TRUE 3D DIGITAL HOLOGRAPHIC TOMOGRAPHY FOR VIRTUAL REALITY APPLICATIONS

Thesis

Submitted to

The School of Engineering of the

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree of

Master of Science in Electro-Optics

By

Alexander Downham

UNIVERSITY OF DAYTON

Dayton, Ohio

December 2017
TRUE 3D DIGITAL HOLOGRAPHIC TOMOGRAPHY FOR VIRTUAL REALITY APPLICATIONS

Name: Downham, Alexander, David

APPROVED BY:

Partha P. Banerjee, Ph.D.
Advisor
Advisory Committee Chairman
Chairperson
Electro-Optics and Photonics

Paul McManamon, Ph.D.
Committee Member
Technical Director, Ladar and Optical Communications Institute
Electro-Optics and Photonics

Joseph Haus, Ph.D.
Committee Member
Professor
Electro-Optics and Photonics

Ron Ewart
Committee Member
Lead Scientific Engineer
Protobox LLC

Robert J. Wilkens, Ph.D., P.E.
Associate Dean for Research and Innovation, Professor
School of Engineering

Eddy M. Rojas, Ph.D., M.A., P.E.
Dean
School of Engineering
ABSTRACT

TRUE 3D DIGITAL HOLOGRAPHIC TOMOGRAPHY FOR VIRTUAL REALITY APPLICATIONS

Name: Downham, Alexander D.
University of Dayton

Adviser: Dr. Partha P. Banerjee

In previous work, a sparse object has been reconstructed using digital holographic tomography by recording the holograms with an ‘on-axis’ system, while the object is rotated about a single axis. The recorded holograms are processed through MATLAB® using the multiplicative technique. In this work an ‘off-axis’ Mach-Zehnder system is used to record holograms of each side of a die, by rotating the object along the x and y axes. Then the holograms of all six sides of the die are imported and reconstructed in MATLAB® using Fresnel techniques. The depth information of each side of the die is obtained by using phase unwrapping techniques. The unwrapped depth information is then converted into an $N \times 3$ array, and then into a point cloud image. Finally, the true three-dimensional image is constructed using point cloud image processing techniques. Once the true three-dimensional image is constructed, it is exported to a Microsoft HoloLens. The true three-dimensional image is now viewable in virtual reality while still containing the depth
information, which is difficult to see with current virtual reality techniques used today, along with the capability of using hand gestures for image manipulation.
ACKNOWLEDGEMENTS

I would like to thank Dr. Partha P. Banerjee for the use of his laboratory and equipment. I would also like to thank him for the time he put in answering questions and editing my thesis and the ensuing paper. I am very appreciative of Dr. Ujitha Abeywickrema’s help in setting up the various hologram recording systems, and for answering questions when reconstructing the holograms. I would like to thank everyone at Protobox LLC for sponsoring my graduate research during my master’s program, and for allowing me to use the Microsoft HoloLens as the display system for viewing the final three-dimensional reconstruction. Finally, I am thankful to all my friends and family for their continued support when working through the master’s program. Without everyone’s help, I would not have been able to complete my thesis and my graduate degree.
TABLE OF CONTENTS

ABSTRACT...........................................................................................................iii

ACKNOWLEDGEMENTS .......................................................................................v

LIST OF FIGURES.................................................................................................xi

LIST OF ABBREVIATIONS.....................................................................................xvi

LIST OF SYMBOLS...............................................................................................xvii

CHAPTER 1: INTRODUCTION................................................................................1

CHAPTER 2: THEORETICAL BACKGROUND...........................................................5

2.1 Introduction......................................................................................................5

2.2 Digital Hologram Recording.........................................................................5

2.3 Fresnel Reconstruction..................................................................................9

2.4 Non-Paraxial Reconstruction.......................................................................13

2.5 Phase Unwrapping......................................................................................14

2.6 SHOT-MT for Tomographic Reconstruction.............................................15

2.7 Conclusion.....................................................................................................16
5.1 Introduction........................................................................................................49
5.2 Experimental Setup..........................................................................................50
5.3 Experimental Results.......................................................................................51
  5.3.1 Hologram Reconstruction............................................................................52
  5.3.2 Phase-Unwrapping....................................................................................53
  5.3.3 Point Cloud Reconstruction......................................................................54
  5.3.4 Hololens Implementation..........................................................................55
5.4 Conclusion..........................................................................................................56

