_{i,j}P_{k,l}}$ is the vector from $P_{i,j}$ to $P_{k,l}$, $||\overrightarrow{P_{i,j}P_{k,l}}||$ is the length of the vector, and $l^0_{i,j,k,l}$ is the initial resting length of the spring connecting $P_{i,j}$ and $P_{k,l}$. The vector $\overrightarrow{P_{i,j}P_{k,l}}$ is simply found by:

$$\overrightarrow{P_{i,j}P_{k,l}} = \langle u_{k,l,1} - u_{i,j,1}, u_{k,l,2} - u_{i,j,2} \dots u_{k,l,n} - u_{i,j,n} \rangle$$

Where n is the number of dimensions which for the cloth simulation most commonly is 3 dimensional.
More intuitively explained, expression 4 corresponds to the difference between the current length of a spring, and its initial resting length $l^0_{i,j,k,l}$. As seen topmost in figure 30, a relaxed spring connects point $P_{i,j}$ and $P_{k,l}$ having the initial length $l^0_{i,j,k,l}$ this is also called the spring's state of equilibrium. In the middle $P_{i,j}$ and $P_{k,l}$ has moved closer and their connecting spring is compressed. If we leave out the spring constant $k_{i,j,k,l}$, $\mathbf{f}_{int}(P_{i,j}, P_{k,l})$ corresponds to the pushing force illustrated by length of the two green vectors. Seen bottom most, the spring is stretched and the length of the red vectors represents the pulling force of $\mathbf{f}_{int}(P_{i,j}, P_{k,l})$. Here a pushing force is positive and a pulling force is negative, and introducing the spring constant k_{ij} will scale the push/pull forces and thus the length of the vectors.

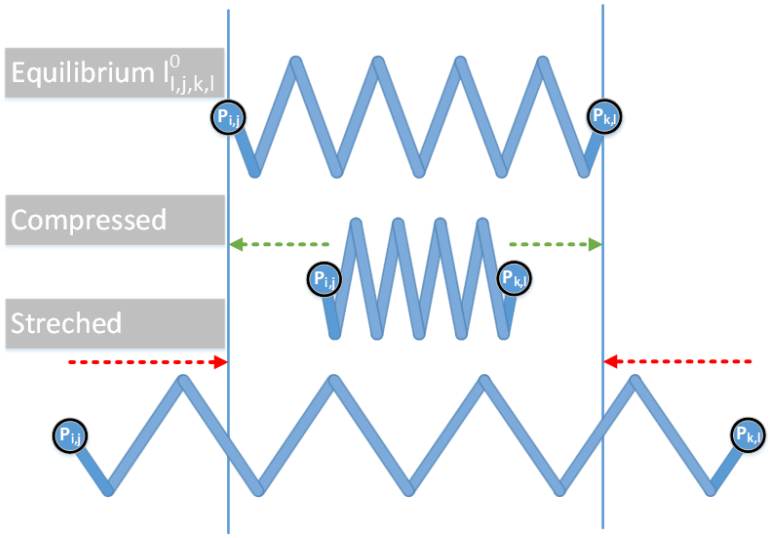


Figure 30: The three states of a spring: equilibrium, compressed and stretched. The green vectors represent a pushing force and the red a pulling force.

Equation 4 however, does only cover the force between two points connected via. a spring. If we define the set of particles that a particle $P_{i,j}$ is connected to via the springs as R, then the internal force $\mathbf{f}_{int}(P_{i,j})$ is equal to

the sum of forces of the connected springs:

$$\mathbf{f}_{int}(P_{i,j}) = -\sum_{k,l \in R} k_{i,j,k,l}(\overrightarrow{P_{i,j}P_{k,l}} - l^0_{i,j,k,l})\frac{\overrightarrow{P_{i,j}P_{k,l}}}{||\overrightarrow{P_{i,j}P_{k,l}}||} \tag{5}$$

External Forces
The external forces $\mathbf{f}_{ext}$ is composed of gravity, dampening and viscosity. Gravity is defined by:

$$\mathbf{f}_{gra}(P_{i,j}) = \mathbf{mg} \tag{6}$$

Where g is a gravitational acceleration constant, which here on earth is approximately 9.8.
Dampening is introduced to approximate the dissipation of the mechanical energy of the model, and helps eliminate oscillation of the springs improving numerical convergence:

$$\mathbf{f}_{damp}(P_{i,j}) = -\mathbf{c}_{dis}\mathbf{v}_{i,j} \tag{7}$$

Where c_{dis} is a dampening coefficient and $v_{i,j}$ is the velocity of the point $P_{i,j}$.
Finally a viscous fluid moving at a uniform velocity $\mathbf{v}_{fluid}$ applies a force $\mathbf{f}_{visc}$, to the surface of a moving body with velocity $\mathbf{v}$ and unit normal n, given by:

$$\mathbf{f}_{visc} = \mathbf{c}_{vi}[\mathbf{n}(\mathbf{u}_{fluid} - \mathbf{v})]\mathbf{n} \tag{8}$$

So in our case for a point of the mesh:

$$\mathbf{f}_{visc}(P_{i,j}) = \mathbf{c}_{vi}[\mathbf{n}_{i,j}(\mathbf{v}_{fluid} - \mathbf{v}_{i,j})]\mathbf{n}_{i,j} \tag{9}$$

Where the $\mathbf{n}_{i,j}$ is the unit normal of the point $P_{i,j}$

Solving the Differential Equation
Isolating the acceleration from equation 2 yields:

$$\ddot{\mathbf{u}} = \frac{\mathbf{f}}{\mathbf{m}}$$

Remembering that acceleration, or $\ddot{u}_{i,j}$, is the 2nd derivative of the position with respect to time, we can use this expression. We can separate the 2nd order differential equation into two coupled first order equations:

$$\dot{\mathbf{v}} = \frac{\mathbf{f}(\mathbf{u}, \mathbf{v})}{\mathbf{m}} \tag{10}$$

$$\dot{\mathbf{u}} = \mathbf{v} \tag{11}$$

which are solved analytically by:

$$\mathbf{v}(t) = \mathbf{v}_0 + \int_{t_0}^{t} \frac{\mathbf{f}(t)}{\mathbf{m}}\, dt$$

$$\mathbf{u}(t) = \mathbf{u}_0 + \int_{t_0}^{t} \mathbf{v}(t)\, dt$$

Meaning that starting from the initial position $u(t) = u_0$ and velocity $v(t) = v_0$ the integrals sum the infinitesimal changes up to time t.
Simulation of the cloth corresponds to updating the position of the particle in the mesh continuously done by calculating $u(t)$ and $v(t)$ from t_0 to the current time.

The simplest way to solve the equations numerically is to approximate the derivatives with finite differences as seen below:

$$\dot{\mathbf{v}} = \frac{\mathbf{v}^{t+1} - \mathbf{v}^t}{\Delta t} + O(\Delta t^2) \tag{12}$$

$$\dot{\mathbf{u}} = \frac{\mathbf{u}^{t+1} - \mathbf{u}^t}{\Delta t} + O(\Delta t^2) \tag{13}$$

Where t is the current step in the simulation(or frame number) and Δt is a discrete timestep. If we substitute these approximations into the equations 10 and 11 we get the update rules:

$$\mathbf{v}^{t+1} = \mathbf{v}^t + \Delta t \frac{\mathbf{f}(u^t, v^t)}{m} \tag{14}$$

$$\mathbf{u}^{t+1} = \mathbf{u}^t + \Delta t\ \mathbf{v}^t \tag{15}$$

Equation 14 and 15 is one of the simplest explicit integration schemes commonly used called the explicit Euler integration scheme[29]. This method has the disadvantage that it needs really small time steps for the equations not to blow up and reach infinity.
Several other both explicit(e.g Runge-Kutta[29], Verlet integration[29] [27]) and implicit integration(e.g.Implicit Euler [3] [2]) schemes exists. Implicit methods are generally more accurate but are also slower, and as we are using it for a real-time simulation in a pipeline running multiple other algorithms, speed is prioritized here. Yet, for this project, we have chosen to use explicit Verlet integration as it is more stable and accurate than the explicit Euler method. Verlet is chosen over the also widely used explicit Runge-Kutta method, which uses 4. order Taylor series approximation of the derivatives since comparison on their ability to deliver stable real-time results have shown that Verlet integration is superior taking computation time into account[26][27].

Verlet Integration
The idea of the method is to keep the position of the previous time step $\mathbf{u}(t-\Delta t)$ to improve the approximation of the derivatives from equations 12 and 13. The Verlet integration scheme uses two third order Taylor expansions of the position, one in each direction of time:

$$\mathbf{u}(t + \Delta t) = \mathbf{u}(t) + \dot{\mathbf{u}}(t)\Delta t + \frac{1}{2}\ddot{\mathbf{u}}(t)\Delta t^2 + \frac{1}{6}\dddot{\mathbf{u}}(t)\Delta t^3 + O(\Delta t^4)$$

$$\mathbf{u}(t - \Delta t) = \mathbf{u}(t) - \dot{\mathbf{u}}(t)\Delta t - \frac{1}{2}\ddot{\mathbf{u}}(t)\Delta t^2 - \frac{1}{6}\dddot{\mathbf{u}}(t)\Delta t^3 + O(\Delta t^4)$$

Adding the two equations and isolating $\mathbf{u}(t + \Delta t)$ gives us the update rule for the postion of a point:

$$\begin{aligned}\mathbf{u}(t + \Delta t) &= 2\mathbf{u}(t) - \mathbf{u}(t - \Delta t) + \ddot{\mathbf{u}}(t)\Delta t^2 + O(\Delta t^4) \\ &= \mathbf{u}(t) + [\mathbf{u}(t) - \mathbf{u}(t - \Delta t)] + \frac{\mathbf{f}(t)}{\mathbf{m}}\Delta t^2\end{aligned} \tag{16}$$

Thus the model does not use the velocity of a point directly for estimating the next time step.

The Mass Spring Model and the Deformable Screen
We wish to use the Mass Spring model to estimate missing/occluded points of the deformable screen, which is quite different to its traditional use for cloth simulation where all points of the mesh are estimated and updated for each frame. Also, the forces are different from a conventional simulation, and can actually be simplified. As the screen is made of a highly retractable fabric fastened to a rigid frame, the gravity force can be canceled. Also, the $\mathbf{f}_{visc}$ can be removed from the model as we do not want to simulate e.g. wind or other forces of flow

applied to the screen.
Equation 16 gave us an updating scheme for the position of a point $P_{i,j}$, and the basic Verlet integration scheme does not compute the velocity. But the dampening force $\mathbf{f}_{damp}(P_{i,j})$ from expression 7 relies on the velocity, so we need to use an ad-hoc method scaling the change in position to create a dampening effect:

$$\mathbf{u}(t + \Delta t) = \mathbf{u}(t) + \mathbf{c}_{dis}[\mathbf{u}(t) - \mathbf{u}(t - \Delta t)] + \frac{\mathbf{f}(t)}{\mathbf{m}}\Delta t^2 \tag{17}$$

Remembering $\mathbf{c}_{dis}$ is the dampening coefficient, which here is multiplied to the change in position for a single timestep.
Lastly, for our deformable screen, the actual position of the majority of the points are known(non-occluded points), so they do not need to be estimated. Though we need keep track of the previous positions to be able to compute an estimate using the Verlet integration when markers are lost in the visual tracking due to occlusion. Like the Extrapolation model, this model allows estimates of estimates(estimates produces based on an occluded neighborhood).

6.1.8 Update Previous Points

For each iteration of the pipeline in figure 19 the positions of the points are updated.
When the post-processing is done, the points positions are passed on to the visualization program covered in the section.

6.1.9 Limitations of the Screen Tracker Pipeline

Early in the project, before settling for a specific design, it became apparent that the **Point Matching** and **Estimate Missing Points** steps in the pipeline would be the hardest to successfully implement, and central to the tracking. We also quickly experienced how the two tasks have a heavy influence on each other, because a bad estimate can result in that the position of an occluded marker is estimated close to a visible marker, potentially causing them to swap ID's in the **Point Matching**. Contrary, if a marker gets its ID swapped with another in the **Point Matching**, it is very likely to move away from its neighboring markers, and its actual position, causing bad estimates if it is used to estimate other occluded points. This can even cause a chain reaction, where a swapped ID causes bad estimates for occluded markers, which is then matched incorrectly in the **Point Matching** step.
As described we experimented with different more or less sophisticated **Point Matching** approaches requiring a substantial amount of work, and experienced that it was hard to develop a **Point Matching** method without heaving a good estimation model in place. Thus, after obtaining a fairly good **Point Matching** using the Hungarian Algorithm, we decided to dedicate the rest of the project to investigate models for estimating the position of occluded markers. This work has resulted in the three estimation models presented earlier in this section, which we have implemented and we evaluate later in this report.
Unfortunately, as the **Point Matching** is not as good as we would like because it is prone to swapping ID's of visible markers when they pass each other and has the same issue if a marker is estimated close to another, the tracking becomes unstable when a dense pattern of markers is used. Later in the report, we will test the breaking point of the system, meaning how much we can deform the screen before the ID's of the markers are swapped around.

Inspecting the estimation models shows a weakness in the Spring and Displacement in their ability to estimate an occluded marker if its positions it further out on the on the Z-axis than its neighbors. Figure 31 shows how the models estimate an occluded marker when the screen is pulled directly on an occluded marker in the z-axis. As the figure contain a lot of information we will explain it step by step from the left. Seen in the leftmost side at (**a) No Estimate** we have drawn the screen seen from a 2D sideways perspective(Y, Z), and with only a single strip of markers in the y-axis. The gray line shows the relaxed screen with 5 reflective markers shown as gray

circles, and the black line shows how the screen looks when being pulled in the z-direction. The black circles on the line indicate visible markers, while the red circle is an occluded marker. In this case, the screen is pulled directly on the occluded marker.
The second illustration from the left shows how **(b) Displacement model** estimates the occluded marker. The blue arrows show the **Displacement Vectors** and the green arrow shows the **Mean Displacement vector** explained earlier. The green circle shows the **Estimated marker** positioned in the original position of the marker plus the **Displacement Vector**. We see that the estimated position is quite far away from the actual position, further away on the z-axis than its immediate neighbors.
Moving into the second illustration from the right, the **(c) Mass Spring model**, we see how the model estimates the position of the occluded marker. The blue dashed line indicates the **Flexion Springs** and the green lines the **Structural Springs**. Because the illustration is in 2D, we cannot show the sheer springs here. As the estimates of the Mass Spring model are integrated over time, the estimate in the figure illustrates when the marker has found a resting place where the forces of the springs cancel each other out. Before finding this resting place, the estimated position of the occluded marker can oscillate in both directions on the z-axis giving estimates of varying quality.
Finally the estimate of the **(d) Extrapolation model** rightmost in figure 31, given by the intersection of the **Extrapolation Lines** is very close to the actual position.

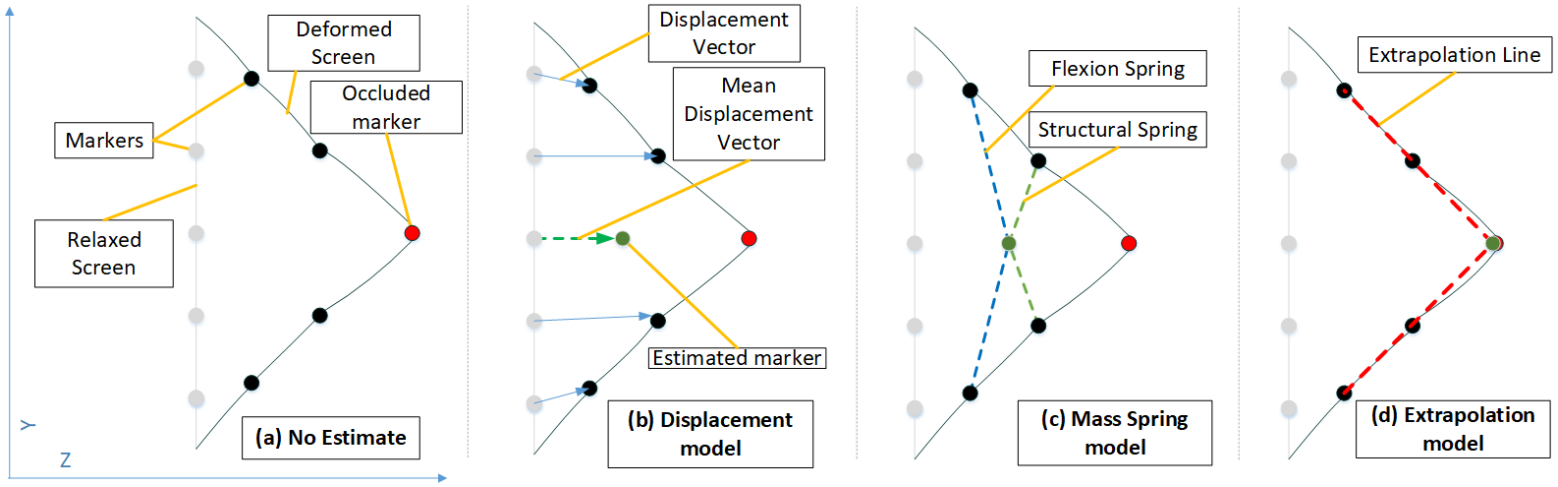


*Figure 31: (a) A 2D(Y, Z) Illustration of the screen begin grabbed directly on a marker and deformed in the z-direction. Here the gray line is the relaxed screen prior to the deformation, and the black line is the deformed screen. (b) Shows the estimate of the **Displacement model** as the green circle. (c) Shows the estimate of the **Mass Spring model**. (d) Shows the estimate of the **Extrapolation model**.*

6.2 Visualization

The second part of the system models the tracked screen as a 3D-model which can be projected back onto the screen. In figure 32, we see the visualization pipeline.

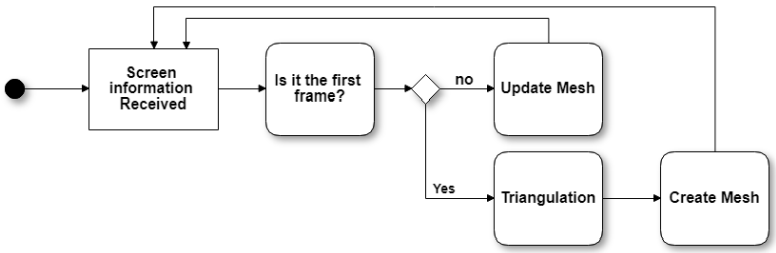


Figure 32: Visulazation pipeline.

Starting from the first step **Screen Information Received**, when data is received from the ScreenTracker it is parsed as points with 3D coordinates and ID. Starting from the first frame, we construct the 3D-model of the screen using what is known in 3D Computer graphics as a polygon mesh. A mesh is a collection of vertices, edges, and faces that together define the shape of an object in a scene. A mesh can be constructed of all sorts of polygons, but most commonly triangles are used to obtain the best fidelity. Thus the first task in creating the 3D-model is to define a list of triangles from the received coordinates, also simply called triangulation. Often the Delaunay algorithm[1] is used to produce a triangulation where the minimal angles of the triangles are maximized to obtain higher fidelity and to minimize tearing of the texture. Since we are dealing with a mostly flat surface, we can simply perform a 2D triangulation of the x- and y-coordinates, and add the z-coordinate afterward. When the mesh has been constructed, the pipeline returns back to the **Screen Information Received** step listening for the next frame. When the next frame is received and onwards, we already have a triangulated mesh, and we only need to update the positions of the vertices in the mesh.

6.3 Calibration of the Projector and the Depth Camera

The ProCam setup requires calibration in order to work because the camera and Kinect do not have the same position and orientation in the scene as seen in figure 34. Also, even if the projector and camera could be placed in exactly the same position and orientation, specific features in the construction of both the camera and projector can cause differences to the perspective in which they observe the scene respectively.
The Kinect is not only able to capture an IR-frame with intensities and depth but also capable of finding the 3D real-world coordinates of a given place in the frame. However, if we projected an object seen by the Kinect directly in the same coordinates, there would be an offset and difference in shape between the observed and the projected object.
To fully understand why the calibration is necessary, we need to take a look at how a camera works. This section includes a lot of terminologies and we have tried to ease the readability by highlighting technical terms with a **bold font** the first time they are used.

6.3.1 Pinhole Camera Model

Taking a look at the **pinhole camera model** seen in figure 33 helps clarify. When looking at a **3D object** in a scene through a **pinhole**, light is cast from the object through the hole and projected upside-down on the back-wall or **image plane** as a **2D image**. The distance between the pinhole and the image plane is called the **focal length**, and often a **virtual image plane** is defined as the image plane turned back upside-up, placed perpendicular to the image plane, on the other side of the pinhole at the same distance. In the following text, we will refer to the virtual image plane just as the image plane.

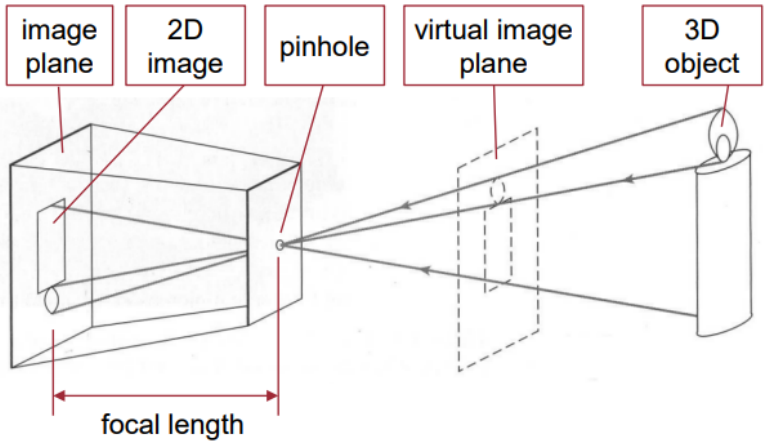


Figure 33: Illustration of the pinhole model (From Kheng [19]).

Figure 34 shows a simplified version of the ProCam-setup where the projector projects from one position onto the 3D-scene, and the camera captures the same scene just from another position and angle. The projector can be considered as an inverse camera where light from the image plane is projected instead of captured. In front of the camera and projector, their image planes are seen, and it is rather clear that a point in the scene is placed very differently on their image planes.

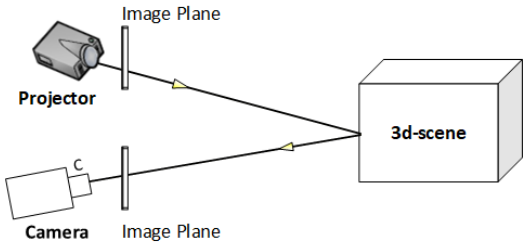


Figure 34: Projector and camera in the ProCam-setup.

It is very important that the camera and projector perceive the scene identically when matching the tracking and projection of the system. Thus we need to perform a transformation between what is seen by the camera and emitted by the projector which is where the calibration comes in.

6.3.2 Central Projection Model

The **Central Projection Model** seen in figure 35 expands on the basic pinhole model, and describes more specifically how points from the real 3D world are mapped to the 2D world of the image plane(or vice versa). The figure shows a coordinate-system (x_c, y_c, z_c) defining the 3D **Camera Space** of the Central Projection Model with origin in the **center of projection C**(the pinhole) and the distance $z = f$(focal length) to the image plane. In figure 35 we see a blue 2D coordinate system (x', y') spanning the image plane, which we will call **Optical Space**. The origin of the Optical Space is called the **principal point** and lies perpendicularly from the Center Of Projection onto the image plane.

[19] `https://www.comp.nus.edu.sg/~cs4243/lecture/camera.pdf` Accessed: 19/04-2018

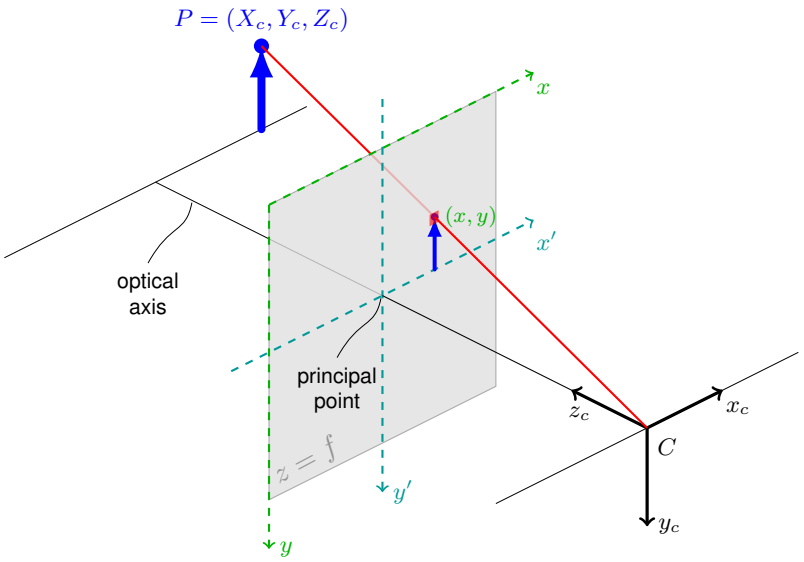


Figure 35: Point P in the 3D coordinate space. The light blue coordinate system defines the Optical Space, and the light green coordinate system describes the discrete pixel coordinates of the Pixel Space (Figure adapted from [20]).

The 2D position(Optical Space) of the 3D-point $P = (X_c, Y_c, Z_c)$ seen from the camera(Camera Space) on the image plane can be computed geometrically as:

$$x' = f\frac{X_c}{Z_c} \tag{18}$$

$$y' = f\frac{Y_c}{Z_c} \tag{19}$$

This gives us the projection equations:

$$Zx' = fX \tag{20}$$

$$Zy' = fY \tag{21}$$

$$Z = Z \tag{22}$$

Which can be expressed as the homogeneous transformation matrix:

$$z_c \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \underbrace{\left[\begin{array}{ccc|c} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right]}_{P_0} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{23}$$

Where P_0 is called the **camera projection matrix**.
In a digital camera the image plane is discretized as pixels, and the image plane is most often defined by their own 2D pixel coordinate system, which we will call **Pixel Space** (x,y), with origin in the left upper corner of the Optical Space. So we want to convert from mm to pixels, and move the center of the coordinate system up in the left corner of the image plane seen in figure 36.

[20] `https://tex.stackexchange.com/questions/96074/more-elegant-way-to-achieve-this-same-camera-perspective-projection-model` Accessed: 19/04-2018

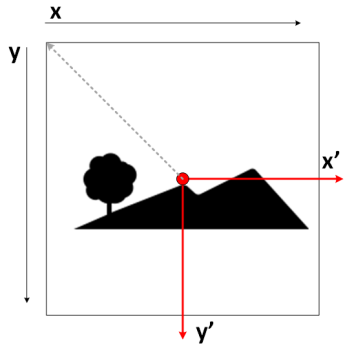


Figure 36: The image plane of a camera. The x,y-coordinates represents the discrete coordinates of the Screen space, and the x',y'-coordinates represent the metric Optical space.

This is done simply by shifting the position and scaling the coordinates into pixels:

$$(x - O_x) = \underbrace{\frac{x'}{s_x}}_{a_x} \tag{24}$$

$$(y - O_y) = \underbrace{\frac{y'}{s_y}}_{a_y} \tag{25}$$

Where x and y are the columns and rows of the discrete image plane, O_x and O_y defines the optical center(the principal point), and s_x and s_y is the pixel width and height respectively. The optical center should preferably be in the center of the image plane, but can have a small offset due to misalignment in the construction of the camera. Also, misalignment, as e.g. a lens dragged downwards by gravity, can produce skew in the image. Thus we need to shift the image into the left corner, scale from mm to pixels, and correct the skew. Bringing this together gives us:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_x & s & p_x \\ 0 & a_y & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \tag{26}$$

Where a_x, a_y is pixel scaling factor, p_x and p_y is the principle point, and s is a skew factor.
The two matrices (from eq. 26 and 23) can be combined to:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_x & s & p_x \\ 0 & a_y & p_y \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{array}{ccc|c} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right] \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{27}$$

Simplified to:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \underbrace{\begin{bmatrix} a_x f & sf & p_x \\ 0 & a_y f & p_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right] \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{28}$$

Where **K** is called the **calibration matrix**.
To summarize, a coordinate can be transformed from the 3D Camera Space into the 2D Pixel Space by:

- First, transforming the 3D camera coordinates to the 2D Optical Space.
- Discretizing the 2D point from the Optical Space, to the Pixel Space by the scale factor.
- Correcting the offset from between the Optical Space and the Pixel Space by shifting.
- Applying the potential skew using a skew factor.

