

Master's Thesis

Correct Projection Mapping on - and Interaction
Techniques with - Cloth-based Deformable
Surfaces

Academic Supervisors: Kasper Hornbaek and Esben Pedersen

July 28, 2014

Acknowledgements

...

Abstract

In recent years, natural user interfaces have become more and more common. These interfaces are very easy to learn by new users, since they rely on gestures or actions which are already known, unlike physical input devices which have a learning curve and also affect the user's experience, since they have to rely on external, artificial input methods. One of the important developments is the growing research around deformable displays. These displays allow the users to interact with an application as they would do in the real world, by directly manipulating the virtual world with their hands. These types of interfaces build upon the touch paradigm, adding an additional layer of flexibility, allowing gestures to be performed in a 3D space. Unlike mid-air methods, these deformable displays also provide haptic feedback while interacting, which can be useful for certain applications.

One of the main problems with deformable displays is that the image contents become distorted when deforming the surface - the image will follow the shape of the display. For many applications this is not desired: for example, if we want to move an object by pushing and dragging it around, the image should appear as on a flat screen - otherwise the object will also move around and become skewed, due to the deformation induced by the display's shape.

While research has been done in this area, to my knowledge, the problem is still not completely solved. Although the issue was addressed, for example, in *The Deformable Workspace*, they rely on a custom image processor. Further, they only show some images and videos which compare the correction with a normal image, but they have not done a study to determine how users perceive the compensation. In this project, the aim was to build a distortion correction prototype using off the shelf components. Moreover, an empirical study has been done to determine if participants thought that the compensation is working and if they prefer it over a regular method of displaying the image. Originally, the plan was to also create some dynamic applications on which users can test different interaction techniques. However, the distortion problem was more complex than originally thought and, due to several issues, no time was left to develop this part.

Unfortunately, the system has a number of limitations. The main issue is that it has not been calibrated, but manually aligned, which means that the compensation is offset due to misalignments of the physical setup. Several projector-camera calibration techniques were tried, but a workable solution was not employed in the allotted time. Moreover, even though two methods of noise reduction were applied(spatial and temporal averaging), dynamic artifacts were still present in the final image, albeit less pronounced. Finally, another issue is that there is some observable latency from the time of interaction, until the image compensates. Although the image has been updated at around 30 frames per second(**fps**) with the noise reduction techniques applied, and around 60 fps without.

These issues all constitute candidates for future work. Finally, the compensation only works well from one viewpoint. In order to fix this issue, the user must be tracked and the image has to be rotated depending on the viewpoint.

Although the prototype has several important limitations, the experiment results indicate both that the compensation is working(to some extent) and that the compensated method was almost universally preferred. For this setup, the results convey that the compensation worked best in the center of the image, because other distortions were introduced on the sides, due to the compensation offset. This issue can be fixed by calibrating the system.

Contents

1	Introduction	1
1.1	Problem Description	1
1.2	Hardware Setup	1
2	Related Work	2
2.1	Tracking and Projection on Deformable Surfaces	2
2.2	Interaction Techniques with Deformable Surfaces	8
3	Prototype Design	9
4	Prototype Development	11
4.1	Calibration of the Projector and the Depth Camera	12
4.1.1	Approaches to Calibration	14
4.2	Creating a Point Cloud from the Depth Data	16
4.2.1	Rendering	17
4.2.2	Limitations	18
4.2.3	Unprojecting to World Coordinates	19
4.3	Object Reconstruction by Mesh Triangulation	19
4.3.1	Connect a Grid of Points with a Triangle Strip	21
4.3.2	Other Methods for 2.5D Triangulation	22
4.3.3	Marching Cubes on GPU	22
4.3.4	Greedy Triangulation	26
4.3.5	Kinect Fusion and Kinfu	28
4.4	Projective Texture Mapping	28
4.4.1	Object Linear Texgen	29
4.4.2	Eye Linear Texgen	29
4.5	Capturing Dynamic Contents	29
5	Prototype Implementation	30
5.1	Tools and Technologies Used	30
5.2	Virtual World Setup	31
5.3	Point Cloud Generation	32
5.4	Mesh Rendering	33
5.4.1	Rendering Process	33
5.4.2	Triangulation	35
5.5	Depth Smoothing	37
5.5.1	Spatial Averaging	38
5.5.2	Temporal Averaging	39
5.5.3	Results	40
5.6	Projective Texture Mapping	41
5.6.1	Setting Uniforms and Computing the Texture Coordinates in the Vertex Shader	41
5.6.2	Applying the Texture in the Fragment Shader	43
5.6.3	Results	44

5.7	Manual System Alignment (Calibration Not Used)	44
5.7.1	Physical Projector Alignment to the Physical Screen(Cloth area)	45
5.7.2	Physical Depth Camera Alignment to the Physical Screen (Cloth area)	47
5.7.3	Virtual projector alignment to the cloth area	49
5.7.4	Virtual camera alignment to the cloth area	51
5.7.5	Testing precision of alignments	51
6	Empirical Study	54
6.1	Experiment Description	54
6.1.1	Grid, Text and Columns Scenarios	57
6.1.2	Map Scenario	58
6.1.3	CT Brain Scan Scenario	58
6.1.4	The Questionnaire	60
7	Results	62
7.1	Experimental Study Results	62
7.1.1	Quantitative Analysis	63
7.1.2	Qualitative Analysis	73
7.2	Comparison of Compensated and Uncompensated Images	76
8	Conclusions	77
9	Future Work	78
A	Glossary	82
B	Additional Images	83
C	Empirical Study Documents	84

List of Figures

1	Physical setup of the prototype	2
2	Projection on Holger the Dane. Courtesy of [14]	3
3	Interaction with Flexpad. Courtesy of [42]	4
4	Deformation model of Flexpad(left) and deformations it can express(right). Courtesy of [42]	4
5	The Future Office. Courtesy of [23]	6
6	The Deformable Workspace Setup. Courtesy of [28]	7
7	Compensation of image warp caused by screen deformation. Courtesy of [28]	9
8	a) Only a portion of the object can be magnified with a small fisheye lens; b) A large fisheye magnifies almost the entire object, but at the cost of added distortion for surrounding objects; c) JellyLens - adapting the relevant information of the object, while preserving the surrounding areas. Courtesy of [21]	10
9	Distortion Correction Pipeline	11
10	A setup that needs projector-camera calibration	13
11	The projection frustum. Courtesy of [24]	14
12	A 4x4 matrix that represents a position(column 4) and orientation(columns 1-3) in space. Courtesy of [24]	15
13	General Pinhole Camera Model	16
14	Vertex Transformation Pipeline. Courtesy of [24]	17
15	3D Graphics Pipeline	17
16	An example point cloud, corresponding to the depth stream, updated in real-time.	18
17	Perspective View of Some Terrain. Courtesy of [18, Chapter 9] . .	20
18	Polyhedral(solid object with flat surfaces) Terrain. Smoothness will depend on point density. Courtesy of [18, Chapter 9] . .	20
19	Ordering of vertices in a triangle strip for a grid. Courtesy of [15].	21
20	a)Mountain ridge; b) Narrow valley. Courtesy of [18, Chapter 9]	22
21	Voxel data representation. Courtesy of [38]	23
22	Marching Cubes - The 14 Patterns of Triangulated Cubes. Courtesy of [35]	24
23	Marching Cubes - Cube Numbering. Courtesy of [35]	25
24	Mesh from point cloud reconstruction of a bunny.	27
25	Virtual Projector Transformation Pipeline. Courtesy of [30] . . .	28
26	Projective Texture Coordinates from a 4x4 matrix and the vertices in object or eye space. Courtesy of [30]	29
27	Object Linear Texgen. Courtesy of [30]	29
28	Eye Linear Texgen. Courtesy of [30]	30
29	Vertex array layout with two vertex attributes. Courtesy of [33] .	33
30	Transfers of vertex array object data using DMA.	34
31	A grid texture which shows the noise reduction obtained by averaging.	40
32	Reverse projection after applying PTM. Courtesy of [30]	44

33	PTM applied on a Suzanne monkey head from Blender, and on a floor with a small cube. The frustum represents the virtual projector.	45
34	A grid with four corners and a red border, used to test the physical alignment of the projector to the cloth.	46
35	The alignment grid texture projected on the cloth.	46
36	Illustration of alignments in the four corners of the frame.	47
37	The depth camera is setup on a tripod. By reversing the column we can get the center of the depth camera to roughly correspond to the center of the projected image.	48
38	A comparison which shows that pushing in the center roughly modifies the same area of both the depth stream and physical image.	48
39	The frame placement. Distance is displayed in red.	48
40	Points for which depth had been extracted and compared.	49
41	Texture Aligned to Cloth Surface. The area inside the red rectangle shows an offset.	50
42	The area has been magnified. An arrow points towards the line that separates the frame from the cloth.	50
43	Correspondence between physical and virtual images.	52
44	Correspondence between physical and virtual images.	53
45	The input device for switching between the methods.	55
46	A 5×5 latin square that denotes the order in which scenarios were shown to the user.	56
47	Grid, Text and Column Textures. Red dots indicate interaction areas.	57
48	A map showing a region of New York City. Courtesy of Google Maps.	58
49	CT scan of a brain. Example of two different textures(slices). Courtesy of [34]	59
50	Mean of preservation of appearance, per scenario.	65
51	Mean of preservation of appearance, overall.	67
52	An area chart showing the perceived overall preservation of appearance, for both methods and all participants.	68
53	Consistency mean, per scenario.	69
54	Means of method preferences, per scenario and overall.	71
55	An area chart showing the perceived overall method preference.	72
56	Mean of perceived differences, per scenario.	73
57	Results for the center point.	74
58	Results for the left-most point.	75
59	Results for the right-most point.	76
60	Qualitative results for the overall case, grouped in a few categories.	77

List of Tables

1	Frame Rate Comparison When Using Noise Reduction Techniques	41
2	Characteristics of pair data for preservation of appearance.	63
3	T-test results for preservation of appearance.	64
4	Characteristics of pair data for overall preservation of appearance.	65
5	T-test results for overall preservation of appearance.	66
6	Characteristics of pair data for consistency.	67
7	T-test results for consistency.	68
8	Characteristics of pair data for method preference.	70
9	T-test results for method preference.	70

1 Introduction

Deformable displays allow new methods of interaction with virtual scenes, which simply are not possible on flat displays. These interactions are more immersive and feel more natural to the user as they touch, push, pull or pinch the deformable surface. The interaction with the scene can now be done in a 3D space, similarly to how we interact with objects on a daily basis, while also providing a degree of haptic feedback.

The first part of the master thesis project deals with the development of a deformable display prototype on which undistorted visuals can be projected. The distortion of visuals due to irregularities in the surface of an arbitrary-shaped object (the display surface) is the main problem that must be solved.

1.1 Problem Description

One of the main challenges when dealing with deformable surfaces, is correcting the distortions which arise from the user's interaction with the display. The image must be changed(the pixels must be shifted) so that the contents still appear as on a flat surface, even when the surface becomes an arbitrary 2.5D object.

If, for example, we are sculpting into or pulling out of a piece of cloth on which visuals are projected, some degree of distortion will appear on the final image. The image should be undistorted in the sense that, the visuals would still look like a fixed display (rather than being deformed with the display), after the previously mentioned pull or pinch gesture is performed. The solution will involve tracking of the cloth material and the modeling of a 3D representation of the material that is deformed based on depth sensor data.

In order to solve the problem we need to first setup a scenario where one can test how images are deformed when interacting with deformable displays. In order to do this we need: a surface that can be deformed by a user(e.g.: a cloth material) which serves as our display, a projector that is used to display contents to the end user and a depth camera that is used to track the display and surface and detect changes.

1.2 Hardware Setup

The deformable display prototype(see figure 1) consists of the following components:

- Cloth display (that can be deformed) + frame
- Projector that is used to display the visuals onto the cloth surface
- Depth camera that is used to retrieve the cloth displays depth information

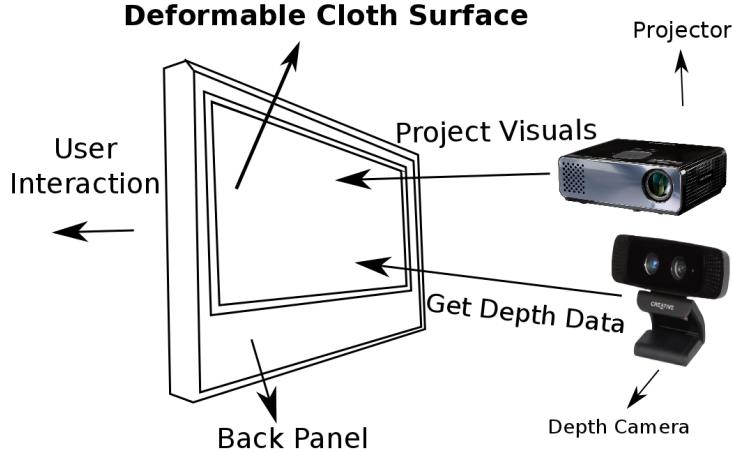


Figure 1: Physical setup of the prototype

2 Related Work

In this chapter, relevant research and related works are highlighted, mainly in relation to deformable user interfaces, tracking and projection techniques on deformable surfaces, interaction techniques with deformable displays, 3D projection and the design of affordances in order to convey information or suggest particular interaction methods. Investigating related research will help in finding solutions to projection issues with deformable surfaces. Moreover, this will also be useful in order to analyze, compare and contrast different approaches to tracking as well as correcting the projection distortion. Further, existing research on interaction techniques and affordances with deformable surfaces will provide a starting point to test, design and implement relevant interaction scenarios for the prototype. Finally, work related to stereoscopic and autostereoscopic projections will be consulted, in order to arrive at a sensible projection technique that can be used in the present case.

2.1 Tracking and Projection on Deformable Surfaces

In this section, literature related to projections on deformable surfaces as well as possible distortion correction and tracking techniques techniques will be presented.

As a starting point, it makes sense to consider existing approaches of projection onto complex or non-planar physical shapes. As discussed in [14], calibration is needed in order to correct distortions arising from a difference between the digital and the physical object. The study deals with projection on several

physical objects, including a complex, organic, non-planar shape, **Holger the Dane**, as shown in figure 2.



Figure 2: Projection on Holger the Dane. Courtesy of [14]

Further, it also proposes a few design themes for 3D projection on physical objects:

- New potentials for well-known 3D effects (lightning, particle systems, shadows, sound etc).
- Dynamics between the digital and physical world and switching between 2D and 3D projections allow to focus the projection on hotspots, while customizing properties allows outlines and switching material textures.
- Relations between object, content and context.

Although the information about the necessity of device calibration as well as the projection on a complex surface provide some insights into the projection distortion issue at hand, the study was only concerned with projection on static objects and thus, does not address problems arising from dynamic surface deformation, where the projection would need to be adapted in real-time according to the deformable surface's depth information.

FlexPad(figure 3) is a real-time interactive system with a highly deformable hand-held paper display. A projector is used to display the visuals, while tracking the display surface with a depth camera provides the required depth information for detecting changes in the surface. Further, as described in [42], any 2D image from Microsoft Windows can be projected and correctly warped unto the display.

Moreover, a deformation model(figure 4) is described, in their case using a 25×25 vertex plane at the size of the surface. The model can be represented by a 15 dimensional vector consisting of the angles of 8 basic deformations, a z

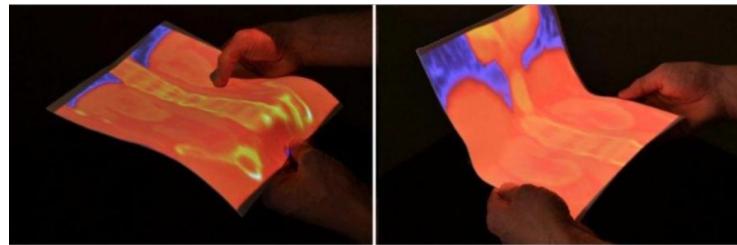


Figure 3: Interaction with Flexpad. Courtesy of [42]

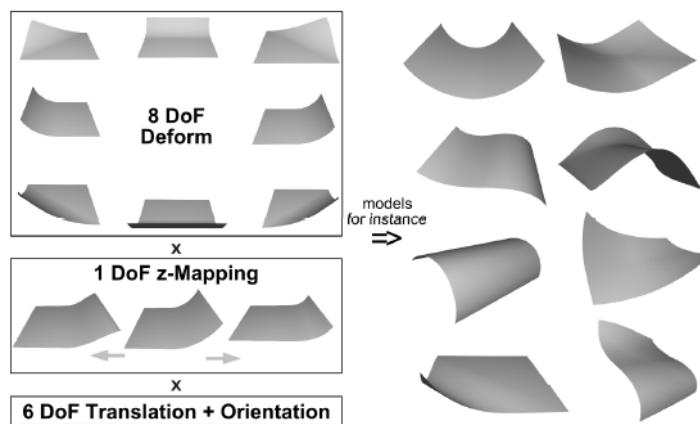


Figure 4: Deformation model of Flexpad(left) and deformations it can express(right). Courtesy of [42]

mapping parameter as well as 6 variables for DOF (degrees of freedom) required for affine 3D transformations.

After obtaining a deformation model, the tracking problem can be defined as finding parameters of the model such that, when synthesized as a depth image, they match the input depth image best(Analysis by Synthesis or AbS). This represents an optimization problem of finding the best parameter vector. The CMA-ES optimization scheme is used because it is a global optimization algorithm and it applies a smart distribution scheme, minimizing function evaluations needed to find an optimum. More information about these tracking methods will be presented below.

The primary limitations of the FlexPad come from those of the Kinect depth camera not being able to track very sharp bends, however these limitations could be overcome by installing several depth sensors or a more powerful sensor. The hand occlusion issue is not particularly relevant to the current project, since the depth camera will be installed in the back of the display, while interaction shall be from the front - thus completely removing this problem.

As discussed in [32], the deformation model can use NURBS surfaces, which can be easily handled via 3D control points, while being able to approximate every surface with arbitrary accuracy. These are widely used in computer graphics and can reduce the high dimensional space of all possible triangle mesh deformations.

The first step is to create an arbitrary NURBS surface using control points and knot vectors, using the algorithm discussed in [32]. Next, once we have a triangle mesh(which can be supplied from a modeling program or computed via OpenGL), we can approximate the mesh with a NURBS surface. The NURBS functions must be fitted to the mesh by minimizing the distance between the surface face function and the vertices of the mesh, or, in other words, finding control points for a given set of vertices, that minimize the sum over all squared distances. Mesh registration can then be performed by associating each 3D vertex coordinates to the previously generated NURBS surface. After this step, the mesh can be translated, rotated and deformed by displacing the control points of the NURBS surface function. Now we can apply an AbS method with a CMA-ES optimization scheme.

DeforMe is a projection-based mixed reality (MR) technique, which provides a method for augmenting deformable surfaces with deformation rendering graphics, as described in [4]. Projected graphics are realistically deformed to match the deformed surfaces. Mixed reality creates the illusion of a virtual modification of the physical surfaces. In this study, a flexible surface is used to allow the user to interact with the projected graphics while obtaining tactile feedback. Further, they present some drawbacks of existing tracking methods such as measuring surface deformation via a depth camera, which is not applicable to tangential deformations that can appear on a surface in a horizontal direction during interactions (e.g.: touching, pulling, etc.). To summarize, DeforMe is a projection-based MR system which allows projected graphics to be deformed to the deformation of the real object, while performing real-time es-

timations of the tangential deformation flows with a small number of feature points. Various deformable materials can be used without prior knowledge of their mechanical properties or deformation models. However, since depth information can be measured for a cloth-based deformable display (with the material properties being known), it is unclear if there is an immediate advantage of this method, for the current project, over the tracking techniques employed in the FlexPad system.

Another project that provides useful information on projecting onto irregular 3D surfaces, as well as tracking them using depth data, is **The Office of the Future**(see [23]). One of their objectives is to use real surfaces in the office, as spatially immersive display surfaces. High-resolution graphics is projected onto these surfaces(see figure 5).



Figure 5: The Future Office. Courtesy of [23]

They extract depth using imperceptible structured light (projecting binary-coded patterns that are not detected by the visible eye and recording the images with video cameras) - this process can be replaced in the current project by the use of an off-the-shelf depth camera. Further, they describe how, by knowing the object's surface, the viewer's location, projector calibration parameters and the contents to be projected, they can render on irregular 3D surfaces by the use of conventional 3-D methods(e.g.: projective texture mapping), in real-time. Finally, they state that their system can be used to detect display surface changes at non-interactive rates. However, due to a large increase in computing power in recent years, and the possibility to write graphics programs that run directly on the hardware(e.g.: shader programs), similar techniques can be used today, to detect display surface changes in real-time.

The Deformable Workspace(see figure 6) is another approach to creating a deformable screen that acts as a boundary surface between the real and virtual

worlds and is presented in [5]. The workspace is set up in similar ways to other techniques by using an IR camera, IR projector and an LCD projector. Basically, the IR camera and projector could be replaced by a depth sensor such as the Kinect.