CHAPTER 6: CONCLUSION..................................................................................59
REFERENCES.........................................................................................................61
APPENDIX A: MATLAB Code..............................................................................63
  A.1 Non-MATLAB Functions...............................................................................63
    A.1.1 myFresnel..............................................................................................63
    A.1.2 myLoftAndRotate..................................................................................66
    A.1.3 myDisplay3d..........................................................................................69
    A.1.4 saveobjmesh..........................................................................................71
    A.1.5 puma_ho...............................................................................................72
  A.2 Spring Reconstruction Code..........................................................................78
    A.2.1 Fresnel Reconstruction..........................................................................78
    A.2.2 Non-Paraxial Reconstruction..................................................................84
  A.3 Die Reconstruction.........................................................................................90
    A.3.1 Hologram Reconstruction......................................................................90
    A.3.2 Phase Unwrapping.................................................................................97
<table>
<thead>
<tr>
<th>Section</th>
<th>File Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.2.3 ObjectSurfaceObserver.cs</td>
<td></td>
<td>206</td>
</tr>
<tr>
<td>B.3 Main Camera Scripts</td>
<td></td>
<td>208</td>
</tr>
<tr>
<td>B.3.1 ZoomOutGUI.cs</td>
<td></td>
<td>208</td>
</tr>
<tr>
<td>B.3.2 ZoomInGUI.cs</td>
<td></td>
<td>208</td>
</tr>
<tr>
<td>B.4 Cursor Script</td>
<td></td>
<td>209</td>
</tr>
<tr>
<td>B.4.1 CursorManager.cs</td>
<td></td>
<td>209</td>
</tr>
<tr>
<td>B.5 Object Scripts</td>
<td></td>
<td>210</td>
</tr>
<tr>
<td>B.5.1 GestureManipulator.cs</td>
<td></td>
<td>210</td>
</tr>
<tr>
<td>B.5.2 TapToPlace.cs</td>
<td></td>
<td>214</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1: Diagram of the Gabor setup................................................................. 7
Figure 2.2: Mach-Zehnder interferometer............................................................ 8
Figure 2.3: Diagram of the modified Mach-Zehnder interferometer...................... 9
Figure 2.4: Diagram showing the relation between the object plane, hologram plane,
and the image plane ......................................................................................... 10
Figure 3.1: Gabor setup used to generate the holograms using a spring as the object.
The holograms are produced using a 514.5 nm Lexel 2 − 7 Watt
TEM$_{00}$ Ar- ion laser. The beam is collimated using a 50cm collimating
lens. M1 and M2 are mirrors used to reflect the beam in the
appropriate direction. The object beam and the reference beam travel
along the same optical path to the CCD....................................................... 19
Figure 3.2: Recorded holograms using the Gabor setup.................................. 20
Figure 3.3: Reconstruction using (a) Fresnel, (b) non-paraxial transfer function
techniques at 0 degrees rotation................................................................. 21
Figure 3.4: Tomographic reconstruction of a pen spring for a recording distance of
9.5 cm, using (a) the Fresnel reconstruction approach, (b)
non-paraxial approach. Tomographic reconstruction for a recording
distance of 19.5 cm, using (c) the Fresnel reconstruction approach, (d) non-paraxial approach. Tomographic reconstruction for a recording distance of 29.5 cm, using (e) the Fresnel reconstruction approach, (f) non-paraxial approach.

Figure 3.5: Reconstructed hologram of a pen spring. Distance is 19.5 cm @ 0 degrees of rotation using the (a) Fresnel approximation. (b) non-paraxial approach.