6.3.3 Radial Distortion

Some cameras have curved lenses(e.g. fisheye lenses with a large field of view) which leads to distortion of the image proportionally to distance from the center of the image, which is called radial distortion. The image can be rectified using the formula:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_d \\ y_d \end{bmatrix} (1 + k_1 r^2 + k_2 r^4 + k_3 r^6 ...) \tag{29}$$

Where (x, y) are the undistorted coordinates, (x_d, y_d) are the distorted, k_i are distortion coefficients obtained through a calibration process, and r is the radius from the center of projection.
Together with the values from the calibrations matrix, the radial coefficients form what is called the **camera intrinsic parameters** or just **camera intrinsic**.

6.3.4 Transforming Coordinate System

In computer graphics, often a universal coordinate system called world coordinates or **World Space** is used. The world coordinate system is independent of the camera coordinate system and other coordinate systems defining objects in a scene(Model Space). This means that the camera, having its own coordinate system(Camera Space), can be placed arbitrarily in the World Space. Thus a point in World space X_w has to be transformed into the Camera Space X_c, which is done by performing a rotation with a 3X3 rotation matrix **R**, and translation with a 3x1 translation vector **t**:

$$X_c = \begin{bmatrix} R_{3x3} & t_{3x1} \\ 0 & 1 \end{bmatrix} X_w \tag{30}$$

The parameters of the rotation matrix and translation vector are called the **camera extrinsic parameters** or just **camera extrinsics**

So if we want to represent a point from X_w(World Space) in Pixel Space coordinates x we have to: 1) transform the world coordinate into Camera Space, 2) transform the camera coordinate into Pixel Space. If we ignore the skew, this is done by combining:

$$x_{pixel} = K_{3x3}[I; 0]_{3x4} X_c \tag{31}$$

$$= K[R, t] X_w \tag{32}$$

Or written with all the parameters:

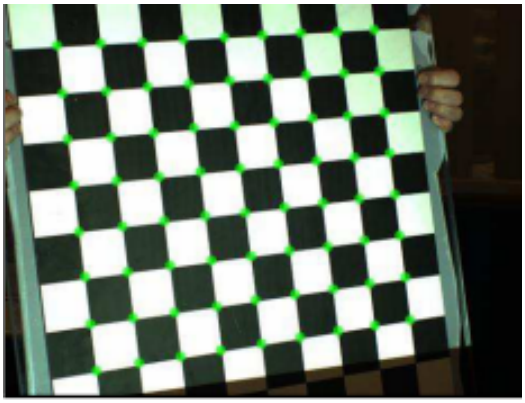
$$z \underbrace{\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}_{x_{pixel}} = \underbrace{\begin{bmatrix} a_x f & sf & p_x \\ 0 & a_y f & p_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K}} \underbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{33} & r_{32} & r_{33} & t_3 \end{bmatrix}}_{\mathbf{R} \in \mathbb{R}^{3x3} \quad \mathbf{t}} \underbrace{\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}}_{X_{world}} \tag{33}$$

6.3.5 Camera Calibration

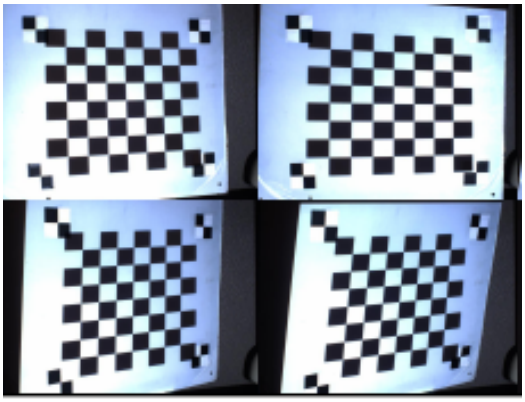
Camera calibration is the task of estimating the intrinsic and extrinsic parameters:

- **Intrinsic Parameters**
  - f focal length (in pixels).
  - The skew factor s.
  - (p_x,p_y) image center (in pixels).
  - k_1, k_2,... radial distortion parameters.
- **Extrinsic Parameters**
  - The 9 values of the rotation matrix **R**.
  - The 3 values of the translations vector **t**.

From these parameters, we can construct the matrices of Eq. 33 and their inverse counterparts, making us able to move from a coordinate system and perspective to another.
To obtain the parameters, often an approach by Zhang[59] is used, where a 2D pattern with known dimensions, e.g. a printed checkerboard(figure 37a), is photographed multiple times in varying positions and angles. By examining the correlations between the known positions of points, in this case, points in the intersections of the pattern of the checkerboard, and their positions on the projected and captured images, the algorithm is able to compute the intrinsic parameters as well as the extrinsic parameters. An extension of this metod[20] first calibrate the camera using Zhang's method, and then treats a projector as an inverse camera, and obtains the "camera" intrinsic and extrinsic parameters of the projector. This is done by projecting a checkerboard onto a plane surface, again with various angles and positions(figure 37b), capturing it with a camera, and examining the correlation of the pattern in the projected checkerboard and the captured frames from the camera.



(a) Physical checkerboard with known dimensions for camera calibration [20, fig 3.3(a)].



(b) Projector calibration sequence containing multiple views of a checkerboard projected on a white plane marked with four printed fiducials in the corners. As for camera calibration, the plane must be moved to various positions and orientations throughout the scene [20, fig 3.5].

Figure 37: Checkerboards used for camera c-alibration.

Another popular method of achieving the parameters in a ProCam-setup is by pattern codification[41] using structured light. Often gray codes [57] [22], here seen as a series of vertical and horizontal lines(e.g. in figure 38) are used to uniquely encode each pixel of the projected image. For a given frame of the gray codes, a pixel is assigned its binary value from the image, 1 if it is lit up, and 0 otherwise. The projected image is

captured by the camera, and the frame is decoded to get the correspondence in position from each pixel in the projected image to the captured image. The mapping from the projected to the captured pixels can then be used to compute the intrinsic and extrinsic parameters of the camera and projector.

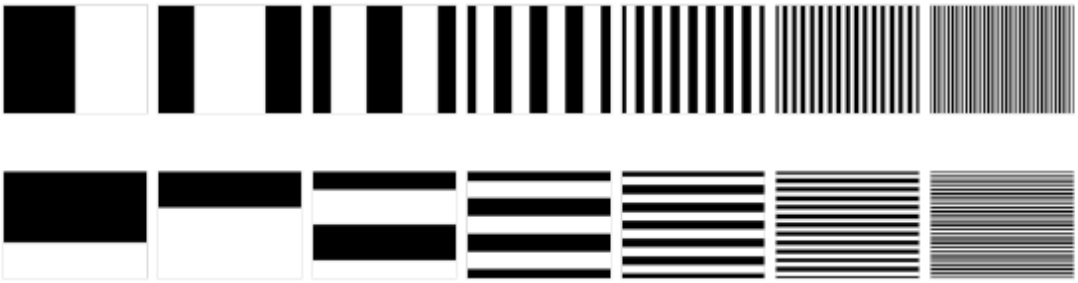


Figure 38: Example of gray codes used for camera calibration (from Yamazaki et al.[57]).

7 System Implementation

In this section, we will go through the content of the Design section and describe how it has been translated into an actual implementation. Although this section will be more code specific, we will still use some abstraction to give a better overview of the system. This is achieved with UML class diagrams, and references to the UML inspired pipelines from the design section.
Since class diagrams of entire systems often are complex and cumbersome, we have only included the class names and associations in our complete diagrams, leaving out methods and fields. In the text, we will mark packages and references to elements in figures with a **bold** font. Classes, interfaces, and objects will be marked with ***bold italic***, and fields and methods with a `typewriter font`. Moreover, we will differentiate between fields and methods in the text by adding a set of empty parentheses after the name of methods as: `method()` (not meaning that the method does not take any arguments).
Like the Design section, we have divided the section into three parts: ScreenTracker, Visualization, and Calibration of the ProCam. Furthermore, We will start each section by providing an overview of the libraries used in the implementations.

7.1 ScreenTracker - C#

We have developed the program for tracking the reflective markers of the screen in C#, and we will simply call the program ScreenTracker[21]. The choice of language is due to Microsoft providing their Kinect SDK[22] in C#, making it the natural choice for development.
Besides the SDK, the program uses the libraries:

- **EmguCV** -framework[23] which is a C#-wrapper for the widely used Computer Vision C++ library OpenCV[24].
- **Accord.net**-framework[25] has been used for experimenting with kd-trees, and is still used for statistical methods such as median, which is not a part of the standard math library.
- **CSVHelper**, Finally the CSVHelper library[26] has been used for creating log-files with data used for evaluation of the ScreenTracker program.

We have divided the ScreenTracker program into four major packages, as seen in the class diagram in figure 39, with the following functionality:

- **DataReceiver** - Responsible for getting the data/frames from the camera and convert it into a format the rest of the program supports.
- **DataProcessing** - Processes the data and tracks the reflective markers in the images delivered by the **DataReceiver**.
- **GUI - Debugging** - Is a Graphical User Interface(GUI) enabling the user to see the frames from the camera with the tracking information on top.
- **Communication** - Is responsible for communicating with the visualization program.

[21] Available at `https://github.com/MultiPeden/ScreenTracker` Accessed: 19/04-2018
[22] `https://msdn.microsoft.com/en-us/library/dn799271.aspx` Accessed: 19/04-2018
[23] `http://www.emgu.com/wiki/index.php/Main_Page` Accessed: 19/04-2018
[24] `https://opencv.org/` Accessed: 19/04-2018
[25] `http://accord-framework.net/` Accessed: 19/04-2018
[26] `http://joshclose.github.io/CsvHelper/` Accessed: 19/04-2018

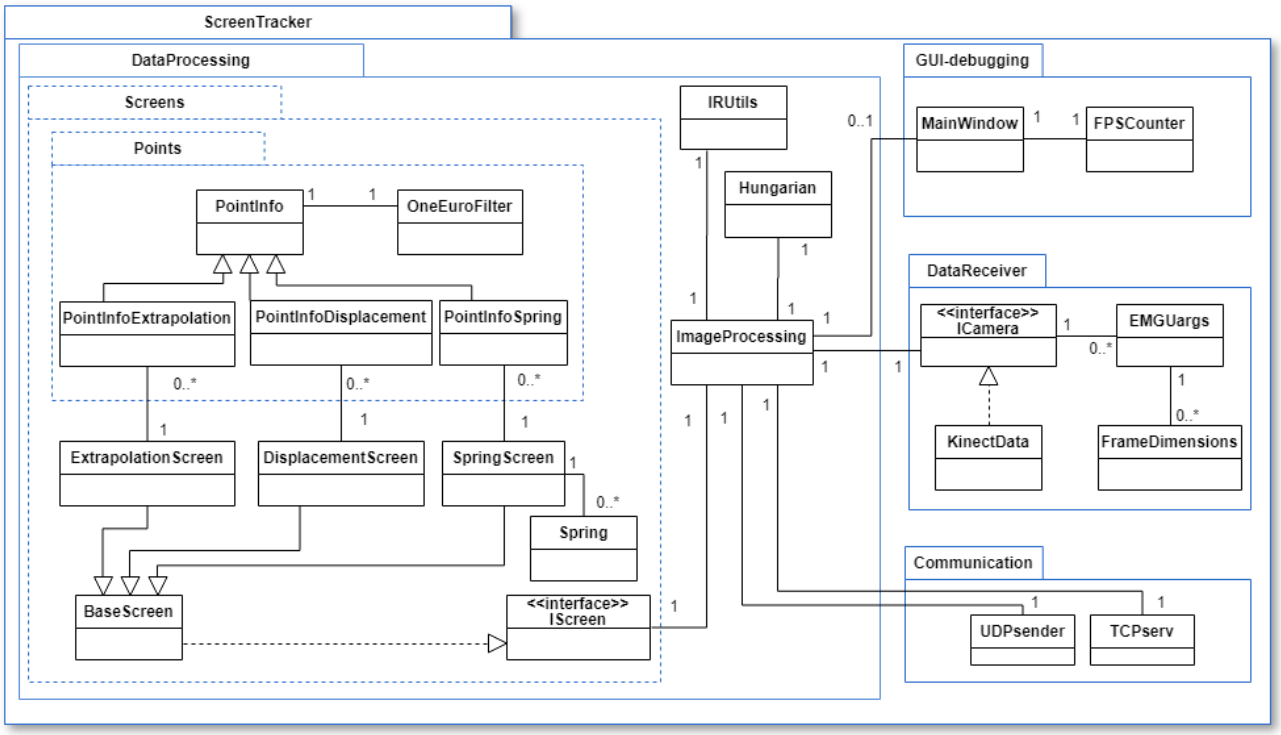


Figure 39: The ScreenTracker Class-diagram shown here without methods and fields to provide a simplified overview of the system. The Dashed lines for the Screen- and Point-package indicates that they are sub-packages of the DataProcessing package.

7.1.1 GUI - Debugging

As we would like for others to be able to expand upon the framework in the future, and let them create their own deformable screens to the specifications they need, we decided to include an optional GUI to visualize the tracking for the user. The **GUI-Debugging**-package contains the class ***MainWindow*** used to control a Windows Presentation Foundation (WPF, a part of the .NET framework) view seen in figure 40. The other class of the package, the ***FPSCounter***, is used to maintain an FPS-counter shown in the bottom right in the GUI as seen in figure 40.

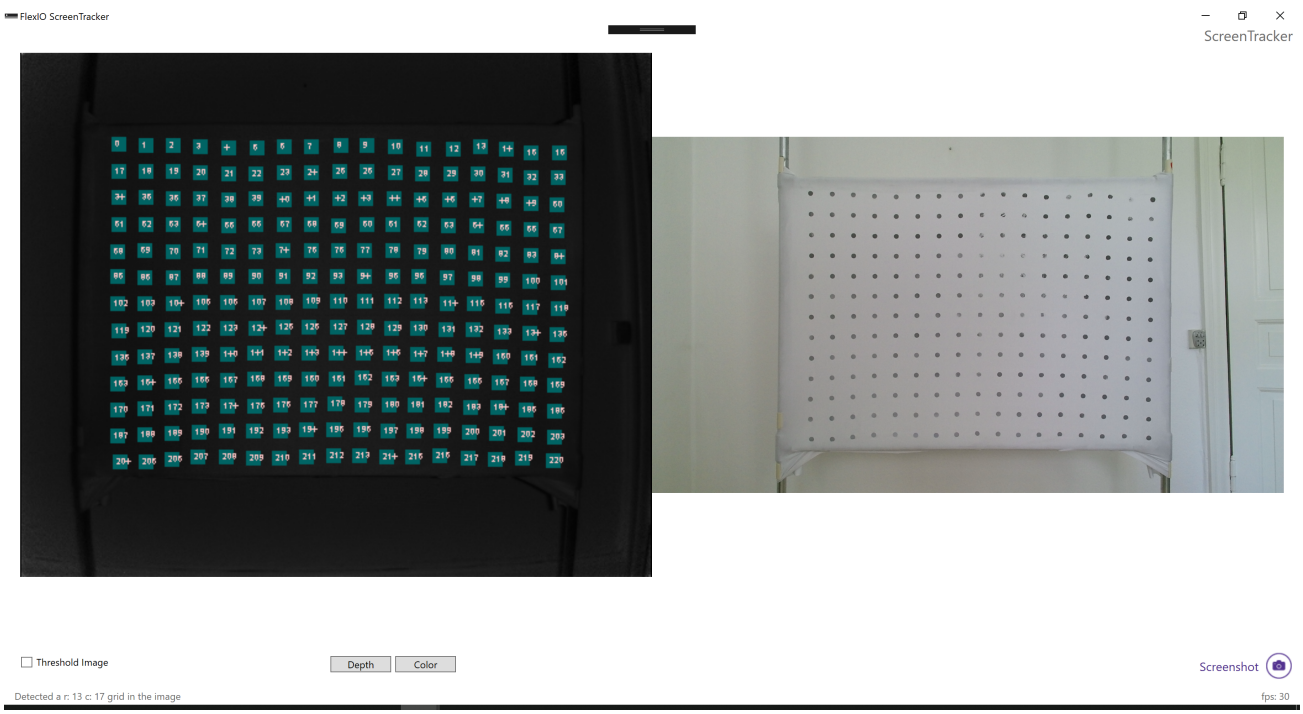


Figure 40: Screenshot from the debugging GUI having an IR-image with the tracked data to the left, and color image to the right.

To the left, in figure 40, we see the infrared frame overlaid with cyan colored rectangles surrounding the tracked markers. Also, the markers IDs are seen as the white numbers inside each rectangle. To the right, we see the color-image from the Kinect, and in the bottom, we see some control buttons and a status text.
The Control buttons include a checkbox for switching the left image to a thresholded version, as well as two buttons for switching the right image between color image en depth image, as seen in figure 41.

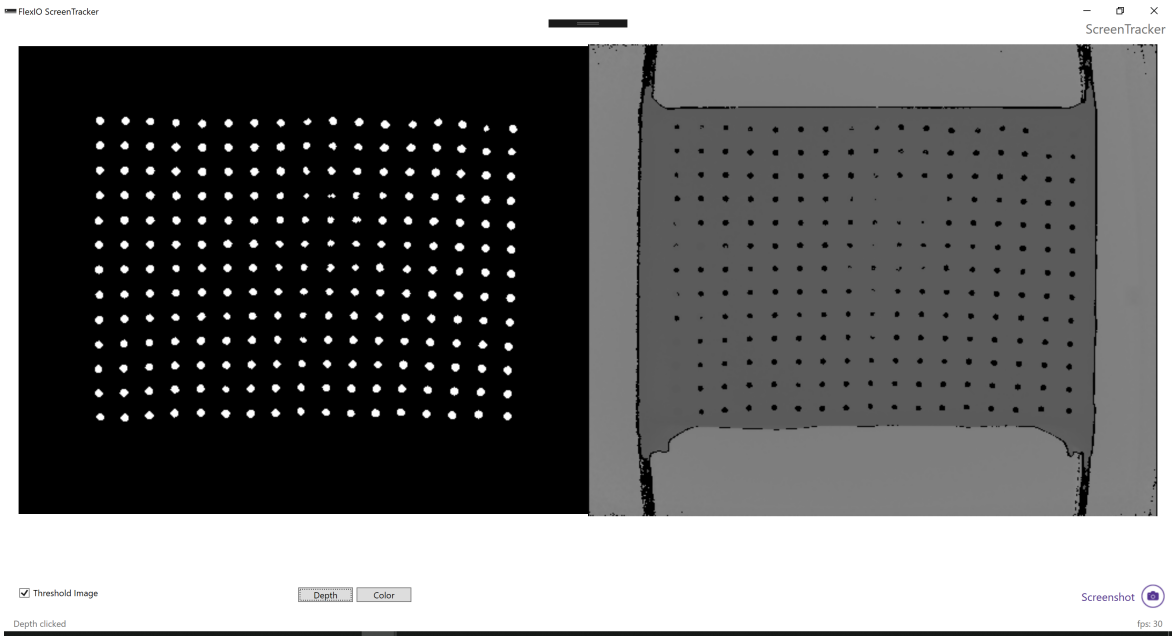


Figure 41: Screenshot from the debugging GUI showing the thresholded version of the infrared frame to the left, and the Kinect's depth image to the right.

The status text in the bottom-left corner of the GUI from figure 40 and 41 is updated to display any messages from the program. The variables of the ***MainWindow***-class are used to store the state of buttons clicked and the status text. The methods of the class are used to ensure that the frames are shown according to which buttons are pressed.

7.1.2 DataReceiver

Figure 42 shows the **DataReceiver**-package, from the class diagram from figure 39, here with the most important fields and methods. As mentioned in section 5.2, about the camera and physical design, we would like our implementation to be as independent of the Kinect camera as possible why we define an interface for cameras. The ***ICamera*** interface is responsible for ensuring a contract such that various IR and depth cameras can be used together with the rest of the program, simply by writing a class implementing this interface. This means that the ***ICamera*** interface is the link between the **DataReceiver** package and the **DataProcessing** package also seen in figure 42. Most importantly, a class implementing the interface, needs to convert frames(color-, infrared-, and depth-frames) received from the camera into ***Emgu.Mat*** matrix structures(from the computer vision package) and wrap them in the ***EMGUargs*** class. When all frames are ready, the event `EmguArgsProcessed` should be triggered to pass on the frames from the **DataReceiver** package to the **DataProcessing** package.
The other event `ChangeStatusText` is only used to pass on information if the Camera's status has changed e.g. if it has been connected/disconnected. The `GenerateColorImage()` method is used to toggle if the color-frame should be generated as it is only used for the debugging GUI.
The `ScreenToWorldCoordinates()` method maps from the 2D screen space of the IR-frame to the metric world 3D coordinates, seen from the cameras view, provided a list of the 2D coordinates and their associated depth/z-coordinate. `CameraToIR()` maps the other way around. `IrFrameDimension()` returns the dimensions of the camera's Infrared frame, used for bounds checking in the **DataProcessing** package. Finally `CameraToColor()` maps coordinates from the 3D metric camera space, to the 2D color frame.

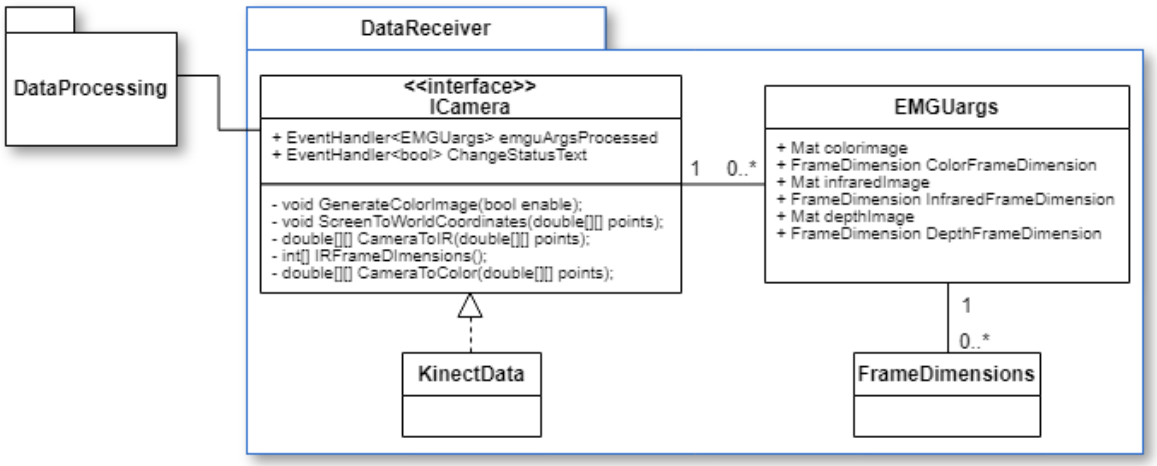


*Figure 42: The **DataReceiver**-package from the class diagram from figure 39 with the most important fields and methods.*

The KinectData class implements the ***ICamera***-interface, listens for events generated through the Kinect's SDK when frames are received, converts the frames to the ***EMGU***-args, and pass them on using the `EmguArgsProcessed` event, linking the Kinect to the rest of the program. Furthermore, it uses the SDK's mapping functions to implement the three coordinate mapping functions of the interface.

7.1.3 Data Processing

As the name implies, it is where the data received from the **DataReceiver** is processed. It is in the ***ImageProcessing*** class the main logic of the ScreenTracker program is found, and as the class is rather complex, we will describe it referring to diagrams for clarity. Figure 43 shows the class diagram with focus on the ***ImageProcessing*** class of the **DataProcessing** package. Only the most central methods of the class is shown in the figure to keep it more manageable.

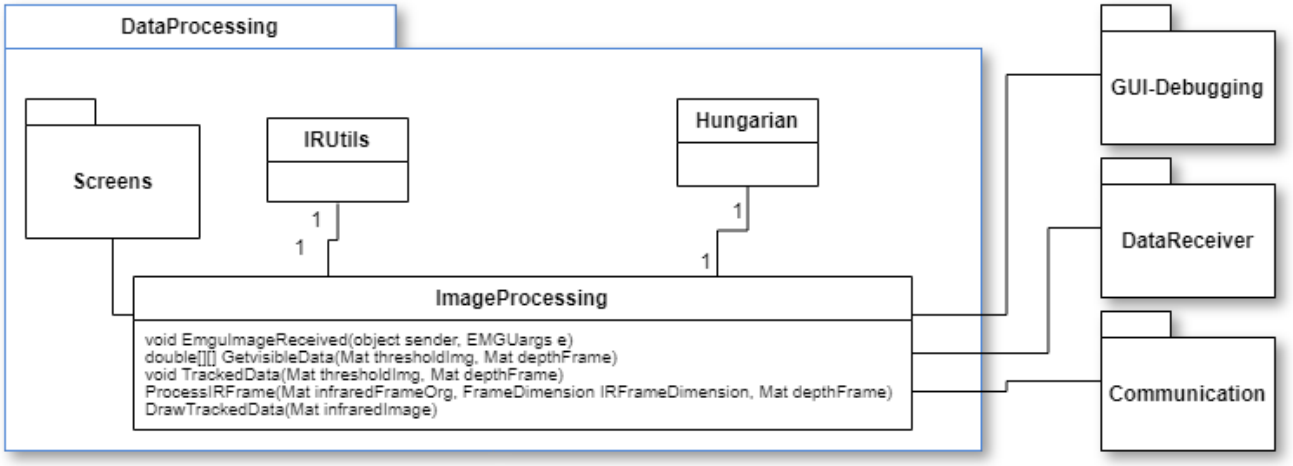


*Figure 43: The **DataProcessing**-package from the class diagram from figure 39 here with focus in the **ImageProcessing** class implementing the main logic of the ScreenTracker program. The **ImageProcessing** class is shown here only with the most essential methods.*

It is also in the ***ImageProcessing*** class the main tasks of the ScreenTracker pipeline, explained in the design section, are performed. The pipeline is shown again here in figure 44, and this time the steps are annotated with the methods in the ***ImageProcessing*** class performing the tasks.

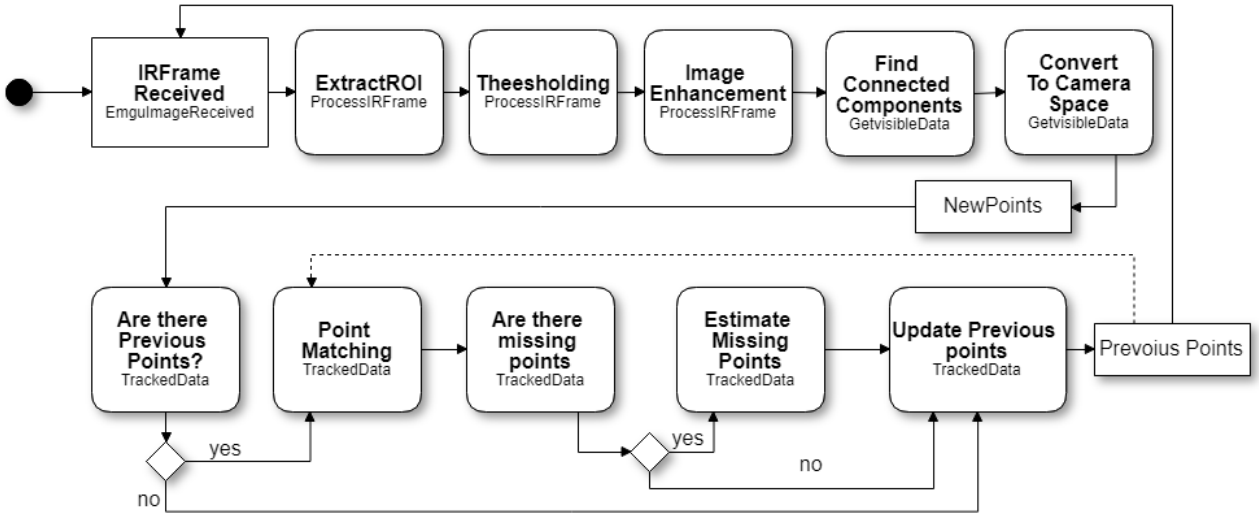


*Figure 44: The ImageProssesing pipeline from the design section, now annotated with associated methods of the **ImageProcessing** class performing the step in the pipeline.*

The following sections will cover the implementation of the pipeline step by step.

7.1.3.1 IRFrame Received

When the class implementing the ***IScreen*** interface, in this case the ***KinectData*** class, raises the `emguArgsProcessed`, it is handled by the ***ImageProcessing***'s `EmguImageRecived()` method which passes the IR- and depth-frame on to the `ProcessIRframe()` method. The IR-frame is delivered as (16-bit gray-scale), and the depth frame arrives as the distance in mm for the pixels(also 16-bit).