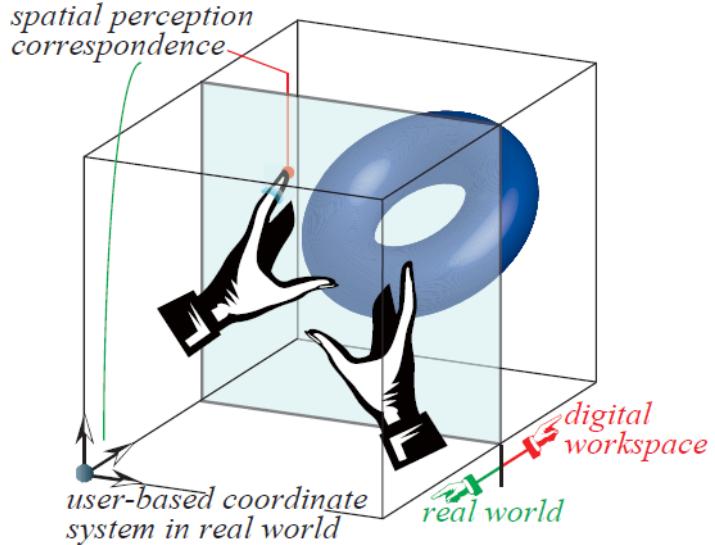


Figure 6: The Deformable Workspace Setup. Courtesy of [28]

Tracking is based on triangulation using structured light. A single snapshot of the illuminated surfaces is sufficient to grasp complete screen deformation - a high-speed camera acquires the image, and a dedicated co-processor for high-speed image processing calculates the coordinates of a 3D point for each spot of light in the pattern (real-time performance at 955fps with a latency of 4.5ms), as described in [27]. Although the frame rate achieved is incredibly high, this is a custom solution, that requires separate hardware. The previous method shall not be employed for this project as the deformable display can now be tracked with a standard depth camera(either a structured light or time-of-flight depth camera).

In [5] they also provide a method to compensate for projection distortion, given that both the calibration parameters and the shape of the deformed screen are known. An observation model in human vision is required to solve this issue, in this case, perspective projection. The technique involves pre-warping the image by shifting each projected point. Further, they claim that projective texture mapping can be used to calculate and render the warped image to be projected in real-time, efficiently and easily. A limitation of the previous approach is that an assumption has been made that the users eyes are fixed. However, a viewpoint-based projection should be employed to accommodate realistic scenarios - the virtual space must be rotated and transformed according

to the change of viewpoint.

As a final note, we can observe that the techniques used to project onto irregular objects and of correcting the visuals are similar in [28] and [23]: both extract depth information and form a polygonal model of the target object, they use the same rendering technique(projective texture mapping) and require the same calibration parameters.

2.2 Interaction Techniques with Deformable Surfaces

3 Prototype Design

Both the projector and the depth camera will be fixed in place relative to the deformable screen. Further, the projected visuals will be interactive applications that need to be rendered continuously. Finally, due to the fact that the projected image will be warped (distorted) according to the screen deformation, an algorithm is needed to shift the projected image pixels corresponding to the display surfaces depth data.

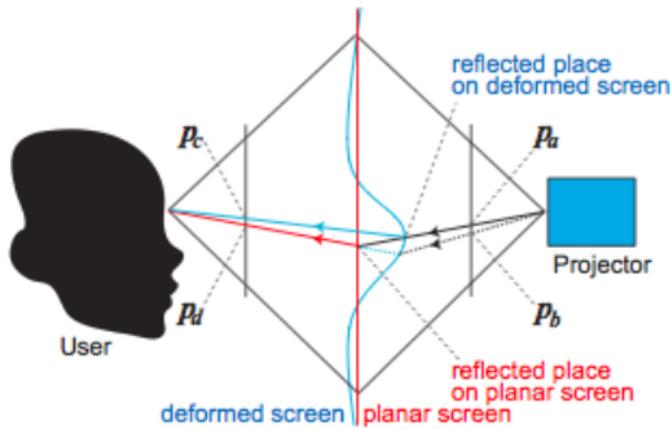


Figure 7: Compensation of image warp caused by screen deformation. Courtesy of [28]

In figure 7, p_d is the position where the ray is observed for a planar screen, while p_c is that of the deformed screen, if the projection is from p_a . To observe p_d for the deformed screen, projection must be from point p_b . Consequently, the image point needs to be translated to this position, in order to compensate for the distortion.

A possible approach to solving the distortion issue is discussed in [28], where the authors state that in order to compensate for the distortions of the display one needs to know the shape of the deformed screen (depth data) and the projector calibration parameters.

Perhaps another solution would be to use some kind of spatial distortion, similar to the approach used for the **JellyLens** (see [21]). The focus + context lens (uses two levels of detail to connect a magnified region to the selected context) can dynamically adapt to the shape of the objects of interest. As we can see in figure 8, the object of interest adapts its shape (is magnified), while the context is preserved - surrounding objects are almost untouched by the applied distortion.



Figure 8: a) Only a portion of the object can be magnified with a small fisheye lens; b) A large fisheye magnifies almost the entire object, but at the cost of added distortion for surrounding objects; c) JellyLens - adapting the relevant information of the object, while preserving the surrounding areas. Courtesy of [21]

The JellyLens can adapt to match the geometry in a three-step process:

- Obtaining information about the geometry of the objects (in our case, this can be accomplished by transforming our object depth data to 3D geometry)
- Computing the lens shape according to its position and visualization and to the geometry of the nearby objects of interest (in our case, the depth data can be used to determine how to shape the lens)
- Rendering of the region seen through the lens (in our case, we can apply a normal texture on the deformable mesh's geometry - that represents the contents, and distort it according to the lens shape obtained previously)

The authors of [21] also describe in detail how to model and adapt the lens using two techniques known as *AreaLens* and *PathLens*. For more detail about the process, please consult this reference.

For this project, a solution similar to that employed in [28] has been implemented. In order to get a better overview and understanding of the whole process required to correct the distortions, I have created a system pipeline(see figure 9) that describes the whole process, at a high-level. Each part will be described individually in subsequent chapters.

Deformable Display Distortion Correction Pipeline

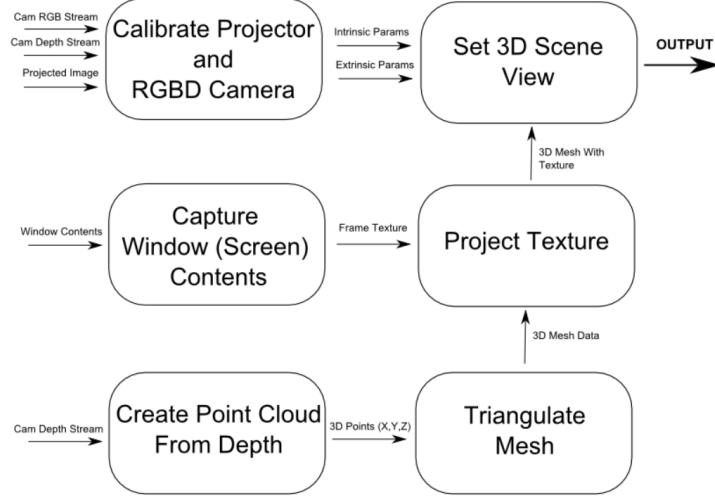


Figure 9: Distortion Correction Pipeline

4 Prototype Development

Considering our system pipeline, I will explain briefly, how the whole components work and why they are needed, in this section. The first process is calibration. Note that the order in which components are explained does not represent the flow in the pipeline, since it will be easier to see, for example, why calibration is needed once some of the other steps have been explained.

The idea followed in [28] consists of displaying the final application contents on a 3D model representation of the deformable surface. In order to obtain a relatively stable model, one can convert the depth data into a 3D mesh. An intermediary step required in order to do so, is to first create a point cloud from the depth data, or in other words, to use the depth value as the z-coordinate and create a cloud of points with coordinates (X, Y, Z) corresponding to the image pixels and depth.

Further, the application contents must be displayed in such a way that they distort with the object. Like with most problems in computer graphics, there is more than one solution. One could, for example, use raytracing to directly project rays of light on the object(also mentioned by the authors of [23]. However, this approach is known to be slow and may not run in real-time. Alternatively, there exists a technique which has been tried and tested before, known in computer graphics, which projects a texture onto an object as from a slide projector. This technique can also be implemented in real-time on modern

hardware, and, since the texture is projected from a virtual projector in the 3D scene, it will update with the distortions corresponding to the deformation on the object.

Finally, we need to make sure that our virtual camera corresponds, as closely as possible to the area we project onto with our physical display. Further, we also need to place our virtual projector, which also requires information about where the mesh(3D model) is placed, and how it is oriented. In addition to position and orientation we must also know the field of view(FOV), or the "*zoom level*" of our camera(as the other parameters depend on the FOV). In addition to the calibration required above, one also needs to manually(physically) align the projector so that it correctly fills the deformable display area. Furthermore, in case of manual calibration, the depth camera must also be placed as orthogonal as possible to the deformable surface area, so that it is easy to obtain acceptable parameters for the virtual camera and projector described above.

At this point, we basically have a solution to the distortion correction problem. The only remaining issue is to update the application contents dynamically, by capturing window/screen contents. This can be accomplished by using textures.

In the next sections, each of the above will be described in greater detail.

4.1 Calibration of the Projector and the Depth Camera

In order to properly display the contents on the deformable surface, three things must be ensured:

- The projector must be calibrated in order to find the intrinsic and extrinsic parameters that will be used to compute a transformation matrix that defines the world view of the 3D scene
- The physical projector must be aligned with the deformable screen area. Further, the depth stream must be adjusted manually so that the deformable display is almost perfectly in view, if the calibration is done manually, by trial and error;
- The projector and camera are static and must not be moved, otherwise the calibration process has to be repeated

The calibration problem is a projector-camera calibration problem, similar to the illustration shown in figure 10. The only difference being that, for this project, an RGBD camera(RGB stream + depth stream) is used. This means that there are two cameras: a normal RGB video camera, and a depth camera. If the calibration is done between the RGB camera and the projector, the RGB camera must then be correctly mapped to the depth camera or vice-versa.

In order to properly align the physical projector/camera with the virtual projector/camera we need to compute two matrices in a virtual world, based on

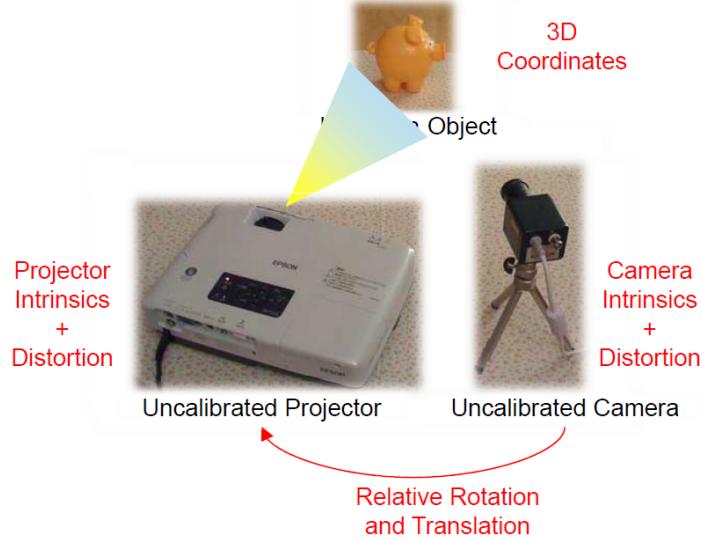


Figure 10: A setup that needs projector-camera calibration

parameters extracted from the calibration procedure. The first is the projection matrix, which can be represented as a frustum(see figure 11). We can define the projection matrix in OpenGL either by providing all the frustum parameters(the near and far parameters can be entered manually), or a field of view and aspect ratio, from which they can be computed.

It essentially defines the viewing volume of our 3D scene. In order to compute this matrix, one needs to know the intrinsic parameters of the physical projector. **Intrinsic parameters** refer to different aspects of a camera/projector such as focal length, scale parameters, a tilt compensation parameter and the image coordinates which define the image center or principal point. In addition, other parameters can be included to account for lens distortion. Lens distortion will not be addressed in the current project. The images projected with a modern device have low distortion.

Secondly, we need to compute the modelview matrix(see figure 12, for the structure of such a matrix). This matrix has information that relates to the position of the model(3D object space) as well as the camera(eye space). In this case, we essentially need to compute the view matrix from the extrinsic parameters, so that our virtual camera correctly points to the deformable display area on which our target contents are projected. **Extrinsic parameters** represent the **location** and the **orientation** of the projector/camera with respect to a world reference frame. For example, if we choose the world origin to be the position of the depth camera, then we can compute the rotation and translation

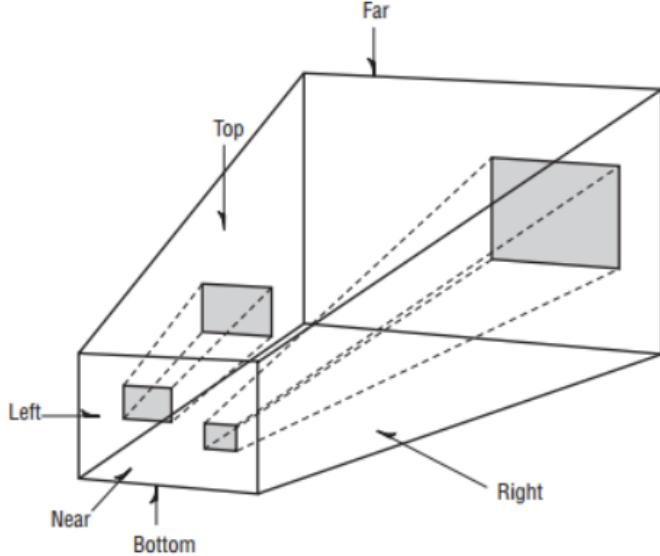


Figure 11: The projection frustum. Courtesy of [24]

of the projector with respect to the depth camera by finding the rotation matrix R and translation coefficients T . To obtain camera coordinates from world coordinates we simply need to multiply a position by R and add the translation T .

The pinhole camera model can be used to represent the projector, only the direction of the light rays are opposite, so the projector is in fact, an inverse camera model. The general pinhole camera model, where the center of projection does NOT have to be the world coordinate system is shown in figure 13.

4.1.1 Approaches to Calibration

- Structured Light Illumination Patterns

The correspondences between 2D pixels and 3D world coordinates are found in both [22] and [29] by employing the structured lighting approach which involves projecting gray coded patterns that represent projector rows/columns onto the surface. This approach requires a camera to capture the projected patterns.

An advantage of this approach is that it does not require any other physical items (such as printed checkerboard patterns), neither moving objects around, for calibration.

$$\begin{bmatrix}
 X_x & Y_x & Z_x & T_x \\
 X_y & Y_y & Z_y & T_y \\
 X_z & Y_z & Z_z & T_z \\
 0 & 0 & 0 & 1
 \end{bmatrix}$$

↓ X axis direction ↓ Y axis direction ↓ Z axis direction ↓ Translation/location

Figure 12: A 4x4 matrix that represents a position(column 4) and orientation(columns 1-3) in space. Courtesy of [24]

- Checkerboard Pattern

Another approach for calibrating a projector-camera system is to use printed checkerboard patterns, extending from Zhang's method, as discussed in Chapter 3 of [16]. The known checkerboard patterns are projected onto a diffuse rigid object and their distorted appearance is photographed. Since a projector is the inverse of a camera, points on the image plane are mapped to outgoing light rays that pass through the center of projection. About 10-20 images with different positions and pose should be recorded.

In [19] a plane-based calibration of a projector-camera system is described. It can be used to obtain the intrinsic and extrinsic parameters of both the camera and the projector. The steps involved are:

- Calibrate the camera using Zhang's method
- Recover calibration plane in camera coordinate system
- Project a checkerboard on calibration board and detect corners
- Apply ray-plane intersection to recover 3D position for each projected corner
- Calibrate the projector using the correspondences between the 2D points of the image that is projected and the 3D projected points

The procamlib (<https://code.google.com/p/procamcalib/>) software package for Matlab can be used to calibrate the projector-camera system using this approach. The steps are explained in [19].

- Manual Adjustment

It is also possible to adjust the physical projector and camera so that one can deduce the extrinsic parameters. The intrinsic parameters may also be obtained by looking into the respective product's datasheet and adjusting the value depending on distance, lens size etc. Otherwise, one must use a trial and error approach, and some tests need to be done to confirm that the accuracy is acceptable. Manual adjustment is likely to produce results of inferior quality, compared to the previous two methods.

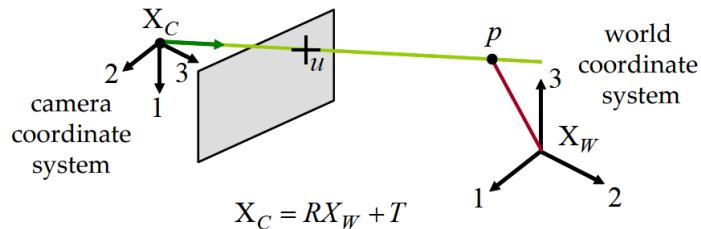


Figure 13: General Pinhole Camera Model

4.2 Creating a Point Cloud from the Depth Data

This step is an intermediary one towards creating a mesh structure on which textures can be projected. A 3D point cloud is a set of data points in the third dimension, usually defined by (X, Y, Z) coordinates.

The depth camera retrieves an image of 320x240 depth pixels. A point cloud can be created by using x, y values from 0 to the width and height of the depth image, respectively. The z value is simply equal to the depth, if the depth value is in a certain range(after it has been filtered). If the depth value is invalid, the current point is discarded and will not be rendered.

Further, the point cloud must be updated in real-time. Rendering is being done with OpenGL, and the previous can be accomplished by using simple hardware graphics shader programs and OpenGL's Vertex Buffer Objects(VBO). The OpenGL point cloud also has to be rotated, as it is upside down(unless the depth camera is positioned upside down).

In the following, I will include a subsection with information about the general rendering process, which is applied to render 3D objects with the programmable OpenGL pipeline. The process is similar, whether we wish to render a point cloud, or a 3D mesh, only the vertex attributes(position, color, texture

coordinates, normals etc.) which are stored in the VBOs and the programmable shaders will change.

4.2.1 Rendering

The transformation pipeline(see figure 14) describes how the vertex information is transformed from raw data, to window coordinates that can be viewed on a screen. The raw vertices are first multiplied by the modelview matrix, which yields the transformed eye(view) coordinates. Then, we multiply the result by the projection matrix and obtain clip coordinates. All data outside the clipping space(frustum) is removed. Then, coordinates are divided by the w value to yield normalized device coordinates in the $+/-1.0$ range, for all axes. Finally, OpenGL internally maps the scene to a 2D plane by the viewport transformation matrix, based on the viewport parameters provided.

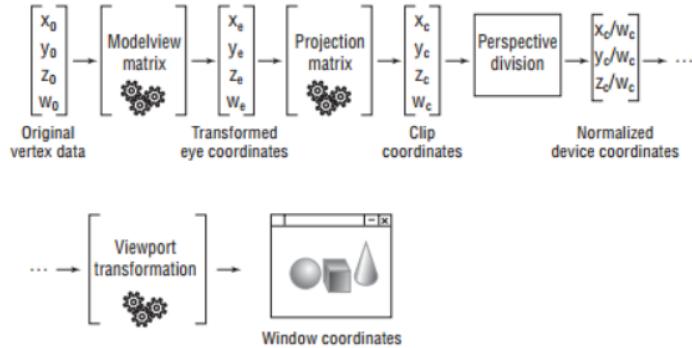
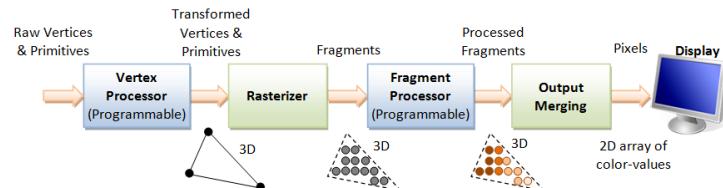


Figure 14: Vertex Transformation Pipeline. Courtesy of [24]

In the following, I will briefly describe the 3D graphics pipeline, shown in figure 15. It explains the stages required to process the vertices in order to obtain colored pixels and how vertex attributes can be programmed in the two shader stages(using the vertex and fragment processors).



3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z) , and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

Figure 15: 3D Graphics Pipeline

As we can see from the pipeline, the raw vertices are processed by a vertex processor which manipulates vertex related attributes, most generally, position. For example, a vertex shader(a program that runs on this processor) can be used to apply the modelview matrix to an input vertex. Next, the data is rasterized into fragments.

A fragment contains the data necessary to generate a single pixel's worth of a drawing primitive in the frame buffer(the frame buffer is a portion of memory that contains a bitmap that will be sent to the video display - it represents a complete frame of data): raster position, depth and, after the fragment shader stage, it also contains interpolated attributes such as color and texture coordinates as well as a stencil value, an alpha value and the window ID. Note that a fragment only corresponds to a pixel if antialiasing is turned off, otherwise there will be K fragments for a pixel for Kx level antialiasing. As we can see from the above, the fragment shader receives fragments and processes them, computing color and texture coordinates. This shader can be programmed to color the geometry in any way desired. One can also apply textures and lighting at this stage. Finally, fragment shaders are also useful for visual debugging, as shader programs can usually not be debugged, since they run on the GPU hardware.

4.2.2 Limitations

Unfortunately, unless the point cloud is of sub-pixel density(which would also require a lot of processing power, due to the very large number of OpenGL primitives), it is easy to see that the target object is in fact made of discrete points(see figure 16). This breaks immersion, especially after applying a texture to the object. Due to this, we must then proceed to the following step, triangulation of the points into a 3D mesh.