Figure 3.6: Reconstructed hologram of a pen spring. Distance is 29.5 cm @ 0 degrees of rotation using (a) the Fresnel approximation. (b) non-paraxial approach.

Figure 3.7: Fresnel reconstruction of a pen spring. Recording distance of 29.5 cm, using (a) 5 holograms at 40-degree increments, (b) 9 holograms at 20 degree increments, (c) 19 holograms at 10-degree increments, (d) 18 holograms at 20 degree increments. (e) 36 holograms at 10-degree increments.

Figure 4.1: Mach-Zehnder interferometer setup used to generate the holograms for the die. The holograms are produced using a 514.5 nm Lexel 2 – 7 Watt TEM$_{00}$ Ar- ion laser. The beam is collimated using a 50cm collimating lens. M1, M2, and M3 are mirrors used to reflect the beam in the appropriate directions. BS1 represents the beam splitter that is used to split the main beam into a reference beam and a second beam which illuminates the object. BS2 represents the beam splitter used to project the interference of the reference beam and the object beam onto
the CCD camera……………………………………………………………………..31

Figure 4.2: Tomographic reconstruction of the 8.4mm die. The reconstruction process is done using myFresnel() and myLoftAndRotate() in a fashion similar to the reconstruction process in Chapter 3 [5]……………….32

Figure 4.3: Fresnel reconstruction (intensities only shown) of (a) side one, (b) side two, (c) side three, (d) side four, (e) side five, (f) side six. The recording distance is set to be 34.75cm w/ the scaling factor applied…………………………………………………………………………34

Figure 4.4: Surface plots of the phase-unwrapped reconstruction side one using PUMA: (a) side one, (b) side two, (c) side three, (d) side four, (e) side five, (f) side six. The x and y axes are measured by the pixel value, and the z axis represents the converted phase information into depth. The depth information z is measured in meters…………………………………..36

Figure 4.5: Point cloud reconstruction of die surfaces after smoothing process. (a) side one, (b) side two, (c) side three, (d) side four, (e) side five, and (f) side six. Side six has been inverted for future reconstruction purposes. All die faces have been smoothed to remove any curvature. There will still be some curvature due to the accuracy of the computer……………………………39

Figure 4.6: Die reconstruction using the point cloud method. The reconstruction is missing the edges due to the process used to generate the holograms for this reconstruction…………………………………………………………………41

Figure 4.7: Final die reconstruction, including the addition of the stitched edges showing, for example, sides (a) one, five and two, (b) one, three and
Figure 4.8: (a) Blender™ reconstruction of the die containing the surface normals needed for the use in the Unity application. (b) Die reconstruction once the Blender™ reconstruction has been imported into Unity and run in an application on the HoloLens. The blue circle in (b) is the courser of the HoloLens interacting with the box collider surrounding the die.

Figure 5.1: Experimental setup for recoding holograms of the die. A regular Mach-Zehnder interferometer is used with one arm as the object and other arm as the reference. A 505.5 nm and 596.5 nm Ar-ion laser is used as a source and collimated using a 50 cm convex lens after spatial filtering. Holograms are recorded using a CCD. M1, M2, M3 represent mirrors and BS1 and BS2 represent beam-splitters. MO and PH represent microscope objective and pinhole, respectively. Holograms are recorded for all sides of the dime and the intensity and phase are reconstructed using Fresnel propagation techniques. Each wavelength were fired separately.

Figure 5.2: Reconstruction of the dime (a) front and (b) back using a wavelength of 505.5 nm and a recording distance of 50-cm, (b) dime back. Also shown are reconstructions of (a) front and (b) back using a wavelength of 505.5-nm and the same recording distance.

Figure 5.3: (a) PUMA surface plot of the dime front, (b) back side.

Figure 5.4: (a) Front of the dime reconstructed in point cloud, (b) back side. Due to
the flattening and the color mapping in MATLAB®, some of the details
seen in Figure 5.2 are not distinguishable.