7.1.3.2 ExtractROI & Thresholding

The ExtractROI is a new step in the pipeline(compared to the design section), and it has been introduced to remedy unfavorable features of the Kinect camera. The IR-camera leaves unwanted artifacts in the image marked with red circles in the IR-frame seen in figure 45. The artifacts are seen as small, almost white, dots in the corners of the frame. Also, there is an area of around 4 pixels, marked in the figure with the red arrow, that flickers and is almost white for every third frame. The artifacts in the corners are probably due to distortion caused by the lens, and the flickering pixels might be due to a small defect in the ir-camera's sensor.

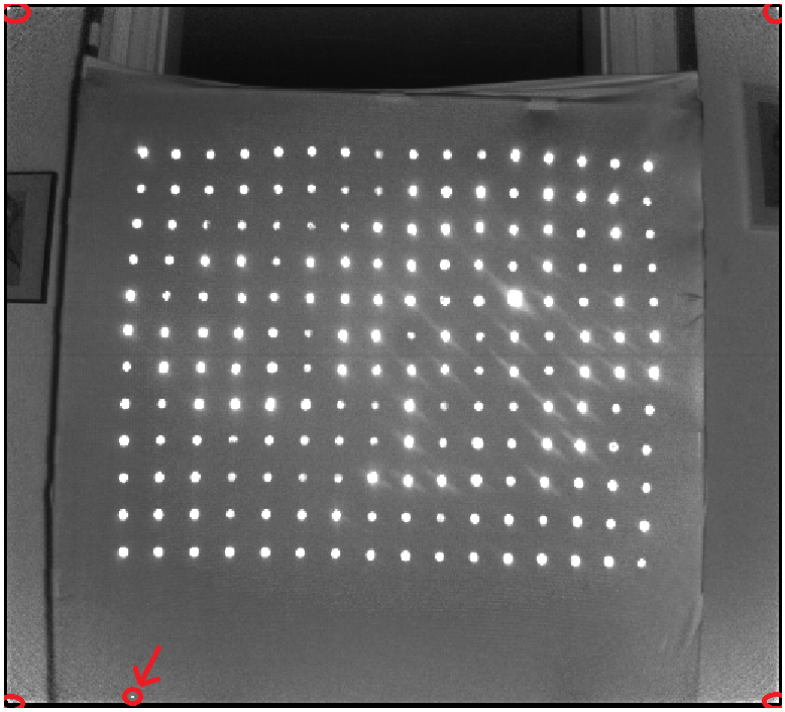


Figure 45: A frame from the Kinect's IR-camera showing some bright artifacts marked with red.

The artifacts are hard to spot on the image visually, but caused some issues in the tracking pipeline as they sometimes appeared as tracked markers for single frames, affecting the tracking heavily. We could filter out the artifacts later on in the pipeline, but as we do not necessarily need the entire frame, we decided to completely discard 10 pixels around the borders of the frame. This is done by setting the Region of Interest(ROI) of the EMGU Mat to a rectangular mask[27] with 10 pixels of padding. One could argue that this step is specific to the Kinect camera, and thus should be placed in the ***KinectData*** class, however we suspect that similar artifacts could be found for other cameras. Thus we have made the padding customizable, and the width can be adjusted by changing the value `IRPixelPadding` in the user settings file(appendix B).
From now on we will just refer to the ROI, as the IR-frame. Next, the IR frame is intensity thresholded using a Gaussian adaptive binary threshold method from the EMGU library with the parameters seen in the settings file(appendix B).

[27] with parameters X=10, Y=10, Width=492, Height=404

7.1.3.3 Image Enhancement

As explained in the design section, we perform morphological operations with a circular kernel to remove noise and enhance circular objects. In the implementation, we first dilate for two iterations enhancing the circular shape of the detected markers and then erode the image for a single iteration to shrink the detected markers in size again. The size of the kernel used for the morphology operations is 3x3 pixel but can be changed in the user setting file as needed. The morphological operations are performed using the EMGU-framework's `MorphologyEx()` method.

7.1.3.4 Find Connected Components

After the opening, the function `GetvisibleData()` is called with the IR- and depth-frame as arguments. `GetvisibleData()` calls EMGU's `ConnectedComponentsWithStats()` which finds statistical data such as centroid position, height, and width, for every connected component in the image. This gives us information about the positions of the reflective markers in the image, which we store as a jagged double array(`double[][]`) called `newPoints`.

7.1.3.5 Convert to Camera Space

The array `newPoints` currently contains the positions of the visible markers in the 2D pixel screen space of the IR-frame, but as described in the design section, we would like to track the markers by their metric 3D Camera coordinates giving us the real world position of the markers relative to the camera. First of all we need to obtain the Z-coordinate from the depth-frame which is found by looking up the position of a point(in screen-space) in the depth-image received from the "ICamera"-interface together with the IR-image. Since the IR-image and the Depth-image are captured using the same camera, they have identical resolutions and parameters(intrinsic and extrinsic), and we do not need any mapping from the IR-image to the Depth-image. However, as described in the design section, the reflectively of the markers causes errors in the depth image, and we need to estimate the depth of a marker using surrounding sample points. Here the data from `ConnectedComponentsWithStats()` comes in handy as it contains the width and height of a bounding box for the detected component, in our case a reflective marker. In figure 46 we see a an illustration of a reflective marker found in the IR-image surrounded by the bounding rectangular box. The red circles along the surrounding box in the cardinal and inter-cardinal directions defines 8 sample points which median z-value is used as an estimate for the depth of a marker. The median is preferred over the mean to minimize potential outliers' effect on the estimate. The sample points are taken 1 pixel from the surrounding box, as this gave more stable z-values. We have experimented with more sample-points, but this did not improve the stability, so only the 8 sample are used in the implementation.

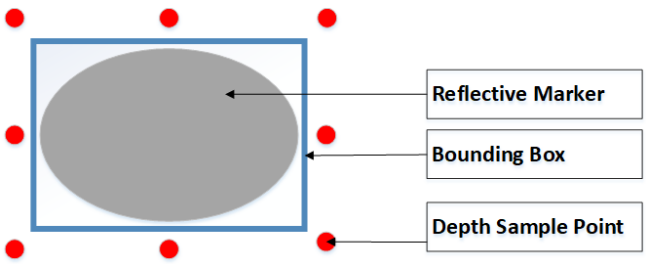


Figure 46: Illustration of how the Z-coordinate of a marker is found using a bounding box and 8 sample points.

As the X- and Y-coordinates are still screen-coordinates, we convert them to the same real-world coordinate system as the Z-coordinates. This conversion is done using the ***ICamera***-interface's method `ScreenToWorldCooordinates()` already covered.

7.1.3.6 Are There Previous Points

At this point the `TrackedData()` is called. In the ***ImageProcessing*** class we maintain two ar-

rays of points(`with type double[][]`), the points from a potential previous frame which we call `prevoisPoints`, and the `newPoints` with the points just found for the current frame.
In the first frame, where we have no `previousPoints`, we look for an image with a number of detected markers equal to the number of markers of the actual deformable screen. The number of rows and columns of markers on the actual screen has to be set in the user settings file(appendix B) in order for the tracking system to work. If the correct number of markers is detected in a frame, we would like to hold the values in a structure that include the markers relative placement to each other. If we look at a screen with a grid of 4x4 markers, we can enumerate them row-wise like seen below:

$$\begin{matrix} 0 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix}$$

We can use the enumerations as indices of the 1-dimensional array `newPoints`, and conform the unsorted array of points to this structure by sorting the detected markers on their X and Y positions. This is done by first sorting the list of points by their Y-coordinate descendingly, and then for each segment of the list with size equal to the number of columns, sort them on the X-coordinate ascendingly.
When the sorting has been performed `previousPoints` are updated to hold `newPoints`, and the pipeline returns to **IRFrameReceived**, and waits for the next frame.

In the later frames, both `previousPoints` and `newPoints` will hold information about the points from the previous and current frame respectively, and the pipeline will continue to **PointsMatching**.

7.1.3.7 PointMatching

The **PointMatching** is performed using by the ***Hungarian*** Class' `Solve()` method. The ***Hungarian***-class is a modified version of Robert A. Pilgrim's implementation [28] of the Hungarian algorithm adapted to our codebase. Furthermore, the original code used new array structures for each run, and as the algorithm use a great portion of memory, this caused very rapid garbage collection. To remedy this, we have converted the code to re-use arrays, without requiring extra work.
The first step in the `solve()` method is to construct a cost-matrix with distances between all combinations of points from the current and previous frame. The cost-matrix has the form seen in table 2 where n_i and p_i denotes the i'th point from the current and previous frame respectively.

	n_0	n_1	n_2	..
p_0	$dist(p_0, n_0)$	$dist(p_0, n_1)$	$dist(p_0, n_2)$	...
p_1	$dist(p_1, n_0)$	$dist(p_1, n_1)$	$dist(p_1, n_2)$	...
p_2	$dist(p_2, n_0)$	$dist(p_2, n_1)$	$dist(p_2, n_2)$	...
...	...	...	...	...

Table 2: Cost Matrix used as input for the Hungarian algorithm.

The cost matrix is generated using the class ***IRUtils***' method called `GetCostMatrixArray`, and then the assignment problem is solved using the ***Hungarian*** class' internal method `RunMunkres()`. Next the `GetMinimizedIndicies()` method is called, which returns an array with a index-mappings from the points from the current frame to the previous frame. Back in the ***ImageProcessing*** class(in `TrackedData()`) the `newPoints` are rearranged according to the index-mappings to match the order from previous points using the ***IRUtils***'s `ReaarrangeArray()` method. If every point from the previous frame is matched to a point in the new frame, we can continue to **Update Previous points**, otherwise, we need to estimate the missing points.

[28] https://github.com/acshi/AccentTutor/blob/master/SpectrumAnalyzer/Munkres.cs

7.1.3.8 Are There Missing Points & Estimate Missing Points

In this step, we estimate the position of potential missing markers (due to occlusion). As mentioned, we have tried out different approaches for estimation, and we want it to be easy to try out other algorithms in the future. Thus, as seen in the class diagram in figure 47, we have made the screen's representation as an interface called "IScreen". The classes implementing the "IScreen" needs to implement the fields:

- `Pointinfo[] pointInfo` which is an array of objects needed for the estimation of missing markers.
- `double[] previousPoints`, it is here the points identified in the previous frame are stored.

Furthermore these methods should be implemented:

- `Initialize()` Used to initialize one of the screen types.
- `UpdateScreen()` Used to update the Screen, here meaning adding estimated positions to occluded markers in the `newPoints` array.
- `InFrame()` A simple boolean check to ensure that we do not go out of bound of the frame.

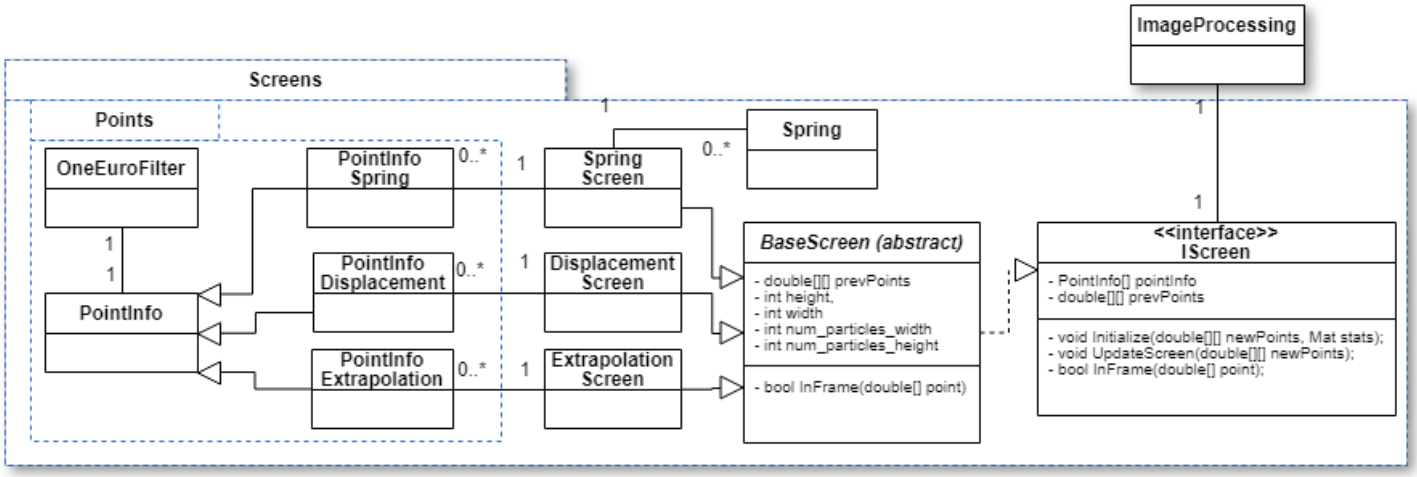


*Figure 47: The **Screens**-package from the class diagram from figure 39 re-arranged for a better overview, and with focus on the **IScreen** interface and the abstract **BaseScreen** Class.*

As seen in the figure, the abstract class ***BaseScreen*** implements the interface partly, holding the fields and methods common for all screen types. ***BaseScreen*** is implemented by the three classes ***SpringScreen***, ***DisplacementScreen*** and ***ExtrapolationsScreen***, respectively implementing the Mass Spring model, the Displacement Vector Model, and the Extrapolation Model covered in the design section.

For simplicity, we will explain the screens bottom-up from the ***PointInfo*** class which holds basic information about a tracked marker, such as if the marker is visible or not, and also the height and width of the detected marker. Furthermore, it contains a range of methods allowing it to find its cardinal-points in the 1D array `previousPoints`. ***PointInfo*** does not hold the positions as these are already present in the `previousPoints` array. Finally, it holds a ***OneEuroFilter***[8], which is a simple algorithm to filter noisy signals for high precision and responsiveness, used to smooth the depth measures very little as they are slightly unstable. The parameters of the filter are found in the user settings file(appendix B), and can be tuned to adjust the filtering. The implementation of the filter is inspired by the code published in connection to the paper[29].
We will now take a closer look at the implementation of each of the estimation methods.

[29] `http://cristal.univ-lille.fr/~casiez/1euro/` Accessed: 07/05-2018

DisplacementScreen
Still starting from the bottom, the ***DisplacementScreen*** contains an array of ***PointInfoDisplacement*** objects. Besides the information from its base class ***PointInfo***, ***PointInfoDisplacement*** holds references to its neighbors(also ***PointInfoDisplacement*** objects) defined in the design section and seen in figure 26 on page 29. A method called `AssignCardinalPoints()` is used to initialize these references to the neighboring points.
The main functionality in the class is found in a method called `EstimatePostitionDisplacement()` which computes a vector describing the displacement from the point's original position, to its current position, for all the neighboring points. It then computes the sum of the vectors, and adds this sum to the position of the occluded point itself.
The ***DisplacementScreen***, holding the array of ***PointInfoDisplacement*** objects, implements the ***IScreen*** interface's methods `Initialize()` and `Update()`. The `Initialize()` method is used to create the array of ***PointInfoDisplacement*** objects using the information from the initial tracked frame of the tracking pipeline. The `Update()` method receives the updated list of points after the **PointMatching** step, and if there are missing any points, here indicated by if there are `null`-values in the array `double[][] newPoints`, their positions are estimated using the ***PointInfoDisplacement*** `EstimatePostition()` method. From now on we will refer to the Vector Displacement model simply as the "Displacement Model".

ExtrapolationScreen
The procedure of the ***ExtrapolationScreen*** is almost identical to the ***DisplacementScreen***, expect that it holds an array of ***PointInfoExtrapolation*** objects instead. ***PointInfoExtrapolation*** are also very similar to ***PointInfoDisplacement*** also having a method called `EstimatePostition()`. However this method uses the Extrapolation approach to produce the estimate based on its neighbors defined in the design section on page 28.

SpringScreen
Our Mass Spring model has been inspired by Jesper Mosegaard's C++ implementation[30] for cloth simulation. Mosegaard's implementation was developed in relation to his Ph.D. thesis [27] where he used a Mass spring model to create a 3D model of a heart for cardiac surgery simulation. We did, however, have to make fundamental changes to large parts of the model and re-write the code for C#. Mosegaard's implementation has been developed for showing a piece of cloth in a 3D scene, where every vertex defining the mesh that makes up the cloth are updated for each frame according to the Mass Spring model.
We will first present how the Mass Spring model is implemented for traditional cloth simulation, and then describe how our implementation deviates. Implementations of the model often consist of a mesh of vertices defining a piece of cloth, where the vertices correspond to the particles of the Mass Spring model. Each vertex in the mesh contains its current and previous position, and additionally, a force variable is used for summing up the forces applied to a vertex. A list of Spring objects, connected to the mesh vertices like defined in figure 28 page 31, each holds references to its two endpoints. The spring class contains a method that calculates the internal forces defined in Eq. 4, seen again below, and adds it to the force variable of the springs endpoints.

$$\mathbf{f}_{int}(P_{i,j}, P_{k,l}) = -\mathbf{k}_{i,j,k,l}(\overrightarrow{P_{i,j}P_{k,l}} - l^0_{i,j,k,l})\frac{\overrightarrow{P_{i,j}P_{k,l}}}{||\overrightarrow{P_{i,j}P_{k,l}}||} \tag{34}$$

The internal force for the entire mesh is thus computed by running though all the springs and calling this method, after with each vertex has the internal force stored in its force variable. In practise, this process is often repeated a number of times for each frame to reduce oscillation between the springs faster, if a more stiff behavior of the cloth is desired. After the internal forces have been added to the vertices of the mesh, the external forces are calculated for each vertex, and added to the force variable as well, so that it holds the sum of all forces for a iteration. Finally the actual position of the vertices are updated according to eq. 17, shown again below:

$$\mathbf{u}(t + \Delta t) = \mathbf{u}(t) + \mathbf{c}_{dis}[\mathbf{u}(t) - \mathbf{u}(t - \Delta t)] + \frac{\mathbf{f}(t)}{\mathbf{m}}\Delta t^2 \tag{35}$$

[30] `https://viscomp.alexandra.dk/?p=147` Accessed 04/05-2018

Where f is the accumulated force, stored in the vertex' force variable.

In our implementation, we define each marker of the deformable screen as particles of the Mass Spring model. The main difference in our implementation is that we know the actual position of most the markers, and only need the model to estimate the position of occluded markers, speeding up the algorithm considerably. Moreover, in traditional cloth simulation, the force produced by every spring is computed and then applied to the all particles without having any reference from the vertices to the spring. We need to be able to find only the springs associated with the occluded points, calculate the force, and then update the position. Thus we keep a dictionary used for obtaining the springs linked to a missing point, using the point's ID. Another difference in our implementation is that f_{gra} and f_{visc} can be omitted, as discussed in the design section. Thus the only forces that need to be computed for each frame are the internal force f_{int} and the damping force f_{damp}.
Figure 48 sums up the estimation process of our Mass spring implementation. Prior to the steps of the pipeline, the model has been initialized by creating a ***SpringScreen***(class from figure 47) object with an array of ***PointInfoSpring*** objects, one for each marker detected in the initial frame when the tracking was started. Additionally the ***SpringScreen*** is initialized with a dictionary containing ***Spring*** objects linking the ***PointInfoSpring*** objects according the spring setting of the Mass Spring model(seen in figure 28 page 31). First, we receive an array of tracked points, in which "null"-values indicates occluded points in our implementation. As we would like visible markers to keep their actual positions, we make them unmovable such that they cannot be moved by a connecting spring. Then we update the ***PointInfoSpring***'s position to the actually detected position and its previous position to its former position. This is important because the estimates rely on the position of the tracked markers, and because we need to maintain these variables if the point is lost in the tracking, and needs to be estimated in later frames.
Next, the internal force for each occluded point is computed using the associated springs found in the dictionary. Finally, the missing points are estimated using Eq. 35 and the force variable contained in the ***PointInfoSpring*** objects. This produces a complete array of all tracked points, containing both visible and estimates. The pipeline is repeated for every frame when using the ***SpringScreen*** for estimation.

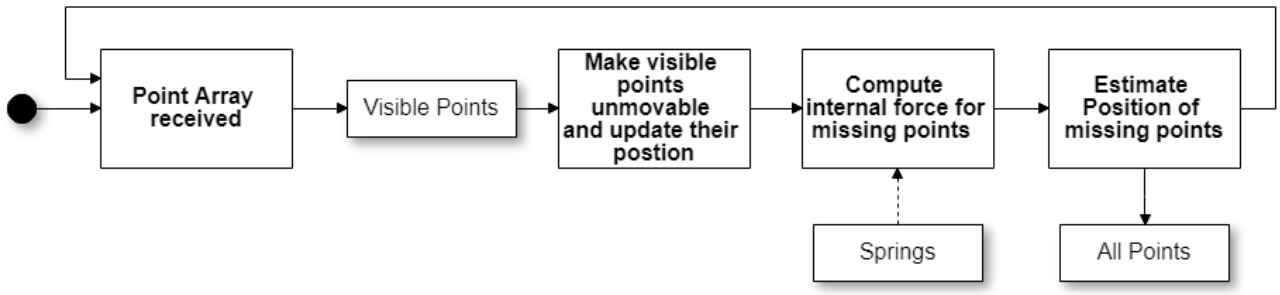


Figure 48: Illustration of the pipeline used for estimating the position of occluded markers using the Mass Spring model.

From now on we will refer to our implementation of the Mass Spring model simply as the "Spring model".

7.1.3.9 Update Previous Points & Post-processing

Ending the **ImageProcessing** pipeline from figure 44, the array `previousPoints` is overwritten with the `newPoints` to update the position of the tracked markers.
Additionally, a copy of the markers positions is converted back to the IR-frame's 2D coordinates for visualization in the **GUI**. The captured frames are overlaid with the IDs and positions of the points, and sent to the "MainWindow" if it has been enabled.
After this, the point's positions are passed on to the Communication package's ***UDPsender*** object.

7.1.4 Communication

The communication package is responsible for communicating with external programs. The ***UDPsender*** converts the tracked data to a flattened array of 32-bit floating points with the structure:

$$[x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, z_2, x_n, y_n, z_n]$$

Where the ID of the markers is given the coordinates indices in the array. The flattened array is sent directly as a byte-array to keep the data as compact as possible.
We have chosen to use UDP communication(port 11000) for the purpose of sending the tracked data as it is sent very rapidly, and if a frame's data is lost it does not matter much. Thus, here we want to focus on speed and not reliability. Also, this keeps a very clean and simple interface for if anyone would like to use the ScreenTracker with other programs in the future. The tracked data can be obtained simply by listening for UDP-data on the correct port and decode the data as a float array.
Vice versa the "TCPserver" works as an interface from other programs to the "ScreenTracker"-program, and handles incoming commands. We use the same port number 11000 for the TCP-connection.
The TCPserver parses and supports the command:

- **resetMesh** - which resets the tracking.

Here we have chosen to communicate using a TCP-connection as the commands are sent rarely, and it is important both that the command is received correctly, but also that the program can send a response that the command has been executed or not.
The TCP-connection and command parsing was added even though we currently only support a single command with the intention that it will be useful if others would like to use the ScreenTracker program in future work.

7.2 Visualization - Unity

We have implemented the visualization of the screen in the game engine Unity[31]. The triangulation defining the screen mesh described in the design section is performed using the Triangle.net[32] library implementing Delaunay triangulation[1].

7.2.1 Working in Unity

As we ourselves had no prior experience in using Unity we will explain the basics very briefly. Figure 49 shows the unity editor with a minimal example project, not related to the deformable screen. Looking at the **Hierarchy** tab we have the current **Scene** "MainScene" we are working on, seen in the Scene tab, containing what Unity calls Game Objects. The Game object is the base class for entities in a unity scene, and do not have much functionality in themselves, but they work as containers for **Components** implementing the real functionality. In the rest of this section, GameObject will be marked with an *italic* font for clarity. The **Inspector** tab shows the components of the game object marked in the scene called *Cube*, with the components: **Transform, Mesh Filter, Box Collider, Mesh Renderer**. The Transform describes the position and orientation of the Game Object in relation to the world coordinate system of the scene. The Mesh Filter takes a mesh from the assets(containing resources for building a scene) and passes it to the Mesh Renderer, rendering the mesh in the screen. The Box Collider is a basic cube-shaped collision primitive for handling collisions with other objects in the scene. Additionally, the game object could have a **Script** component attached, for controlling the behavior of the game object. Unity scripts are special classes that extends the `Monobehavior` class[33] which methods are overwritten as needed. As an example, we could add a script that would move the *Cube*'s Transform-component for each frame by overwriting Monobehavior's `Update` method which is called for each frame automatically, and make the *Cube* e.g. bounce in the scene. This scene also has a *Main Camera* and a *Directional Light* game object

[31] Version 2017.4.1, released: 6 April 2018
[32] `https://archive.codeplex.com/?p=triangle` Accessed: 22/04-2018
[33] `https://docs.unity3d.com/ScriptReference/MonoBehaviour.html` Accessed 07/05-2018

defining the perspective from where the program will be observing the scene, and a light source, respectively. The Game-tab shows the scene from the *Camera*'s perspective, and finally the **Project** tab show the directory of the project.

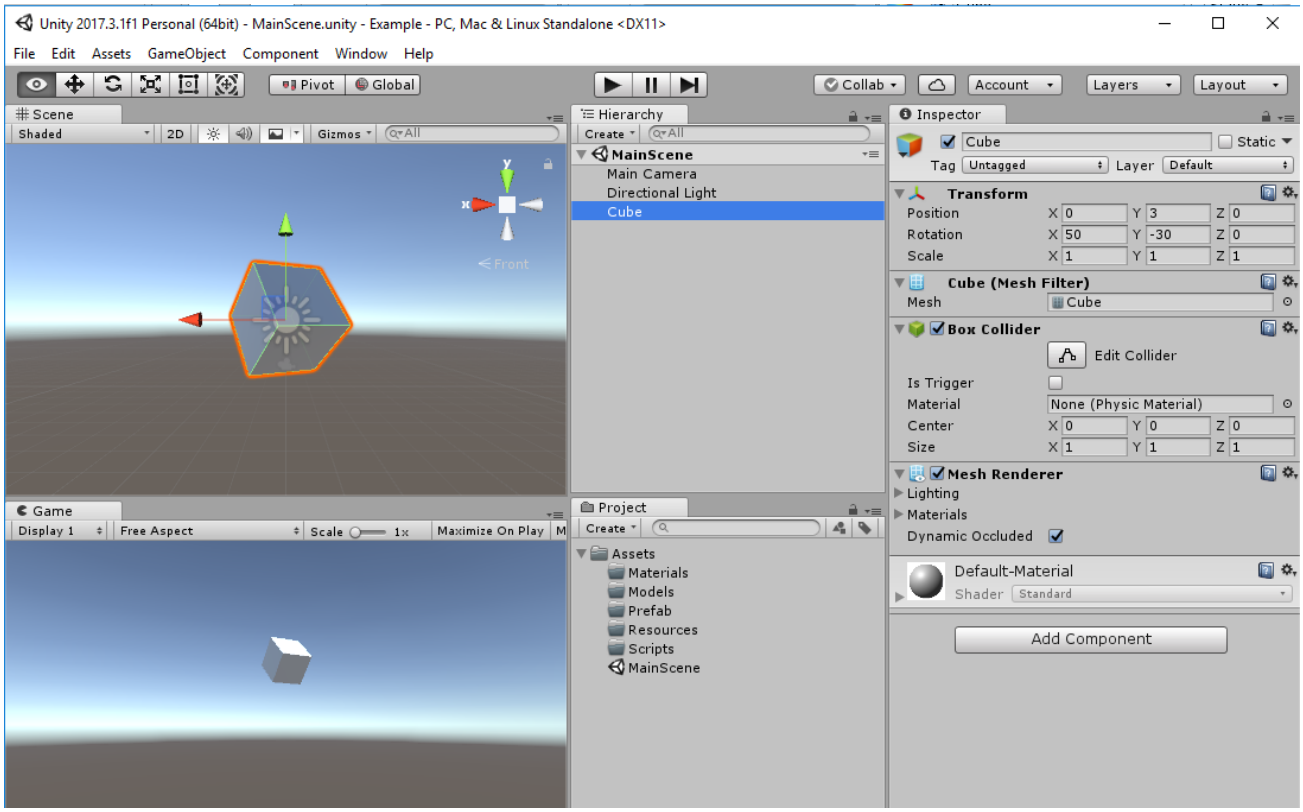


Figure 49: Example of a Scene with a Box Game Object.

7.2.2 Architecture and Implementation

Unity's architecture of game objects and components presents a challenge when trying to describe it using a traditional UML Class Diagram as each game object often has numerous components attached, and because scripts, instead of traditional classes, are used to implement the functionality. We have created a modified class diagram seen in figure 50 where we present game objects as blue boxes and their attached script components as white boxes. Naturally, scripts can have traditional classes associated, which are shown as gray boxes. We will leave out other components e.g Transform, Mesh Collider, Mesh Renderer etc. for simplicity. Also, the game objects' inheritance to the basic game object has been left out to avoid cluttering the diagram. Lastly, fields and methods have been omitted for the diagram of the entire system like in previous sections.