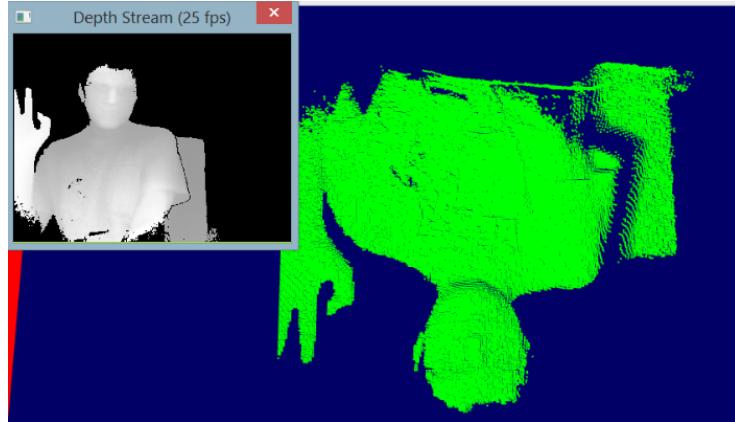


Figure 16: An example point cloud, corresponding to the depth stream, updated in real-time.

4.2.3 Unprojecting to World Coordinates

Before we proceed to triangulation, we must also unproject the points to world coordinates(we just had pixel positions and depth values, but we need to obtain the coordinates in regular metric space, so that the distances between the points are correct). This operation requires knowledge about intrinsic camera parameters, since we need to know the projection matrix, in order to unproject the points:

$$point \Rightarrow Viewport^{-1} \Rightarrow Projection^{-1} \Rightarrow ModelView^{-1} \Rightarrow PointInWorldSpace$$

The matrix should be the identity matrix at this point, since we have not manipulated the points. More information about the unprojecting process is available in [20], although the authors unproject a 2D mesh with a set of sparse depth points. Nevertheless, the authors explain that unprojecting is the inverse process of how normal images are obtained, by projecting rays into a camera. In order to compute the rays, one needs to know the focal length and image center(intrinsic parameters), as well as the distance of the unprojection(ray length, which, in our case, corresponds to the depth for that pixel). They explain that the 3D position for a vertex $v(x, y)$ can be denoted by $p(v)$ and is determined as:

$$p(v) = t(v) * \vec{r}(v),$$

where $t(v)$ represents the ray length and $\vec{r}(v)$ represents the ray direction and is equal to

$$\vec{r}(v) = \left(\frac{x - x_c}{f}, \frac{y - y_c}{f}, 1 \right)^T,$$

where (x_c, y_c) is the principal point or image center and f represents the focal distance.

However, the official library for the depth camera used in this project provides a very useful method that computes the unprojection given the array and number of points. So in the end, that is all we really need to know in order to un project the points, if we are using the *Creative Senz3D* depth camera.

4.3 Object Reconstruction by Mesh Triangulation

Before delving into the different methods of object reconstruction, it should be mentioned that there is an alternative to the full reconstruction approach. In [31], a method of tracking deformable objects using depth and color streams is described. The authors of this paper have used a NURBS(Non-uniform rational b-spline - common in computer graphics; can be used to approximate curved surfaces) based deformation function to decouple the geometrical object complexity from the complexity of the deformation. They deform the mesh using a number of control points(low dimensional space when compared to the number of vertices) on the NURBS surface to which the object is mapped. However, it is

still needed to have an initial mesh and one way to obtain it is to define a 3D object from the initial frame of the video streams. The desired object can first be segmented out of the frame, and then the 3D vertices can be unprojected to 3D space and connected according to their adjacencies in the depth image(actually, in section 4.3.1 a similar method is used, but it is running on the GPU and updates the mesh every frame). This mesh can be subsequently deformed using the control points, so that it matches the depth and/or color data. The mesh is also mapped to the color image to retrieve the colors of the vertices, but this is not needed for the deformable display, since a texture will be projected onto the mesh, completely replacing any color and lighting information.

Coming back to the general topic of object reconstruction, there are several techniques for reconstructing a 3D mesh from a point cloud. An important aspect for our deformable display is that we only need a 2.5D correct mesh, similar to a terrain(see figure 17) or heightmap.

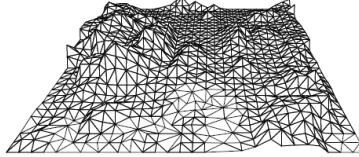


Figure 17: Perspective View of Some Terrain. Courtesy of [18, Chapter 9]

In order to obtain a full 3D model of an object, we would, anyway, need to rotate the depth camera around the object, or use multiple depth cameras, that look at the object from different angles. We always look from the front of the deformable display, as we would, down on a terrain, so such 3D detail is not needed. Therefore, the most straight-forward, naive approach, is to simply find any triangulation for the point cloud. In order to do this we can consider a grid of points, that can be connected either by using triangles or a triangle strip. The main idea of this method is also shown in figure 18, where a terrain is triangulated from some sample points.

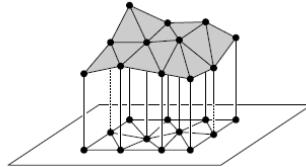


Figure 18: Polyhedral(solid object with flat surfaces) Terrain. Smoothness will depend on point density. Courtesy of [18, Chapter 9]

Although the naive approach has been implemented and used in this project, some more sophisticated alternatives will also be described and contrasted, as this approach was not the first one attempted.

4.3.1 Connect a Grid of Points with a Triangle Strip

Since the depth data captured via a depth camera is ordered, the easiest way is to consider the point cloud as a grid of points. Then, one can connect the vertices together using triangle strips. Triangle strip representation is a method of drawing triangles that can save some of the memory bandwidth, by connecting a new triangle to an existing strip with each new point. This means, that, for example, if we want to draw two triangles, we only use 4 points, instead of 6, which we would normally need. An important thing to keep in mind is triangle winding(the combination of order and direction in which the vertices are specified). By default, OpenGL considers counterclockwise winding to represent front faces, and clockwise winding to represent backfaces. This enables the use of back-face culling for optimisation(we do not need to render triangles which are not visible), and are also important for normal calculation, and thus lighting. Although we are not concerned with lightning in this project, it is always good practice to keep the winding consistent. For more information about triangle strips and other OpenGL methods of drawing primitives please consult [24, Chapter 3].

Coming back to the current problem, an internet article which explains how a triangle strip can be constructed from a grid of points is [15]. Figure 19 provides an overview, on how we can connect the vertices. Note that each point in the figure, numbered from 0-15 is a 3D vertex with coordinates (x, y, z) .

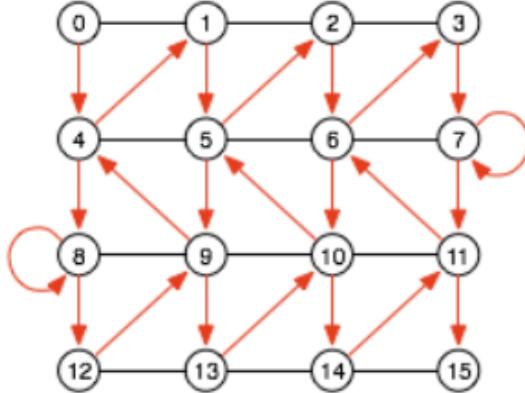


Figure 19: Ordering of vertices in a triangle strip for a grid. Courtesy of [15].

In this example, we can observe that on our path, we alternatively add 4 and subtract 3 for computing the next index, on the first row. On the next row, we still add 4, but subtract 5. The number 4 represents the number of vertices in the row or column, depending on how we advance($n + 1$ vertices, where $n =$ number of divisions). The numbers 3 and 5, correspond to $m = (n + 1) + 1$ or $m = (n + 1) - 1$ and represent changes between adjacent columns. The number alternates, depending on the order of traversal on that particular row.

This approach provides a decent reconstruction, for a 2.5D object using data from a single depth camera. Normals can also be computed after we have the geometry(at an additional computational cost), however, there is no need for lighting in our 3D scene, since we will be applying textures(which represent the desired image contents) on the mesh before rendering. For improving the visual quality of the mesh, one could also look at more sophisticated methods of triangulation such as *Delaunay Triangulation*, or use standard 3D reconstruction techniques, like the *Marching Cubes* algorithm.

4.3.2 Other Methods for 2.5D Triangulation

The naive triangulation, by connecting the triangles without considering some measure of how natural it looks, may not provide the intuitively *best-looking* results. If the triangulation is not acceptable, it can be improved with a technique such as **Delaunay Triangulation**(explained in detail in [18, Chapter 9]), where triangulations are ranked by comparing their smallest angle(small angles, result in *skinniness*). The issue is shown in figure 20 where two triangulations of the same point set are shown. From the heights of the sample points, it seems as though the points were taken from a *mountain ridge*. If an edge is flipped to obtain very small angles for our triangles, we get what looks as a *narrow valley*(as you can see, the height value is now very small). The latter is a suboptimal triangulation and can be improved. Delaunay triangulation aims to find an optimal triangulation, given the fact that, for any set of points, there is a finite amount of possible triangulations.

However, for a dense set of points, such that obtained from the depth data, a naive triangle strip triangulation has provided adequate results - the deformation is clearly visible, both in the mesh and on the texture applied to the mesh. For this reason, Delaunay triangulation has not been attempted, although it may improve visual quality. It remains to be determined, as future work, if the computational cost involved is worth the possible increase in visual quality.

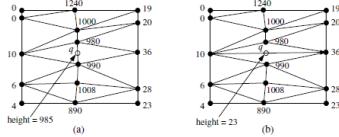


Figure 20: a)Mountain ridge; b) Narrow valley. Courtesy of [18, Chapter 9]

4.3.3 Marching Cubes on GPU

One of the classic computer algorithms for surface reconstruction is the *Marching Cubes* algorithm. An attempt was made to implement this algorithm. In order to run it on a GPU, one can use a *geometry shader* or *histogram pyramids* with a regular vertex and fragment shader. The attempt consisted of porting an existing implementation of the geometry shader for fixed-pipeline OpenGL to

modern OpenGL. Unfortunately, I did not get any geometry as output after running the algorithm, but there were also no errors. The problem is that I could not debug the shaders, since the tools I have tried did not work - some because of no support for modern OpenGL(3.0+) like *glslDevil*, and others did not work on my laptop because it uses the *nVidia Optimus* technology, as is the case for the *nVidia nSight* debugging tool for *Microsoft Visual Studio*. As this was taking too much time, I decided to go for a simpler approach, that should still prove appropriate for the real-time 3D reconstruction of the cloth, described in section 4.3.1. Nevertheless I will briefly describe the algorithm in the following section, as well as how it can be implemented on a GPU. This may be useful to look into, as an extension to the current project, if a higher quality mesh is desired(this method works well for 3D objects in general, and is not restricted to 2.5D, as is the case for the currently used method).

The Marching Cubes Algorithm

It was originally designed to create triangle models from 3D medical data - usually volumetric data, using voxels for representation(voxels are similar to pixels, only they exist in 3D space and can be represented as a cube). The algorithm also needs each vertex of the dataset to have a scalar value, that usually represents a property of the underlying data set. An example of such a voxel set is shown in figure 21. This value will be used to construct an isosurface, by comparing where vertices of the voxel edges intersect the surface(given a constant user-defined threshold or isovalue).

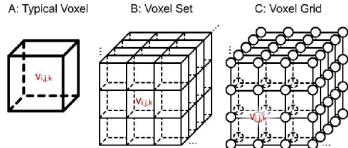


Figure 21: Voxel data representation. Courtesy of [38]

The authors of the original paper on the topic state in [35] that two primary steps were used in their approach:

- Locating the surface corresponding to a user-specified value(isolevel) and creating triangles.
- Calculating normals to the surface at each vertex of each triangle, in order to ensure a high quality image.

Since only the 3D mesh coordinates are required in order to capture the distortion of the cloth, the issue of calculating normals will not be described further, and we shall focus on the first part of the algorithm(although it should be mentioned that a fully shaded, reconstructed model could be useful for some

calibration approaches, since, at the time of this writing there are no reconstruction libraries for the depth camera used in this project).

The algorithm uses a divide-and-conquer approach by using logical cubes of eight pixels to locate the surface. How the surface intersects every such cube needs to be determined. In order to do this, a value of one is assigned to a cube's vertex, if the data value at that vertex exceeds the value of the surface we are constructing(if the scalar value is above a threshold called the isolevel). These vertices are inside the surface. Conversely, a zero value is assigned to values below, and thus outside, the surface. The surface intersects the cube edges where one vertex is inside and the other outside.

Moreover, there are eight vertices in each cube and two states(inside and outside) so we have a total of $2^8 = 256$ cases for intersection with such a cube. However, by eliminating symmetries, like complementary cases and rotational symmetry, the number of distinct cases of triangulation can be reduced to 14 patterns.

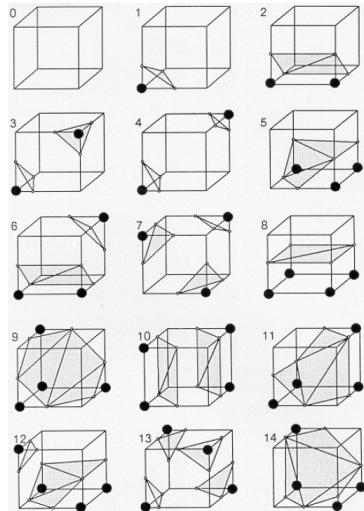


Figure 22: Marching Cubes - The 14 Patterns of Triangulated Cubes. Courtesy of [35]

As an example, the cube for pattern one in figure 22 contains one of the surface vertices(the lower-left vertex of the front face) and results in one triangle defined by three edge intersections. An eight bit index is created for each case, and it corresponds to the cube's vertex states(see figure 23). This index points to an edge tables that retrieves all of the edge intersections for the given cube configuration. The triangles are calculated using linear interpolation of the scalar values along the vertices of the respective edges(to determine the exact location of the triangle vertices on the cube edges). As described in [38], if $P1$ and $P2$ are the vertices of an intersected edge and $V1$ and $V2$ are the scalar

values of each vertex, the intersection point P is calculated as following:

$$P = P_1 + (isovalue - V_1)(P_2 - P_1)/(V_2 - V_1)$$

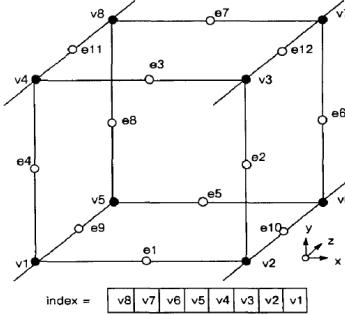


Figure 23: Marching Cubes - Cube Numbering. Courtesy of [35]

GPU Implementation

Geometry Shader

Many aspects from a classical implementation of marching cubes can also be used for a GPU implementation. This is the case with the basic look-up tables used to store edges and triangles. A version of these tables used in several implementations is available in an on-line article(see [12]), where the full source code is also available. Although, there is no version implemented in a programmable shader on the GPU available, an implementation using a geometry shader is described in another on-line article(see [13]). A geometry shader is simply a shader program that can be used to manipulate geometry directly - it allows the GPU to create geometry. This shader takes a single primitive as input(e.g.: a vertex) and outputs zero or more primitives(e.g.: zero or more triangles). This implementation is based on [12], with the main marching cubes logic inside the geometry shader. However, this implementation also uses deprecated OpenGL functionality. The main idea is that data can be stored inside 2D and 3D textures and can then be passed to the geometry shader as uniforms:

```
//Volume data field texture
uniform sampler3D dataFieldTex;
//Edge table texture
uniform isampler2D edgeTableTex;
//Triangles table texture
```

```

uniform isampler2D triTableTex;
//Global iso level
uniform float isolevel;
//Marching cubes vertices decal
uniform vec3 vertDecals[8];

```

In this code fragment the *dataFieldTex* contains the 3D voxel scalar field data, the *edgeTableTex* contains which edges intersect the current cube index, and the *triTableTex* contains all the triangle indices needed, corresponding to each case. Next, the cube index is calculated by comparing the value in the grid(retrieved from *dataFieldTex*) with the *isolevel* uniform. The position is retrieved by adding to the current vertex(to move the vertex at the appropriate position in the cube), the value in *vertDecals*. This adds the required offset in the *x*, *y* or *z* directions by an amount of *cubeStep*, which is defined initially. Having the cube vertex positions and scalar values, the interpolated values can now be calculated, for each edge. The final triangle vertices are retrieved by using the indices in *triTableTex*, for the previously computed interpolated values.

Histogram Pyramids

Another approach is to use histogram pyramids, as described in [17]. The authors state that their algorithm outperforms the known geometry shader approaches and it does not take much more effort to implement. The hierarchical data structured called the *HistoPyramid*, used for data expansion and compaction of 2D textures, is the focus of the previously mentioned paper. The term texel was used to denote single data elements, and the 3D array of voxels has been mapped to a 2D domain.

Implicit Surface Generation

As stated before, in order to use the marching cubes algorithm, one also needs to obtain a scalar value at each point in the data set, and, since we only have a point cloud, a function is needed to provide this - such a dataset is called an implicit surface. These functions were not a focus for this project, as the marching cubes algorithms was not successfully run on datasets that already represented implicit surfaces. However, in [38] the Hermite Radial Basis Function(HRBF) is mentioned as a workable method, please consult this resource for more information on the topic. Another function that can determine the scalar field is the Newtonian gravity field equation, please consult [41] for further information.

4.3.4 Greedy Triangulation

Another triangulation method that was briefly considered was the greedy projection algorithm described in [36]. As described in the previously mentioned source as well as in [38], the algorithm grows the mesh from a list of points, until

all points have been considered or there are no more valid triangles to be formed - in case of the latter, the algorithm is restarted somewhere in the unconnected area with vertices from a new *seed* triangle. The triangulation algorithm is of the greedy type(looks for simple solutions to a complex, multi-step problem by deciding which next step will provide the most obvious benefit) and incremental - it considers the neighborhood of the current point and projects it onto the plane that is tangential to the surface formed by the neighborhood. Next, points are pruned(removed if they do not meet certain criteria) by visibility and then, the remaining points are connected to the current point(and consecutive points) by edges, to form triangles. An implementation of the algorithm was readily available, and as such, the algorithm was not studied in detail, although the two previously mentioned sources can be consulted for more information.

An implementation is available in the PCL library, and a description on how to use it can be found in [40]. The algorithm has been tested on the bunny mesh that is mentioned in the PCL tutorial. The result was a triangulated mesh of the provided point cloud. The algorithm also computes the normals and a shaded model of the object is shown in figure 24. However, this was a static object and it was only comprised of about 400 vertices. When the test was run on the point cloud retrieved by the depth camera, the algorithm was running very slow, and it was clear it could not achieve an interactive framerate - it was taking seconds or even minutes per frame, before giving warnings/errors about the dataset; the size of the point cloud was of about 70000 vertices. Indeed, in [36] it is mentioned that the algorithm is (only) near real-time. Further, a performance analysis is provided, where a point cloud of comparable size was reconstructed in about 9 seconds. Some of the parameters of the algorithm could perhaps be changed in order to improve performance(and remove the warnings), although, it is unlikely that this method will reach an interactive frame rate of about 30 frames per second or more. This is an important aspect to consider, as the distortion of the cloth display needs to update reasonably fast(at interactive speeds) in order for the system to be usable. Due to the previous reason and time constraints, the alternative described in section 4.3.1 was employed.



Figure 24: Mesh from point cloud reconstruction of a bunny.

4.3.5 Kinect Fusion and Kinfu

4.4 Projective Texture Mapping

An article by NVIDIA which also references [26], describes projective texture mapping and how it can be implemented using OpenGL (see [30]). A short summary of this is presented below.

For projective texture mapping we need to use homogeneous texture coordinates. For example, for a projective 2D texture we have the coordinates (s,t,q) where the interpolated homogeneous coordinate is projected to a real 2D texture coordinate by dividing both s and t by q to obtain, $(s/q, t/q)$. The homogeneous texture coordinates have to be assigned per-vertex. The final step implies a range mapping with $[0,1]$ for each coordinate (using a scale and bias, so that the texture coordinates change from $[-1,1]$ to $[0,1]$). In figure 25 the transforms that are applied to a vertex position in order to compute projective texture coordinates are displayed.

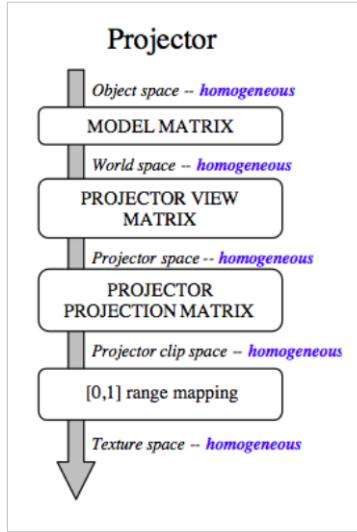


Figure 25: Virtual Projector Transformation Pipeline. Courtesy of [30]

The vertex position in object and eye space is used to generate the texture coordinate. In OpenGL this can be accomplished with the *texgen* facility (generates texture coordinates from vector attributes). In the *object* and *eye linear* modes, the texture coordinate is computed by solving a plane equation at the vertex position. Evaluating a plane equation is equivalent to a 4-component dot product, so the texgen planes form a 4x4 matrix, T :

where s, t, r, q are the texture coordinates, v are the vertices in *eye* or *object* space and T is the texgen matrix in *eye* or *object* space.

$$\begin{aligned} \begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} &= \mathbf{T}_o \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}_{object} \quad (1) \\ \begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} &= \mathbf{T}_e \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}_{eye} \quad (2) \end{aligned}$$

Figure 26: Projective Texture Coordinates from a 4x4 matrix and the vertices in object or eye space. Courtesy of [30]

4.4.1 Object Linear Texgen

The object linear texgen can be represented as shown in figure 27, where the first matrix is the scale-bias matrix, M is the model matrix (in our case this should be the model matrix of the deformable display), V_p is the view matrix for the projector and P_p is the projector's projection matrix.

$$\mathbf{T}_o = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}_p \mathbf{V}_p \mathbf{M} \quad (3)$$

Figure 27: Object Linear Texgen. Courtesy of [30]

4.4.2 Eye Linear Texgen

The eye linear texgen can be represented as shown in figure 28, where the same notations are used as in the previous section, but instead of multiplying with the model matrix M in the end, we multiply with V_e^{-1} which is the inverse of the eye(or camera) view matrix.