Figure 5.5: Final reconstruction of dime (a) front and (b) back in Blender™. The
same reconstructions in Hololens (c) front, (d) back.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D</td>
<td>Three dimensional</td>
</tr>
<tr>
<td>2D</td>
<td>Two dimensional</td>
</tr>
<tr>
<td>BS</td>
<td>Beam splitter</td>
</tr>
<tr>
<td>CCD</td>
<td>Charged coupled device</td>
</tr>
<tr>
<td>DH</td>
<td>Digital holography</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete cosine transform</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>IDCT</td>
<td>Inverse discrete cosine transform</td>
</tr>
<tr>
<td>M</td>
<td>Mirror</td>
</tr>
<tr>
<td>MT</td>
<td>Multiplicative technique</td>
</tr>
<tr>
<td>ND</td>
<td>Neutral density</td>
</tr>
<tr>
<td>RTT</td>
<td>Radon transform technique</td>
</tr>
<tr>
<td>RHS</td>
<td>Right hand side</td>
</tr>
<tr>
<td>SHOT-MT</td>
<td>Single beam holographic tomography multiplicative technique</td>
</tr>
</tbody>
</table>
LIST OF SYMBOLS

$a$  amplitude

$\varphi$  phase

$E$  optical field

$I$  intensity

$h$  hologram function

$d$  recording distance

$m,n$  pixel coordinate in reconstruction plane

$k,l$  pixel coordinate in hologram plane

$\lambda$  wavelength

$\Delta \nu, \Delta \mu$  spatial frequency

$\Delta x, \Delta y$  pixel size

$k$  wave propagation/number

$z$  reconstruction distance

$H$  Fourier transform of hologram

$z_R$  Rayleigh range
Holography was originally invented by Gabor in 1948. Holography was used to record and reconstruct the amplitude and phase of a given optical field. Gabor named the process of recording and reconstructing the amplitude “holography” \[1,2\]. Although holograms can be recorded by simulating a spatially coherent source, further progress in holography did not occur until the development of the laser. Traditional holograms record the interference pattern between the scattered wave off the object and the reference wave. A three-dimensional (3D) object can have its information recorded onto a two-dimensional (2D) film which is the hologram. When the hologram is illuminated by a reading wave which could be the same as the reference wave that is used for recording, 3D features of the object can be reconstructed.

In the original Gabor setup, the object wave and the reference wave traveled along the same optical path. The Gabor setup is considered to an “in-line” system for recording holograms of semi-transparent objects. With this setup, reflection and dc noise can make its way into the recording. This can cause errors in the reconstruction process. A different holographic recording process needed to be developed to allow for the elimination of these defects. Leith and Upatnieks developed an “off-axis” holography system. In this setup,
the optical wave is split into a reference wave and what finally becomes an “object” wave. Once this wave scatters off the object, the object and reference waves are made to travel noncollinearly and the interference pattern is recorded onto a medium. Since these two waves travel separate from one another, when recording the object’s hologram, a separate recording of the reference wave can be used in later reconstruction process to remove any dc component in the object hologram. Using this method, the reconstruction results are greatly improved [3]. However, the off-axis system suffers from the fact that the fringes or spatial frequencies resulting from interference of noncollinear object and reference beams also need to be properly recorded to spatially separate the dc and the twin images during reconstruction. This can be an issue when recording the hologram using a charged coupled device (CCD) since it depends on the pixel size and spacing.

In digital holography (DH), a coherent light source is used to generate a hologram that is conveniently recorded onto a CCD. Unlike analog holography, DH has no film to process, but it has its own shortcomings. One of the biggest issues is the resolution that can be achieved through DH is limited by the CCD camera. A typical CCD camera has a resolution of approximately 1000 lines/mm, which is much less that photographic film [4]. Also, in DH, phase-unwrapping becomes important in the reconstruction process, which is now numerically performed and mimics the forward and back propagation of the optical fields immediately behind the illuminated hologram. When phase-unwrapping a reconstruction, the phase information is limited between a modulo of \(2\pi\) (\(-\pi\) to \(\pi\)). The phase of a reconstruction gives the depth information of a given object. There are several phase unwraping algorithms that are available, the most common are IDCT (Inverse Discrete Cosine Transform) and PUMA [5].
There are several different mathematical algorithms that have been used to reconstruct digital holograms. The Fresnel reconstruction technique [6,7] and the convolution approach are some of the more commonly used methods [6] in DH reconstructions. The non-paraxial transfer function approach has also been used in DH reconstructions, and is more accurate than the transfer function approach [3]. The Fresnel and the non-paraxial techniques are discussed later in the thesis.