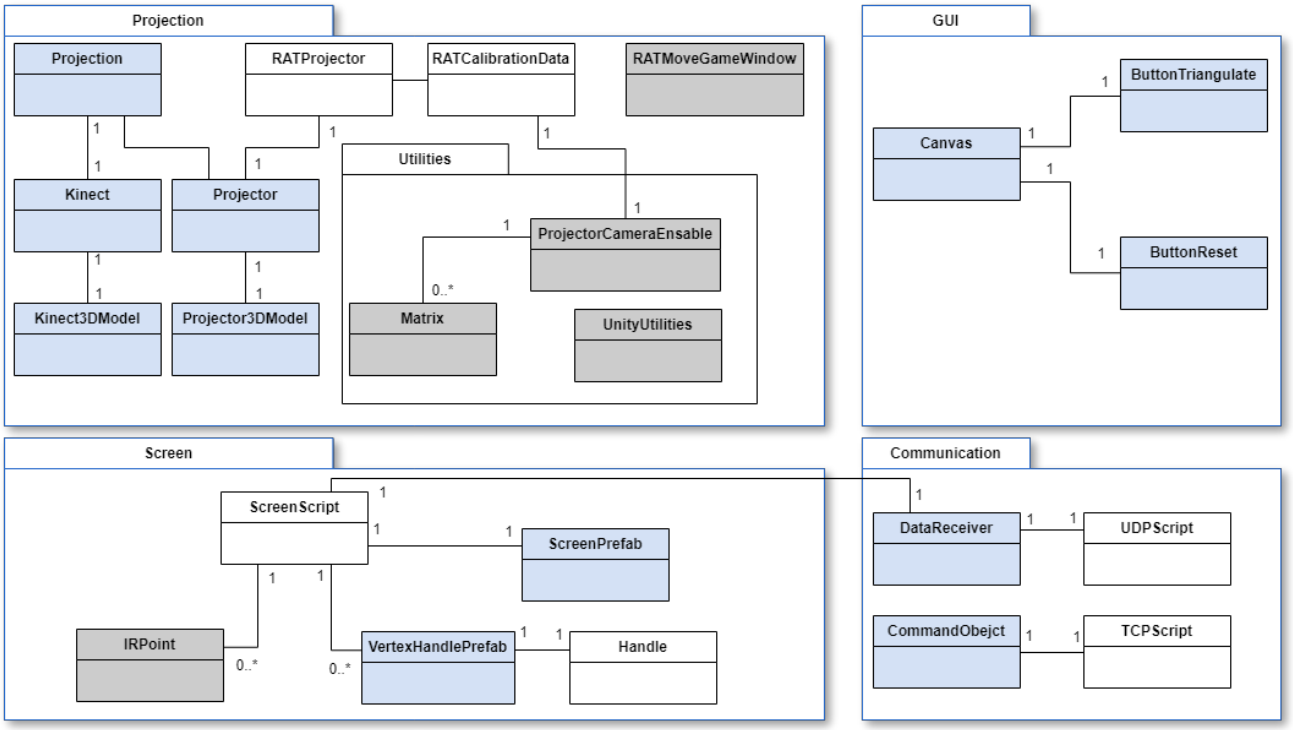


Figure 50: Class diagram for the Unity Visualization program. Game objects are shown as blue boxes, and their attached script components as white boxes.

We can see some of the game objects of the class diagram in figure 51 showing the actual scene. The scene is seen from a sideways perspective to show the game objects. In the small preview on the bottom right side, we see the scene from the perspective of the projector, representing the actual projection.
The *Projection* game object contains the *Kinect* and *Projector*, which again contains their associated 3D models seen to the right in the figure. The multicolored grid to the left in the scene represents the 3D model of the screen, here deformed by a pull gesture. The screen consists of a single *ScreenPrefab* game object with multiple *VertexHandlePrefab*'s representing the vertices of the screen's triangulated mesh. The *VertexHandlePrefab*'s corresponds to the markers of the physical screen, and when they move, the *VertexHandlePrefab*'s moves, deforming the 3D representation of the screen equivalently. The *DataReceiver* and *CommandObject* game objects are merely containers for their scripts and do not have any visual features in the scene. Finally, the *Canvas* with the two buttons are not visible in the scene itself, but will only be shown in the actual projection.

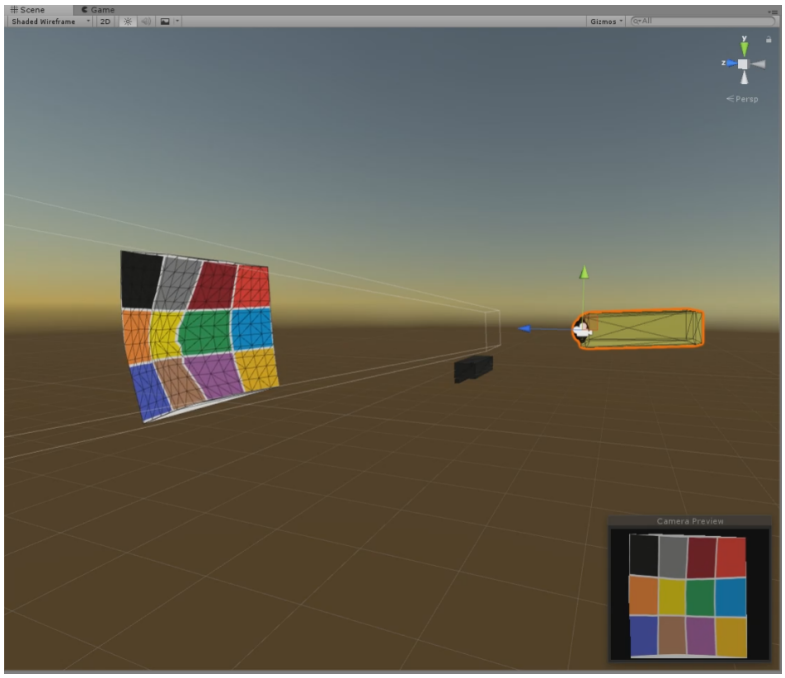


Figure 51: The Unity scene with the 3D model of the deformable screen.

Looking at the class diagram in figure 50 again, we have four main packages:

- **Projection** - Responsible for showing the scene from the projectors view.
- **Screen** - Creating a 3D model of the deformable screen.
- **GUI** - For allowing simple user input/outputs related to the model.
- **Communication** - For communication with the ScreenTracker.

The Projection package will be addressed in a separate section concerning the entire ProCam-calibration implementation.
Figure 52 shows an updated version of the visualization pipeline(from figure 32) specific for the Unity implementation. As seen in the figure, the pipeline has been extended quite a lot, and we have colored the diagram to indicate which packages the processes belongs to.

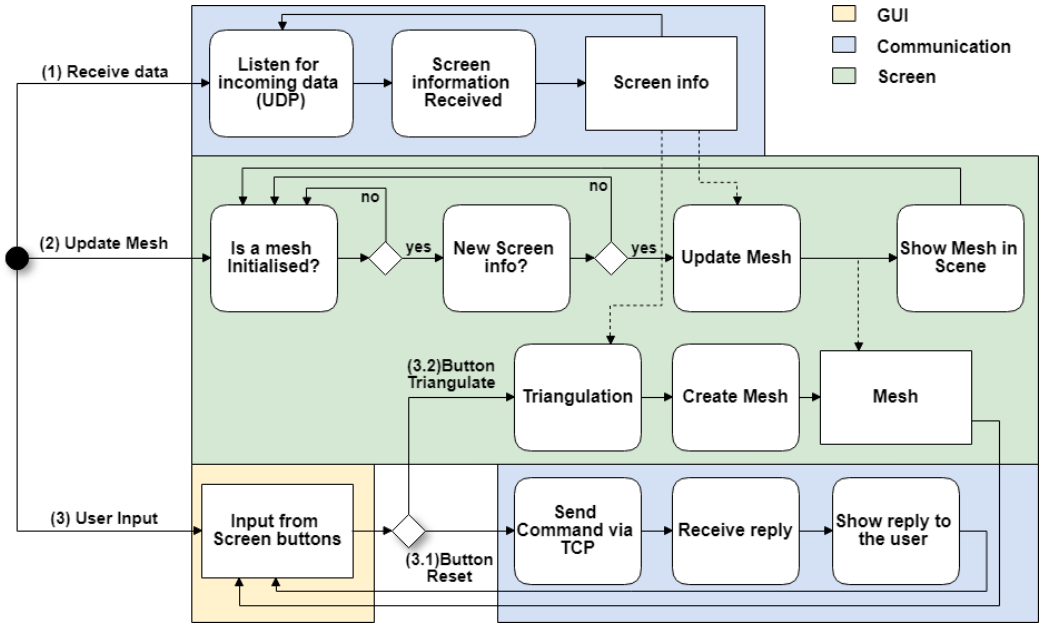


Figure 52: Unity visualization pipeline.

We will start by explaining the pipeline from a high abstraction level, and later dig a little deeper into the script controlling the 3D model of the screen. As seen in the figure, the pipeline is divided up in three lanes: **(1)Receive Data**, **(2)Update Mesh** and **(3)User input** which are running separately and repeatedly while the program is running.
First in lane 1, the program listens for incoming data using a UDP-socket in the ***UDPSript*** of the *DataReceiver* game object. This lane is traversed for every time data is received through the UDP connection to the ScreenTracker program.
Before continuing on to lane 2, we will like to explain lane 3 which accepts user input from the two button game objects; *ButtonTriangulate* and *ButtonReset* splitting the lane once more. The **(3.1)ButtonReset** lane uses the *CommandObject*'s ***TCPscript*** to send a message to the C# ScreenTracker program, which carries out the action by resetting the tracking information, and then reply if the action was successful or not. When the response from the ScreenTracker is received, it is shown on the screen for the user to see.
The **(3.2)ButtonTriangulate** lane, part of the Screen package, performs the triangulation of the **Screen info** from the **(1)Receive data** lane, and then creates a Unity ***Mesh*** from the triangulation before it returns to listen for input again.
The **(2)Update Mesh** is executed for each frame that Unity is able to produce, with the purpose of updating the screen mesh. First, it checks if a **Mesh** has been initialized(in the (3.2)Button Triangulate lane), and if not, returns to check again. If a Mesh has been created previously, it checks if the **(1)Receive data** lane has received new **Screen info** since last time the ***Mesh*** was updated. This is done because Unity's frame-rate is expected to be higher than the ScreenTracker's(of max 30 fps because of the Kinect). If the **Screen info** has not been updated since last time, the pipeline returns to the initial state in the lane. And if the **Screen info** has been updated, the data is used to update the **Mesh** which is then shown in the Unity scene.

The main logic of the 3D screen model is controlled by the ***ScreenScript*** class seen in figure53 which we will cover here in more details. As mentioned, Unity scripts all extend the ***Monobehavior*** class, and in the figure, the overwritten methods are marked with blue. The `Start()` method is called when the associated game object is initialized, and the `Update()` method is called for each frame automatically by Unity. Thus, the `Start()` method is most often used for initializing resources used to control a game object, and `Update()` actually controls the object by updating its variables for each frame.

Figure 53: The ScreenScript class, with methods extending the MonoBehaviour class marked in blue.

The `Start()` method in the ***ScreenScript*** class seen in listing 1 is very short, and does simply obtain a reference to the ***UDPScript*** from the *DataReceiver* game object. The ***Monobehavior*** classes are not passed variables through constructors as traditional, so we first need to find the game object containing the ***UDPScript*** component. This is done in line 4 where the scene is searched for the *DataReceiver* game object, and in line 5, the class variable ***udpScript*** is assigned the `UDPScript` component.

```
1 // Initializes reference to the UDPScript in the DataReceiver GO
2 void Start()
3 {
4    GameObject go = GameObject.Find("DataReceiver");
5    udpScript = (UDPScript)go.GetComponent(typeof(UDPScript));
6 }
```

Listing 1: Code from ScreenScript.cs for the method `Start()`

We will explain the rest of the ***ScreenScript*** class referring to the pipeline from figure 52 for cohesion and clarity. The ***ScreenScript*** implements the entire **Screen** package, meaning that it also covers both the **(2) Update Mesh** and the **(3.2)Button Triangulate** lane.
Starting from the **(3.2)Button Triangulate** lane, the `TaskOnClick()` method is called when the `ButtonTriangulate` game object is clicked. This method first cleans up, and removes objects from potential previous generated meshes. Then the internal method `Triangulate()` is called using the TriangleNet library to convert a list containing the screen markers 2-dimensional coordinates(x,y) into a triangulated `TriangleNet.Mesh` by calling TriangleNet's `Triangulate()`. The returned ***TriangleNet.Mesh*** contains a list of triangles and their vertices forming the screen in 2D.
Then the internal `MakeMesh()` method is called which runs through the triangles of the ***TriangleNet.Mesh*** converting it into a ***Unity Mesh*** requiring an array of the vertices, an array of UV-coordinates, an array of triangles defined by the indices of their vertices, and an array of vertex normals. The vertices and triangles are given from the ***TriangleNet.Mesh***, but the UV coordinate has to be computed. UV-coordinates are used for texture-mapping, meaning mapping a 2D texture to the faces of the triangles in the mesh, and ranges from 0-1 in both x- and -y coordinate. As the coordinates of the screen-markers are given in the metric coordinates of the camera, we need need to normalize the markers' coordinates into the range 0-1 to obtain their UV-coordinates.

This is done for both for the x- and y-coordinates using the formula:

$$x_{norm} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

Where x_{norm} is the normalized value, x_i is the metric value, and x_{min}, x_{max} are the global minimum and maximum coordinate for an axis of the mesh respectively. The triangles' normals are not important here because we let Unity calculate them automatically on the GPU in the `Update()` method later.
Finally, we create a *ScreenPrefab* game object, and assign the constructed Unity mesh to it as a `MeshFilter` component.

The **(2) Update Mesh** lane is carried out by the `Update()` method. `Update()` is responsible for updating the *ScreenPrefab*'s mesh according to the coordinates from the ScreenTracker program for each frame. This is done by checking if the UPDScript has received new data since the last frame, and if it has, then all the vertices of the mesh are updated to their new positions. Lastly, the normals are computed using the Unity Mesh's build-in `RecalculateNormals()` running directly on the GPU.

The Unity program supports using both images and videos as the texture of the deformable screen. The texture can easily be interchanged simply by first placing an image or video in the Assets-folder, and then dragging it onto the *ScreenPrefab*.

This section has covered how the Unity Visualization program receives the tracked data from the Screentracker program, and how this is used to create a real-time responsive 3D model of the actual screen. The next section describes how we calibrate the ProCam-setup in order to correctly project the 3D model back onto the deformable screen.

7.3 Calibration of Projector and the Depth Camera

In connection with the RoomAlive[18] project, where a ProCam setup was used to augment a living-room with projected computer graphics, a toolkit to ease the development of Augmented Reality(AR) applications for the Kinect was published.
The toolkit[34] consist of two main components:

- **ProCamCalibration** A C# project enabling calibration of multiple projectors and Kinect cameras using gray codes and depth images.
- **RoomAlive Toolkit for Unity** A set of Unity scripts and tools that enable immersive, dynamic projection mapping experiences, based on the projection-camera calibration from ProCamCalibration. The project also includes a tool to stream and render Kinect depth data to Unity.

We have chosen to use the ProCamCalibration-project for performing the calibration to obtain the intrinsic- and extrinsic-parameters of the camera and projector. This choice is based on that it uses the structured light approach for calibration so that the calibration is automated, fast, and easy with no need for physical objects as a checkerboard.

7.3.1 Using the RoomAlive Toolkit

To calibrate the projector and Kinect, the ProCamCalibration project needs three different processes running:

- **KinectServer** A process to capture the frames for each connected Kinect camera.
- **ProjectorServer** A process for projecting the gray codes for each connected projector.

[34] `https://github.com/Microsoft/RoomAliveToolkit` Accessed: 19/04-2018

- **CalibrateEnsemple** A GUI letting the user calibrate the system.

We will now go through the process of ProCam calibration using the RoomAliveTookit's ProCamCalibration program. Our setup has only a single projector and camera, thus we only have to run one instance of each of the two servers and the GUI. Prior to calibration, the Kinect should be placed such that it covers the projection.

1. First A new calibration file in XML-format with one projector and one camera is created in the GUI from the menu **File→New**.

2. Then the calibration XML file has to be manually edited so that the tag "displayIndex" has the value of projector's screen index, meaning if the projector is the second screen, it should have index 2. It is also possible to run the projector and camera server on other computers separately, so the "hostNameOrAddress" should have the IP-address of the computer running the server. As we use a single computer in our setup, we can just leave the default value "localhost" in this field.

3. Next **Calibrate→Acquire** is selected to start the projection and capturing of gray codes. The acquire phase takes about 30 seconds for a projector with a resolution of 1920 × 1080 on our setup(described in the Physical Design section). Since each pixel has to be uniquely encoded, the number of gray codes required is dependent on the resolution of the projector.

4. Then **Calibrate→Solve** is selected to start the calibration by computing the camera-, transformation-matrices, and distortion parameters for both the Kinect's sensors(IR and color) and the projector, taking about 20 seconds.

5. Finally **File→Save** is selected to save the calibration XML-file.

After running the acquiring step 3), the GUI looks like seen in figure 54. To the left, we see a part of the Kinect's view, and an output log to the right. Here, the output contains the info that 100 depth images were captured and that the acquire phase is complete. In the view, we see a part of the deformable screen with a projected rectangular frame with a "0" in its center. We also see black spots on top of some of the markers, which is due to the fact that the Kinect masks out areas with IR-reflection outside a certain spectrum, so that only what is considered valid depth-information is used for the calibration later. Since the reflective markers have very high intensities in the IR-image from the Kinect, most of the reflective markers have been masked out.

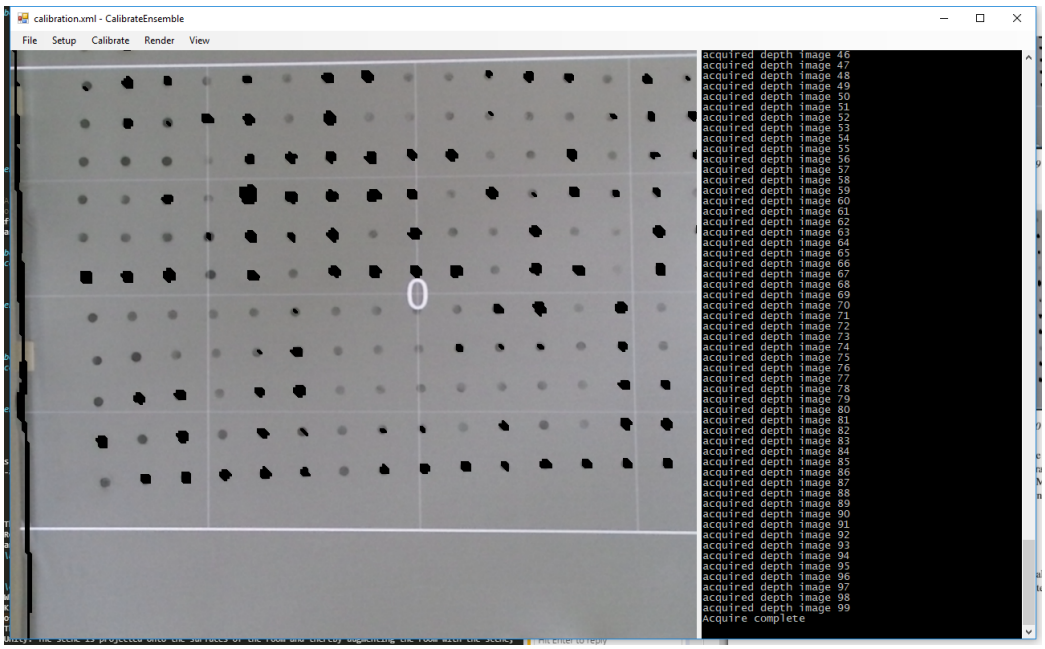


Figure 54: The CalibrateEnsemple GUI after running the acquiring step 3).

Zooming out in the GUI lets us see the frame captured by the Kinect in figure 55, and it is now apparent that the program has matched the Kinect's color frame to the valid depth-information. As we do not see the scene directly from the Kinect's point of view, there are large black gaps where there is a large change in the depth. The calibration has been performed in front of a doorway, causing a large black area above the deformable screen as this area is further away in the image. Like seen before, we observe that the image is curved between its outer corners, implying that we should expect that the lens of the depth camera is distorted around the edge, which is revealed in the calibration results later.

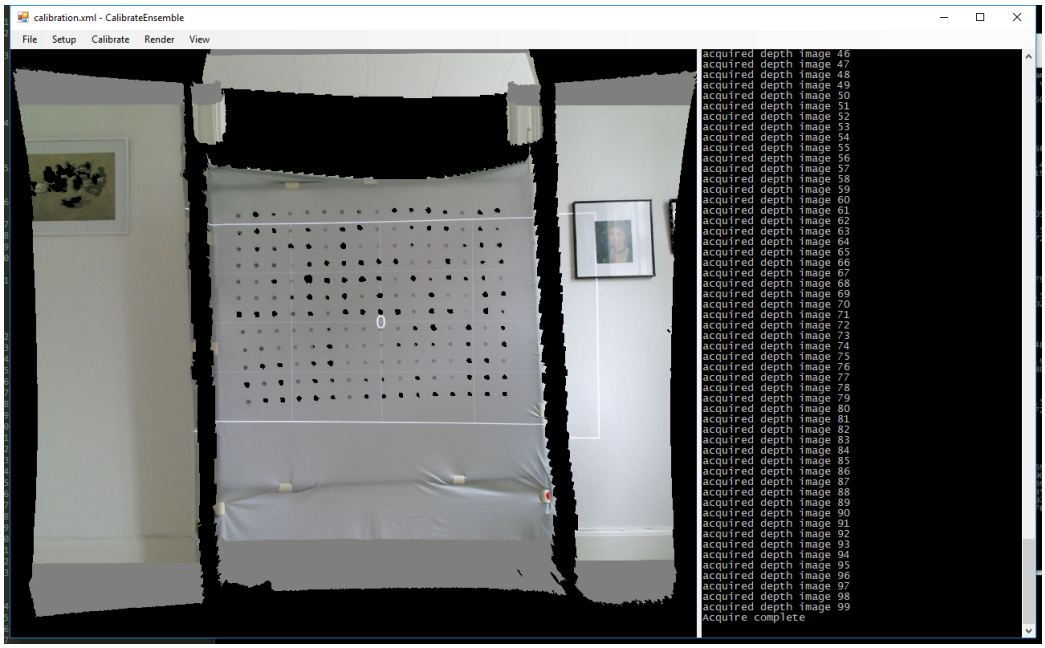


Figure 55: The CalibrateEnsemple GUI after running the acquiring step 3) and zooming out.

After the calibration has been performed in step 4), we can select **Render→Camera 0** to see the scene from the projectors perspective, seen in figure 56. What should be noticed here is that the view now only covers the projected frame seen in the previous figures 54 and 55, which corresponds to the view of the projector. Looking at the log, it now contains information about the calibration and the obtained precision. If the solve has been successful(like here), the last line of the log says "Solve complete".

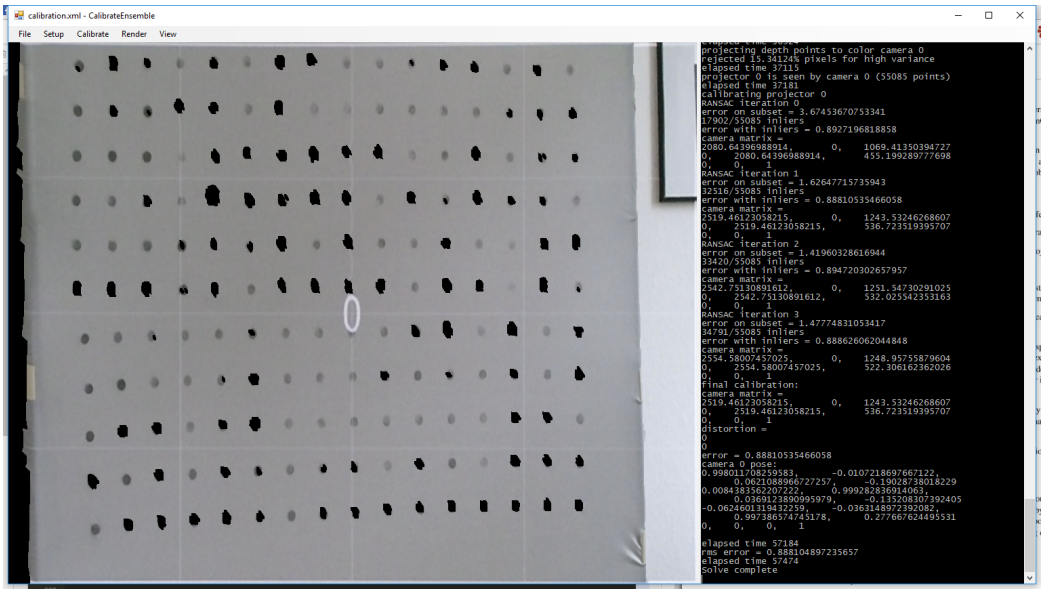


Figure 56: The CalibrateEnsemple GUI after running the solve step 4).

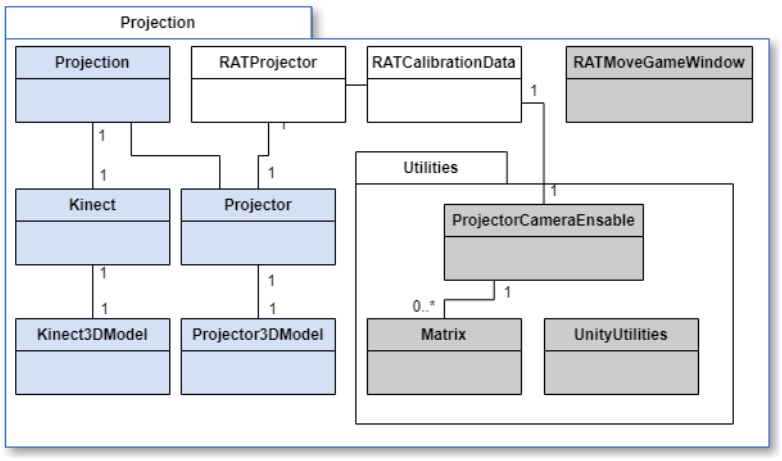
After the solve have been completed, the calibration info is saved in step 5), writing the calibration info to the XML file which contains the info:

- **Kinect**
  - **colorCameraMatrix** - The 3x3 color camera's calibration matrix **K** (eq. 33).
  - **colorLensDistortion** - A 1x2 vector with radial distortion parameters (eq. 29) for the color camera.
  - **depthCameraMatrix** - The 3x3 depth/IR camera's calibration matrix **K** (eq. 33).
  - **depthLensDistortion** - A 1x2 vector with radial distortion parameters (eq. 29) for the depth/IR-camera.
  - **depthToColorTransform** - A 4x4 transformation matrix from the depth Camera's Space to the color camera's coordinate space (eq. 30).
  - **pose** - A 4x4 transformation matrix (eq. 30) from the Kinect's color camera's coordinate space to World Space. When only one Kinect is present in the setup, the World Space = Camera Space meaning this matrix is the identity matrix I_{4x4} and that the Camera's coordinate space is located in (0,0,0) of the World Space.
- **Projector**
  - **cameraMatrix** - The 3x3 projectors camera's calibration matrix **K** (eq. 33).
  - **height** The height of the projection in pixels.
  - **lensDistortion** A 1x2 vector with radial distortion parameters (eq. 29) for the projectors lens.
  - **pose** - A 4x4 transformation matrix (eq. 30) from the projectors "camera space" to the World Space.
  - **width** - the width of the projection in pixels.

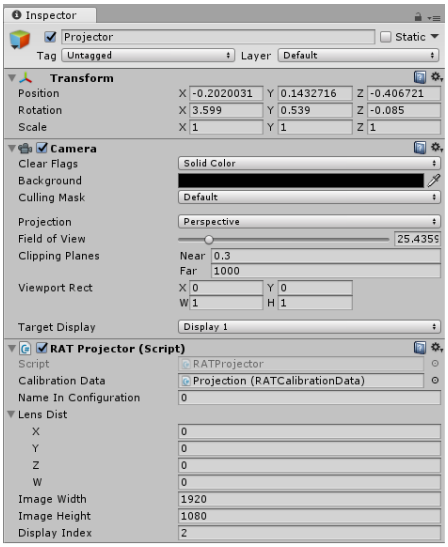
Besides that the **pose** matrix of the camera is fixed to I_{4x4}, the width and height are fixed to the resolution of the projector. Not surprisingly we have observed that the lens distortion of the Kinect's camera's sensors is fixed, and we see that they are both distorted as expected. Another observation we have made is that the lens distortion parameters of the projector are consistently $k_1 = 0, k_2 = 0$. This is also not surprising since the projector has a high projection ratio(described in section 5.3) indicating that the lens does not curve significantly. The values of the camera- and transformation-matrices changes expectedly when the Kinect or projector is moved in relation to each other. An example of a calibration file can be found in appendix C.