4.5 Capturing Dynamic Contents

$$\mathbf{T}_e = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}_p \mathbf{V}_p \mathbf{V}_e^{-1}$$

Figure 28: Eye Linear Texgen. Courtesy of [30]

5 Prototype Implementation

5.1 Tools and Technologies Used

The prototype has been implemented in C++ using *OpenGL v3.3*, which uses the programmable OpenGL pipeline(the vertex and fragment shaders must be programmed to suit the problem's specific needs). This means that shaders must be used to render graphics on the screen. This choice was made because of performance improvements over the OpenGL immediate mode, as well as allowing the code to take advantage of the latest OpenGL features, in possible future extensions.

The development environment used was the *Microsoft Visual Studio IDE*. Initially, development started on the 2013 version of the IDE, but due to incompatibilities with some of libraries used, the toolchain was switched to that of *Visual Studio 2012*(while still using the 2013 IDE - in order to do this, both versions need to be installed). Note that the professional edition of *Visual Studio* is needed in order to be able to install the plug-in for syntax highlighting of shader files.

Further, several libraries were used to create a simple graphics engine that can be used to render objects, import meshes, move around a virtual world etc. The *GLEW*(see [2]) library was used for importing the OpenGL functions and extensions into the project, while the *GLFW*(available at [3]) library was used for managing the window context, as well as for managing keyboard and mouse input.

Another useful feature is to be able to import textures and apply them to objects in the 3D world. The *FreeImage* library(available at [7]) has been used for reading in images and checking if the format is correct. Further, objects can be imported from standard graphics programs through ASSIMP(details at [1]), using well-known file formats like *WaveFront OBJ* and *Autodesk 3DS*. This feature was useful when debugging different aspects of the graphics engine and it was also used to test projective texture mapping on a physical ball, by using an imported mesh of the interior half of a sphere.

Another important library is *GLM*(available at [4]), which contains very useful math functions, like the ability to work with matrices and vectors, calcu-

lating the inverse of a matrix etc. In addition, this library uses the same syntax as that of GLSL(OpenGL Shading Language), which means that there is more consistency in the code and there is no need to learn a third-party syntax.

Further, the *Point Cloud Library*(PCL, available at [9]) was used for trying out some reconstructions techniques, like the greedy projection triangulation. However, this library has not been used in the final implementation, as the mesh was triangulated by connecting a triangle strip, using the fact that the point cloud was ordered, to connect adjacent points.

Finally, the only other library used was the *Intel Perceptual Computing SDK*(details at [6]) which can be used to retrieve the depth and color streams, and do some basic processing on the respective stream. For this project, only the depth stream was used. The data was quite noisy, so in order to reduce noise, some depth smoothing functions were applied. These will be described in one of the future sections. Otherwise, the library allows one to easily convert the data from the depth camera to a point cloud, as well as to unproject it to 3D space, as described in the following sections.

5.2 Virtual World Setup

The process to set up the virtual world will not be described in detail here. The structure of the code was inspired by the OpenGL tutorials available at [8], as well as several other on-line articles/tutorials describing OpenGL functionality. Basic functionality is contained in a class, or cpp file - for example, all texture related processing is the responsibility of the *Texture* class, shader programs are compiled and linked by the *LoadShader* method defined in *loadShader.h/.cpp*, etc. Next, the 3D world is created as a scene(indeed, several such scenes were creating during development, for testing purposes). Each scene must implement three different methods, inheriting from a *SimpleScene* class:

- *initScene(GLFWwindow * window)* - used to initialize everything that is needed before rendering, for example, the compilation and linking of shader programs is done at this stage
- *renderScene(GLFWwindow * window)* - this method loops continuously; it is called every frame, and it is responsible for the actual rendering that can be observed on the screen
- *releaseScene()* - this is mainly used for memory deallocation and is called before the application exits

This template can be used to create new scenes, and all basic functionality such as texture and object importing, camera movement, shader compilation and linking, shaded/unshaded rendering by using different vertex and fragment shader programs, are already available and can be easily reused.

5.3 Point Cloud Generation

The *RawDepthPipeline* class is responsible for retrieving video frames from the depth stream. A method has been added, *renderFrame(..)* that simply acquires the current frame so that the data can be converted to a point cloud, and subsequently a mesh. If the frame data is valid, a call is made to *createPointCloudMappedToWorld* which is the function responsible for converting the raw data into a 3D point cloud, in world coordinates.

An array is needed to store the point cloud:

```
pcPos = (PXCPoint3DF32*)new PXCPoint3DF32[nPoints].
```

The data type is from the Intel Perceptual Computing library and represents 3D points with x, y, z coordinates and floating point values. Further, the number of points, namely **nPoints**, is set to the maximum number of elements we could have in the cloud, which is equal to **depthCamWidth*depthCamHeight**, where the previous two variables represent the width and height of the depth frame, respectively. Then, we just need to iterate over all the depth values and retrieve the depth value:

```
pxcU16 depthValue = ((pxcU16*)ddepth.planes[0])[y * depthStride + x],
```

where **ddepth.planes[0]** is an array containing the depth data, and **x** and **y** represent the current index on the height, and width of the image, respectively. The **depthStride** is used to properly access a two-dimensional array using one index, and it represents the width of the row. Finally, we check if the depth value is in a certain range, and, if so, it is stored in our **pcPos** array:

```
if (depthValue > 10 && depthValue < 2000)
{
    pcPos[n].x = (pxcF32)x;
    pcPos[n].y = (pxcF32)y;
    pcPos[n].z = (pxcF32)depthValue;
}
n++;
```

The elements in the array for which we do not have valid depth values remain uninitialized(**n** is incremented irrespective of the depth value). This is required, because when we send the array later to an *OpenGL VBO(Vertex Buffer Object)*, all valid elements are rendered. This means that initializing the values with, for example $(0, 0, 0)$, would mean that a lot of points would be connected to the origin point $(0, 0, 0)$, which is not desired.

The last thing that needs to be done at this stage is to unproject the points to 3D space. In order to do this, one needs to know the depth camera's intrinsic calibration parameters(these parameters relate to the field of view of the camera, distortion coefficients, principal point etc.). However, the Intel Perceptual Computing SDK provides a very handy method that does the necessary computation for us:

```
projection->ProjectImageToRealWorld(nPoints, &pcPos[0], worldPos.data());
```

The *worldPos* vector array will contain the final points that will be rendered on screen. If we were to render the image now, we would get a 3D point cloud representation of our depth stream. Therefore, the final issue which remains is to connect the points in order to form a mesh, so that the texture will be projected on a polygonal surface. This process will be described in the following section.

5.4 Mesh Rendering

5.4.1 Rendering Process

In order to render a mesh that changes every frame at an acceptable speed, the use of OpenGL VBOs has been employed. This object is nothing more than a contiguous untyped block of memory(untyped means it does not have a particular structure; the structure is defined by the user as we shall see shortly), which can be used to store several vertex arrays and upload them to the GPU. A vertex array is usually used to store a specific vertex property, such as position, color etc. For an example of an interleaved vertex array with two attributes(position and color), please consult figure 29. Note that we are only interested in position for our mesh, since we will only use one full color of the mesh for debugging purposes and, the user will only see the color of the texture.

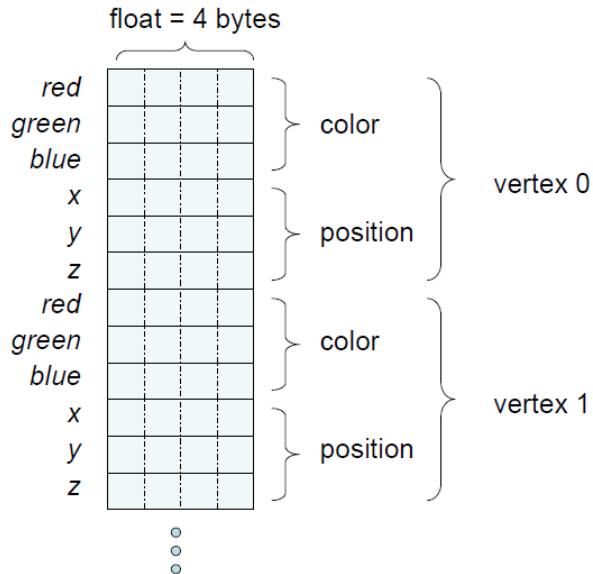


Figure 29: Vertex array layout with two vertex attributes. Courtesy of [33]

A good resource that explains how to properly use VBOs in OpenGL and

why they are needed is [33]. The author explains that the GPU doesn't have arbitrary access to conventional system memory(so called client memory). With VBOs one can point the GPU at the vertex arrays and let it read the vertex attributes from the arrays contained inside it(the arrays are stored in spatially negotiated regions of memory, maintained as buffer objects, that both the application and the OpenGL implementation can access). An example of how the VBO data is transferred to the GPU via DMA(direct memory access) transfers is shown in figure 30. As you can see from the figure, there is no communication between the data and the CPU, all transfers are made using DMA transfers directly to the GPU. This means that the performance can be greatly improved by avoiding unnecessary communication with the CPU. This is in stark contrast with the OpenGL immediate mode, where each call to `glBegin()` and `glEnd()` had to be sent first to the CPU, in addition to a full list of commands for drawing(now we just draw the VBO with one call to `glDrawArrays` or `glDrawElements`).

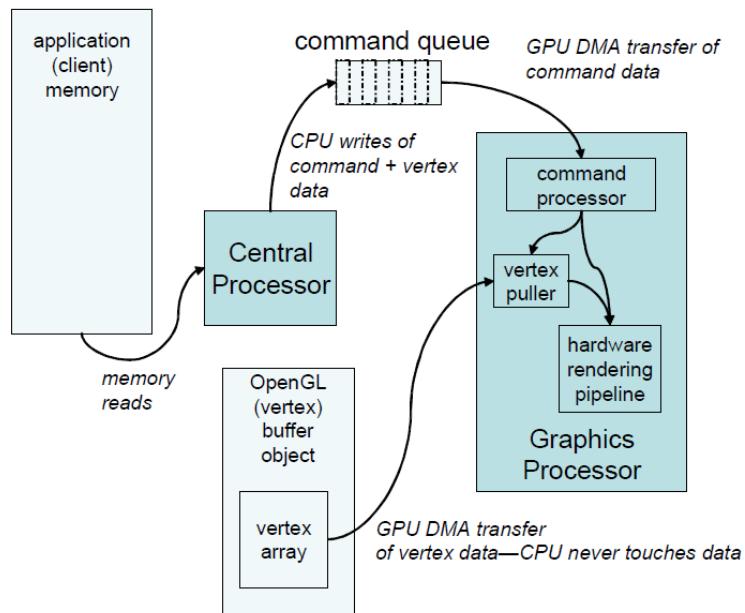


Figure 30: Transfers of vertex array object data using DMA.

We need to define the type of data and how much of it needs to be stored in a VBO. In order to do this we use a vertex attribute pointer:

```
// vertex positions start at index 0
// the distance between two consecutive vertices is
// sizeof whole vertex data
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(PXCPPoint3DF32), 0);
```

The signature of the vertex attribute pointer method is as follows:

```
void glVertexAttribPointer(GLuint index, GLint size, GLenum type,
    GLboolean normalized, GLsizei stride, const GLvoid * pointer)
```

In the previous code we enable the vertex array with index 0. The *glVertexAttribPointer* is also set to index 0(first parameter), since it is the one and only attribute(for example, we could interleave data, if we also had texture coordinates; then the index would be 1 since 0 would have already been used by the position attribute). In it we can store data with 3 components: x, y, z and we specify the size of *PXCPPoint3DF32*. The size will be used as an offset to store the next vertex(stride). We only store the point cloud elements in this VBO and, therefore, we can just set the pointer to 0, as there is no other data stored in the VBO.

Before a VBO has to be used, it must be bound, so that it is made the active VBO. The custom class *VertexBufferObject* provides methods so that we can easily create and bind VBOs, like so:

```
// create and bind VBO
vboSceneObjects.createVBO();
vboSceneObjects.bindVBO();
```

Now that our VBO is bound, we need to add the actual data:

```
glBufferData(GL_ARRAY_BUFFER, rdp->nPoints * sizeof(PXCPPoint3DF32),
NULL, GL_STREAM_DRAW);
```

This line stores an array buffer of n points of the Intel Perceptual Computing type. Further, the hint *GL_STREAM_DRAW* indicates that our data will be constantly updating(every frame). This makes OpenGL optimize the VBO for dynamic contents. Another flag, *GL_DYNAMIC_DRAW* is available, but this is actually for dynamic data that does not change every frame, only at specific moments. So, in this case, the first flag is the appropriate one, since the mesh will be updated every frame.

OpenGL also has another data type, namely the VAO(Vertex Array Object) which can store multiple VBOs. These VAOs are mainly used to avoid setting the vertex attribute pointers each time we bind the VBO. We just bind the VAO, and we do not have to do anything else. Now, if we were to render our memory contents now, using *glDrawArrays* we would need to draw the type *GL_POINTS*. This is because we have not triangulated the point cloud yet.

5.4.2 Triangulation

As described in section 4.3.1 a triangle strip has been used to triangulate the point cloud. In order to do so, an *IBO(Index Buffer Object)* has been used to store a *GL_ELEMENT_ARRAY_BUFFER*. This object is very similar to a VBO, the only difference is that we do not store vertex attributes in the element array, but actual element indices. In other words, we need to compute an array of

indices, that defines how the triangle strip should be connected, and then just simply bind the data to the IBO:

```
ibo.addData(&(rdp->indices[0]), rdp->indices.size() * sizeof(unsigned int));
ibo.uploadDataToGPU(GL_STATIC_DRAW);
```

We add an array of ints into the buffer object of the size of the depth camera's resolution, in pixels(`width*height`). Note that the index computation can only be done once! We always have the same indices, for the same grid, since the x and y values are fixed for a vertex - the indices are computed for all points, even for invalid ones, this means we do not have to recompute the indices if the state of the points changes(since the VBO data for those invalid points is not initialized, they will not be drawn). This means the flag for data optimization has been set to `GL_STATIC_DRAW`. Further, the `uploadDataToGPU` method internally calls `glBufferData` with the previously added element array data. Finally, the IBO is bound and setup similarly to a VBO.

The drawing command in order to render a mesh from the indices(connected using a triangle strip), is:

```
glDrawElements(
    GL_TRIANGLE_STRIP,           // mode
    rdp->indices.size(),       // count
    GL_UNSIGNED_INT,            // type
    (void*)0                   // element array buffer offset
);
```

The parameters are briefly explained in the comments, but to detail briefly:

- the first parameter is the type of primitive OpenGL will use for rendering
- the second parameter represents the number of elements required to draw the mesh
- the third parameter defines the type of the elements
- finally, the last parameter is simply a pointer that defines an offset, but since we only have the mesh data to render with indices, this can be set to 0

The only thing left to be is explained is how the index array has been populated. In order to understand how the code works, it is useful to review figure 19. The logic is contained in the `addIndexData` method from the *RawDepth-Pipeline* class. We traverse the two-dimesional grid with one index(in the same manner as we have traversed the depth frame before):

```
t = c + r * colSteps,
```

where `t` will represent our current position in the grid. As we can see from the previously mentioned figure, the indices are connected by using elements

from adjacent rows, alternatively. We can then simply check if this position is divisible by 2 or not, so we know if we need to move to the other row. We first push index 0, and then we need to add the width of the grid to our current index(since it is divisible by 2):

```
if (t % 2 == 0) {
    n += width;
}
```

In the next iteration, we go in the `else` loop and check if the index of the row is divisible by 2. This is required, since we traverse the grid from both directions and we need to subtract `width-1` or `width+1`, in order to traverse up with one row, depending on whether we are traversing the current row from left to right, or right to left, respectively:

```
(r % 2 == 0) ? n -= (width - 1) : n -= (width + 1);
```

Then, the process is simply repeated. Note that, since we pass through two rows in one direction, `colSteps` is set to `width * 2`(for example, the last element in the second row in the figure is 7, which is element number $4*2 = 8$, since our width is 4 in that example). Further, we stop one row before the last, since we traverse through the last one, when we start on the one right before it, so `rowSteps = height - 1`.

Finally, we need to remember to add the degenerate triangle by adding the last index twice(once as shown below, and again at the start of the next loop, as `r` is incremented):

```
if (c == colSteps - 1) {
    indices.push_back(n);
}
```

At the end of this process, the `indices` vector will contain all of the triangle strip indices and in the right order. Therefore, it can be added to the index buffer object, so that the data is available for rendering later(degenerate triangles will simply be discarded by the GPU, they will not introduce artifacts).

5.5 Depth Smoothing

The depth data is quite noisy and, due to this, the final textured image is shimmering(vertices are moving around between frames, which creates a distracting effect). The effect is similar to aliasing(the "staircase effect").

In order to reduce this noise, smoothing of the depth data can be applied. A whitepaper from Canesta(see [25]) discusses aliasing issues, as well as artefacts for time-of-flight cameras. The authors describe how the noise can be reduced through *spatial* and *temporal averaging*.

Further, they suggest the use of a median filter for improving resolution, as a solution for spatial averaging. They applied a 5×5 median filter and noticed a two-fold increase in resolution.

Next, they present a case where 10 frames were averaged over time. Their results indicate that this technique of temporal averaging made the image much clearer and considerably increased the resolution.

In the following sections the implementations for spatial and temporal averaging employed in this project will be described.

5.5.1 Spatial Averaging

A median filter is used as a means for spatial averaging. The basic idea is to form a window of size $N \times N$, where N is an odd number(greater than 1, otherwise we would have just one element in the window). Typical values for N are 3, 5 or 7. We traverse with such a window through the whole image, with the current pixel(or vertex) being in the center of the window, and the other elements being the immediate neighbors. Further, we need to obtain the `median` of the window. The median is the center element of a sorted array. Therefore, we also need to apply a sorting algorithm. Insertion sort is typically used for median filter implementations, since it is efficient for small data sets.

The median filter can be turned off and on by the press of a key, using a boolean variable. The size of the window must be specified before, since it is represented as a constant. The `medianFilter` method in the `RawDepthPipeline` computes the result for every frame, based on the specified `windowSize`. In order to do this, the fact that both index digits of the center element in the window are: `midIndex = (size - 1) / 2` is used(here `size` is equal to the N mentioned above). To compute the window elements, we traverse it using `for` loops and then assign the values like so:

```
window[index] = pcPos[(y - midIndex + i) * depthCamWidth
    + (x - midIndex + j)].z;
index++;
```

We simply need to subtract `midIndex` from the current position, in order to get the neighbor elements. For example, `midIndex` would be equal to 1, for a 3×3 window. Then `window[0] = pcPos[(y-1)* depthCamWidth + (x-1)]`, ..., `window[8] = pcPos[(y+1)* depthCamWidth +(x+1)]`.

Once we have the elements, we need to sort them, in order to find the median. Insertion sort was used to accomplish this: `insertionSort(window)`. Now we simply need to assign the median as the new depth value for the current point, and continue to iterate through the rest of the points.

Additionally, we must only apply the algorithm if all the neighbors of the current point have valid depth values(since certain areas of the image do not contain depth information), so this must also be checked.

The sorting method works by iterating through each element of the array, storing it(`temp = window[i]`) and then comparing it against the size of all elements to the left of `temp`. The left side represents the sorted part of the array(we start with the left-most element). If the current element is less than another one from the left of the list, the larger element is shifted one place to the right.

```

for (j = i - 1; j >= 0 && temp < window[j]; j--) {
    window[j + 1] = window[j];
}
window[j + 1] = temp;

```

The last line places the current element in its proper place. If it is the largest element in the left side of the list, j will not be decremented and it remains on its current position. Otherwise, it is moved to the left, depending on how many elements are less than the current value.

5.5.2 Temporal Averaging

A WMA(Weighted Moving Average) algorithm has been used for temporal averaging. The main purpose of this is to average the depth information over the last N frames. We assign weights or priorities to the frames, with the last frame having the highest priority when averaging.

The frame data is stored in a list from which we can push and pop elements(individual frame data arrays):

```

// push array to the back of the list
averageList.push_back(pcPos);

```

Once we push a new frame in the list, we also need to check if the size of the list is greater than the number of frames we need to average over(we call *checkForDequeue*). If so, we remove the extra frames from the list.

```

if (averageList.size() > averageFrameCount)
{
    delete[] averageList.front();
    averageList.pop_front();
    checkForDequeue();
}

```

Next, we iterate through all of the arrays in the list, and compute the sum of the depth data for each element, with a weight that is 1 for the oldest frame, and N (currently 10) for the newest:

```

// count denotes the weight
sumDepthArray[index] += (pxcF32)item[index].z * count;

```

Further, we also store a denominator we will use to compute the average: *denominator* += *count*(we count how many times in total we add depth values from frames). Note that we need to finish this pass through all frames, in order to obtain all the sum values and the denominator.