It has been stated that holography records the amplitude, phase, and depth information of a 3D object. However, reconstruction only contains the information from the recorded scattered wave, which is generated from only the part of the object that is illuminated. To help generate a true 3D reconstruction, 2D holograms from all different illumination angles need to be reconstructed and combined to generate the 3D reconstruction [2,8]. This process is known as tomography. The multiplicative technique (MT) and Radon-transform technique (RTT) are techniques used in tomography to derive and display true 3D information [8]. These methods are discussed later in the thesis.

In this thesis, the theory of DH reconstruction, phase-unwrapping, and multiplicative techniques in tomography are discussed in Chapter 2. In Chapter 3, the Fresnel and the non-paraxial techniques are simulated in MATLAB® and compared to one another. Later in Chapter 3, the quality of the tomographic reconstruction is investigated as a function of the number of holograms needed to create an accurate reconstruction. Chapter 4 introduces a new reconstruction technique, using the point cloud, to reconstruct objects and maintain depth information. The reconstruction is imported into the Microsoft® HoloLens to view the image in a true 3D format and in a virtual reality environment.
Chapter 5 introduces multi-wavelength digital holographic tomography as a more accurate way to record digital holograms. Chapter 6 concludes the thesis and discusses future work.
2.1 Introduction

In this Chapter, the theoretical background of digital holography, holographic tomography, and the various reconstruction processes are introduced. In Section 2.2, the mathematical background behind recording holograms is described, along with a few recording techniques. Section 2.3 and Section 2.4 discuss some of the different reconstruction techniques that can be used for reconstruction in DH, such as the discrete Fresnel approximation and the “non-paraxial” propagation/transfer function method.

2.2 Digital Hologram Recording

Before the use of digital holograms, holograms had to be recorded using a photographic film in an optical system that included a collimated light source, typically a laser [2]. This coherent beam is split into two separate beams by a beam splitter. The first beam is used to illuminate the object and the second beam illuminates the recording medium directly. This beam is also known as the reference beam. The two waves interfere and are recorded onto the medium. The final recorded interference pattern is known as a hologram. A generic holographic recording system is shown in Figure 2.1.
To reconstruct the object from the hologram, the hologram is illuminated by the reference wave of the recording system. With the hologram illuminated by the reference wave, the observer then sees a virtual image of the recorded object [2]. The reconstructed image will exhibit all the effects of perspective and depth of focus [2]. A generic system used to reconstruct the hologram can be seen in Figure 2.2.

The recording process of the hologram can be represented mathematically using the participating complex optical fields. The optical field of the object wave can be described as

$$E_o(x, y) = a_o(x, y)e^{i\varphi_o(x, y)}$$  \hspace{1cm} (2.1)

with the $a_o$ being the real amplitude and $\varphi_o$ as the phase, while

$$E_R(x, y) = a_R(x, y)e^{i\varphi_R(x, y)}$$  \hspace{1cm} (2.2)

is the complex electric field of the reference beam with $a_R$ denoting the amplitude and $\varphi_R$ representing the phase. The optical fields represented by Equations (2.1) and (2.2) interfere on the surface of the recording medium. The total intensity, proportional to the hologram amplitude transmittance function, can be calculated as [2]

$$I(x, y) \propto h(x, y) \propto |E_o(x, y) + E_R(x, y)|^2$$

$$= (E_o(x, y) + E_R(x, y))(E_o(x, y) + E_R(x, y))^*$$

$$= E_R(x, y) E_R^*(x, y) + E_o(x, y) E_o^*(x, y)$$

$$+ E_R(x, y) E_o^*(x, y) + E_o(x, y) E_R^*(x, y).$$  \hspace{1cm} (2.3)

The expanded equation contains two dc terms, ($E_R(x, y) E_R^*(x, y)$ and $E_o(x, y) E_o^*(x, y)$), and the real and virtual images, $E_R(x, y) E_o^*(x, y)$ and $E_o(x, y) E_R^*(x, y)$, respectively.
There are various experimental setups that can be used to generate digital holograms. Two of these designs are discussed here, viz., the Gabor setup and the modified Mach-Zehnder setup.