7.3.2 Calibration and Unity

Returning to the Unity visualization implementation, receiving coordinates in the Kinect color camera's 3D coordinate space, which has the same origin as the World Space. What we want to do is to transform the 3D model of the deformable screen, from the Kinect's camera space to projector's camera space.
Looking at the class diagram for the **Projection package**, seen again in figure 57a, the *Projector* game object defines where we want to view the scene from.



*(a) Class diagram for the **Projection package** cut out from class diagram in figure 50 covering the entire system.*



*(b) The Unity inspection panel showing the **Projector** Game Object.*

*Figure 57: The **Projection** package(left), and Projector Game Object(right).*

Figure 57b shows the Unity inspector view for the *Projector* game object with the three components: `Transform`, `Camera` and `RATProjector`(Script). The Transform component holds the Projectors position and rotation in the world space, and the Camera component holds the intrinsic values for the projection. The script RATProjector.cs is responsible for setting the values of the Camera and Transform.
When the Unity program is started, the RATProjector.cs script first loads the calibration data, using the RATCalibrationData.cs class. The script then sets the Transform's position and rotation using the [R—t] matrices from the calibration. This is done like seen in the code in listing 2 but due to differences in the format of the calibration data, and what Unity's transforms needs, we have to make a conversion:

```
1 Matrix4x4 worldToLocal = RAT2Unity.Convert(projConfig.pose);
2 worldToLocal = UnityUtilities.ConvertRHtoLH(worldToLocal);
3 this.transform.localPosition = worldToLocal.ExtractTranslation();
4 this.transform.localRotation = worldToLocal.ExtractRotation();
```

Listing 2: Code from RATProjector.cs setting values of the transform

First in line 1 the transformation matrix [R—t] `projConfig.pose` is converted from the calibration file's row-major matrix to column major using the utility functions from ***UnityUtilities.cs***. In line 2 the Matrix is converted from defining the projection in a right-handed coordinate system to a left-handed. Then in line 3 the transform's coordinates are set to the translation vector t using `ExtractTranslation()`. Finally, in line 4, `ExtractRotation()` converts the rotation matrix R to a Unity ***Quaternion*** describing the rotation, and sets the transform's rotation field to this value. Now objects in the scene will be transformed from world space, which is the same as the Kinect's camera space, to the projectors camera space when running the program.
Next, we have to set the `Projector Camera`'s perspective to the view of the physical projector using the camera calibration matrix K, so that we transform the 3D points down in the 2D screen space, as discussed

in section 6.3.2. However, among other programs, Unity uses a normalized OpenGL perspective projection viewpoint representation of the screen ranging from [-1:1] as seen in figure 58

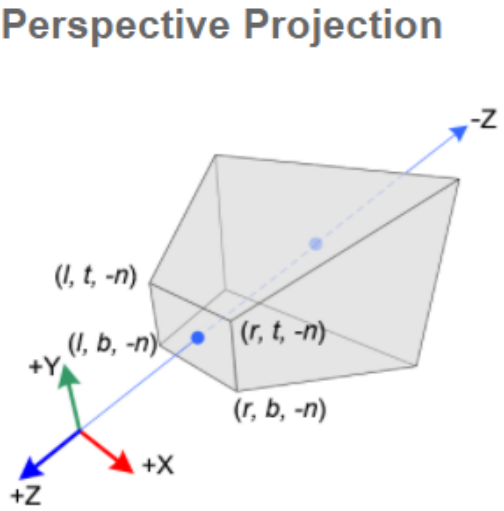


Figure 58: OpenGL Perspective Projection model (From Song [36]).

Where the range of x-coordinate in figure 58 goes from [l, r] to [-1, 1], the y-coordinate from [b, t] to [-1, 1] and the z-coordinate from [n, f] to [-1, 1].
We will not describe in details how the camera calibration matrix is converted to use this normalized perspective representation, but below the OpenGL perspective projection maxtrix(eq. 36) used in RATProjector.cs is seen:

$$\begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} = \underbrace{\begin{bmatrix} 2\frac{a_x f}{w_s} & 0 & -1+2\frac{(w_s-p_x)}{w_s} & 0 \\ 0 & 2\frac{a_y f}{h_s} & -1+2\frac{h_s-p_y}{h_s} & 0 \\ 0 & 0 & \frac{-(z_{far}+z_{near})}{(z_{far}-z_{near})} & -2\cdot z_{near}\frac{z_{far}}{(z_{far}-z_{near})} \\ 0 & 0 & -1 & 0 \end{bmatrix}}_{\textbf{K converted to OpenGL perspective projection matrix}} [R_{3x3}|t_{1x3}] \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \quad (36)$$

The above perspective projection matrix is based on the two online articles concerning this conversion[35] [36]. Where w_s, h_s is the screen width and height, and z_{far}, z_{near} is the far and near clipping plane defining how much of the scene depth should be covered in the projection. Looking at figure 58 again, this means where the viewing frustum starts and ends. The matrix is converted in the `GetProjectionMatrix()` method and assigned to the Unity-Camera's `projectionMatrix` attribute.
Now objects seen by the Kinect are transformed into the view of the projector, or if we look at it another way, now the projector "sees" the world as the camera does. The class remaining not explained yet from figure 57a is the RATMoveGameWindow.cs which add the utility tab to the Unity editor seen in figure 59. This utility enables the user to move and re-size the Unity **Game View**, that in our case shows the projection of the 3D screen model, to a specific location in a screen setup. This comes in handy when developing in Unity as we can set the **Game View** to cover the entire projection, without having to build the project for release every time a change is made to the code. In the example in the figure, the tool is set to move the **Game View** 1920 pixels to the right(to a secondary screen if the main screen is 1920 pixels wide), and set the size to the resolution of the projector to 1920x1080.
As the tool has eased the development of our project, we imagine that it can help others in future development as well, thus we leave it as a part of the project.

[35] `http://spottrlabs.blogspot.dk/2012/07/opencv-and-opengl-not-always-friends.html` Accessed: 19/04-2018
[36] `http://www.songho.ca/opengl/gl_projectionmatrix.html` Accessed: 19/04-2018

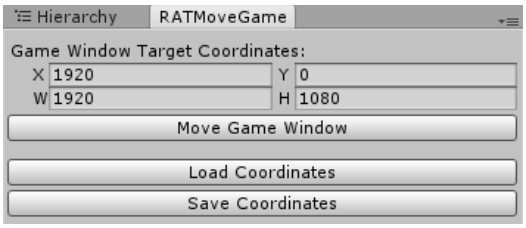


Figure 59: The scene moving tool from RATMoveGameWindow.cs.

The classes in the Projection package has in large extent been inspired by the RoomAlive Toolkit for Unity. Since the toolkit was developed for another purpose, it was much more complex than we needed. We only want to use the part of the project making it possible to import the calibration file and use its content to calibrate the ProCam setup. We removed large parts of the project and rewrote relevant classes to fit our needs. Thus, all the classes in the Projection package have been taken from the Toolkit and adapted to our screen visualization project.

8 The Deformable Screen

This section will show our final prototype of the Deformable Screen consisting of the FlexIO framework and the physical elements put together. The section will mainly show photographs of the setup to provide a better understanding of how the final setups look, and functions. Instructions on how to install and use the framework can be found in appendix A. Figure 60 shows the entire system with a computer, projector, Kinect camera, and the screen. On the laptop, we can faintly see the 3D model of the screen in the Unity program. To the left we see the actual screen being deformed with a push gesture. In this case, the texture consists of a video sequence displaying ”an explosion of colors”.

Figure 60: The Deformable Screen, here showing a video while a push gesture is applied to the screen.

It is hard to capture the deformation in figure 60 due to the viewangle. Figure 61 shows the same deformation and texture but seen from the side.

Figure 61: The same gesture and texture as in figure 60, here seen from a sideways perspective.

The two photos show what the screen looks like, but it is hard to see the deformations in the 3D model. To illustrate the deformations better, we can change the texture to a colored grid seen in figure 62:

Figure 62: Colored grid used to illustrate the deformations of the 3D model more clearly.

Figure 63 shows the grid being exposed to a sideways push deformation. It is now more clear that the 3D model is actually deformed according to the deformation of the screen.

Figure 63: The deformable screen with the grid from figure 62 as texture. The screen is here exposed to a sideways push deformation.

The screen can also handle that a user grabs the canvas in the hands. In figure 64 we see the screen from a sideways perspective, while the user has grabbed the screen with both hands, and is performing both a pull and push gesture.

Figure 64: The deformable screen with the grid from figure 62 as texture. The screen is here exposed to both a pull and a push gesture.

The photos were taken in full daylight for us to be able to see the setup, so the projected image is a little faint in the colors compared to when the screen is used in a darkened room.
More photos of the Deformable Screen can be found in appendix E.

9 Breaking Point Analysis

To be able to evaluate the estimations models in our prototype, we needed to perform boundary testing to find out how much we can deform the screen before it breaks. Breaking the screen in this context means that the point matching fails to identify the correct markers between frames. As this leads to all estimations becoming useless in the surrounding area, and thus would prevent evaluation of the estimation models.
We started our breaking point analysis using our more dense screen, described in the Physical Design section. We remind that this screen has the dimensions 120cm X 85cm with a 17x13 grid of reflective markers(n=192) with ø 2 cm, and 5 cm spacing between the markers.

Spring	X	Y	Z+	Z-	XY
Light	1	1	0	4	0
Medium	6	2	2	7	2
Strong	8	8	10	10	10
Extrapolation					
Light	0	0	0	2	1
Medium	1	1	5	10	1
Strong	10	10	10	10	9
Displacement					
Light	0	0	1	0	0
Medium	0	1	3	10	1
Strong	7	6	10	10	3

Table 3: Number of breaks per ten deformations.

We performed a small evaluation of the screen by performing basic deformations given by stretching the fabric on the X (horizontal) and Y (vertical) axes, and push, pull on the Z axis("Z -" reducing the distance to the Kinect, "Z +" increasing the distance to the Kinect). In addition, we performed a stretch on the combined X, Y axes, giving us a total of five deformations to be performed at different intensities. As the fabric can be stretched approximately 24cm in any given direction, we performed light deformations with a stretch of 8cm, medium at 16cm, and finally strong deformations at 24 cm.
The deformations where all performed ten times in the center of the screen, and once a deformation was completed we recorded whether or not the markers still had the correct ID assigned.
Table 3, shows the number of times each deformation broke the screen for these ten attempts, and with only some of the light, and a few of the medium deformations not resulting in general breaks, it was obvious that an evaluation of the estimation models could not be completed with this canvas. Thus we had to change to our other canvas with fewer markers and more spacing between them. We remind that this canvas likewise has the dimensions 120cm X 85cm, but with a 9x7 grid of reflective markers(n=63) with ø 2 cm, and 12 cm spacing between the markers.
We performed the test again with the less dense canvas which resulted in very few to no breaks as each marker had more room to move, before overlaying another marker. This allows us to continue on to evaluating our estimation models.

10 Estimation Evaluation

With our three different estimation models in place, we needed to figure out which one was better, or if it were situational. We would like to be able to compare positions computed by our estimation models to the actual positions of the markers which we do not have. We observed different ways of evaluating models for the estimation of non-visible objects for tracking purpose in the related works, though most articles simply skipped any formal evaluation, and settled for telling the reader how they felt it performed. Actual evaluations, however, range from the visual evaluation[13] where we only see whether or not it works, and what works better, to ProCam setups[12] that tries to see the markers that are otherwise occluded, and get the ground truth for evaluation through there. As we allow for deformations that, no matter how many cameras we use, will occlude some of the markers, we had to find another solution. Inspired by the work of Lin et al.[21], where they use manual labeling as ground truth to evaluate a quite different system, we likewise decided on basing our evaluation on manual labeling of the occluded markers positions. With a ground truth, we are able to evaluate the models both individually and against each other.

10.1 Obtaining Ground Truth

We have developed a setup for getting ground truth data by simple manual labeling. As we cannot repeat the exact same gesture to the deformable screen multiple times, we record the tracked data, the three different estimation techniques' data, and the infrared- and color-image they were detected on.
This data is recorded in the ScreenTracker program and saved to disk as seven files

1. A file containing only the visible points in IR Screen Coordinates
2. A file containing only the visible points in Color Screen Coordinates.
3. Containing all points in metric coordinates, the positions of occluded markers are estimated by the Spring model.
4. Containing all points in metric coordinates, the positions of occluded markers are estimated by the Displacement model.
5. Containing all points in metric coordinates, the positions of occluded markers are estimated by the Extrapolation model.
6. The IR-frame.
7. The color-frame.

We have developed a Matlab script Labelling.m[37] where we load file 1 containing only the visible markers positions, and the associated IR-frame. If there are missing markers in a frame, a GUI shows the IR-frame overlaid with the tracked markers shown as enumerated dots. This is seen in figure 65, where marker no. 87 was not visible for the IR-camera.

[37] Available at `https://github.com/MultiPeden/ScreenTrackerEvaluation/tree/master/Matlab` Accessed: 19/04-2018

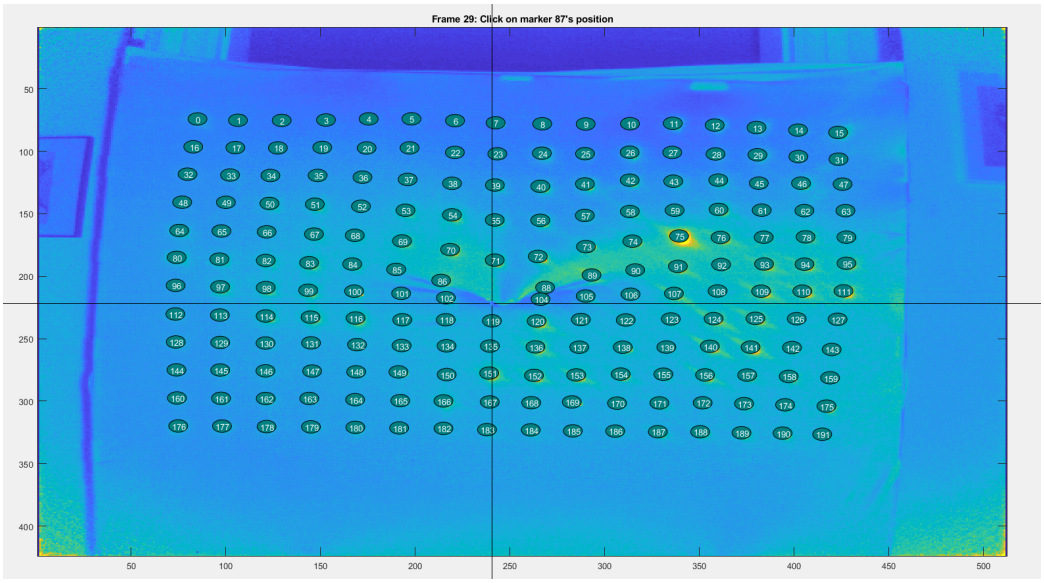


Figure 65: The GUI used for manual labeling showing the IR-frame, and overlaid with enumerated circles for each of the visible markers of that frame.

We would then like to determine the true position of missing points, done by clicking the mouse, where we think the marker actually should be in the image. Since it can be hard to determine where the marker is from the IR-frame alone, the color frame with markers seen in figure 66 is also shown.

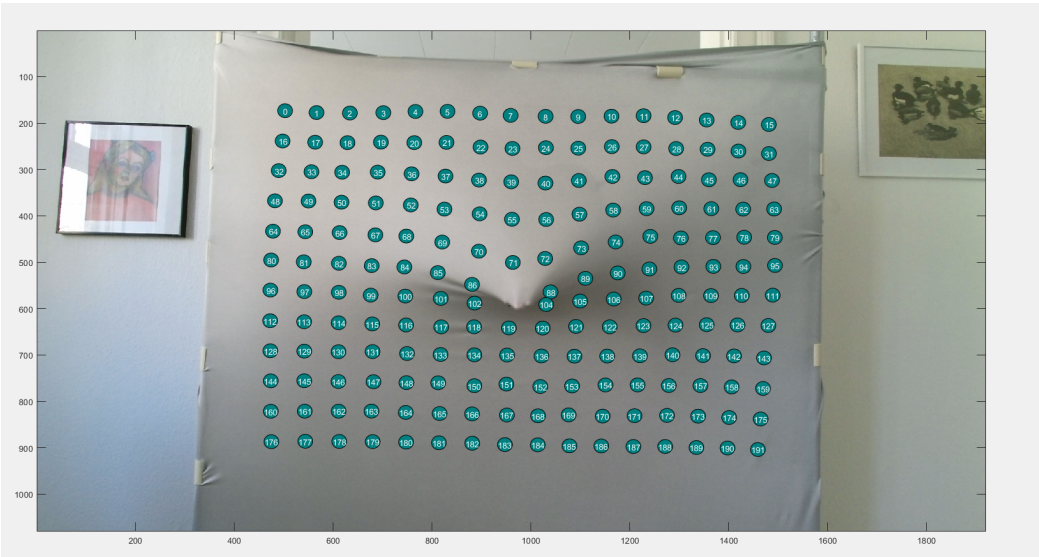
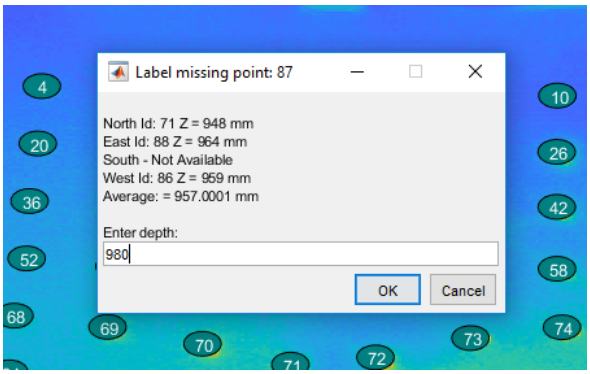
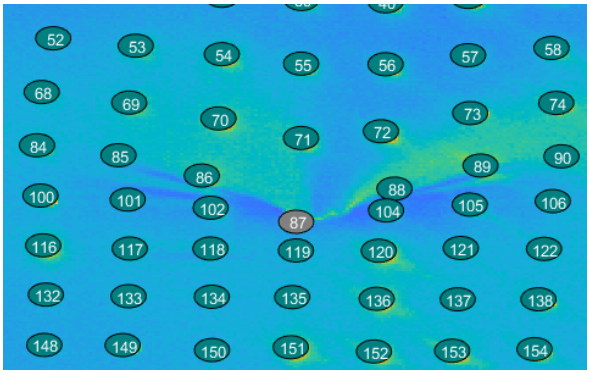


Figure 66: A color version of figure 65 shown to ease the manual labelling.

Clicking the mouse on the marker gives us the x- and y-position, but we also need the depth information given by the z-coordinate. Thus, after obtaining the 2D position, the GUI shows a dialog as seen in figure 67a, letting us see the z-coordinates of the cardinal points if available(and average), and takes as input what we think the actual z-coordinate is. When the z-coordinate has been entered, the manually labeled marker is shown as a gray circle as seen in figure 67b.



(a) Cutout of the same frame as in figure 65, here with an input dialog box prompting for the z-coordinate. This dialog is shown after the 2D(x and y) position has been determined by clicking the mouse on the image. The z-coordinates of the visible cardinal markers, and the average are shown to ease the labelling.



(b) Zoomed in cutout from the frame from figure 67a, but now with the manually labelled marker is shown as an enumerated gray circle. The GUI now prompts for clicking the next missing marker.

Figure 67: Manually labelling, obtaining the z-coordinate.

The process is continued until all missing points in the sequence has been labeled after which the labeled data is saved to disk. Even when having both the IR- and color-frame, it is rather challenging to label the data, not having the particular deformation fresh in memory. Thus the labeling process was performed immediately subsequent to a deformation was recorded.

10.1.1 Baseline

In addition to the three estimations models presented in the report, we added a baseline model for later comparison. The Baseline-model simply lets the estimation be the last known actual position of a marker. The model marks the minimum of which an estimation model must outperform, in terms of offset error, in order to be useful for tracking the screen.

10.1.2 Comparing Estimates to the Labeling

Like described, we compare the manually labeled data to those of the three estimation models. But after the labeling process, the data is defined in the IR-camera screen coordinates, while the estimations live in the 3D metric camera coordinates. Another small program called CoordinateConverter [38] has been developed in C# to map the labeled data from the file on disk, into metric camera coordinates, and output the mapped coordinates to a file again.
Once the labeled data has been converted into metric camera coordinates, we can compare it to the estimates produced by our three estimation models. A final Matlab script ComputeError.m [39] calculates the offset in terms of the Euclidean distance between an estimate and the actual position of a marker obtained through the manual labeling process.

[38] Available at `https://github.com/MultiPeden/ScreenTrackerEvaluation/tree/master/CoordinateConverter` Accessed: 19/04-2018

[39] Available at `https://github.com/MultiPeden/ScreenTrackerEvaluation/tree/master/Matlab` Accessed: 19/04-2018

10.2 Deformations

With the test setup ready to record data, we settled on a specific set of deformations to record, label, and analyze.

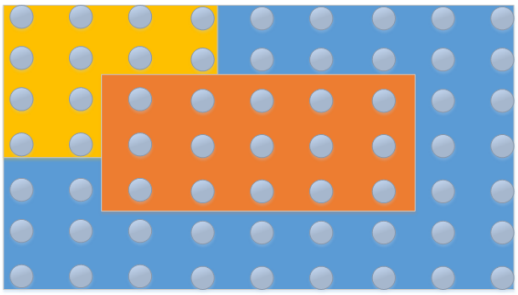


Figure 68: Chosen regions for deformations.

We work with deformations in two different screen regions as seen in figure 68. The yellow regions are chosen as it is close to the edge of the screen, the fabric here is tenser and harder to stretch, while the fabric in the orange region is relaxed. We expected the two regions two deform differently due to the difference in tension in the center opposed to the edge.

In each of these regions, we performed number of different deformations to the Screen. In addition to the deformations described in the breaking point analysis in section 9, we have expanded on the array of deformations used, combining the four single direction deformations, to get eleven different deformations to evaluate, as seen in table 4.

X	Y	Z-	Z+
X Z -	X Z +	Y Z -	Y Z +
X Y	X Y Z -	X Y Z +	

Table 4: All of the deformations performed.

We distinguish between Z in backwards and forward directions as the two deformations, gives us very different occlusion. When pulling the fabric backwards, points simply disappear from view in the area we occlude, while pushing the fabric in, can occlude points in areas which are not within the actual deformation.
As the final variable, we perform each deformation with different intensities. We define three levels of intensities as Strong, Medium and Light. This gives us $2 \times 3 \times 11$ (Location $\times$ Direction $\times$ Intensity) 66 different recordings, for which we as described, will manually label occluded markers for each frame where there are any missing.

10.3 Procedure

To determine the exact nature of the deformation and get the recordings uniform, we measured how much far we could deform the fabric for any of our deformations as seen on figure 69, and dividing that distance into the three intensities as found in table 5.

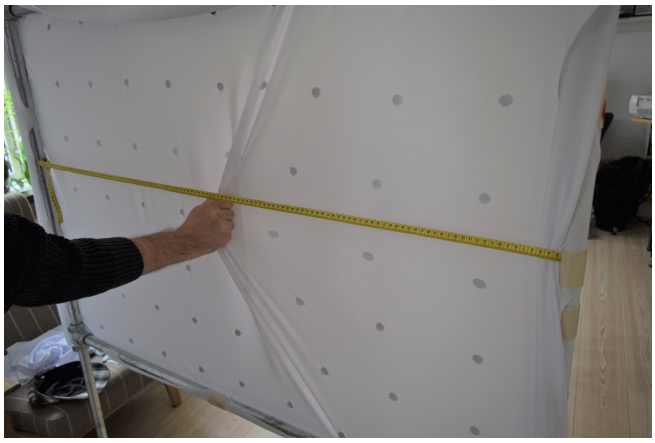


Figure 69: Measuring the distance possible for the center X axis deformation.

Center	**Light**	**Medium**	**Strong**
X	7	14	21
Y	7	14	21
Z-	11	22	33
Z+	11	22	33
XY	4	8	14
XZ-	8	16	24
XZ+	8	16	24
YZ-	8	16	24
YZ+	8	16	24
XYZ-	7	14	21
XYZ+	7	14	21

Edge	**Light**	**Medium**	**Strong**
X	5	10	15
Y	5	10	15
Z-	8	16	24
Z+	8	16	24
XY	4	8	14
XZ-	7	14	21
XZ+	7	14	21
YZ-	7	14	21
YZ+	7	14	21
XYZ-	7	14	21
XYZ+	7	14	21

Table 5: Measurements for each deformations intensities maximum distance in centimeters.

In our experiments, we stored 30 incoming frames spaced over 6 seconds, while performing the deformation as a single smooth action, leaving 10 frames for each intensity. As the Spring model uses temporal information and is influenced if we reduce the frame rate, we had to run the program with the maximal 30 FPS, and obtained the 30 frames over the 6 seconds by only recording the data for every 6^{th} frame.
To get more data, the deformation was applied at two different spots of the screen in the same Location region, and on the same time for each recording.
Once the recordings were completed and manually labelled, we could run the ComputeError.m script to obtain the offset error between the estimates and the ground truth for all the recordings. Unfortunately, due to an issue in the code where the first measure was skipped when recording the data, we only have 9 measurements for all light intensities. However, this means that we still have $4 \times 2 \times 11 \times 29$ (Model $\times$ Location $\times$ Direction $\times$ measures across the Intensity) 2552 measurements.

10.4 Hypotheses

Naturally, we expected the estimation models to outperform the baseline so we hypothesized that (**H1**) Our estimation models all outperform the Baseline in terms of offset. Further, based on our understanding of how the models handle deformations with movement in the Z-axis: (**H2**) For deformations including movement in the Z-axis, the Extrapolation model outperforms the other models in terms of lowest offset error.

10.5 Analysis of Results

In this section, we will report and analyze the results of the estimation model experiment. In the text we will mark variable names with **Bold** and values with *italic*(except numeric values) for clarity. As the experiment is an initial exploration of the performance of the estimations models, we only have two hypotheses to test. However, in addition, we will look for other factors influencing the performance of the estimation models.
The **Baseline** is included in the analysis since only comparing the three estimation models, would mostly reveal their performance relative to each other. Comparing with the **Baseline** however, will reveal if our estimate models are better than keeping the marker stationary and not estimate its position at all.
We performed a $4 \times 2 \times 3 \times 11$ (**Model** $\times$ **Location** $\times$ **Intensity** $\times$ **Direction**) repeated measures analysis of variance (ANOVA), and found significant main effects and interactions for all independent variables (all $p < 0.003$), details is seen table 14 found in appendix D. The dependent variable of the test is the **Error** offset in terms of the Euclidean distance between an estimate and the ground truth in meters. As the assumption of sphericity was violated, we corrected the degrees of freedom using Greenhouse-Geisser. Also for later pair-wise post hoc tests, we have used Bonferroni-corrected confidence intervals to compare against $\alpha = 0.05$.
The significant main effect and interaction mean that all the variables, and combinations, seem to influence the **Error**, and thus need to be considered when working on the models.

10.5.1 Comparing the Models

Looking at the **Model**, we see that it has a significant influence on the **Error**, and further pairwise comparisons shows significant difference between the **Models** ($p < 0.001$) individually, where the **Models** each has the mean **Error** seen in table 6. As we have a significant difference in effect on the **Error** for each **Model**, we can rank the **Models** by the quality of the estimates also seen in the table.

Rank	**Model**	Mean **Error**
1	*Extrapolation*	0.049
2	*Displacement*	0.065
3	*Spring*	0.091
4	*Baseline*	0.105

*Table 6: Ranking the estimation **Models** by performance, lower error is better. Where the **Error** is the offset in terms of euclidean distance between an estimate and the ground truth in meters.*

We observe that the ranking in table 6 supports H1.