Finally, we must iterate through the sum values and compute the average:

```

averagedDepthArray[index].z =
    (pxcF32)(sumDepthArray[index] / denominator);

```

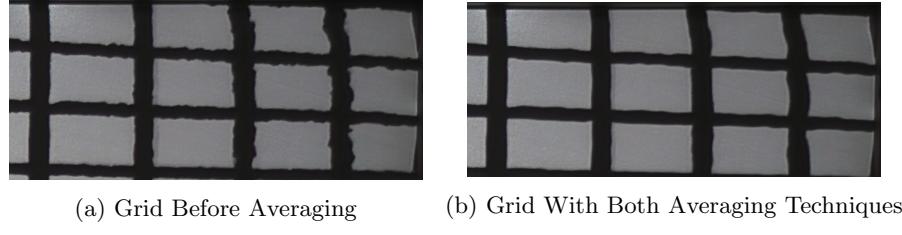


Figure 31: A grid texture which shows the noise reduction obtained by averaging.

In order to speed up the process, we can use a `parallel_for` loop, so that the rows are processed in parallel. This is possible because the calculations are independent.

```
// Process each row in parallel
Concurrency::parallel_for(0, depthCamHeight,
[&](int depthArrayRowIndex)
{
    ...
}
```

5.5.3 Results

The averaging techniques have been tested on the deformable display, by judging the quality of a texture with a grid on it. This makes it easier to notice noise around the grid's lines. In figure 31, the difference between the grid texture with no averaging applied, and with both averaging techniques applied, is shown (the image is zoomed in so that the differences are easier to spot).

Note that it is easier to see the differences in the thesis' video, although it is still possible to see a significant difference in the figures. A downside of averaging is the reduction of frame rate. A 3×3 median filter was used in the final implementation, and a 10 frame moving weighted average was used as a temporal averaging technique. In table 1 some data about the frame rate is presented, when turning on and off different averaging techniques. The frame rate was printed out for a certain number of frames, every second. The tests were also repeated 2-3 times and the results seemed consistent. We can see that the temporal averaging technique only slightly reduced the frame rate, while the median filter drastically reduced it, almost halving it when a 3×3 filter was applied, and almost halving it again, when a 5×5 filter was applied.

In order to improve the performance, the median filter could be applied on the GPU, by using shaders. One such method is described in [39]. They only briefly describe the method, but provide source code and a reference where the full algorithm is explained.

Finally, all averaging techniques can also reduce detail, for example, they could make the distortion correction less pronounced. This issue has not been thoroughly tested, and needs to be determined as future work. In order to

Averaging	Frame Rate
None	[55 - 60]
Temporal	[50 - 57]
Both(3x3 MF)	[30 - 40]
Both(5x5 MF)	[12 - 21]

Table 1: Frame Rate Comparison When Using Noise Reduction Techniques

properly analyze the previous issue, the calibration of the system should also be corrected, because then it would be easier to compare with the expected result. A further issue may be that the system was not calibrated, so the distance and field of view of the virtual camera were set by trial and error. This is important because distance affects both the noise amount, as well as the degree to which the distortion correction can be seen(it needs to be *just right*, and the correct parameters can only be obtained through calibration).

5.6 Projective Texture Mapping

In order to get a final image rendered with the distortion compensation, by using a mesh of the deformable surface, one can apply projective texture mapping(**PTM**), as described in section 4.4. This method was implemented using GLSL shaders.

5.6.1 Setting Uniforms and Computing the Texture Coordinates in the Vertex Shader

Since the programmable OpenGL pipeline was used, there is no predefined **Texgen** matrix. Instead, we can just use any 4×4 matrix. This matrix is defined as a uniform in the vertex shader(OpenGL server side code):

```
uniform mat4 TexGenMat;
uniform mat4 InvViewMat;
```

The matrices are set from the OpenGL client code(the actual C++ application). The inverse view matrix is needed in order to convert from eye to world coordinates. In addition to these PTM specific matrices, we also need the usual projection and modelview matrix, as well as the position of the vertices:

```
uniform mat4 P;
uniform mat4 MV;
layout (location = 0) in vec3 inPosition;
```

We need to compute the texture coordinates for the projected texture and pass them to the fragment shader:

```
out vec4 projCoords;
vec4 posEye      = MV * vec4(inPosition, 1.0);
```

```

vec4 posWorld = InvViewMat * posEye;
projCoords    = TexGenMat * posWorld;

```

The `TexGenMat` has been constructed as described in section 4.4, in the object linear texgen equation, except the final multiplication with `M`:

```

texGenMatrix = biasMatrix * projectorP * projectorV;
invViewMatrix = glm::inverse(mModelView);

```

Actually, by the time we compute the inverse, in the second line of the code above, `mModelView` only contains the view matrix. The modelview matrix uniform is instead stored from another variable, called `mCurrent`:

```
glUniformMatrix4fv(iPTMModelViewLoc, 1, GL_FALSE, glm::value_ptr(mCurrent));
```

The `glUniformMatrix4fv` method binds the given matrix to the location of the modelview matrix in the shader. The 4 in the name, means OpenGL is expecting a 4×4 matrix, and the `v` at the end denotes a vector or pointer array, which we will need to pass in as the data - in this case, `glm::value_ptr(mCurrent)`. The proper location can be retrieved from the shader program ID and a string which corresponds to the name of the variable:

```
int iPTMModelViewLoc = glGetUniformLocation(programID, "MV")
```

The other uniforms are set in a similar way(if we just need to bind an integer we use `glUniform1i`):

```

glUniformMatrix4fv(iPTMProjectionLoc, 1, GL_FALSE,
    glm::value_ptr(*pipeline->getProjectionMatrix()));
glUniformMatrix4fv(iTexGenMatLoc, 1, GL_FALSE,
    glm::value_ptr(texGenMatrix));
glUniformMatrix4fv(iInvViewMatrix, 1, GL_FALSE,
    glm::value_ptr(invViewMatrix));
glUniform1i(iSamplerPTMLoc, 0);
glUniform1i(iProjSamplerLoc, 1);
//black
glUniform4fv(iColorLoc, 1, glm::value_ptr(glm::vec4(0.0f, 0.0f, 0.0f, 1.0f)));
// update debug state
glUniform1i(iDebugLoc, bFragmentShaderDebug);

```

We set the projection matrix, the texgen matrix, the inverse view matrix, and finally sampler locations, color locations and a debug uniform. The samplers, colors and debug uniforms are only used in the fragment shader, while the others are needed in the vertex shader.

Finally we need to apply the model matrix, in order to get world coordinates. First, the eye coordinates are computed(multiplying with the modelview matrix). Then, in order to get the world coordinates, we need to remove the effect of the view matrix. We can do this by multiplying with its inverse(this is equivalent of just multiplying with `M`, but in order to be consistent with other shaders, the modelview matrix is passed as one matrix, denoted `MV`):

```

vec4 posEye    = MV * vec4(inPosition, 1.0);
vec4 posWorld  = InvViewMat * posEye;
projCoords     = TexGenMat * posWorld;

```

Finally, the position is computed as usual, by applying the Model View Projection(MVP) matrix to the input vertex:

```
gl_Position = P * MV * vec4(inPosition, 1.0);
```

5.6.2 Applying the Texture in the Fragment Shader

Next, the fragment shader computes the final texture coordinates for the vertex and extracts the color from the sampler:

```

vec2 finalCoords = projCoords.st / projCoords.q;
vec4 vProjTexColor = texture(projMap, finalCoords);

```

Samplers are used to access the texture data. In this case, we have two `sampler2D` types, since the object may have its own texture, on which we may then apply the projected texture:

```

uniform sampler2D projMap;
uniform sampler2D gSampler;

```

However, for the deformable cloth mesh we do not need to use an additional texture. A constant color value was used for the mesh, in the areas in which it is not covered by the virtual texture projector. Therefore, if the texture coordinates are valid and we do not have a reverse projection the output color is set to the one retrieved from the texture, otherwise it is set to an input color:

```

// supress the reverse projection
if (projCoords.q > 0.0)
{
    // CLAMP PROJECTIVE TEXTURE (for some reason gl_clamp did not work...)
    if(projCoords.s > 0 && projCoords.t > 0 &&
    finalCoords.s < 1 && finalCoords.t < 1)
    {
        outputColor = vProjTexColor;
    }
    else
        outputColor = vColor;
}
else
    outputColor = vColor;

```

The texture was clamped in the shader, so that it does not repeat. This was needed, due to the fact that the normal texture parameters(that can be used to specify the `GL_CLAMP` flag for a particular coordinate) did not work:

```
glTexParameteri( GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_CLAMP );
```

Finally, as mentioned in [30], one of the problems of projective texture mapping is that it also produces a reverse projection when the sign of q becomes negative(see figure 32). However, it is easy to correct this in the shader by just checking that q is positive, as we have seen in the code snippet above.

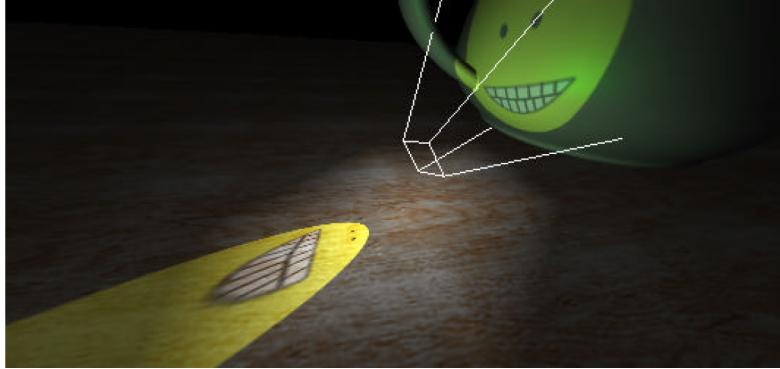


Figure 32: Reverse projection after applying PTM. Courtesy of [30]

5.6.3 Results

In addition to what has been described, a frustum has also been created for debugging purposes, so that we can see the virtual projector in the scene. The full code of the projective texture mapping shaders(for shaded scenes), is available in a *stackoverflow* discussion(available at [11]). I originally started it, while I was stuck on this problem. The previous resource may also describe some aspects in greater detail. The algorithm has been applied on several objects in figure 33. Further, in the accompanying thesis video, there is another example of the projected texture being moved, by transforming the virtual slide projector.

Further, the `debug` flag was used later on in order to change the color of the vertices, if they are within a certain depth range. More on this, in the following sections.

5.7 Manual System Alignment (Calibration Not Used)

I have run out of time to implement a proper calibration solution, I have implemented a manual process. The results are not as precise as with regular calibration, and an extension to this project would be to correctly calibrate the projector-depthcam system. In this section I will outline the main steps and tests needed in order to make sure that the alignment is acceptable(or at least that there is a difference when applying the compensation; tests will be required in order to find out if the compensation works, and to what degree, for the current setup).

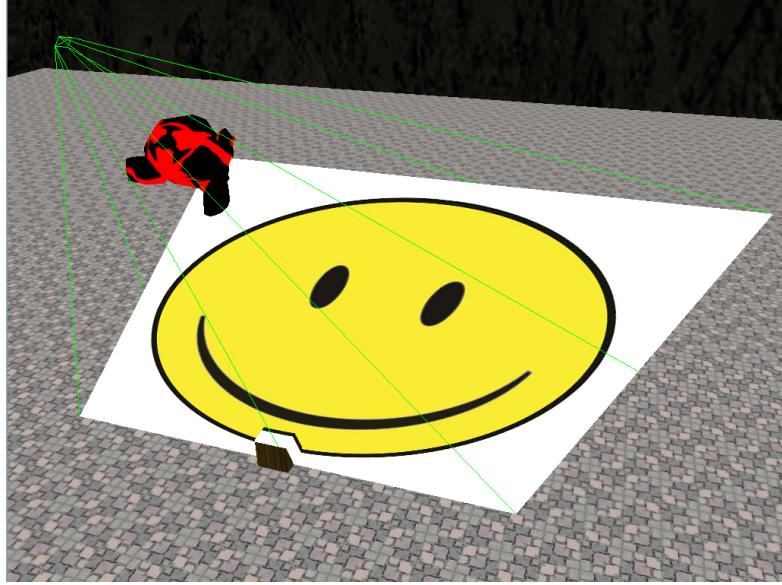


Figure 33: PTM applied on a Suzanne monkey head from Blender, and on a floor with a small cube. The frustum represents the virtual projector.

5.7.1 Physical Projector Alignment to the Physical Screen(Cloth area)

This step consisted of physically moving the display, positioning it, changing lens shift amounts, using keystone correction etc. In order to test this step, I have created a texture with a grid(see figure 34), four colored corners and a red border. The grid and the corners were used to roughly position the projector so that the cloth display is correctly in view. The border was used for some fine tuning.

Some photo shots were taken, in order to show how well figure 34 fits the cloth area. The whole image is shown in figure 35. In this figure, we can see that the image very closely matches the surface, and that the four cyan squares in the corners roughly correspond to those of the visible cloth area. However, it is hard to see how well the red border is aligned(it is not very visible in this figure).

Some shots of the corners of the frame were taken, so that we can more clearly see how well the grid texture is mapped to the surface. These images are shown in figure 36.

From these images, we can see that the texture is reasonably well aligned, but the red borders are not very visible on the sides, and the image is slightly skewed towards the upper left, making the border disappear completely in that part of the image. Some space is also present in the upper central area. This may be due to the cloth not being perfectly flat. Some of these visual cues may

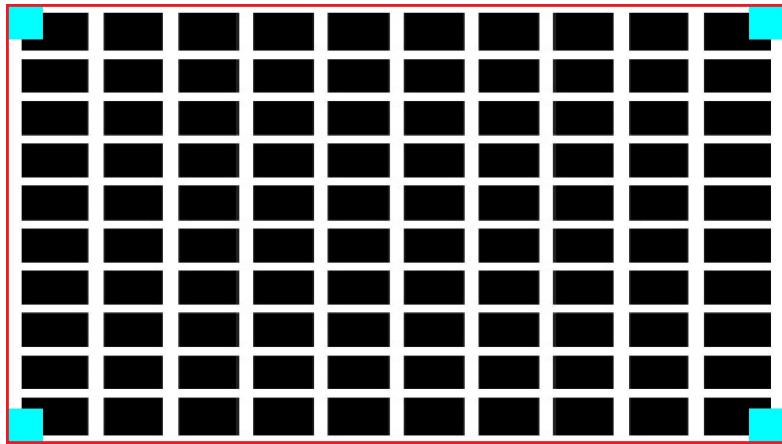


Figure 34: A grid with four corners and a red border, used to test the physical alignment of the projector to the cloth.

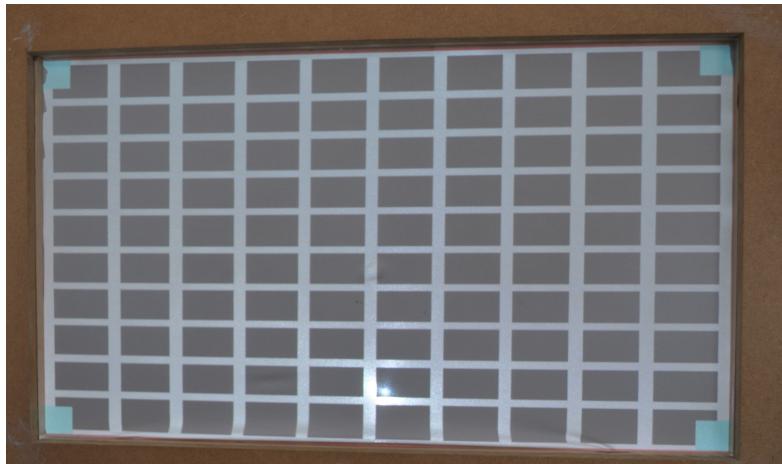


Figure 35: The alignment grid texture projected on the cloth.

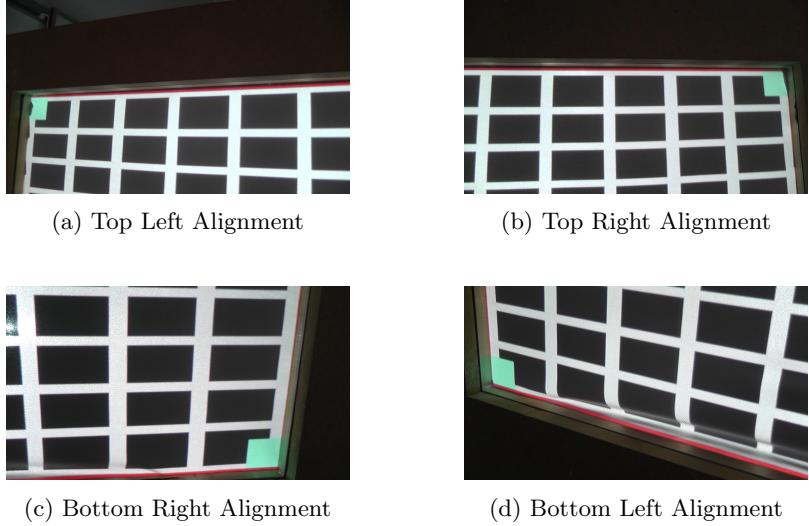


Figure 36: Illustration of alignments in the four corners of the frame.

be hard to spot on these figures, so they will be also displayed in appendix B, in a larger size.

5.7.2 Physical Depth Camera Alignment to the Physical Screen (Cloth area)

The depth camera was also aligned to the cloth manually, by trial and error. The objective was to position the camera as orthogonal as possible onto the cloth display. In order to do this physically, I used a tripod with the camera attached upside down, to an inverse tripod column(see the setup in figure 37).

This setup allowed to position the camera fairly close to the screen and fairly straight on. This has been tested by displaying a pattern on the projector and touching the center, while observing the corresponding changes in the depth stream. As you can see in figure 38, the point is roughly in the middle of both images.

However, this figure only provides information about the alignment of the image's center area. In order to test the sides, we can check the depth values of points, in different areas on the screen. Of course, the depth of the captured object has to be the same in the physical world - we can use the wooden frame for this. The frame has been placed at a distance of approximately 2.5 centimeters from the edge of the desk, as shown in figure 39. However, as you can see, there is a slight offset, with one side being slightly closer to the edge. This issue may also introduce some minor depth discrepancies.

An application that allows a user to manually specify a number of points in the image and subsequently returns the depth values corresponding to these points, has been developed(see figure 40). It is using the OpenCV(see [?])

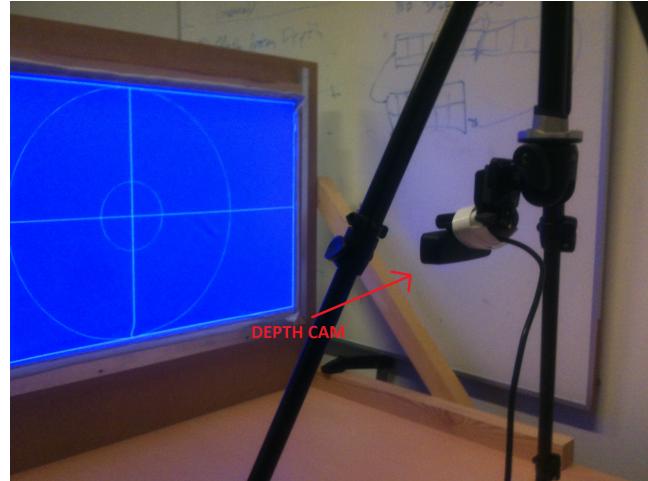


Figure 37: The depth camera is setup on a tripod. By reversing the column we can get the center of the depth camera to roughly correspond to the center of the projected image.

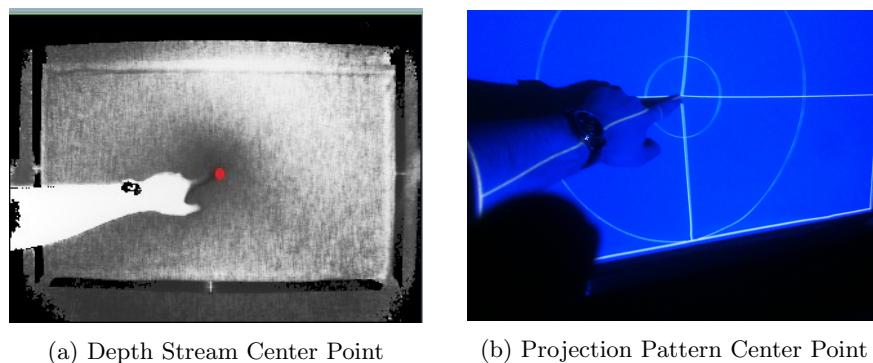


Figure 38: A comparison which shows that pushing in the center roughly modifies the same area of both the depth stream and physical image.



Figure 39: The frame placement. Distance is displayed in red.

and Intel Perceptual Computing Libraries. The former was used for displaying the image, reacting to mouse input and drawing the crosses, while the latter provided the depth frame data from which the image was created. The depth has been returned in millimeters, and for the points in figure 40, the values were: 689 for top left, 685 for bottom left, 701 for top right, 702 for bottom right. From these values, it appears that there is a small vertical offset(1-4 mm), and a larger offset on the horizontal axis(7-11 mm). The test was repeated several times, and the general pattern was the same - larger horizontal offset and smaller vertical offset. However, the obtained values shifted overall with up to 2-3 centimeters, between different frames and points. While the relative offset patterns were largely the same, there were a few exceptions, when the vertical offset was also large on one side. However, this issue, along with that of the points being shifted, may be caused due to noisy data, as the points in those cases were very near to the top or bottom edge of the valid depth data(where we can also see some small dark spots).

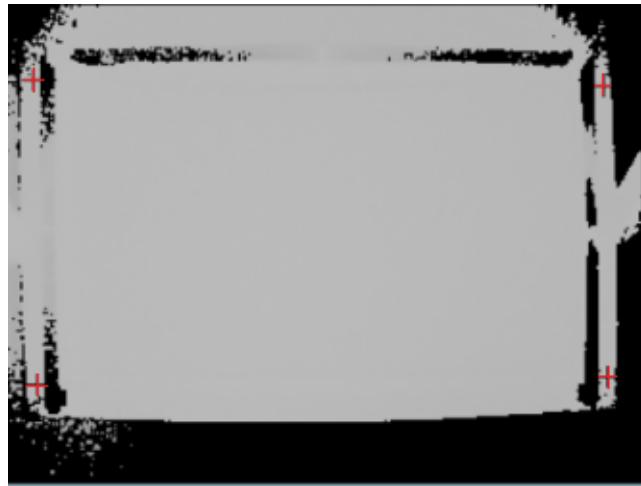


Figure 40: Points for which depth had been extracted and compared.

Note that if calibration is used, the offset introduced from incorrect orientation and positioning is applied to the virtual camera view. Therefore, as long as the projector would be physically aligned to the cloth and the cloth would be fully in view of the depth camera, the setup would work fine. It would not be necessary to have the camera aligned perfectly in the physical world.

5.7.3 Virtual projector alignment to the cloth area

The virtual slide projector displays a texture onto the mesh of the cloth display. When looking at the display's mesh in wireframe mode(see figure 41), we can distinguish what roughly represents the cloth area, because of depth variations with respect to the frame.

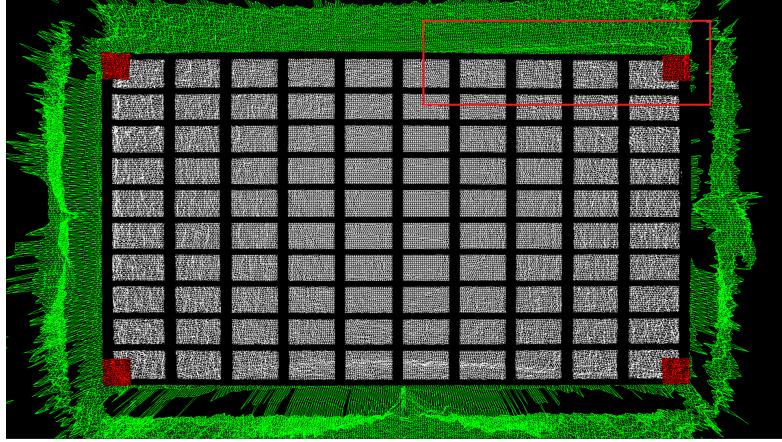


Figure 41: Texture Aligned to Cloth Surface. The area inside the red rectangle shows an offset.

Due to the fact that the cloth mesh is rotated, the texture can not be aligned perfectly (in order to do this, proper calibration is required; see figure 42). Otherwise, a texture with 4 red corners and a grid has been used to test if the grid looks fairly straight and the corners roughly correspond to the edges of the cloth.

The offset is not easy to see in the previously mentioned figure, but an arrow points to the line which separates the wooden frame from the cloth. The line is a bit thicker and we can think of it like a delimiter. If you look closely, you can see that the line is steadily going upwards as it advances towards the right side. If the texture would have been aligned perfectly, that line would correspond to the top line of the grid.

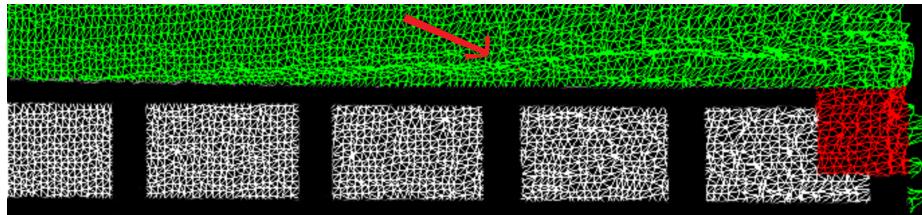


Figure 42: The area has been magnified. An arrow points towards the line that separates the frame from the cloth.

In order to align the texture better, or at least, make the debugging process easier, we could calculate the normals for the mesh and show the shaded object. Further, we could also add color information (then we would have a full 3D model, which resembles the physical object, making it easier to see where the texture should be projected).

5.7.4 Virtual camera alignment to the cloth area

The virtual camera was aligned by computing the projection and view matrices, by trial and error, so that it completely covers the previously aligned texture. The projection matrix was computed via the `glm::perspective` method, which requires the vertical field of view, the aspect ratio and the near and far planes. The aspect ratio was set to 16 : 9, and the near and far to 0.1 and 1000, respectively. The field of view(FOV) was the parameter that was computed by trial and error. If we have the intrinsic parameters, the field of view can be obtained, or alternatively, we can obtain all of the frustum planes with these parameters(*left, right, bottom, top; near and far* can be specified manually). For the latter, `glm::frustum` could have been used.

Next, for the view matrix, we simply use a modelview matrix, of the form described in figure 12. The matrix is set in the code by specifying every element manually. If we have the extrinsic calibration parameters, we need to use the translation parameters for the last column, and the rotation parameters for the first three columns in the matrix. The last row remains unmodified.

As mentioned, the camera was moved around the scene until it roughly had the texture in view. The field of view and distance of the camera have been adjusted to that effect, but, it is important to note that there is an infinity of ways to combine these two values in order to roughly have the texture in view. This is because, we can always increase the field of view, and decrease the z distance, or vice versa. Now, the important issue here, is that the image will not look the same for all different combinations of values! For example, if we move the camera closer to the mesh in the virtual world while decreasing the FOV, we will see more aliasing(noise), but the distortion compensation is also more pronounced. Conversely, if we move the camera further away while increasing the FOV, we will have a more stable image, but the distortion effect is less pronounced. In order to get accurate results, one needs to calibrate the system in order to obtain proper parameters. However, it is still possible to set the value to acceptable levels by trial and error.

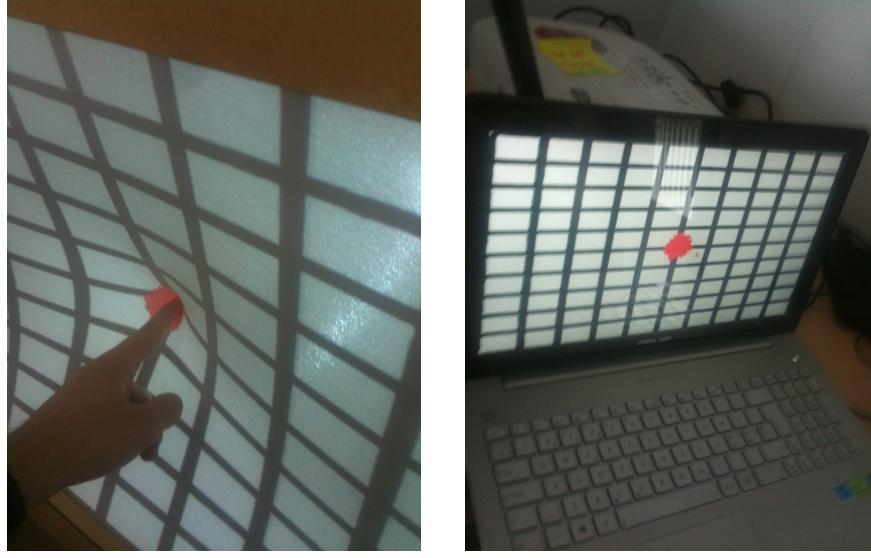
The values were set manually as explained above, and the system was tested via a video, where the difference between an image with distortion compensation and without is shown, and an experimental study. If the results from these tests are positive, meaning that the compensation improves the quality of the image(to some degree), by making it look more like on a flat screen, then it is reasonable to assume that, if the setup is calibrated, the distortion correction will be of higher quality. However, another set of tests would be necessary in order to verify this. The empirical study is described in section 6.

5.7.5 Testing precision of alignments

Finally, in order to test how well the system is aligned, tests were made with single finger interactions, where the pushed finger corresponds to the maximum slope(minimum depth distance from the camera). A color coding scheme has been used for testing. First, just by using a red color for testing the finger's

tip, and then several colors, to distinguish a larger area around the interaction point.

The results were accurate in the center, as you can see from the figure 43. The area surrounding the finger was drawn roughly in the center, in both images(the size of the area was determined by a threshold). In 43a we can also see that after pushing into the display, the red dot still appears around the finger.



(a) Pushed area on cloth, shown red. (b) Pushed area on screen, shown red.

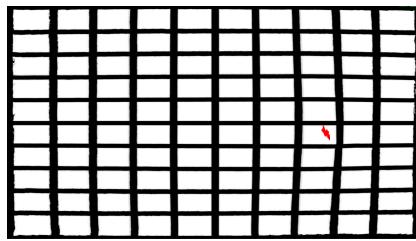
Figure 43: Correspondence between physical and virtual images.

However, when testing towards the sides, the results were worse, and the offset grew larger. This is probably due to the fact that the surface is rotated. The results are shown in figure 44, with the detected area being very small. However, we can still see that the image shifts when seen on the cloth, and the red area is no longer aligned with the finger. Considering this, it is much more likely that the image will be correctly shifted(undistorted) only in the center area. Some further testing and adjustments could perhaps reduce the offsets and increase the quality of the distortion correction.

Please do not try to look for the distortion compensation effect in these figures, because they have been captured at an angle and you will not see the correct effect. For the compensation please take a look at the video and section [TO DO], where the results are also discussed.



(a) Pushed area on right side of cloth,
shown red.



(b) Pushed area on right side (screen view),
shown red.

Figure 44: Correspondence between physical and virtual images.

6 Empirical Study

Despite the fact that the system was not calibrated, it did seem to correct the distortions in certain areas. However, this correction is not perfect, and to find out how well it performs a user study had to be run. Further, the users were also asked if they prefer the correcting method over a normal image.

6.1 Experiment Description

Before starting the experiment, each user was given a short text to read, which presented basic information on the experiment in which they were about to partake. Next, they were introduced to the cloth display and the distortion problem was explained and shown to them (with images outside of the application). Further, they practiced the gestures that they were about to use during the experiment.

Each user was presented two methods of displaying the image onto the cloth: A and B. One of these methods represented the distortion compensation method that is the focus of this report, while the other was just a regular image, projected onto the cloth. The participants had to compare the two methods, regarding the preservation of appearance (proportions similar to the original image, projected onto a 2D surface), or whether they see a difference at all. Further, 5 scenarios were designed, and the users had to test both methods in all scenarios.

Two pilot experiments were performed, before the actual user study. From these, an important issue surfaced. Users could easily spot which method is the normal one, because there was no noise in the image, while the distortion correction methods introduced aliasing-like effects, other noise and black areas, due to invalid depth data (while interacting).

One attempt to solve the problem was to use depth smoothing as described in section 5.5. The final result after spatial and temporal averaging was still not good enough - you could still see the difference between the two methods, because of the dynamic noise in the compensated image. However, the smoothing did reduce the overall noise and improved image quality.

Since noise reduction did not solve the issue, the remaining options were to either introduce noise in the other image, by some function or random pattern, or, the solution which was actually employed, capturing a number of frames rendered with the distortion correction method and displaying them continuously as the normal method. This approach made the images look very much alike when the display was not deformed. Consequently, this served as a good way to avoid any bias coming from comparing a normal image with a noisy one. Most users did not realize that the other method was just a normal image (or actually 10 frames which keep repeating). Even some users, who had the impression that the normal method does not help at all with correcting the distortion, assumed that the method is doing something, because of the dynamic noise in the image.

Coming back to the two methods, A and B, the users had to switch between them by using a mouse as an input device. As seen in figure 45, the mouse had two labels, denoting which button must be pressed to select a particular

method. Moreover, it was attached to the display's frame, which made it easy for users to interact with it. The participants were instructed what task they need to perform, before each interaction point.



Figure 45: The input device for switching between the methods.

This input method allowed the users to compare the visuals while interacting with the display, and at their own pace. Further, this enabled the use of the same images in both methods A and B, without having to use a visual indicator on the textures themselves. All scenarios started with method A, but after some points, a few users forgot which method was A or B; although, this was not a problem, since they could just press the A(or B) button twice, to find if the current image is rendered with that method. For certain scenarios, when using two-hand interactions, I switched between the two methods for the users. Moreover, the scenarios and points were also changed by me.

Next, methods A and B were also given in a random fashion - first A as the compensating method for the first user, then B as the compensating method for the second user, and so on, not necessarily keeping the order(meaning that, for example, the experiment with A as the compensated method could have been performed twice in a row). Moreover, the scenarios have also been shown to the participants in a different order, so that the data is not biased for a particular scenario configuration. Therefore, each scenario was assigned a number between 1 and 5, and a latin square was used in order to decide the scenario configuration for the following experiments. Each row from the square(the same scenario configuration) was used once for each combination of A and B(once when A was the compensating method, and once for B).

The original order of the scenarios was as follows:

1. Grid
2. Text
3. Columns

1	2	3	4	5
2	3	5	1	4
3	5	4	2	1
4	1	2	5	3
5	4	1	3	2

Figure 46: A 5×5 latin square that denotes the order in which scenarios were shown to the user.

4. Map

5. Brain Scan

Finally, I was sitting at a desk behind the users, so I could watch them go through the task, while also keeping notes on the current task. I also asked interview questions, while they were interacting. In addition, a questionnaire was given to the users after each scenario. If the questionnaire would have been only given at the end, it would have been hard for the users to remember the details about each method, for all scenarios. This fact was also uncovered during one of the pilot sessions.

Reiterating, feedback has been captured using several methods:

- Questionnaire
 - Likert scale questions (quantitative feedback)
 - Descriptive questions (more qualitative feedback)
- Questions during the experiment, related to the task at hand. The user was also encouraged to think aloud.

To summarize, the following steps were performed during the experiment:

- Before starting, a certain configuration was picked for methods A and B(e.g.: A compensates, B does not) as well as for the scenarios(choosing a row from the latin square)
- To begin, the users were given an introduction page to read up some basic information
- Then, the cloth display, the distortion issue and gestures were introduced to the participant
- The scenarios were started in a certain order, and the participant kept switching between methods A and B(by using the mouse attached to the frame), in order to compare them

- While the participant was performing a certain task, some interview-like questions were asked, and I noted down important information
- After each scenario, the participant was given a short questionnaire regarding that particular scenario only

In the following sections, each scenario will briefly be described in addition to the other documents used in the study, including the questionnaire. Due to three of the scenarios being very similar, they will all be explained in one section.

6.1.1 Grid, Text and Columns Scenarios

These scenarios all consisted of displaying a static image: of a grid, some text, and a number of columns in a scene, respectively. For each of these scenarios the users had to share their thoughts on methods A and B, for three different points in the image(see figure 47). For these scenarios, they only used one finger interactions to deform the display at the indicated points(by using the index finger). Only one point was shown at a time, and it was made invisible after the participant remembered where the point was positioned.

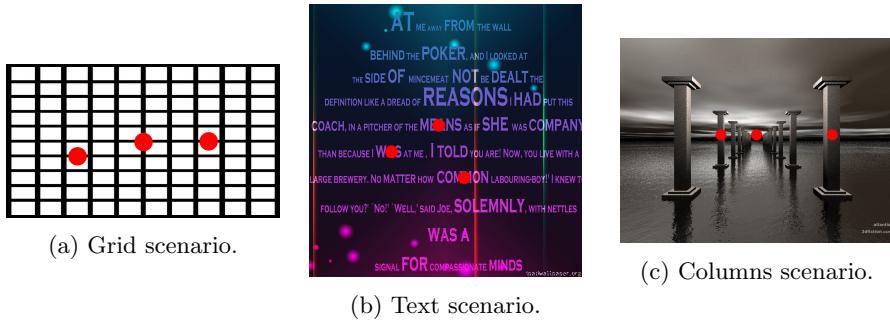


Figure 47: Grid, Text and Column Textures. Red dots indicate interaction areas.

As you can see, the dots are not in the same positions, except for the center. This probably affects the results to some extent, since the compensation seems worse as one advances towards the sides of the image. However, the points were placed with the texture contents in mind - for example, having the red circles on the columns, or on a particular word, for the respective scenarios. The three points were coded: *left*, *center* and *right*(with left meaning left-most and similarly for right). Even though the left and right points are not in the same positions, the same patterns were observed for all three scenarios, as presented in section 7.

Finally, for each scenario, users were told to look at some particular aspect of the current image, in order to give their opinion on how distorted the image is, when compared to how it would look on a flat display:

- For the grid, users were asked how straight the black lines appear with each method
- For the text, users were asked to look at the space between the lines, readability, and how similar the word appears to one on a flat screen
- For the columns, users were asked to look at how straight the columns appear after pushing into the display

6.1.2 Map Scenario

The map scenario simply consisted of a static map texture of a region of *New York City* (see figure 48). Users were asked to push into the display, using two hands. Further, they were also instructed to focus on the central area of the display. Then, the users were asked to compare the proportions and readability of the image, with both methods. Further, users were also instructed to focus on some particular name or landmark to focus on, as this makes it easier to compare.

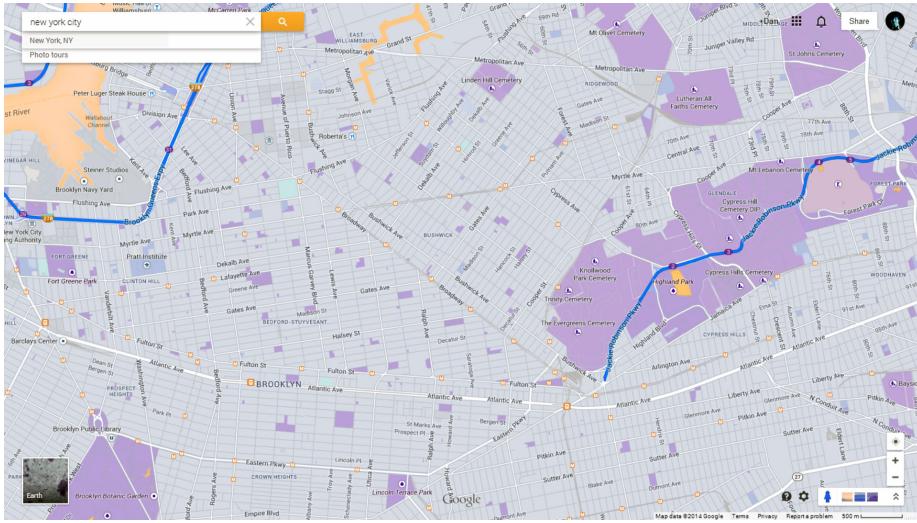


Figure 48: A map showing a region of New York City. Courtesy of Google Maps.

Note that the colors in the image are not the default ones. This is because, initially, methods A and B were also visually different. However, by using the mouse as input, this was no longer necessary, but one of the images which was already loaded into the application was used.

6.1.3 CT Brain Scan Scenario

This scenario consists of a number of textures, representing volume data for a brain scan. The texture set was taken from the *Stanford volume data archive* (available

at [34]). Users had to use two-hand push gestures in order to advance, or two-hand pull gestures to reverse, through the *slices*(textures). The starting texture was in the middle of the dataset. Again, the objective was to compare methods A and B. Since the brain is changing shape throughout the slices and the speed is too fast, it can be hard to see the differences. Several users complained about this, and, if they could not see a difference, they were instructed to compare the methods for the last image(see figure 49), using the push gesture.

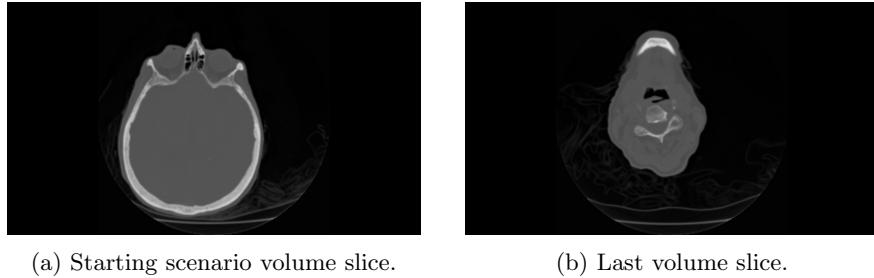


Figure 49: CT scan of a brain. Example of two different textures(slices). Courtesy of [34]

Volume Slice Interaction

The implementation of the interaction for this scenario is naive, with only the depth value of the center point being considered. Therefore, the users could actually use any gesture that moves the display in the center area to change the slices. The two-hand gestures were used because, with one hand, the user might obstruct the object. Both the compensated method and the normal texture are interactive(only one texture has been used for the normal image in this scenario, because the noise is much less apparent).

For the compensated method, we get the initial depth value, given the current modlview and projection matrices, as well as the resolution:

```
initDepth = getGLDepth(1920 / 2, 1080 / 2, mModelView, frustum)
```

This method reads the pixel depth value from the back buffer(with double buffering, drawing commands are typically directed to the back buffer, while the front buffer is used for drawing), at position (x, y) using:

```
glReadBuffer(GL_BACK);
glReadPixels(x, y, 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &depth_z);
```

The result is stored in `depth_z`. However, our depth value must be converted from screen space coordinates back to metric space coordinates by unprojecting:

```
glm::vec3 final = glm::unProject(glm::vec3(x, y, depth_z),
mModelView, frustum, glm::vec4(0, 0, 1920, 1080));
```

The `glm::unProject` method works similarly to how the process has been described in section 4.2.3. Further on, we simply compute the depth difference between frames, and if the value is above or below a certain threshold, the active texture is changed by incrementing, or decrementing a counter, respectively.