10.5.1.1 Intensity

The Anova likewise showed that the **Intensity** has a significant influence on the **Error**. We know that the *Baseline* is dependent on the **Intensity**, because its **Error** is simply proportionally to the **Intensity**, so it is not interesting to consider significance of the **Intensity** with the baseline included. Thus we performed these tests, leaving out the Baseline model and likewise found a significant main effect of ($F_{1.186,9.488} = 2143.772$ $p < 0.001$) for the **Intensity**.
However in pairwise comparisons it is interesting to see if our **Models** performs significantly different than the *Baseline*, so we now look at the Anova including the *Baseline* for pairwise comparisons. The pairwise comparisons also show significant differences ($p < 0.001$) in **Error** between the **Intensities**.
To investigate if the individual **Intensities** and **Models** has an effect on the **Error**, we ran a post hoc test $4 \times 2 \times 11$ (**Model** $\times$ **Location** $\times$ **Direction**) for each **Intensity** for comparison (with the *Baseline* included). Figure 70 shows a plot of the mean **Error** by **Intensity** by **Models**, and the Mean **Errors** are seen in table 7

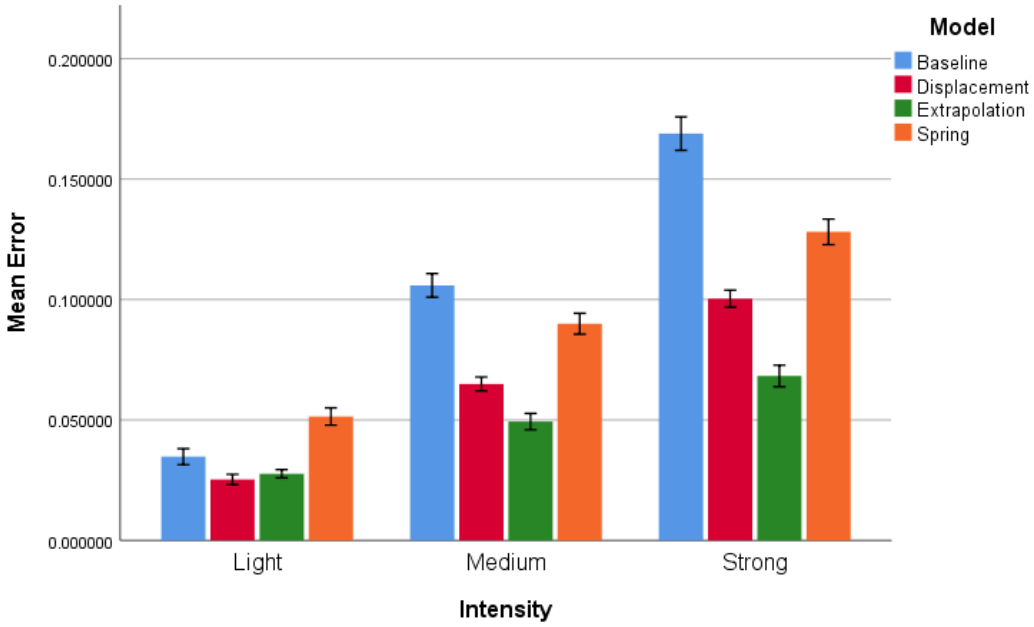


*Figure 70: Clustered Bar Mean of **Error** by **Intensity** by **Model**, with error bars of 95% Confidence Interval(CI), Here the **Error** is the offset in terms of euclidean distance between an estimate and the ground truth in meters, the **Model** is the method used for the estimation, and the **Intensity** is the degree of deformation(Light,Medium, Strong).*

Model\Intensity	*Light*	*Medium*	*Strong*
Baseline	.035	.109	.172
Displacement	.025	.067	.102
Extrapolation	.028	.050	.069
Spring	.051	.092	.130

*Table 7: The mean error for each **Model** by **Intensity**.*

Looking at only the *Light* **Intensity**, we see that the **Model** has a significant influence on the **Error** ($F_{1.027,8.215} = 11.348$ $p < 0.01$). Pairwise comparisons between the **Models** individually shows that there is no significant difference between the performance of the *Baseline-*, *Displacement-* and *Extrapolation-models*. However, we see a significant difference between the performance of the *Spring model* and the other models ($p < 0.023$). Thus we cannot statistically say that our *Extrapolation* and *Displacement-models* are better than the *Baseline*, but only that the *Spring* model performs worse than all the others for the *Light* deformation **Intensity**.

Moving on to the *Medium* **Intensity** we still see that the **Model** has a significant ($F_{1.003,8.025} = 213.248$ $p < 0.001$) effect on the **Error**. But this time, also the pairwise comparisons between the **Models** for the *Medium* **Intensity** all have significants different ($p < 0.001$) effect on the **Error**.

This tendency continues on in the *Strong* **Intensity** where we see the **Model** also has a significant ($F_{1.018,8.144} = 1007.929$ $p < 0.001$) effect on the **Error**. Again here, pairwise comparisons show that the **Models** has a significant ($p < 0.001$) different effect on the **Error** individually.
Summarizing for the **Intensities**, there is significant difference between the performance of the **Models** for the *Medium* and *Strong* **Intensities** with the ranking according to table 6 supporting H1. Another trend we can see from the figure 70, is that even though the error is increased significantly for every step up in the intensities, we do not see the error for estimation models increase proportionally to the baseline. If we for example look a the

baseline in the step from the *Medium*(0.109m) to the *Strong*(0.172m) **Intensity**, we see that the error is increased by 58 %, while the same step in the Extrapolation model only increases the error from 0.05m to 0.069m by 38 %.

10.5.1.2 Model and Direction

Figure 71 shows the **Error** by **Direction** by **Model**(values are seen in table 15 in appendix D) and gives us a better insight of how the **Models** performs for the different types of deformation.

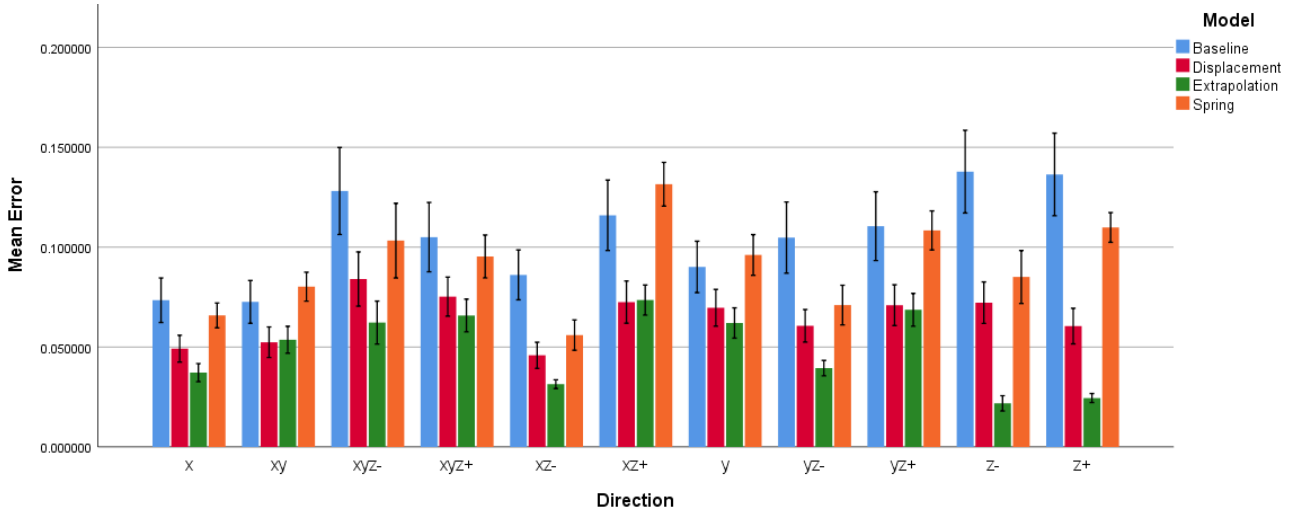


Figure 71: Clustered Bar Mean of Error by Direction by Model, with error bars of 95% Confidence Interval(CI).

As mentioned, the two Anova's showed a significant difference in effect on the **Error** for **Model** and **Direction**. Pairwise comparisons shows that not all directions has a significantly different effect on the **Error** individually, however there are not apparent trends in the correlations, for details see table 16 in appendix D.
If we instead look at the plot in figure 71 again, we generally again see the tendency that the estimates of the Extrapolation model is a little better than the Displacement model, which is at great deal better than the Spring model, which again is a little better than the Baseline. The Baseline however outperforms the spring model for the *XY*,*ZX* and *Y* **Direction**.

If we look at the **Directions** on the z-axis exclusively(*Z-*,*Z+*) we see that the error is considerably lower for the Extrapolation model than the other models. We performed a $4 \times 2 \times 3$ (**Model** $\times$ **Location** $\times$ **Intensity**) post hoc test for each of the two **Directions** *Z-*,*Z+* individually, revealing that the **Models** has a significant effect ($F_{1.058,8.464} = 233.209\ p < 0.001$) for *Z-* and ($F_{1.011,8.089} = 216.832\ p < 0.001$) for *Z+*. Furthermore pairwise comparison showed significant different effects ($p < 0.003$) between the **Models** individually for both **Directions**. This, held together with table 15 (appendix D) supports H2, since we now can say that the Extrapolation model performs significantly better than the other **Models** for **Directions** in the z-axis exclusively.

Even though we have seen that the Extraplation genrally performs significant better than the other models, especially in directions with movement in the z-axis exclusively, we would like to investigate if it also is better for all **Directions** just including movement in the Z-axis to further support H2. Thus we replaced the **Direction** variable with a binary variable **ZMovement**(0,1), where $0 = \{X, Y, XY\}$ and $1 = \{Z-, Z+, XZ-, XZ+, YZ-, YZ+, XYZ-, XYZ+\}$, in our data to indicate if the direction included movement in the Z-axis or not. We aggregated the **Error** for the grouped directions. Table 8 shows the **Error** for the **Models** by the **ZMovement** independent variable, which is also plotted in figure 72.

Zmovement\Model	*Baseline*	*Displacement*	*Extrapolation*	*Spring*
0	0.078	0.057	0.051	0.080
1	0.115	0.068	0.048	0.095

*Table 8: The **Error** for the **Models** by the **ZMovement** independent variable.*

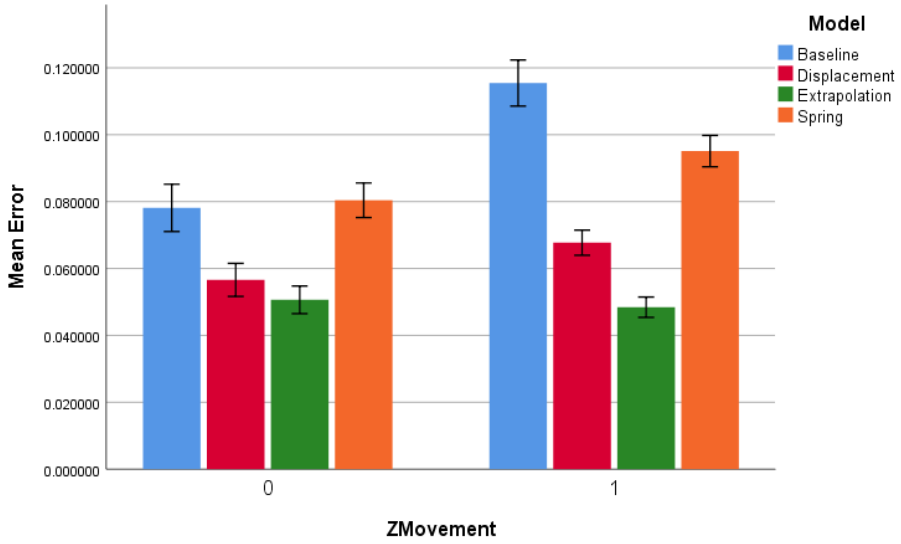


Figure 72: Clustered Bar Mean of Error by Movement in z-direction by Model, with error bars of 95% Confidence Interval(CI) 1 indicates movement in the z-direction and 0 indicates deformations in only the x and y axes exclusively.

We ran another reapaeted measures Anova $4 \times 2 \times 3 \times 2$ (**Model** $\times$ **Location** $\times$ **Intensity** $\times$ **ZMovement** revealing that also **ZMovement** has significant ($F_{1,8} = 269.248$ $p < 0.001$) effect on the **Error**. We performed a post hoc test for each of the two **ZMovement** groups and found that, for **Directions** without movement in the Z-axis(**ZMovement** = 0) the **Models** has a significant effect ($F_{1.016,8.132} = 200.514$ $p < 0.001$), and pairwise comparison likewise showed a significant difference $p < 0.001$ between the **Models** individually. More importantly, supporting H2 further, for **Directions** including movement in the Z-axis(**ZMovement** = 1) the **Models** also has a significant effect ($F_{1.005,8.038} = 205.838$ $p < 0.002$), and pairwise comparison likewise showed a significant difference $p < 0.002$ between the **Models** individually.
This, held together with the **Error** values from table 8, means that the Extrapolation model performs significantly best for both the deformations including movement in the z-axis(supporting H2), and deformations with movement in the X- and Y-axes exclusively.

Also worth noting, the *Spring model* actually perform significantly($p < 0.002$) worse than the *Baseline* for deformation not including movement on the z-axis. Another observation we made when looking at figure 72 is that we see that the **Error** for the *Baseline* for deformations with **ZMovement** = 1 is higher than for **ZMovement** = 0, which is a consequence of that the screen allows larger deformations in directions including movement in the z-axis.

10.5.2 Comparing the Screen Location

Figure 73 shows a plot of the mean **Error** by **Model** by **Location**, and table 9 shows the numerical values.

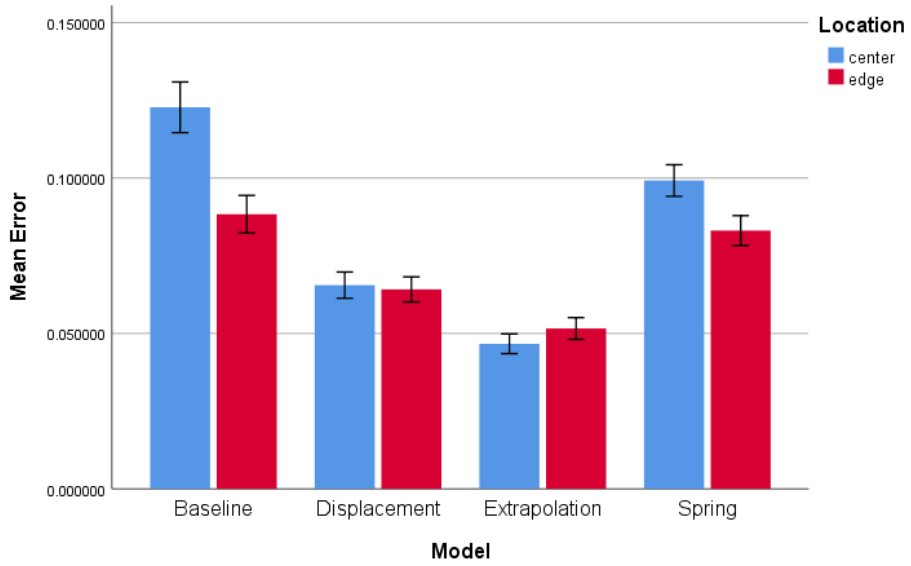


*Figure 73: Clustered Bar Mean of **Error** by **Model** by **Location**, with error bars of 95% Confidence Interval(CI).*

Model Location	*Center*	*Edge*
Baseline	.122	.088
Displacement	.065	.064
Extrapolation	.047	.051
Spring	0.99	0.83

*Table 9: Mean **Error** for **Models** by **Locations***

We know from the initial Anova tests that **Locations** has a significant effect on the **Error**. Pairwise comparisons between the two **Locations** likewise reveals a significant difference($p < 0.001$).
We performed Post hoc tests on each of the **Models** $2 \times 3 \times 11$ (**Location** $\times$ **Intensity** $\times$ **Direction**) for further analysis of the screen **Locations** effect for of each **Models**.

For all the model we see that the two **Locations** has a significant effect:

- *Baseline* ($F_{1,8} = 373.323$ $p < 0.001$).
- *Displacement* ($F_{1,8} = 17.229$ $p < 0.003$).
- *Extrapolation* ($F_{1,8} = 50.998$ $p < 0.003$).
- *Spring* ($F_{1,8} = 4862.331$ $p < 0.001$).

This means that the for each of the **Models**, the **Location** has a significant effect on the **Error**, and since the **Error** is higher for the *Center location* for all the **Models**, we can say that the **Error** for each **Model** is significant higher for deformations in the *Center location* than in the *Edge*. It is not surprising that the *Baseline* has a larger **Error** in the *Center* as the screen allows larger deformations here, compared to the *Edge location*. Moreover, we see that even though the **Location** has a significant effect on the *Extrapolation model*, the mean **Error** is still a lot closer between the two **Locations** than for the *Extrapolation-* and *Spring-model*.

11 Efficiency Evaluation

For the screen to be usable, it is important that it is able to run in real-time. This means that the actual tracking, estimation, and 3D rendering should at least be able to run at the same rate as the camera does.
With the chosen Kinect camera, which at best provides us 30 frames per second, that leaves us 0.0334 seconds to perform all needed actions, to achieve real-time performance.
We will not in this evaluation look into input and output delay provided by the hardware itself, but rather look into the running time of each step, and the scalability of the prototype which will be affected by the amount of markers to track, as well as the estimations that needs to be handled in a frame.
For the evaluation, we are looking at the screen with a 7X9 grid of 63 trackable markers. The timings have been recorded during a full day of testing, leaving us with a decent amount of data, 117.094 data points for the tracking and estimation, and 1.048.575 for the unity visualization program.

	Tracking	Point Matching	Cost Matrix	Displacement	Mass Spring	Extrapolation
Average	2.0706	0.7219	0.5110	0.0027	0.0467	0.0014
Median	1.9550	0.6359	0.4963	0.0030	0.0465	0.0012
High 1%	3.5421	0.9799	0.7336	0.0307	0.7372	0.0170
High 5%	3.1810	0.9433	0.7027	0.0131	0.1479	0.0038
Low 1%	1.8419	0.4689	0.3183	0.0002	0.0046	0.0002
Low 5%	1.8608	0.5769	0.4326	0.0002	0.0046	0.0002

Table 10: Timings for each part of the ScreenTracker C# program, colored according to the steps in the ImageProcessing pipeline seen in figure 74.

11.1 Tracking Speed

The tracking timings from table 10 are given from the time we receive a frame from the camera to the point where our list of visible points have been made, corresponding the orange colored steps in the ImageProcessing Pipeline in figure 74:

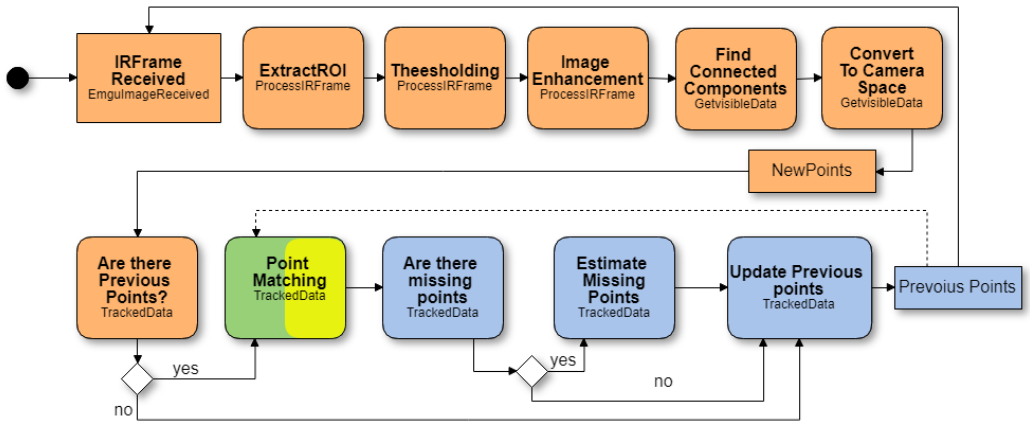


Figure 74: The ImageProcessing Pipeline from the implementation section's figure 44, here colored according to match the timings.

As seen from table 10 this initial part of the tracking is the most costly of the code. This is presumably because it is here we perform image analysis, which has to go through the individual pixels. For our screen this roughly takes 2ms. It does, however, hit higher values of around 3.5ms, but with our low values being very much closer to the median, it is outliers that occur, more likely being due to factors in the computer, out of our control.
Looking at figure 75 we see the tracking for X missing points has a small but defined trend, of running faster with fewer points visible, which is to be expected, as it gives us fewer markers to process.

Figure 75: Shows time cost dependency between Point Tracking, Point Matching and missing markers.

11.2 Point Matching Speed

When looking at the step of Point Matching, colored green and yellow in the pipeline from figure74, we have as seen in table 10 measured the time used for running the entire Hungarian algorithm, but also the time used on constructing the cost matrix used in the Hungarian algorithm. This is done to show that the actual cost of the Hungarian algorithm is dominated by the construction of the cost matrix as it makes up about 80% of the point matching time also seen in figure 75. For point matching, we have a wider span of run times, and it fluctuates between 0.32 and 0.73ms for the 1% high median and 1% low median. This is because of the way it calculates the best matches, where scenarios with multiple points being about equally as close to a point in the old frame, means it will have to sort out these conflicts, requiring a larger number of iterations. The Hungarian algorithm is, as the initial tracking, only run on the visible points of the screen, this on figure 75 shows the same downward trend in time, as the initial tracking and with roughly the same scale.

11.3 Estimation Speed

After the initial tracking steps(orange) and point matching(green/yellow), we have only left in the C# program to look at our estimation models corresponding the blue steps of figure 75 and table 10. Looking at figure 76 it is clear that one of the estimation models is more costly than the others. The Mass Spring model has quite a few more steps involved when it has to compute the internal forces, and even in the event that there is nothing to estimate, it still has to do some variable updates, as it needs current and previous positions to be up to date for when estimation is necessary. While the Mass Spring model is clearly the most costly, we can see the same cost trend for it as for the Displacement- and Extrapolation-model(See figure 77 for a closer look at the last two). All three have a linear looking tendency of cost depending on the number of points to be estimated which should be expected.
Looking just at the time cost for Extrapolation- and Displacement-model, we see one performing slightly better, although they work much the same way. This, however, has a simple explanation as the displacement method looks at the position of more neighbors than the extrapolation does, giving it more lookups and arguments to take into the calculation.

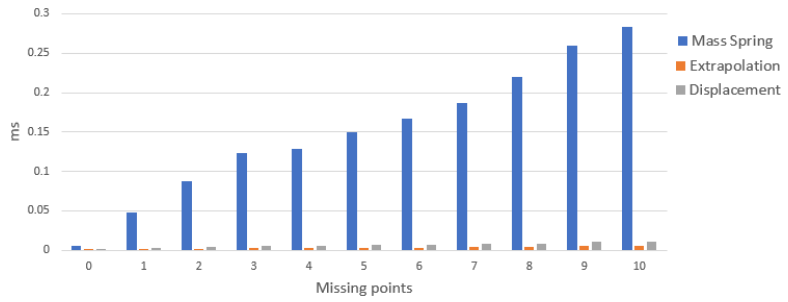


Figure 76: The time cost for each estimation model per missing marker for the three splits of the pipeline.

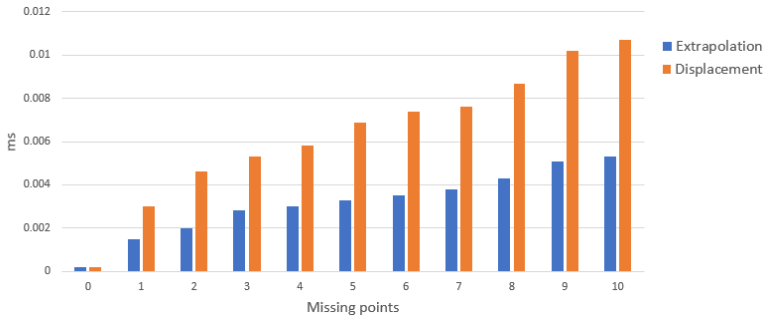


Figure 77: Shows the time cost for Displacement and Extrapolation estimation models per missing marker.

11.4 Visualization Speed

The final part of the project, the Unity visualization, was timed as well from the moment it receives the UDP input, and until each point in the 3D model has been updated. We find really no surprises here, as it is a continuous loop, that updates variables for the unity model and not much else.
As seen on table 11, the program takes about 0.12 ms per iteration, and while it has some dips into the high end of 0.25, they are outliers and can be caused by outside uncontrollable variables.

	Unity time cost
Average	0.1374
Mean	0.1185
High 1%	0.2504
Low 1%	0.1037

Table 11: Time cost of Unity visualization program.

12 Discussion

In this section, we would like to more closely discuss the findings we have made, and the considerations that we have needed to take.
From as early as when we looked into related works, we have had to make some hard choices. There was no chance that we could test and evaluate every method proposed for ourselves. Each choice led down a path that provided even more options. With the goal in mind, to create a deformable display that with consumer hardware can handle both real-time performance and deformations on all axes, we had to cut otherwise promising approaches, such as multi-cam setups, high-speed tracking due to hardware restrictions, and unique markers and patterns, as they would not hold up well under stretch.
In the end, we had to go for an affordable approach, with all the restrictions the cheaper hardware comes with, the results of which we will go through categorically here.

12.1 Physical Design

In section 5 we described how the ProCam-setup and Deformable Screen are physically implemented. The entire setup is rather cumbersome and needs calibration when the projector or camera is moved. As described in the introduction, we do not envision our prototype of the deformable screen as a consumer product, but rather as a tool for further research in deformable screens. Hopefully, the future will bring truly deformable displays, allowing a more compact form factor. Our system does at this stage not recognizes gestures, but is build to easy future projects defining and investigating the interaction space of a deformable screen. Calibration of the ProCam in our setup is automated, which greatly reduces the task of setting up the system.
The frame of the screen consists of steel pipes providing the rigidity needed for keeping the canvas stretched, and the height of the screen is even adjustable which is an important factor in future user testing.
The canvas itself consists of a stretchable and retractable membrane, in this project realized by a piece of cloth. In the process of evaluating our system, we have realized that the cloth in our prototype is slightly hard to grab in the hands, because of a smooth and silky surface. We think that it is essential for the screen to be appealing to users, that the surface should be easy to grab on to, otherwise using it for some tasks can be a frustrating experience. Thus we explored the materials used for other projects and found a study[50] recommending a specific type of cloth based on user input. There was a little confusion about the material actually used in that study, but we assume that has the same stretchable and retractable properties as our cloth, and in addition have a more rubber-feel texture which is easier to grab onto. When evaluating the system we tested the maximal deformation that the screen allows, and we saw that this sometimes left semi-permanent deformations in the cloth as seen in figure 78. Here semi-permanent deformations mean that the cloth does not retract fully without help, and needs to be stretched in other locations and directions to re-obtain a uniform flat canvas.

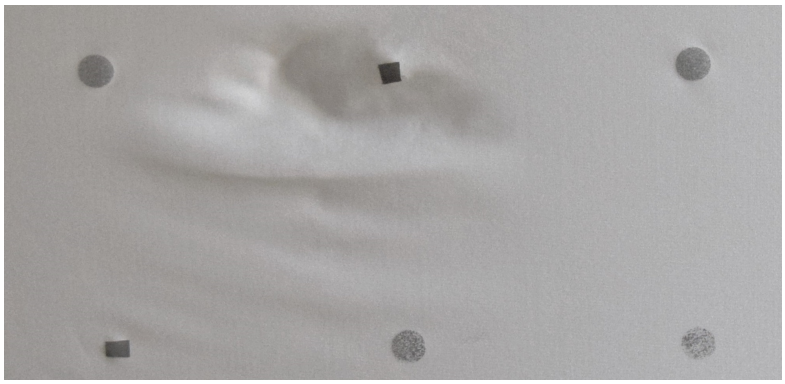


Figure 78: Photo of semi-permanent deformation of the cloth canvas of the deformable screen.

As explained we have not experimented with different materials ourselves, but we suspect that adding too much rubbery material in the cloth compound might compromise the retractability of the screen even more. This

indicates that further research is needed to find a material having all the properties of being stretchable, fully retractable and easy to grab onto.

Figure 78 also shows another issue in the physical design in form of the reflective markers. As explained, we had envisioned that we could realize tracking with fully invisible markers, but we needed to settle for the gray reflective paint. These gray dots will inevitably influence the user-experience as they are very visible in the projection on the screen as seen in figure 79 where the colored grid are projected onto the screen.