For the texture, we cannot access the depth values from OpenGL since our matrices are different and, in this case, the texture is projected on a quad, within the default coordinate space. Therefore, one can just retrieve the raw depth value from the middle of the frame data, in the *RawDepthPipeline* class, when we create the point cloud:

```
if (x == depthCamWidth / 2 && y == depthCamHeight / 2)
centerDepth = depthValue;
```

This value can then be retrieved and converted to metric units (by dividing with 1000, since the depth frame data is in millimeters). Finally, the textures are changed in the same manner as for the compensated method.

6.1.4 The Questionnaire

The questionnaire is composed of several parts. On the first page, the participants were given some basic information about the structure of the questionnaire and they were also asked for demographic data. This page was given to the users after completion of the first scenario, with an additional page with the questions for that particular scenario - questions for each scenario were given on a separate page. Finally, there were some common question in the end, which asked the participant to rate method A and B overall.

The questions mainly related to the following attributes:

- Perceived difference between the two methods
[Not Different - Very Different]
- Preservation of appearance(proportions) for each method
[Very Poor - Very Good]
- Method preference [A - B]

The preservation of appearance is related to the resemblance to the original image, as it would appear on a flat display (how similar are the relative proportions, with respect to the original image?). Next, the subjective preference is asked for, because it is possible that a user does not like a method, even if it works correctly - qualitative feedback has been obtained in order to determine if this is the case. Lastly, the attribute that is being rated is shown in brackets in the section above (the ends of the scale).

To be more concrete, the questionnaire contained the following items:

- Difference between the two methods
- Please explain in what way were the methods different
- Preservation of appearance during deformation - Method A

- Preservation of appearance during deformation - Method B
- Method preference for this scenario
- Please explain why you have preffered the selected method

In addition, the grid, text and columns scenarios had two additional questions, regarding how consistent the behavior was, in the different areas of the display - for the three tested points(e.g.: a method could work very well in the center, but have poor results on the sides):

- Method A's behavior was consistent, for all points
[Not Consistent - Consistent]
- Method B's behavior was consistent, for all points
[Not Consistent - Consistent]

An 11-point Likert scale has been made available for users to rate all the attributes. The scale starts at 0 and ends at 10; it has been originally adapted from the QUIS 10-point scale(examples at [10]). An 11-point scale was used instead, so that the user also has a neutral option otherwise they would have been forced to prefer one of the methods.

7 Results

7.1 Experimental Study Results

The user study consisted of 11 experiments on the deformable cloth display. As described in 6, each participant tested both the corrected method, and a normal image projection. The two methods have been alternated between participants the first had method A as the corrected and B as the normal image, while the second had A as the normal and B as the corrected, etc. Further, the experiment consisted of 5 scenarios, which have also been shown to the participants in a different order, so that the data is not biased for a particular scenario configuration. Finally, smoothing functions were applied on the corrected method in order to reduce noise, but, these were not enough, so noise was added to the normal method, by capturing 10 frames of the corrected method and replaying them in a loop. This meant that the users could not easily identify the method due to visual artifacts, because they were present in both methods.

Three of the participants had seen the display before the experiment. However, a strong bias due to the previous reason is unlikely, since they had not seen the actual experiments/scenarios before.

Since the feedback is of both quantitative and qualitative nature(questionnaire + interview questions), both shall be used in order to draw conclusions on the outcome of the experiment.

In order to analyze the quantitative results, t-tests have been run for both the consistency and preservation of appearance questions. These tests help to determine the statistical significance of that particular test.

Further, t-tests have also been run on the preference question as well as on the overall preservation of appearance question. However, it is very important to note that there was only one question related to the previous items, where the user had to compare between methods A and B on an 11-point Likert scale from 0 to 10. In order to get some meaningful results, the scale has been converted, so that both the corrected and the normal method could be rated from 0 to 5(e.g. 0 represents 5 for method A, while 10 represents 5 for method B, and 5 represents 0). The values were separated into corrected/normal categories and the missing values were replaced with 0. A bias is probably introduced due to the previous reason. However, it is likely that a bias could appear in the degree to which a method is preferred, but it is much less likely to affect the overall result(e.g.: if 80

Moreover, the differences are simply shown in a pie chart.

The qualitative data has been grouped in a few categories(e.g.: distorted, preserves appearance) as the users were stating similar things in many different ways. Only the qualitative data regarding the center, left and right points for the grid, text and columns scenarios and the qualitative data in the overall case, have been analyzed. Most of the qualitative feedback for the three previously mentioned scenarios could be grouped by the three points - it may have been confusing to look at how the scenario was said to behave over all of the points, since in many cases the results differed quite a lot between them(for example,

if a user stated that method A for scenario 1 was distorted twice, and not distorted once; however, feedback such as - "it was not distorted in the center, but distorted to the left or right" is much more useful).

In the following, the full analysis will be described.

7.1.1 Quantitative Analysis

Preservation of Appearance

There are a total of 5 pairs that have been tested for significance using a paired t-test. Each pair corresponds to one of the scenarios, and several useful measures including the mean for each case are shown in table 2. The values were computed from the raw data with both a spreadsheet and IBM SPSS(details at [5]) in order to verify their correctness.

Paired Samples Statistics					
		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	CorrectedGrid	5.3636	11	1.80404	.54394
	NormalGrid	4.1818	11	1.77866	.53629
Pair 2	CorrectedText	6.0909	11	2.02260	.60984
	NormalText	5.2727	11	1.73729	.52381
Pair 3	CorrectedColumns	5.8182	11	2.04050	.61523
	NormalColumns	5.0909	11	1.70027	.51265
Pair 4	CorrectedMap	7.2727	11	1.42063	.42834
	NormalMap	3.4545	11	1.91644	.57783
Pair 5	CorrectedBrainScan	6.5455	11	2.42337	.73067
	NormalBrainScan	4.8182	11	2.13627	.64411

Table 2: Characteristics of pair data for preservation of appearance.

The output of the t-test is shown in table 3. Note that a green color coding has been used for pairs which are statistically significant, while a red color coding indicates that the null hypothesis could not be rejected for those cases. Since, the confidence interval is 95%, we need a p-value(or Sig. In the table) of ≤ 0.05 to have statistical significance. The last pair has not been color coded, since its value is still > 0.1 , which probably suggests a trend. An increase in the number of participants will most likely improve results, at least for pair 5.

From this test we can conclude that, for the Map Scenario, the difference in preservation of appearance is statistically significant. Moreover, it is possible that we have a trend for the Brain Scan scenario, but all other cases are insignificant(a larger user pool may improve results).

Further, the actual results are shown via a bar chart in figure 50. All scenarios have the mean shown for both corrected and normal methods. We can

Paired Samples Test						
		Paired Differences 95% Confidence Interval of the Difference	t	df		
					Upper	Sig. (2-tailed)
Pair 1	CorrectedGrid - NormalGrid	2.90297	1.530	10	.157	
Pair 2	CorrectedText - NormalText	2.64120	1.000	10	.341	
Pair 3	CorrectedColumns – NormalColumns	2.89569	.747	10	.472	
Pair 4	CorrectedMap - NormalMap	5.15568	6.361	10	.000	
Pair 5	CorrectedBrainScan - NormalBrainScan	3.83232	1.828	10	.097	

Table 3: T-test results for preservation of appearance.

also see visually, that the highest differences are in the Map Scenario, with the corrected method being rated much higher. In addition, the vertical error bars show the standard error, and it can be used as another measure for statistical significance by checking if the bars from the two different methods overlap(see [37] for more details on this procedure). We can see that the only scenario for which there is no overlap is Map. Although this does not tell us if the results are statistically significant, it does tell us that all the other cases, are not(when there is an overlap, we can be sure that $p > 0.05$ and the results are insignificant).

To conclude this part, both the table and the chart indicate that the only statistically significant scenario is Map. The table tells us that the result is actually significant, while the chart can also be used to rule out the other cases, as well as providing a helpful visual indicator of the degree of difference for the two methods, for preservation of appearance.

Overall Preservation of Appearance

A separate question was asked for overall preservation of appearance. This case has not been included previously because the results are on a different scale. All other questions about preservation of appearance were asked twice, once for each method. However, there is only one question for the overall case, which asked the participants to rate the preservation of appearance on an 11-point Likert scale from 0 to 10, where 0 denoted the maximum preference for method A, while 10 denoted the maximum preference for method B, with 0 as a neutral option. The results were mapped to $[0,5]$ as mentioned previously, at the beginning of this section. Returning to our t-test, the result is shown in table 5. This time we only have one pair, for which we show aggregated data measures(see table 4).

As mentioned before, the missing values were replaced with 0 when running the t-test(the user was forced to select a preference for either method A or B).

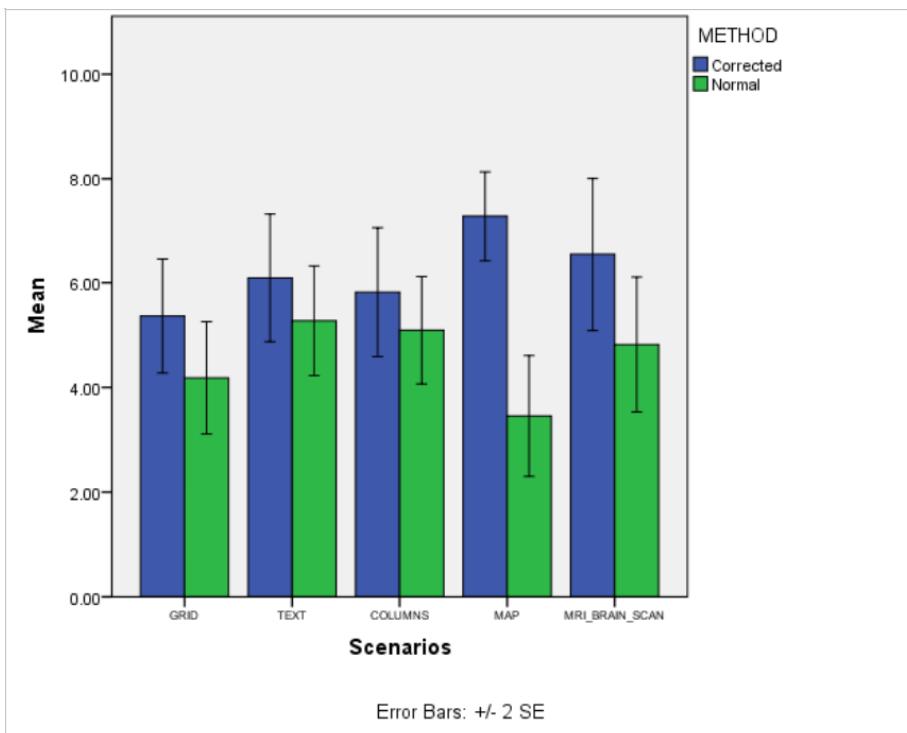


Figure 50: Mean of preservation of appearance, per scenario.

Paired Samples Statistics					
		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	CorrectedOverall	2.4545	11	1.03573	.31228
	NormalOverall	.1818	11	.60302	.18182

Table 4: Characteristics of pair data for overall preservation of appearance.

The results are statistically significant, with a value comfortably below the 0.05 threshold.

Paired Samples Test					
	Paired Differences				
		95% Confidence Interval of the Difference			
		Upper	t	df	Sig. (2-tailed)
Pair 1	CorrectedOverall - NormalOverall	3.31742	4.847	10	.001

Table 5: T-test results for overall preservation of appearance.

Finally, the graph in figure 51 shows the contrast for the results between the two methods. We can see that the preference was clearly for the corrected method, though the degree of preference is maxed somewhere between 2 and 3, out of a maxim of 5, which suggests that, even though the method did help to correct the image so that it appears as on a flat display, there still is room for improvement. Removing some of the visual artifacts, as well as calibrating the system to ensure that the compensation is almost exactly right, should improve the degree to which participants prefer the corrected method.

Moreover, the standard error bars do not overlap, which indicates that the test may be statistically significant. In conjunction with the previous table, we can conclude that there is a substantial, statistically significant preference for the corrected method, albeit with a relatively modest degree of preservation.

Another helpful visual tool is the area graph shown in figure 52. Now we can more clearly see that only one participant preferred the normal method of projecting images onto the deformable display.

Consistency

The consistency refers to the behavior of the method in different areas of the display, and it has only been tested for three scenarios: Grid, Text and Columns, where the participants had to interact with three different points placed towards the center. Left and right side of the screen. The samples are shown below:

Next, the t-test results are shown in the following table. Note that all values have been color coded red. This is to indicate that the consistency test between methods is not significant for any scenario.

The graph below show the results visually. We can see that we got similar values for all scenarios. Further, the standard error bars overlap for all cases, which also indicates inconclusive results.

From the previous analysis it seems that the quantitative consistency tests did not provide significant results. However, the results are quite similar between scenarios and the normal projection method has generally been deemed more consistent, as we could see in the previous chart.

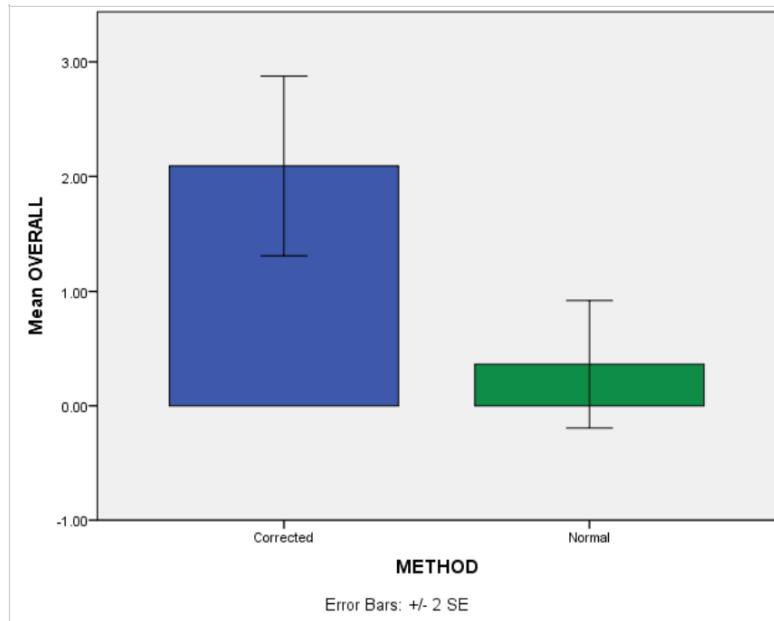


Figure 51: Mean of preservation of appearance, overall.

Paired Samples Statistics					
		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	CorrectedGrid	4.5455	11	2.42337	.73067
	NormalGrid	6.0000	11	2.40832	.72614
Pair 2	CorrectedText	5.0909	11	2.11918	.63896
	NormalText	6.2727	11	2.68667	.81006
Pair 3	CorrectedColumns	5.0909	11	2.84445	.85763
	NormalColumns	6.3636	11	2.54058	.76601

Table 6: Characteristics of pair data for consistency.

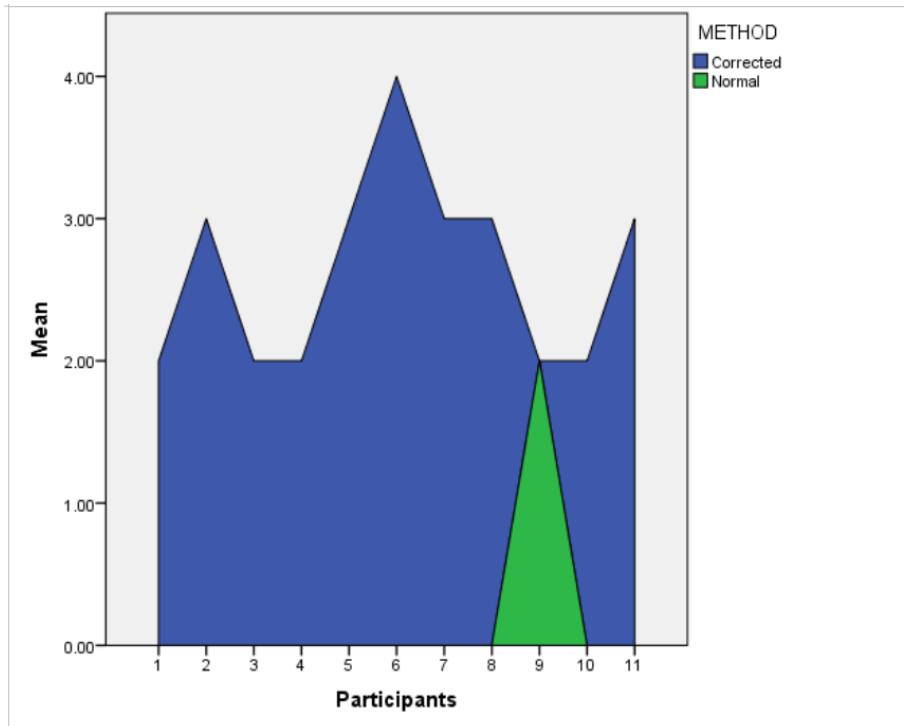


Figure 52: An area chart showing the perceived overall preservation of appearance, for both methods and all participants.

Paired Samples Test						
		Paired Differences	95% Confidence Interval of the Difference	t	df	Sig. (2-tailed)
		Upper				
Pair 1	CorrectedGrid - NormalGrid	1.02958				.221
Pair 2	CorrectedText - NormalText	1.04263				.264
Pair 3	CorrectedColumns - NormalColumns	1.93640				.398

Table 7: T-test results for consistency.

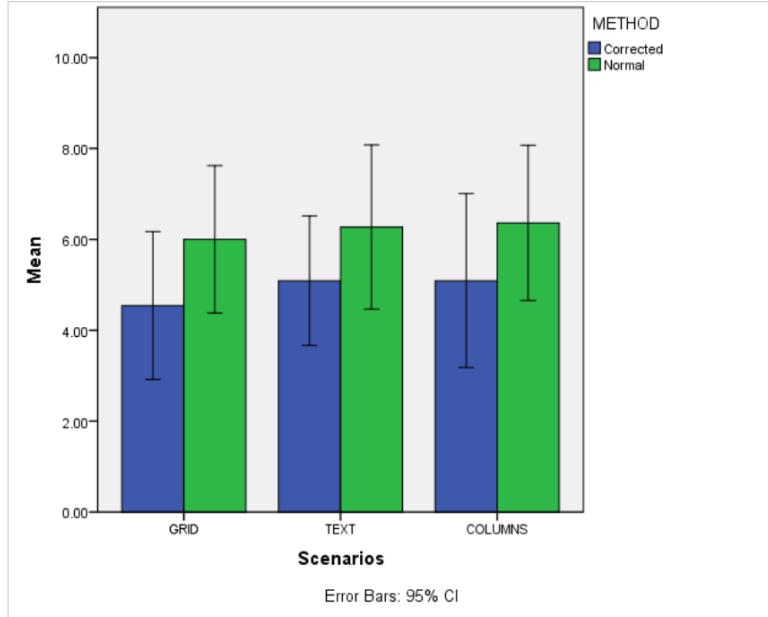


Figure 53: Consistency mean, per scenario.

Finally, qualitative measures may provide more information on consistency, and whether there really was a perceived difference between the three interaction points. We shall look at this aspect in one of the subsequent sections.

Preference

Similarly to the previous cases, the sample statistics used for the t-test are shown in table 8. We can see that in this case we have 6 pairs, 1 for each scenario, and one for the overall case. This is because there was only one combined preference question, where the user had to rate between method A and B on an 11-point scale, with 0 representing strong preference for A and 10 representing strong preference for B.

The results of the t-test can be observed in table 9. Again, rows have been color coded red if the results were not statistically significant for that pair. We observe that we again have the map scenario and the overall case as significant.

The graph in figure 54 shows all the preference ratings, color coded by method and clustered on scenarios. From the standard error bars, we see that only the Map and Overall cases may be statistically significant, since the bars do not overlap for these cases.

Moreover, we can also see the rather large preference for the corrected method, both for the Map and the Overall cases. This analysis indicates that, for the previous two cases, the corrected method has been preferred to a far

Paired Samples Statistics					
		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	CorrectedGrid	1.2727	11	1.27208	.38355
	NormalGrid	.4545	11	.82020	.24730
Pair 2	CorrectedText	.8182	11	1.07872	.32525
	NormalText	1.0909	11	1.57826	.47586
Pair 3	CorrectedColumns	1.5455	11	1.50756	.45455
	NormalColumns	.9091	11	1.37510	.41461
Pair 4	CorrectedMap	3.1818	11	1.60114	.48276
	NormalMap	.2727	11	.90453	.27273
Pair 5	CorrectedBrainScan	1.6364	11	1.43337	.43218
	NormalBrainScan	.6364	11	1.28629	.38783
Pair 6	CorrectedOverall	2.0909	11	1.30035	.39207
	NormalOverall	.3636	11	.92442	.27872

Table 8: Characteristics of pair data for method preference.

Paired Samples Test					
		Paired Differences	t	df	Sig. (2-tailed)
		95% Confidence Interval of the Difference			
		Upper			
Pair 1	CorrectedGrid - NormalGrid	2.08640	1.437	10	.181
Pair 2	CorrectedText - NormalText	1.31964	- .382	10	.711
Pair 3	CorrectedColumns - NormalColumns	2.44583	.784	10	.451
Pair 4	CorrectedMap - NormalMap	4.45438	4.195	10	.002
Pair 5	CorrectedBrainScan - NormalBrainScan	2.64559	1.354	10	.206
Pair 6	CorrectedOverall - NormalOverall	3.10705	2.789	10	.019

Table 9: T-test results for method preference.