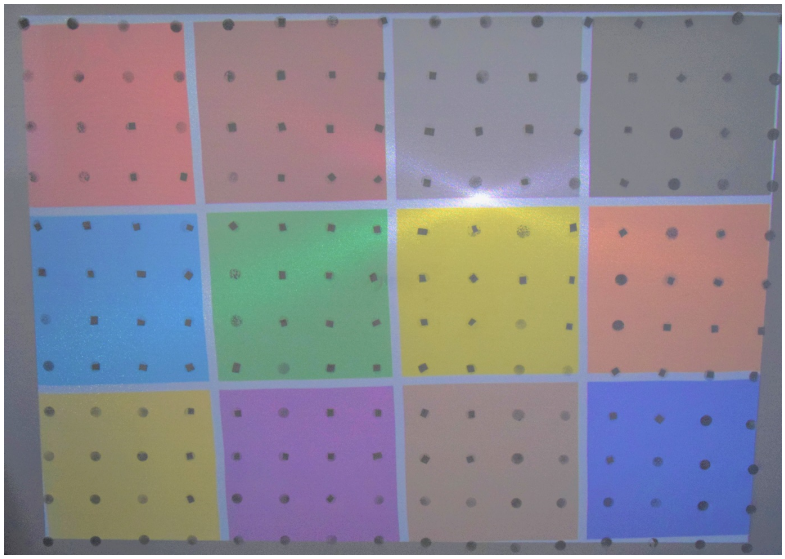


Figure 79: Photo of the colored grid projected on to the deformable screen showing that the reflective markers are visible in the projection.

Even though this was not a problem for our project focusing mainly on tracking the screen, the paint presented another issue of fading over time. In figure 78 we see that the marker in the bottom right corner is almost gone. During the evaluation of the screen, we experienced that the paint disintegrated, and fell off when we repeatedly manipulated the screen, causing the tracking to fail. Thus we had to repaint the markers several times during the project. This tedious process continued on until we ran out of paint after which we simply glued a small piece of reflective material onto the screen, which is also seen in figure 78 for the marker in the bottom left, and in the center in the top of the photo. We saw another project[31] using invisible IR-ink for the tracking, but we experienced that this ink was not completely invisible, and requires more sophisticated equipment than the Kinect to work properly. These issues should preferably be solved before user-involvement. Another observation we made during the evaluation was that placing the projector behind fabric screen causes a bright spot in the projection where the projector's light source is placed, also seen in 79. This issue is however easily solved by placing the projector outside the screen frame, and angling it such that the projection still covers the entire screen as seen in figure 80:

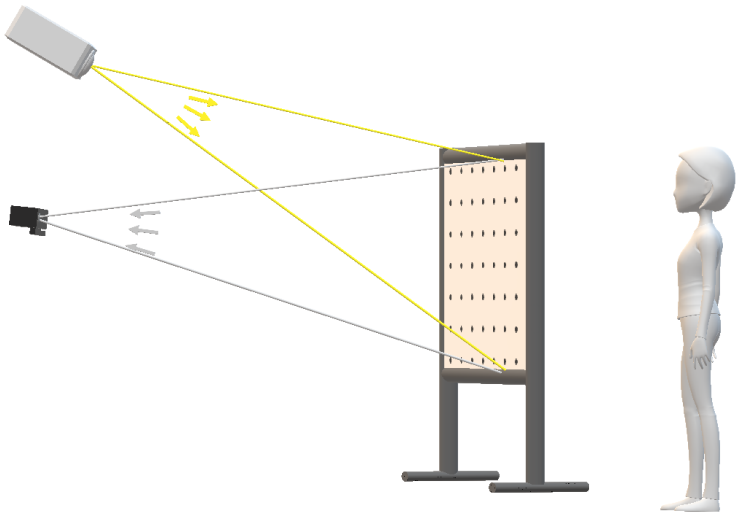


Figure 80: The projector is placed above the frame to avoid bright spots in the projection.

This will however introduce increased occlusions for the projector when the canvas is deformed in the z-axis, a problem we will cover later in the discussion.

12.2 System Design and Implementation

As described, we have implemented the system realizing the deformable screen in two separate programs to keep the tracking and visualization apart. This decision was based on that the Kinect's SDK is only provided in C#, and we would like to allow visualization in any graphics frameworks.
We have chosen develop the visualization in Unity over pure OpenGL(or similar) because it is faster to create prototypes, and the framework is cross-platform meaning that it can easily be compiled for iOS, Linux and Windows operating systems(OS). Another advantage of using Unity is that we do not need to implement the graphics pipeline from scratch, but rather use the built-in pipeline. Furthermore, as we are designing our own framework to be easy to use, the simplicity and broad application space of Unity is a great advantage. Also, developers familiar with Unity will have the benefit of knowing the basic behavior of the framework, compared to a custom made computer graphics pipeline.
Another great benefit of keeping the two programs separate is that other developers can implement their own visualization, without having to consider the tracking, and vice versa, another tracking program can also easily be connected to our visualization framework, both by implementing the interface using UDP and TCP sockets described in the implementation.

The design and implementation sections described the ImageProcessing pipeline for tracking the deformable screen. Early in the project, it became apparent that especially the task of **Point Matching** and **Estimate Missing Points** are non-trivial, and would become a large part of the project. As described, we tried out the different approaches for the **Point Matching** but settled using the Hungarian Algorithm to match markers between successive frames solely on the markers distance from each other in the frames. The choice of abandoning the other approaches in favor of the Hungarian algorithm was based on that these efforts never obtained the same level of stability in the tracking. We had to accept that the **Point Matching** in our pipeline is far from optimal, and probably the main reason for why markers are swapping ID's when tracking the screen, to be able to focus more on estimating the position of occluded markers in the **Estimate Missing Points** step.
We designed the system such that we easily can change between different models for estimating the occluded markers, and we have implemented three models which we evaluated in an experiment.

12.3 Evaluation and Estimation Models

In the Evaluation, we divided the screen into 2 locations, the deformations into 3 intensities and 11 directions, and tested the models for all their combinations.
initially, we had a hard time figuring out how to test the system because it is hard to get the ground truth in the form a the occluded markers actual position, but we found a way to obtain the ground truth by manual labeling. It is important to notice that the manual labeling is not very precise, but rather a guesstimate and thus we cannot measure the error in terms of offset precisely. This is also why we have not focused a lot on the actual size of the offset error since our guesstimates also can be a couple of centimeters off. However as the guesstimates were produced for each deformation frame by frame, we could visually follow the general direction of the occluded markers, and thus it should give us a good idea of how the models perform.

Fortunately, the evaluation confirmed our hypothesis(**H1**) that all three estimation models overall performed better than the Baseline. Moreover, it revealed that the best of our estimation models is the Extrapolation model, followed rather closely by the Displacement model, and then there is a rather large gap down to Spring model. We also had our second hypothesis (**H2**) confirmed as we saw that the extrapolation model was superior to the other models especially for deformations in directions exclusively on the Z-axis(Z+,Z-), and also for all direction including movement in the z-axis.
The analysis of the results also showed significant main effects and interaction for all the independent variables of the experiment (Model, Location,Intensity, and Direction) emphasizing that these factors all are important to consider when developing an estimation model. We saw that the error increased significantly according to the level of intensity of the deformation. We also saw that the estimations of the model were significantly better on the edge than in the center location of the screen. This is probably due to that the edge does not allow as strong deformations as the center location, so we should expect a lower error since we saw that stronger deformations gave a higher error.
Further analysis revealed that models did perform significantly better for the Medium and Strong intensities, but not for the light intensity. This is probably mainly due to that our estimation models all are linearly dependent on their neighborhood, and that elasticity of the fabric leads to the largest level of stretch between the neighboring markers for the light deformation. This means that even though we deform the fabric close to a marker, its neighboring markers do first start to move similarly when the fabric in between reaches some level of stretch. Actually, before we performed the evaluation, we experimented with adding polynomial functions to the estimations models, but we soon realized that the non-linear behavior is very different depending on the type of deformation, and the locations on the screen, so we did never reach a satisfactory result and discarded the approach. Another less critical explanation for why there is no significant difference between the models and the baseline for the light intensity could be that the light intensity simply do not move the markers enough for us to see such a difference.
Furthermore, we observed that even though the error increases significantly according to the three levels of intensity in the deformation, the error for the models did not increase proportionally to the baseline. As the baseline is defined as the distance the occluded marker is moved in the deformation, this trend indicates that the estimates of the models do not increase proportionally to the distance of the deformation. An explanation could be that, the stronger the deformation is, the less elongation of the fabric between and occluded marker and its immediate neighborhood we should expect. Explained in another way, when the fabric surrounding an occluded marker reaches some level of stretch, we expect that its neighbors will behave more linearly, and thus follow the movement of the occluded marker more closely.
In general, we saw that the Extrapolation model is superior in the cases of our evaluation, but we should be careful to declare it the best estimation model for all scenarios, as the three models each have their strengths and weaknesses which we will shortly summarize.

The Extrapolation model considers a very small neighborhood for its estimates, and in the original implementation in the study [32], they had the luxury of being able to completely discard markers if too much of its neighborhood is occluded. As we would like to track or estimate the position of the markers at all times for later

gesture detection, we had to modify the model to allow the estimate of estimates. Remembering that this means that we allow the model to base its estimation on other estimated markers if it has no visible neighbors. Testing the Extrapolation model with a more dense screen with the markers placed closer, and allowing estimates of estimates, could potentially cause a numeric explosion if a larger area is occluded, and a badly estimated marker starts a chain reaction. If nothing is done to handle such cases, the Extrapolation model could possibly perform worse then the other model for the specific scenario. This did not present issues for the model in the evaluation with the 7x9 grid of markers but may cause problems in a more dense grid pattern. Thus the experiment is a little biased in favor of Extrapolation model.
The Displacement model is very simple and provides almost as good estimates as the Extrapolation model. The model contains information of the initial position of the markers, which would be valuable if there were to be added a recovering phase for points lost in the tracking. Also, as it takes a larger neighborhood into consideration for the estimate, it should be less vulnerable to having its neighbors occluded. Moreover, this model does not allow an estimate of estimates, which can both be a strength and a weakness since an estimate based on a badly estimated neighborhood can be a lot worse than just keeping the estimated position stationary. On the contrary, it is not realistic that occluded points keep a stationary position as the presence of occlusion can only mean that the screen is being manipulated/deformed. The main drawback of the model identified in the evaluation and illustrated in figure 31 of the design section, is its weakness of underestimating the position of occluded markers when the screen is deformed in the z-axis.

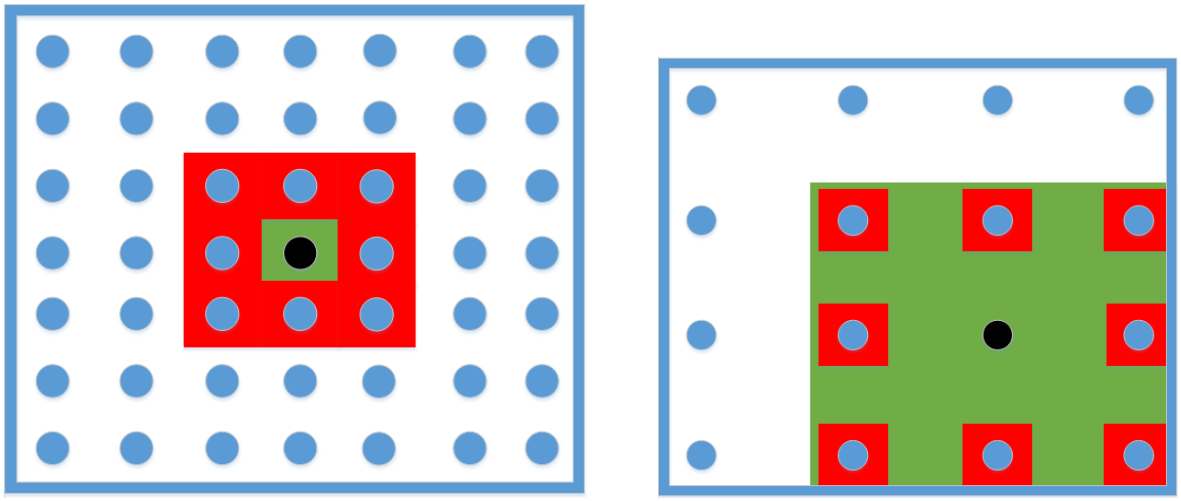
The Spring model clearly performed the worst in the evaluation, sometimes were even worse than the baseline, indicating that this model is not fit for estimation of occluded markers. Furthermore, the Spring model is the most complex, and the only model using temporal information consisting of the markers previous position. The Mass Spring model for cloth simulation is originally based on that all the points of the mesh being estimated by the spring forces for each frame, creating a chain reaction causing a realistic behavior with wrinkles and waves in the fabric. However, in our adapted version of the model, we keep visible markers fixed in their tracked position, causing the ”springs” to oscillate when a marker is occluded. We even observed that if a marker is occluded during a motion in one direction, the ”springs” retracts, causing the marker to be estimated in the opposite direction of the movement. This makes perfect sense as the springs are stretched when the screen is pulled and the marker is visible, causing a counter-movement when the marker is occluded, letting the springs retract. However, this does not mimic the actual behavior of the screen. Nonetheless, if we had a larger part of the screen occluded, we might see the spring model mimic the cloth more realistically than the other models as this better resembles the original use of the model as several neighboring markers would be controlled directly by the springs. If a more sophisticated and stable Point Matching algorithm was implemented, it would be interesting to experiment with a compound model where small occlusions were estimated using the extrapolation method, and when a large neighborhood is occluded, then let the spring model deliver the estimate.

After all, the Extrapolation model was the clear winner in our evaluation. With an offset error of 2.8,5 and 6.9 cm for the Light, Medium and Strong intensities of deformations respectively, we find this offset acceptable for a $120cm \times 85cm$ screen with three degrees of freedom(X,Y,Z).

12.4 Breaking Point

Our breaking point analysis revealed clearly some of the shortcomings of the point matching algorithm. When dealing with occlusions and having only the distance between markers from one frame to another to evaluate, then problems will arise whenever points get to close. We see a few things contributing to this, all of which can be handled to some extent. The first thing we became aware of early in the project when looking at related works, was the fact that the further markers move between frames, the harder it is to determine where they came from. While it is possible to reduce this distance by simply reducing the time between frames, we have hardware restrictions that rule this out as an option for us. We can, however, make the distances shorter relative to the spacing between the markers, which was the way we chose to handle the high amount of breaks we experienced. By increasing the spacing, and effectively reducing the number of markers, we can perform a

stronger deformation, and faster ones, without the markers overlapping and thereby causing breaks. This gives not only a certain freedom to move a marker around in the square given by the surrounding markers but with the greater spacing. A marker can also in many cases move beyond its neighbors without issues, as it can freely move in the between to neighboring markers with a distance to them great enough as to avoid breaks, this theory is illustrated in figure 81. We could achieve the same thing, working with an ultra high-resolution camera and much smaller markers, but this is another hardware issue, and it might prove to make creases and folds generate a lot more occlusions.
With the knowledge, we have gained throughout the project, a new attempt to optimize point matching would probably involve extensive testing of the screen behavior as mentioned in the Point Matching section of System Design (section 6.1.6) or an altogether different tracking approach.



(a) Shows the freedom of movement a marker has in a tight grid.

(b) Shows the freedom of movement a marker has in a relatively highly spaced grid.

Figure 81: Shows the freedom of movement a marker has in a grid, the green being a safe zone for the black marker, while moving to the red might potentially cause issues with point matching.

12.5 Efficiency

As stated from the beginning, we had strict time restrictions for the program as we wanted it to run real-time. However, this restriction was not that demanding, due to the 30 FPS camera we are using. From the timing experiments, we see a worst case scenario cost, of an iteration of the tracking pipeline in table 10, which is 5.5ms without accounting for hardware delay. This gives us a theoretical minimum speed of 180 FPS if the applied camera can deliver that many. Looking at the mean timings instead, and the not so costly displacement estimation method, we could potentially run 370 FPS, giving a bit more freedom to choose a better camera without having to do further optimization of the code.
While this is definitely real-time for the program, the user experience still seems a bit delayed. As we do not see this delay stemming from the program itself, we have to put it down to in and output delay. As these are hardware limitations, there is not much we can do to give a smoother experience.
We did see some unexplained run time spikes in parts of the program, as our testing computer was not dedicated, but was running normal background tasks, we can have run into minor delays due to limitations of the computer itself.
If we were to use an even faster camera we could speed up the program considerably by converting our initial tracking phase(taking up the majority of the computation time) from using the standard EMGU library to the CUDA version which speeds up the image processing by moving the task from the CPU to GPU.
Also, we run the entire tracking pipeline on a single thread, and some task like e.g. constructing the Cost Matrix

for the Hungarian Algorithm could easily be run in parallel on multiple threads, speeding up the process considerably.
Finally, if we increased the ”resolution” of the screen by increasing the number of markers, more heavy estimation models like the Spring model could also be moved from the CPU to the GPU to speed up the process. This is also described in the project by Mosegaard [27] which we used for inspiration for our implementation of the Mass Spring model. However, this should only be considered if we expect a large number of markers to be occluded frequently, because otherwise moving the visibly tracked data from the CPU to the GPU, will take more time than we gain by the faster computational time.

12.6 ProCam and Screen Occlusion

We are using a single camera and a single projector in our setup, each of them having their perspective of the deformable screen. When strong deformations are applied to the screen it can cause occlusion for the projector and camera individually, thus we can experience that the camera cannot see a marker ”visible” from the projectors view and vice versa.
Occlusion for the camera and projector has different characteristics because when the projector's view is hindered by a part of the screen occluding another, it is only the user experience that is decreased because the user will experience dark areas on the screen. If however the camera's view is blocked by the deformation, the occluded markers need to be estimated, which surely also can have an effect on the user experience if the markers are badly estimated.
We can divide the occlusion up in two categories, those which will hinder the view independently of where the camera or projector is placed e.g grasping the screen into the hands, and those where moving the camera or projector would remove the occlusion. For the first category, there is not much else to do than estimate the position of the occluded markers for the camera. This type of deformation does not have a serious impact on the user experience for the projection, because it is caused by that the screen canvas has folds introduced by the users' hands that covers a large part of the occluded area anyway.
The other category, often caused by strong deformations without grabbing the screen into the hand, where some part of the screen blocks other parts from a specific viewpoint. This kind of occlusion has a heavy impact in our system as it often occludes a greater number of markers from the camera's view, and leaves large dark areas where the projection is blocked. This type of occlusions could be reduced significantly by using more cameras and projectors, covering the screen from different viewpoints as illustrated in figure 82:

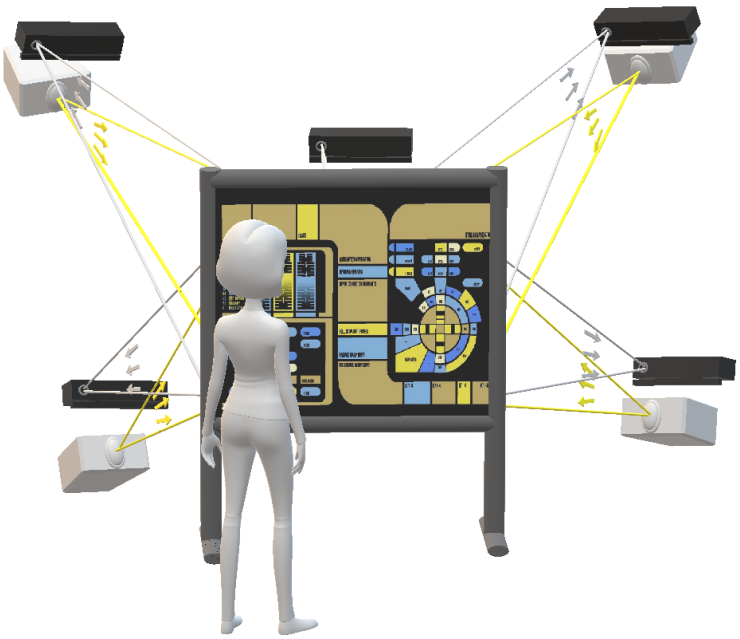


Figure 82: Multi-ProCam setup covering the deformable screen from different viewpoints to reduce occlusions.

We can place the cameras where we want to cover the screen optimally, however, we still need to angle the projector to avoid the bright spots on the screen from the projector lamps. Such a setup requires that all the projectors and cameras are correctly calibrated, and a different number of projectors and cameras can be applied. The RoomAliveToolkit used for the calibration in this project already supports calibration of multiple cameras and projectors, and we could achieve support for a Multi-ProCam setup in two ways. Either by starting an instance of the ScreenTracker program for each camera, modify the Visualization Program to convert the tracked coordinates into the same coordinate space, and then combine the tracking information there. Otherwise, the ScreenTracker program could unify the coordinate space, and use the frames from the multiple cameras directly, allowing us the discover otherwise occluded markers, but also presumably improve the precision of visible markers since we would have more measurements in our data. In both cases, the visualization program should be customized to handle multiple projections, which is just a question of adding more Projector game objects in the Unity scene, and adjust their viewpoint to the calibration data.
Adding more cameras and projectors to a single computer will inevitably increase the computation time, thus prepossessing of frames from the cameras could be distributed out on multiple computers, and the visualization could be handled on a separate machine. Such a modification should be a manageable task as the programs already communicate over network sockets(TCP and UDP).

13 Conclusion and Future Work

With the current technology not allowing for active deformable displays, we have created a prototype which models it instead, using a deformable canvas and a ProCam setup. Our solution runs in real-time and allows for deformations in all axes, as well as local deformations such as stretch between tracking markers. This is achieved using only commodity hardware, and the fact that we track all markers of the deformable screen even when occluded is to our knowledge new to this field of research.

We contributed with three different techniques to estimate the position of occluded markers; Extrapolation, Vector Displacement as well as an implementation of a Mass Spring model. Where the Extrapolation and Mass Spring model are inspired by related work but adapted to handle our specific use.
Exploring related work revealed that other projects rarely had any real evaluation of their estimation models, as obtaining the ground truth of the non-visible markers is challenging at best. We obtained our ground truth by manual labeling. This has a certain amount of inaccuracy, but does, however, allow us to statistically evaluate each model against a common truth.
The statistical evaluation revealed that all the tested independent variables and their combinations have an effect on the precision of an estimation algorithm, and as such further research into estimation models should take these into consideration. With the independent variables being; the intensity of the deformation, the direction of the deformation, and where on the screen the deformation is performed(location).
Throughout the project, we experienced difficulties producing tracking markers, which does not interfere with the user experience, as they cause visible artifacts on the screen. Finding a more translucent solution would be recommended before user involvement. Moreover, the canvas itself also had minor issues of being a little hard to grab into the hands and also leaving semi-permanent deformations when exposed to strong gestures. Preferably, future work should address this issue by finding a better canvas material that is easy to grab onto, highly deformable, and fully retractable.
Our evaluation showed that the Extrapolation model generally performs the best, meaning that it presents the lowest error in terms of offset between an estimation and the ground truth. With the tracking pipeline initially assigning IDs to each marker, the main issue in our implementation is maintaining these IDs between successive frames, especially when two markers are located very close to each other. We defined this problem as Point Matching and considered several different solutions. In the end, however, we chose to use the Hungarian algorithm for solving the assignment problem, in spite of its limitations, so that we could focus on the estimation models instead. As a result, the tracking pipeline only reliably supports a limited density of markers on the screen.
With a fairly good estimation model in place, we would recommend future work to focus on improving the Point Matching algorithm. Another possible improvement would be to get more data to work from and limit the number of occluded markers, as well as the level of occlusion in the projection seen by the user. This could be done by utilizing multiple cameras and projectors in the ProCam-setup, which also can be accomplished using commodity hardware.
Once stable tracking is achieved, future work could address adding gesture detection to the framework, allowing research in the interaction space of a deformable screen.

Appendices

A Using The FlexIO Framework

In this section, we will describe how our FlexIO Framework is installed and run.
As described in the report the system consists of two programs, one for tracking the deformable screen, and one for the visualization and projection of the screen.

A.1 ScreenTracker

The C# tracking program is named ScreenTracker and can be obtained on GitHub via this URL: `https://github.com/MultiPeden/ScreenTracker` Accessed: 19/04-2018
The project requires the following frameworks and libraries to be installed:

- **EmguCV** -framework[40] C#-wrapper for OpenCV[41]. An installation guide is found here: `http://www.emgu.com/wiki/index.php/Download_And_Installation` Accessed: 19/04-2018.
- **Accord.net**-framework[42] can be downloaded directly in Visual Studio's NuGet package manager.
- **CSVHelper**, Finally the CSVHelper library[43] can be downloaded directly in Visual Studio's NuGet package manager.

The ScreenTracker can be started directly in Visual Studio, or it can be build to ScreenTracker.exe and run in the terminal. Running the ScreenTracker.exe without any argument opens the ScreenTracker without the GUI, and adding the "-s" flag will launch the ScreenTracker with the GUI.
For the tracking to work, it is important that the dimensions of the grid have been entered in the user setting file explained in section B.

A.2 Visualization - FlexIO

The visualization Unity program for projecting the image onto the canvas is named FlexIO and is found via this URL `https://github.com/MultiPeden/FlexIO` Accessed: 13/05-2018

This program relies on that the ProCam setup has been calibrated using the RoomAliveToolkit's ProCamCalibration program found at `https://github.com/Microsoft/RoomAliveToolkit` Accessed: 19/04-2018. The calibration process is described in section 7.3 on page 61.
The output file from the calibration has to be named `calibration.xml` and placed inside the FlexIO Unity project's `FlexIO\textbackslashAssets\Resources` folder.
The program can both be started directly in Unity by pressing the "Play" button, and can also be built and run as a separate program. When the ScreenTracker is running, clicking the "Triangulate" button in the Unity Program will initialize the 3D model of the Deformable Screen. Clicking the "Reset" button in the Unity program will reset the tracking in the "ScreenTracker".

[40] `http://www.emgu.com/wiki/index.php/Main_Page` Accessed: 19/04-2018
[41] `https://opencv.org/` Accessed: 19/04-2018
[42] `http://accord-framework.net/` Accessed: 19/04-2018
[43] `http://joshclose.github.io/CsvHelper/` Accessed: 19/04-2018

B ScreenTracker's User Settings

The Implementation section 7 starting on page 44 refers to this section explaining the user settings of the ScreenTracker.
The user settings is a list of variables which are used as parameters for methods in the ScreenStracker. As we have envisioned that other students might further develop the ScreenTracker in the future, we would like the parameters to be easy to tune. Thus we have separated the user settings from the rest of the code to enable tuning of the ScreenTracker without requiring re-compilation of the entire project.

The XML file `ScreenTracker.exe.config` in the ScreenTracker project contains the user setting in the `userSettings` field, and in the below table 12 we have added a short explanation to each field of the settings file. Table 13 in section B.1 shows the actual values of the user settings used for the Estimate Evaluation from section 10

Name	Type	Used for
KernelSize	int	Specifies the size of the kernel used for morphological operations in `ProcessIRFrame()` of the ***Imageprocessing*** class in pixels
DataIndicatorThickness	int	Thickness of bounding boxes around markers drawed in `DrawTrackedData()` for the GUI's visualization in the ***Imageprocessing*** class
GridRows	int	The number of markers i a row in the grid of the deformable screen(***Imageprocessing*** class). Needs to be set correctly for the tracking to work
GridColums	int	The number of markers i a columns in the grid of the deformable screen.(***Imageprocessing*** class) Needs to be set correctly for the tracking to work
Spring_Damping	float	The damping constant for the ***Spring*** class used in the Mass Spring model
Spring_StepSize	float	The step size for the Verlet integration in the ***PointInfoSpring*** class used in the Mass Spring model
Spring_ConstraintIterations	int	The number of iterations we want the run the computation of the internal force for each frame ***SpringScreen*** class used for the Mass Spring model
Spring_constant	float	The spring constant for the ***Spring*** class used for the Mass Spring model
IRPixelPadding	int	The number of pixels of the edge of the IR-frame we want to discard for removing noise generated in the edges of the frame, used in the constructor of the ***Imageprocessing*** class

Table 12: The fields of the user settings file explained briefly.

B.1 User Settings for Evaluation

Table 13 shows the actual values of the user settings used for the Estimate Evaluation section 10. The user settings XML file is seen in section B.2

Name	Type	Value
KernelSize	int	3
DataIndicatorThickness	int	4
GridRows	int	7
GridColums	int	9
Spring_Damping	float	0.1
Spring_StepSize	float	0.0333
Spring_ConstraintIterations	int	20
Spring_constant	float	1
IRPixelPadding	int	20

Table 13: User settings used in the evaluation of the system.

B.2 User setting XML file

The actual user setting XML file used for the Estimate Evaluation section 10

```
<userSettings>
  <ScreenTracker.Properties.UserSettings>
    <setting name="kernelSize" serializeAs="String">
      <value>3</value>
    </setting>
    <setting name="DataIndicatorMinimumArea" serializeAs="String">
      <value>2</value>
    </setting>
    <setting name="DataIndicatorThickness" serializeAs="String">
      <value>4</value>
    </setting>
    <setting name="GridRows" serializeAs="String">
      <value>7</value>
    </setting>
    <setting name="GridColums" serializeAs="String">
      <value>9</value>
    </setting>
    <setting name="Spring_Damping" serializeAs="String">
      <value>0.1</value>
    </setting>
    <setting name="Spring_StepSize" serializeAs="String">
      <value>0.0333</value>
    </setting>
    <setting name="Spring_ConstraintIterations" serializeAs="String">
      <value>20</value>
    </setting>
    <setting name="Spring_constant" serializeAs="String">
      <value>1</value>
    </setting>
    <setting name="IRPixelPadding" serializeAs="String">
      <value>20</value>
    </setting>
  </ScreenTracker.Properties.UserSettings>
</userSettings>
```

xml/usersettings.xml

C RoomAliveToolkit's ProCamCalibration File

```
<?xml version="1.0" encoding="utf-8"?>
<ProjectorCameraEnsemble xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/RoomAliveToolkit">
  <cameras>
    <ProjectorCameraEnsemble.Camera>
      <calibration>
        <colorCameraMatrix>
          <ValuesByColumn xmlns:d6p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
            <d6p1:ArrayOfdouble>
              <d6p1:double>1060.9328797317123</d6p1:double>
              <d6p1:double>0</d6p1:double>
              <d6p1:double>0</d6p1:double>
            </d6p1:ArrayOfdouble>
            <d6p1:ArrayOfdouble>
              <d6p1:double>0</d6p1:double>
              <d6p1:double>1061.5557527397523</d6p1:double>
              <d6p1:double>0</d6p1:double>
            </d6p1:ArrayOfdouble>
            <d6p1:ArrayOfdouble>
              <d6p1:double>947.34672921042977</d6p1:double>
              <d6p1:double>528.86120189436781</d6p1:double>
              <d6p1:double>1</d6p1:double>
            </d6p1:ArrayOfdouble>
          </ValuesByColumn>
        </colorCameraMatrix>
        <colorLensDistortion>
          <ValuesByColumn xmlns:d6p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
            <d6p1:ArrayOfdouble>
              <d6p1:double>0.014585398814289306</d6p1:double>
              <d6p1:double>-0.0017344508974270292</d6p1:double>
            </d6p1:ArrayOfdouble>
          </ValuesByColumn>
        </colorLensDistortion>
        <depthCameraMatrix>
          <ValuesByColumn xmlns:d6p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
            <d6p1:ArrayOfdouble>
              <d6p1:double>366.19356705460592</d6p1:double>
              <d6p1:double>0</d6p1:double>
              <d6p1:double>0</d6p1:double>
            </d6p1:ArrayOfdouble>
            <d6p1:ArrayOfdouble>
              <d6p1:double>0</d6p1:double>
              <d6p1:double>366.2171602922046</d6p1:double>
              <d6p1:double>0</d6p1:double>
            </d6p1:ArrayOfdouble>
            <d6p1:ArrayOfdouble>
              <d6p1:double>257.1041282985118</d6p1:double>
              <d6p1:double>218.95675898009807</d6p1:double>
              <d6p1:double>1</d6p1:double>
            </d6p1:ArrayOfdouble>
          </ValuesByColumn>
        </depthCameraMatrix>
        <depthLensDistortion>
          <ValuesByColumn xmlns:d6p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
            <d6p1:ArrayOfdouble>
```

```
55            <d6p1:double>0.068531581990569371</d6p1:double>
56            <d6p1:double>-0.17692115629766447</d6p1:double>
57          </d6p1:ArrayOfdouble>
58        </ValuesByColumn>
59      </depthLensDistortion>
60      <depthToColorTransform>
61        <ValuesByColumn xmlns:d6p1="http://schemas.microsoft.com/2003/10/Serialization/
              Arrays">
62          <d6p1:ArrayOfdouble>
63            <d6p1:double>0.99999743700027466</d6p1:double>
64            <d6p1:double>0.0020087773445993662</d6p1:double>
65            <d6p1:double>-0.0010338653810322285</d6p1:double>
66            <d6p1:double>0</d6p1:double>
67          </d6p1:ArrayOfdouble>
68          <d6p1:ArrayOfdouble>
69            <d6p1:double>-0.0020074362400919 2</d6p1:double>
70            <d6p1:double>0.99999713897705078</d6p1:double>
71            <d6p1:double>0.0012859050184488297</d6p1:double>
72            <d6p1:double>0</d6p1:double>
73          </d6p1:ArrayOfdouble>
74          <d6p1:ArrayOfdouble>
75            <d6p1:double>0.0010364879854023457</d6p1:double>
76            <d6p1:double>-0.0012838188558816 91</d6p1:double>
77            <d6p1:double>0.99999862909317017</d6p1:double>
78            <d6p1:double>0</d6p1:double>
79          </d6p1:ArrayOfdouble>
80          <d6p1:ArrayOfdouble>
81            <d6p1:double>0.052635055018407831</d6p1:double>
82            <d6p1:double>9.1107834368623367E-05</d6p1:double>
83            <d6p1:double>7.1376111093023414E-05</d6p1:double>
84            <d6p1:double>1</d6p1:double>
85          </d6p1:ArrayOfdouble>
86        </ValuesByColumn>
87      </depthToColorTransform>
88    </calibration>
89    <hostNameOrAddress>localhost</hostNameOrAddress>
90    <name>0</name>
91    <pose>
92      <ValuesByColumn xmlns:d5p1="http://schemas.microsoft.com/2003/10/Serialization/
            Arrays">
93        <d5p1:ArrayOfdouble>
94          <d5p1:double>1</d5p1:double>
95          <d5p1:double>0</d5p1:double>
96          <d5p1:double>0</d5p1:double>
97          <d5p1:double>0</d5p1:double>
98        </d5p1:ArrayOfdouble>
99        <d5p1:ArrayOfdouble>
100         <d5p1:double>0</d5p1:double>
101         <d5p1:double>1</d5p1:double>
102         <d5p1:double>0</d5p1:double>
103         <d5p1:double>0</d5p1:double>
104       </d5p1:ArrayOfdouble>
105       <d5p1:ArrayOfdouble>
106         <d5p1:double>0</d5p1:double>
107         <d5p1:double>0</d5p1:double>
108         <d5p1:double>1</d5p1:double>
109         <d5p1:double>0</d5p1:double>
110       </d5p1:ArrayOfdouble>
111       <d5p1:ArrayOfdouble>
112         <d5p1:double>0</d5p1:double>
113         <d5p1:double>0</d5p1:double>
114         <d5p1:double>0</d5p1:double>
```

```
115           <d5p1:double>1</d5p1:double>
116         </d5p1:ArrayOfdouble>
117       </ValuesByColumn>
118     </pose>
119   </ProjectorCameraEnsemble.Camera>
120 </cameras>
121 <name>Untitled</name>
122 <projectors>
123   <ProjectorCameraEnsemble.Projector>
124     <cameraMatrix>
125       <ValuesByColumn xmlns:d5p1="http://schemas.microsoft.com/2003/10/Serialization/
            Arrays">
126         <d5p1:ArrayOfdouble>
127           <d5p1:double>2158.3246049903328</d5p1:double>
128           <d5p1:double>0</d5p1:double>
129           <d5p1:double>0</d5p1:double>
130         </d5p1:ArrayOfdouble>
131         <d5p1:ArrayOfdouble>
132           <d5p1:double>0</d5p1:double>
133           <d5p1:double>2158.3246049903328</d5p1:double>
134           <d5p1:double>0</d5p1:double>
135         </d5p1:ArrayOfdouble>
136         <d5p1:ArrayOfdouble>
137           <d5p1:double>1122.2755848506731</d5p1:double>
138           <d5p1:double>495.41790673852597</d5p1:double>
139           <d5p1:double>1</d5p1:double>
140         </d5p1:ArrayOfdouble>
141       </ValuesByColumn>
142     </cameraMatrix>
143     <displayIndex>2</displayIndex>
144     <height>1080</height>
145     <hostNameOrAddress>localhost</hostNameOrAddress>
146     <lensDistortion>
147       <ValuesByColumn xmlns:d5p1="http://schemas.microsoft.com/2003/10/Serialization/
            Arrays">
148         <d5p1:ArrayOfdouble>
149           <d5p1:double>0</d5p1:double>
150           <d5p1:double>0</d5p1:double>
151         </d5p1:ArrayOfdouble>
152       </ValuesByColumn>
153     </lensDistortion>
154     <lockIntrinsics>false</lockIntrinsics>
155     <name>0</name>
156     <pose>
157       <ValuesByColumn xmlns:d5p1="http://schemas.microsoft.com/2003/10/Serialization/
            Arrays">
158         <d5p1:ArrayOfdouble>
159           <d5p1:double>0.99880623817443848</d5p1:double>
160           <d5p1:double>-0.0076080029830336571</d5p1:double>
161           <d5p1:double>0.04825226217508316</d5p1:double>
162           <d5p1:double>0</d5p1:double>
163         </d5p1:ArrayOfdouble>
164         <d5p1:ArrayOfdouble>
165           <d5p1:double>0.0054789846763014793</d5p1:double>
166           <d5p1:double>0.99901199340820313</d5p1:double>
167           <d5p1:double>0.044102743268013</d5p1:double>
168           <d5p1:double>0</d5p1:double>
169         </d5p1:ArrayOfdouble>
170         <d5p1:ArrayOfdouble>
171           <d5p1:double>-0.048540122807025909</d5p1:double>
172           <d5p1:double>-0.043785721063613892</d5p1:double>
173           <d5p1:double>0.99786108732223511</d5p1:double>
```

```
          <d5p1:double>0</d5p1:double>
        </d5p1:ArrayOfdouble>
        <d5p1:ArrayOfdouble>
          <d5p1:double>0.084802281238860591</d5p1:double>
          <d5p1:double>0.11869591082533781</d5p1:double>
          <d5p1:double>-0.018691528581482812</d5p1:double>
          <d5p1:double>1</d5p1:double>
        </d5p1:ArrayOfdouble>
      </ValuesByColumn>
    </pose>
    <width>1920</width>
  </ProjectorCameraEnsemble.Projector>
 </projectors>
</ProjectorCameraEnsemble>
```

xml/calibration.xml

D Evaluation Tables

Factor	**Significance**
Model(4)	$(F_{1.005,8.037} = 202.170\ p < 0.001)$
Location(2)	$(F_{1,8} = 1231.668\ p < 0.001)$
Intensity(3)	$(F_{1.153,9.225} = 2277.808\ p < 0.001)$
Direction(11)	$(F_{2.253,18.022} = 232.495\ p < 0.001)$
Model * Location	$(F_{1.020,8.162} = 273.146\ p < 0.001)$
Model * Intensity	$(F_{1.583,12.664} = 1088.559\ p < 0.001)$
Location * Intensity	$(F_{9.873,14.134} = 9.873\ p < 0.003)$
Model * Location * Intensity	$(F_{1.497,11.976} = 459.918\ p < 0.001)$
Model * Direction	$(F_{1.356,10.849} = 264.527\ p < 0.001)$
Location * Direction	$(F_{1.587,12.693} = 150.139\ p < 0.001)$
Model * Location * Direction	$(F_{1.764,14.115} = 237.525\ p < 0.001)$
Intensity * Direction	$(F_{2.132,17.058} = 38.581\ p < 0.001)$
Model * Intensity * Direction	$(F_{2.178,17.422} = 96.090\ p < 0.001)$
Location * Intensity * Direction	$(F_{1.925,15.398} = 30.341\ p < 0.001)$
Model * Location * Intensity * Direction	$(F_{2.364,18.915} = 48.841\ p < 0.001)$

Table 14: Significance levels for the repeated measures analysis of variance (ANOVA), with significant main effects and interactions for all independent variables, with the independent variables seen to the left and the significance level to the right with corrected degrees of freedom according to Greenhouse-Geisser.

Model \Direction	X	XY	XYZ-	XYZ+	XZ-	XZ+	Y	YZ-	YZ+	Z-	Z+
Baseline	0.073	0.072	0.128	0.105	0.086	0.116	0.089	0.105	0.110	0.138	0.136
Displacement	0.049	0.052	0.084	0.075	0.046	0.072	0.069	0.061	0.071	0.073	0.061
Extrapolation	0.037	0.053	0.062	0.066	0.031	0.073	0.062	0.039	0.068	0.022	0.025
Spring	0.066	0.080	0.104	0.095	0.056	0.131	0.096	0.071	0.108	0.085	0.110

Table 15: Mean errors for model by direction.

Direction	Non-significant to	p values for rest
X		$(p < 0.002)$
XY	Z-,Z+	$(p < 0.045)$
XYZ-	XYZ+,YZ+	$(p < 0.021)$
XYZ+	XYZ-,Y,YZ+	$(p < 0.001)$
XZ-		$(p < 0.001)$
XZ+		$(p < 0.021)$
Y	XYZ+	$(p < 0.008)$
YZ-	Z-	$(p < 0.001)$
YZ+	XYZ-,XYZ+	$(p < 0.008)$
Z-	XY,YZ-	$(p < 0.045)$
Z+	XY	$(p < 0.045)$

Table 16: Directions pairwise Bonferroni corrected comparison from the $3 \times 2 \times 3 \times 11$ (***Model*** $\times$ ***Location*** $\times$ ***Intensity*** $\times$ ***Direction***) *repeated measures ANOVA(With the Baseline model excluded). To the left we see each direction, in the middle we see which other direction it in not significantly different to, and to the right we see the bounded* p *value for the remaining directions.*

E Photos of the Deformable Screen

Figure 83: The deformable screen with a colored grid as graphical texture. Here the screen is exposed to push deformations by both hands of the user.

Figure 84: Same scenario as in figure 83, here from a sideways perspective.

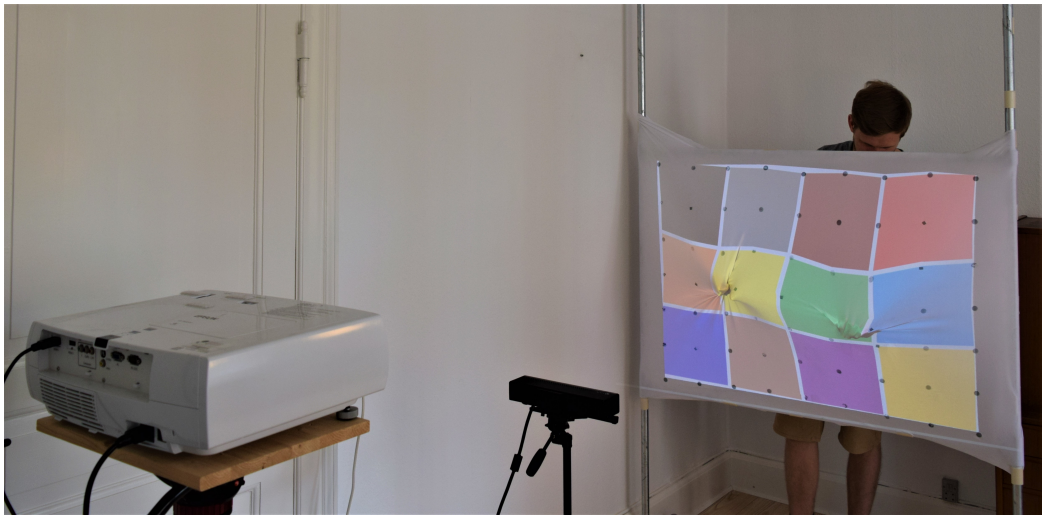


Figure 85: The deformable screen with a colored grid as graphical texture. Here the screen is grabbed into the user's hands while being dragged downwards.

Figure 86: Here the user is dragging the screen sideways as much a the fabric allows.

Figure 87: The deformable screen with a video of "a color explosion" as graphical texture. Here the user tries to push the color explosion away.

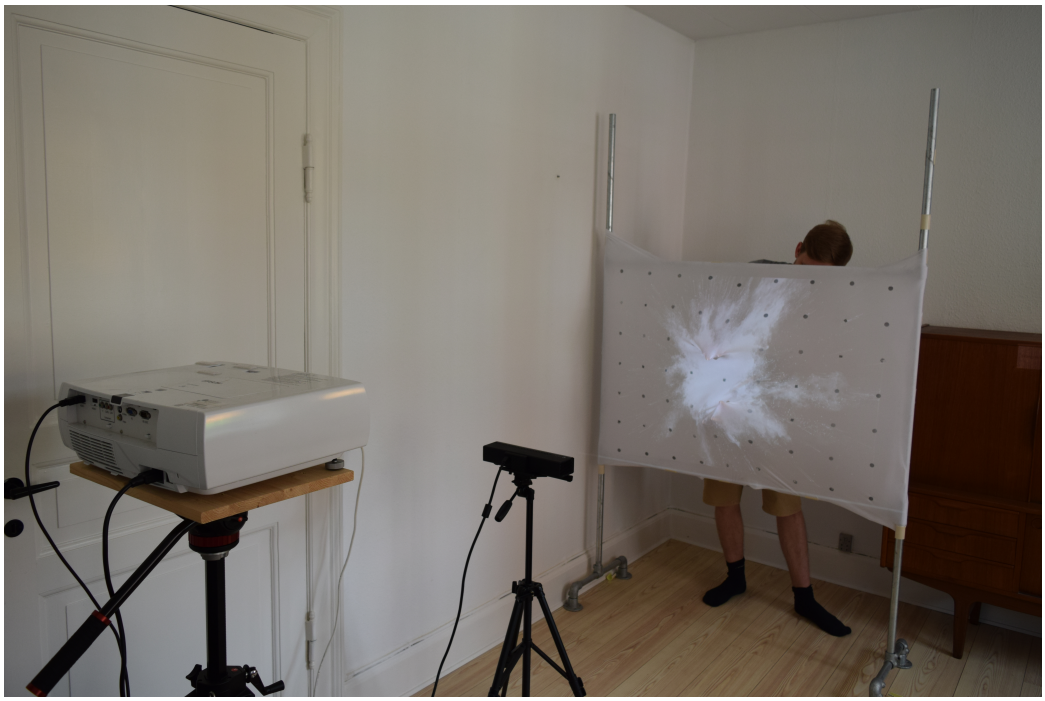


Figure 88: The deformable screen with a video of "a black and white explosion" as graphical texture. Here the user tries to grab the explosion between his hands.

References

[1] F. Aurenhammer, R. Klein, and D.-T. Lee. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1st edition, 2013.

[2] D. Baraff. Physically based modeling. *SIGGRAPH Course Notes, ACM SIGGRAPH*.

[3] D. Baraff and A. Witkin. Large steps in cloth simulation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 43–54, New York, NY, USA, 1998. ACM.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[5] D. Bradley, G. Roth, and P. Bose. Augmented reality on cloth with realistic illumination. *Machine Vision and Applications*, 20(2):85–92, 2007.

[6] D. E. Breen, D. H. House, and P. H. Getto. A physically-based particle model of woven cloth. *The Visual Computer*, 8(5):264–277, Sep 1992.

[7] M. Carignan, Y. Yang, N. M. Thalmann, and D. Thalmann. Dressing animated synthetic actors with complex deformable clothes. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 99–104, New York, NY, USA, 1992. ACM.

[8] G. Casiez, N. Roussel, and D. Vogel. 1 € filter: A simple speed-based low-pass filter for noisy input in interactive systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2527–2530, New York, NY, USA, 2012. ACM.

[9] A. Cassinelli and M. Ishikawa. Khronos projector. In *ACM SIGGRAPH 2005 Emerging technologies*, page 10. ACM, 2005.

[10] Y. Chan, A. Hu, and J. Plant. A kalman filter based tracking scheme with input estimation. *IEEE Transactions on Aerospace and Electronic Systems*, AES-15(2):237–244, 1979.

[11] C. R. Feynman. *Modeling the appearance of cloth*. PhD thesis, Massachusetts Institute of Technology, 1986.

[12] Y. Fujimoto, R. T. Smith, T. Taketomi, G. Yamamoto, J. Miyazaki, H. Kato, and B. H. Thomas. Geometrically-correct projection-based texture mapping onto a deformable object. *IEEE Transactions on Visualization and Computer Graphics*, 20(4):540–549, 2014.

[13] V. Gay-Bellile, A. Bartoli, and P. Sayd. Direct estimation of non-rigid registrations with image-based self-occlusion reasoning. *2007 IEEE 11th International Conference on Computer Vision*, 2007.

[14] M. Gupta, S. K. Nayar, M. B. Hullin, and J. Martin. Phasor imaging: A generalization of correlation-based time-of-flight imaging. *ACM Trans. Graph.*, 34(5):156:1–156:18, Nov. 2015.

[15] A. Hilsmann and P. Eisert. Tracking and retexturing cloth for real-time virtual clothing applications. *Computer Vision/Computer Graphics CollaborationTechniques Lecture Notes in Computer Science*, page 94–105, 2009.

[16] X. P. Institut and X. Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *In Graphics Interface*, pages 147–154, 1996.

[17] D. Joele, G. van der Mast, and M. C. Juan-Lizandra. Development of an augmented reality system using artoolkit and user invisible markers. *Research Assignment, Master Programme Media & Knowledge Engineering, Valencia*, 2005.

[18] B. Jones, R. Sodhi, M. Murdock, R. Mehra, H. Benko, A. Wilson, E. Ofek, B. MacIntyre, N. Raghuvanshi, and L. Shapira. Roomalive: Magical experiences enabled by scalable, adaptive projector-camera units. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 637–644, New York, NY, USA, 2014. ACM.

[19] B. Lahey, A. Girouard, W. Burleson, and R. Vertegaal. Paperphone: Understanding the use of bend gestures in mobile devices with flexible electronic paper displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1303–1312, New York, NY, USA, 2011. ACM.

[20] D. Lanman and G. Taubin. Build your own 3d scanner: 3d photography for beginners. In *SIGGRAPH '09*, 2009.

[21] W.-C. Lin and Y. Liu. A lattice-based mrf model for dynamic near-regular texture tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(5):777–792, 2007.

[22] R. B. Madhkour, J. Leroy, and F. Zajéga. Kosei: a kinect observation system based on kinect and projector calibration. 2012.

[23] J. Meisner. Collaboration, expertise produce enhanced sensing in xbox one. `https://blogs.technet.microsoft.com/microsoft_blog/2013/10/02/collaboration-expertise-produce-enhanced-sensing-in-xbox-one/`. Accessed: 2010-09-30.

[24] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler. MOT16: A benchmark for multi-object tracking. *arXiv:1603.00831 [cs]*, Mar. 2016. arXiv: 1603.00831.

[25] Y. Mo, D. Yu, J. Song, K. Zheng, and Y. Guo. Vehicle position updating strategy based on kalman filter prediction in vanet environment. *Discrete Dynamics in Nature and Society*, 2016:1–9, 2016.

[26] A. B. Mor and T. Kanade. Modifying soft tissue models: Progressive cutting with minimal new element creation. In S. L. Delp, A. M. DiGoia, and B. Jaramaz, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2000*, pages 598–607, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[27] J. Mosegaard. Cardiac surgery simulation-graphics hardware meets congenital heart disease. *Department of Computer Science. Aarhus, Denmark: University of Aarhus*, 2006.

[28] X. Mu, J. Che, T. Hu, and Z. Wang. A video object tracking algorithm combined kalman filter and adaptive least squares under occlusion. *2016 9th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, 2016.

[29] M. Müller, J. Stam, D. James, and N. Thürey. Real time physics: Class notes. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, pages 88:1–88:90, New York, NY, USA, 2008. ACM.

[30] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[31] G. Narita, Y. Watanabe, and M. Ishikawa. Dynamic projection mapping onto a deformable object with occlusion based on high-speed tracking of dot marker array. *Proceedings of the 21st ACM Symposium on Virtual Reality Software and Technology - VRST 15*, 2015.

[32] G. Narita, Y. Watanabe, and M. Ishikawa. Dynamic projection mapping onto a deformable object with occlusion based on high-speed tracking of dot marker array. In *Proceedings of the 21st ACM Symposium on Virtual Reality Software and Technology*, VRST '15, pages 149–152, New York, NY, USA, 2015. ACM.

[33] T. Omirou, A. Marzo, S. A. Seah, and S. Subramanian. Levipath. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI 15*, 2015.

[34] H. Park and J.-I. Park. Invisible marker–based augmented reality. 26:829–848, 08 2010.

[35] A. Payne, A. Daniel, A. Mehta, B. Thompson, C. S. Bamji, D. Snow, H. Oshima, L. Prather, M. Fenton, L. Kordus, et al. 7.6 a 512× 424 cmos 3d time-of-flight image sensor with multi-frequency photo-demodulation up to 130mhz and 2gs/s adc. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 134–135. IEEE, 2014.

[36] J. Peschke, F. Göbel, T. Gründer, M. Keck, D. Kammer, and R. Groh. Depthtouch: an elastic surface for tangible computing. In *Proceedings of the international working conference on advanced visual interfaces*, pages 770–771. ACM, 2012.

[37] J. Pilet, V. Lepetit, and P. Fua. Fast non-rigid surface detection, registration and realistic augmentation. *International Journal of Computer Vision*, 76(2):109–122, Mar 2007.

[38] D. Pritchard and W. Heidrich. Cloth motion capture. *Proceedings of the SIGGRAPH 2003 conference on Sketches & applications in conjunction with the 30th annual conference on Computer graphics and interactive techniques - GRAPH 03*, 2003.

[39] P. Punpongsanon, D. Iwai, and K. Sato. Deforme. *SIGGRAPH Asia 2013 Emerging Technologies on - SA 13*, 2013.

[40] A. Rosenfeld and J. L. Pfaltz. Sequential operations in digital picture processing. *J. ACM*, 13(4):471–494, Oct. 1966.

[41] J. Salvi, J. Pagès, and J. Batlle. Pattern codification strategies in structured light systems. *Pattern Recognition*, 37(4):827 – 849, 2004. Agent Based Computer Vision.

[42] V. Scholz, T. Stich, M. Keckeisen, M. Wacker, and M. Magnor. Garment motion capture using color-coded patterns. *Computer Graphics Forum*, 24(3):439–447, 2005.

[43] J. Sell and P. O'Connor. The xbox one system on a chip and kinect sensor. *IEEE Micro*, 34(2):44–53, 2014.

[44] C. Solomon and T. Breckon. *Fundamentals of Digital Image Processing: A Practical Approach with Examples in Matlab*. Wiley Publishing, 1st edition, 2011.

[45] J. Steimle, A. Jordt, and P. Maes. Flexpad. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI 13*, 2013.

[46] A. Stevenson, C. Perez, and R. Vertegaal. An inflatable hemispherical multi-touch display. In *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction*, pages 289–292. ACM, 2011.

[47] J. Stuhmer, S. Nowozin, A. Fitzgibbon, R. Szeliski, T. Perry, S. Acharya, D. Cremers, and J. Shotton. Model-based tracking at 300hz using raw time-of-flight observations. In *ICCV 2015 - International Conference on Computer Vision*. IEEE – Institute of Electrical and Electronics Engineers, October 2015.

[48] F. Taher, J. Hardy, A. Karnik, C. Weichel, Y. Jansen, K. Hornbæk, and J. Alexander. Exploring interactions with physically dynamic bar charts. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI 15*, 2015.

[49] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 205–214, New York, NY, USA, 1987. ACM.

[50] G. M. Troiano, E. W. Pedersen, and K. Hornbæk. User-defined gestures for elastic, deformable displays. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*, AVI '14, pages 1–8, New York, NY, USA, 2014. ACM.

[51] H. Uchiyama and E. Marchand. Deformable random dot markers. *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, 2011.

[52] F. Vogt, T. Chen, R. Hoskinson, and S. Fels. A malleable surface touch interface. *ACM SIGGRAPH 2004 Sketches on - SIGGRAPH 04*, 2004.

[53] H. Wang, J. F. O'Brien, and R. Ramamoorthi. Data-driven elastic models for cloth: Modeling and measurement. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 71:1–71:12, New York, NY, USA, 2011. ACM.

[54] Y. Watanabe, A. Cassinelli, T. Komuro, and M. Ishikawa. The deformable workspace: A membrane between real and virtual space. *2008 3rd IEEE International Workshop on Horizontal Interactive Human Computer Systems*, 2008.

[55] J. Weil. The synthesis of cloth objects. *SIGGRAPH Comput. Graph.*, 20(4):49–54, Aug. 1986.

[56] J. O. Wobbrock, M. R. Morris, and A. D. Wilson. User-defined gestures for surface computing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1083–1092. ACM, 2009.

[57] S. Yamazaki, M. Mochimaru, and T. Kanade. Simultaneous self-calibration of a projector and a camera using structured light. pages 60 – 67, 07 2011.

[58] K. Yun, J. Song, K. Youn, S. Cho, and H. Bang. Elascreen: exploring multi-dimensional data using elastic screen. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 1311–1316. ACM, 2013.

[59] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22(11):1330–1334, 2000.