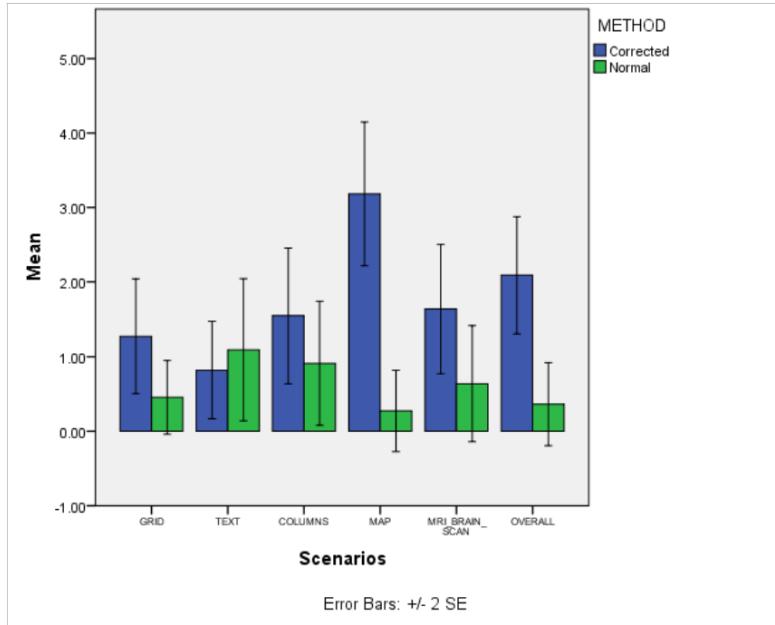


Figure 54: Means of method preferences, per scenario and overall.

greater extent than the other one, and the results are statistically significant. In order to better illustrate how participants rated their preference for the overall case, the area graph in figure 55 can be consulted. We can see that only 2 participants preferred the normal method, while 9 preferred the corrected method. Further, one of the ratings for normal is of only 1 point, which may suggest that the normal method was preferred not because it was inherently better, but because there are some problems with the other.

Difference

One of the items on the questionnaire asked users whether they can see a difference between the corrected and normal methods. The means of the results are shown in the pie chart in figure 56, for each scenario. We can see that the map scenario had the highest perceived differences, while the brain scan scenario had the lowest. For the latter, a problem was that the shape was changing too fast as the user was interacting, which made it hard to make out differences. However, in order to solve this issue, the user was then told to only compare the methods for the last slice in the set, which remained static.

The mean values are approximately in the [4.2, 7] interval. This shows that at least moderate differences were observed between the two methods.

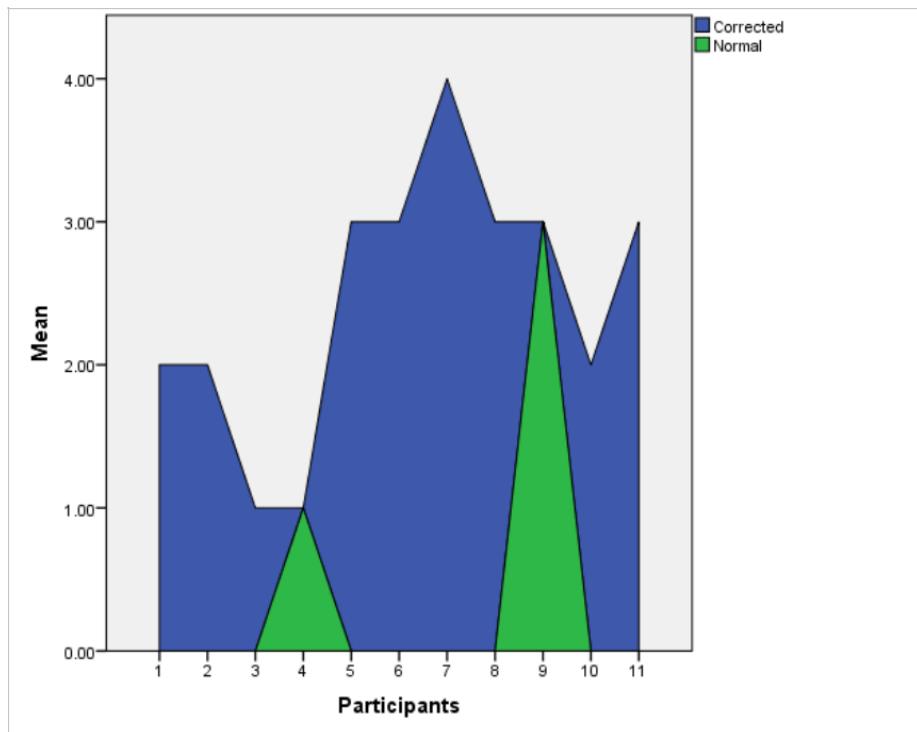


Figure 55: An area chart showing the perceived overall method preference.

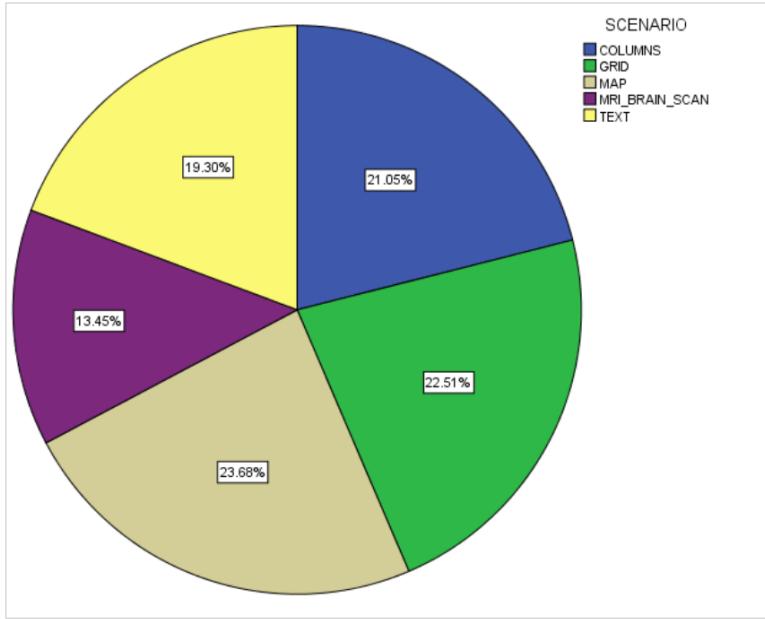


Figure 56: Mean of perceived differences, per scenario.

7.1.2 Qualitative Analysis

Qualitative data has been retrieved with two methods:

- Descriptive questions about all of the previous attributes: consistency, difference, preference, preservation of appearance.
- Interview-like questions asked while the user was interacting. Some scenario specific questions were introduced, in order to make the user think how well the image is preserved.

A substantial amount of data has been collected for each scenario. However, due to the nature of this experiment, participants described their experience using different words, even in the face of very similar questions. This makes it difficult to obtain meaningful measures for each scenario. Further, in some cases, for example, when the participants were stating that something is distorted, they repeated the test for each point(for the scenarios with three interaction points), which means that just knowing how many participants stated that a method is distorted is not as relevant as also knowing for which point the image was distorted. For example, the center point could have worked very well, but maybe, due to a high number of distortions on one of the sides, the overall scenario would have been ranked as distorted. Therefore, we start by analyzing the distortions around the interaction points, from the interview. Naturally, this is only applicable to the three scenarios which consisted of a task for each point(shown as a red circles in the scenarios)

Center Point

From the graph in figure 57 we can observe that the corrected method was preserving the appearance much more often than the normal one. In addition, the normal method also appeared distorted to users a much larger amount of times. Finally, the *Follow the Finger* measure was also in favor of the corrected method, but there were only few results and not as significant. This last measure refers to the user's finger being in the sample place after deforming the display, which would normally happen if the image correctly compensates for the distortion. However, in the center area of the display it is more difficult to tell the difference, unless a more significant force is applied to the display.

The three qualitative measure do not necessarily appear under the same name in the survey results. For example, if a method was moving the image to the left, it was added as distortion for the respective method; or, if the grid lines appeared straighter that note was added to preservation of appearance. Otherwise, there would have been too many different measures, and it would have been hard to analyze and classify the data.

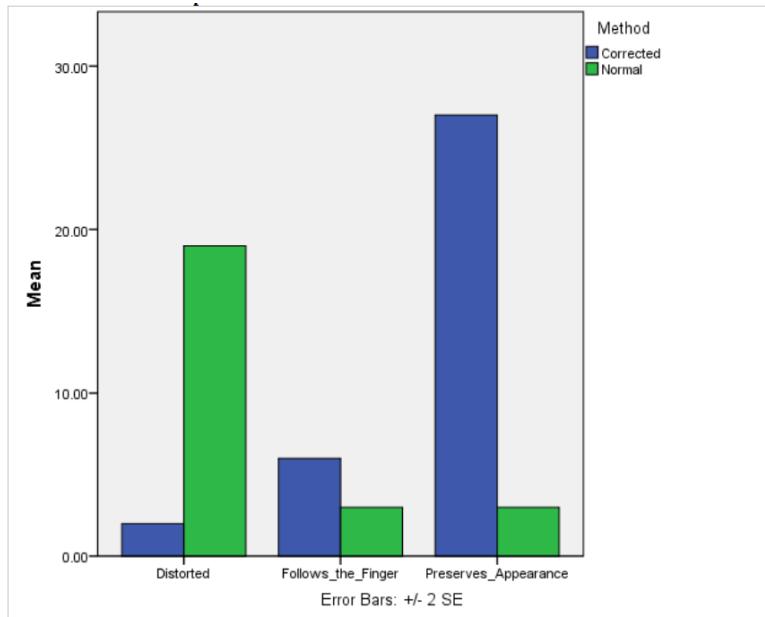


Figure 57: Results for the center point.

Left Point

For this point we almost see a reversal of the distortion and preservation of appearance measures, while the other remained similar, as we can see in figure

58. This happens because the setup is not calibrated, and the depth camera is not 100% orthogonal to the display. This introduces misalignments near the sides(which means the compensation will take place in a wrong area with an offset; several users complained of distortions underneath the finger), and these can be larger if the camera is tilted toward one side(even if only slightly).

Even though the results were much poorer for this point, we also have a moderate amount of candidates for preservation of appearance for the corrected method, in contrast to the previous point, where the differences were more pronounced between the methods.

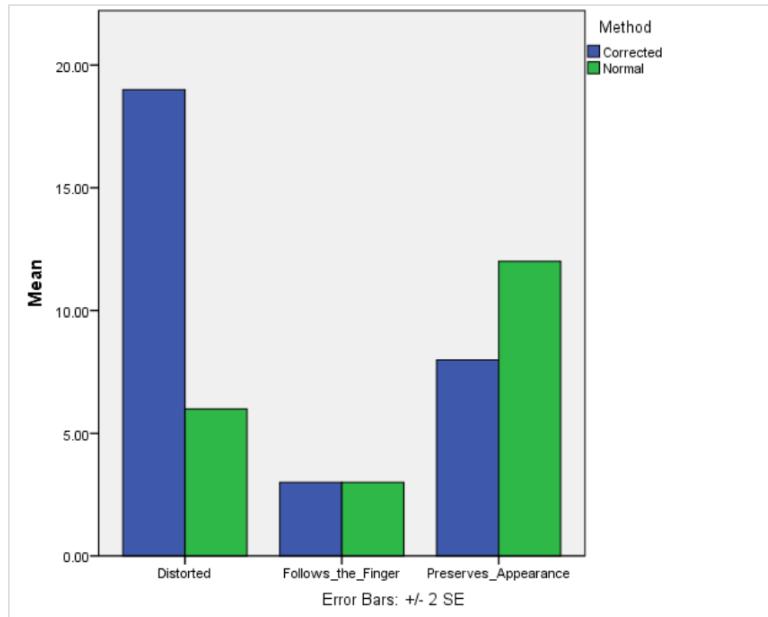


Figure 58: Results for the left-most point.

Right Point

For this point, we interestingly get a large number of preservation of appearance statements for the corrected method, while also getting large numbers of distortions, for both methods, as seen in figure 59. This effect can maybe be explained by the fact that users felt the corrected method introduces a different kind of distortion. Therefore, even though there was compensation, which helped made the image look more like it would on a flat display, some parts of the image were distorted in some way(e.g.: underneath the finger).

Overall

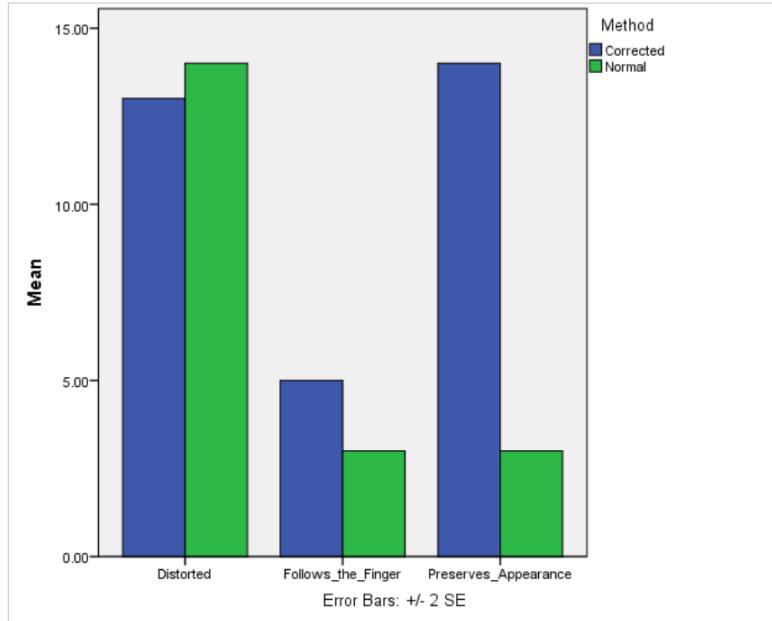


Figure 59: Results for the right-most point.

Finally, some qualitative measures regarding overall differences and preference are shown in the graph below.

Better simply represents which method participants thought is subjectively better. There is no item for the normal method, which may indicate that participants didn't necessarily feel the normal method was more appropriate, but simply that the other method has some issues. Next, we only have a few entries for distortion, but we can see that the normal method has a slightly larger number of distorted results. Finally, there is a very substantial difference between preservation of appearance of the two methods, with almost all participants preferring the corrected method.

7.2 Comparison of Compensated and Uncompensated Images

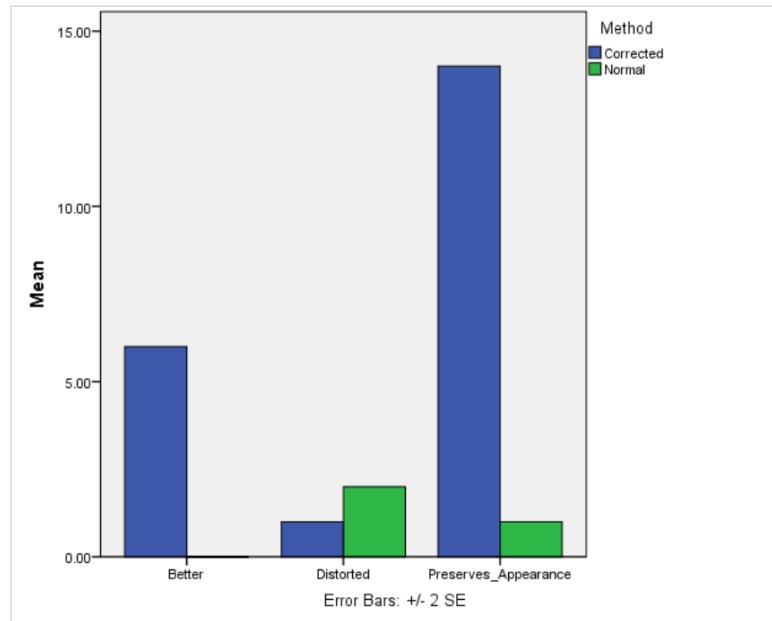


Figure 60: Qualitative results for the overall case, grouped in a few categories.

8 Conclusions

9 Future Work

References

- [1] Assimp: Open asset import library. <http://assimp.sourceforge.net/>.
- [2] Glew: The opengl extension wrangler library. <http://glew.sourceforge.net/>.
- [3] Glfw. <http://www.glfw.org/>.
- [4] Glm: Opengl mathematics. <http://glm.g-truc.net/0.9.5/index.html>.
- [5] Ibm spss. <http://www-01.ibm.com/software/analytics/spss/>.
- [6] Intel perceptual computing sdk. <https://software.intel.com/en-us/vcsource/tools/perceptual-computing-sdk/home>.
- [7] Open asset import library. <http://freeimage.sourceforge.net/>.
- [8] Opengl tutorials. <http://www.mbstworks.sk/index.php?page=tutorials&series=1>.
- [9] Point cloud library. <http://pointclouds.org/>.
- [10] Questionnaire for user interation satisfaction. <http://lap.umd.edu/quis/>.
- [11] Dan Avram. Projective texture mapping via shaders. <http://stackoverflow.com/questions/22732717/opengl-projective-texture-mapping-via-shaders>.
- [12] Paul Bourke. Polygonising a scalar field. May 2013.
- [13] Cyril Crassin. Opengl geometry shader marching cubes. January 2007.
- [14] Daalsgard and Halskov. *3D Projection on Physical Objects: Design Insights from Five Real Life Cases*. CHI 2011, Aarhus University, Denmark.
- [15] dan.lecocq. *Triangle Strip for Grids - A Construction*. <http://dan.lecocq.us/wordpress/2009/12/25/triangle-strip-for-grids-a-construction/>.
- [16] Gabriel Taubin Douglas Lanman. *Build Your Own 3D Scanner: 3D Photography for Beginners*. SIGGRAPH 2009 Course Notes.
- [17] Christopher Dyken and Gernot Ziegler. High-speed marching cubes using histogram pyramids. 2007.
- [18] Berg et al. *Computational Geometry: Algorithms and Applications, Third Edition*. Springer, 2008.
- [19] Falcao et al. *Plane-based calibration of a projector-camera system*. VIBOT Master 2008.

- [20] M. Sung et al. Image unprojection for 3d surface reconstruction: A triangulation-based approach. 2013.
- [21] Pindat et al. *JellyLens: Content-Aware Adaptive Lenses*. UIST - 25th Symposium on User Interface Software and Technology, 2012.
- [22] Radhwan et al. *KOSEI: A KINECT OBSERVATION SYSTEM BASED ON KINECT AND PROJECTOR CALIBRATION*. QPSR of the numediart research program, Vol. 4, No. 4, December 2011.
- [23] Raskar et al. *The Office of the Future: A Unified Approach to Image-Based Modeling and Spatially Immersive Displays*. SIGGRAPH 98, Orlando, Florida, 2008.
- [24] Richard S. Wright Jr. et al. *OpenGL SuperBible, Fifth Edition: Comprehensive Tutorial and Reference*. Addison-Wesley, 2011.
- [25] S. Burak Gokturk et al. A time-of-flight depth sensor - system description, issues and solutions.
- [26] Segal et al. *Fast Shadows and Lighting Effects Using Texture Mapping*. Computer Graphics 26, 2 July 1992.
- [27] Watanabe et al. *955-fps Real-time Shape Measurement of a Moving/Deforming Object using High-speed Vision for Numerous-point Analysis*. IEEE International Conference on Robotics and Automation, Roma, Italy, 2007.
- [28] Watanabe et al. *The Deformable Workspace: a Membrane between Real and Virtual Space*. IEEE International Workshop on Horizontal Interactive Computer Systems, 2008.
- [29] Yamazaki et al. *Simultaneous Self-Calibration of a Projector and a Camera using Structured Light*. In Proceedings Projector Camera Systems 2011, National Institute of Advanced Industrial Science and Technology, Japan, June, 2011.
- [30] Cass Everitt. *Projective Texture Mapping*. NVIDIA.
- [31] Andreas Jordt and Reinhard Koch. Fast tracking of deformable objects in depth and colour video. BMVC 2011.
- [32] Andreas Jordt and Reinhard Koch. *Direct Model Based Tracking of 3D Object Deformations in Depth and Color Video*. Int J Comput Vis, September, 2012.
- [33] Mark J. Kilgard. Modern opengl usage: Using vertex buffer objects well. September 2009.
- [34] Marc Levoy. The stanford volume data archive. <http://graphics.stanford.edu/data/voldata/>.

- [35] William E. Lorensen and Harvey E. Cline. Marching cubes: a high resolution 3d surface construction algorithm. July 1987.
- [36] Zoltan Csaba Marton, Radu Bogdan Rusu, and Michael Beetz. On Fast Surface Reconstruction Methods for Large and Noisy Datasets. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Kobe, Japan, May 12-17 2009.
- [37] Dr. Harvey Motulsky. Graphpad software: The link between error bars and statistical significance. http://egret.psychol.cam.ac.uk/statistics/local_copies_of_sources_Cardinal_and_Aitken_ANOVA/errorbars.htm.
- [38] Jon Macey Navpreet Kaur Pawar. Surface reconstruction from point clouds. August 2013.
- [39] Jongtae Park. A fast, small-radius gpu median filter.
- [40] PCL. Fast triangulation of unordered point clouds.
- [41] Max Roth. Metaballs - rendering a scalar field with marching cubes. May 2013.
- [42] Flexpad Steimle et al. *Highly Flexible Bending Interactions for Projected Handheld Displays*. CHI 2013, Paris, France.

A Glossary

B Additional Images

C Empirical Study Documents