The Gabor setup is an “on-axis” set up, shown in Figure 2.1. The incident beam passes over the object; the incident beam also acts as the reference field. Light that passes over the object then begins to scatter, generating the object field. This field depends on the characteristics of the object. The reference field and the object field interfere with one another and are picked up by the CCD camera. The resulting interference pattern generates a holographic image of the object. Opaque objects record differently, the resulted scattered field is that of the edges of the object.

The Mach-Zehnder setup can either be considered an “on-axis” or “off-axis” depending on the setup. The traditional Mach-Zehnder has the object is set at an angle $\theta$, shown in Figure 2.2, where $2\theta$ is the nominal angle between the light illuminating the object and the scattered light. Light from a laser passes through a beam splitter BS1,
allowing for the beam to be split into two beams. The first beam is considered to be the reference field and the other, once it hits the object, is considered as the object field. The reference field passes through BS1, hits the mirror M1, and is reflected into the second beam splitter BS2, where it partially reflects and falls on the CCD camera. The second beam is reflected from BS1, hits the object at an angle $\theta$, and is then reflected into BS2. The two fields after BS2 interfere and are collected on the CCD camera. To eliminate the $dc$, the reference field intensity must be deducted from the overall hologram captured by the CCD. The setup is called on-axis if the angle between the scattered light and the reference beam is nominally equal to zero.

There are some disadvantages when using the Mach-Zehnder setup. Due to the tilt in the object, once the incident light hits the object’s surface, the angle of incidence has to be taken into account in the reconstruction. Another concern with the angle of the object are possible unwanted shadows caused by the light not hitting normal to the surface of the object.

![Mach-Zehnder interferometer](image)

*Figure 2.2: Mach-Zehnder interferometer.*
To eliminate the angle of the object and the shadowing effect, a modified Mach-Zehnder setup can be used, shown in Figure 2.3. Using this modified Mach-Zehnder setup does come with one major drawback. Multiple reflections generated by the mirrors and beam splitters can affect the resulting holograms generated before the light can interfere on the CCD, thereby contaminating the final results.

![Diagram of the modified Mach-Zehnder interferometer.](image)

2.3 Fresnel Reconstruction

In this Section, a reconstruction technique via the Fresnel transform is shown. Figure 2.4 represents the typical coordinate system for the object, hologram and image planes.
When reconstructing a hologram, the diffracted light can be represented by the Fresnel-Kirchhoff integral, shown by [2]:

\[
E'(x', y') = \frac{1}{\lambda} \iint_{-\infty}^{\infty} h(x, y) E^*_R(x, y) \frac{\exp\left[-i \frac{2\pi \rho}{\lambda}\right]}{\rho} \, dx \, dy
\] (2.4)

where

\[
\rho = \sqrt{(x - x')^2 + (y - y')^2 + d^2}
\] (2.5)

represents the distance between the different points on the hologram plane and the reconstruction plane. \( \lambda \) is the wavelength of the beam used to generate and reconstruct the hologram, \( d \) is the recording distance, and \( E^*_R(x, y) \) is the optical field of the reconstruction beam, taken to be the complex conjugate of the recording field, for convenience. In most cases, the \( E \)-field is considered to be a plane wave, therefore the value is a constant value.

The expression in Equation (2.5) can be expanded through a Taylor series [2]:

\[
\rho = d + \frac{(x-x')^2}{2d} + \frac{(y-y')^2}{2d} - \frac{1}{8} \frac{[(x-x')^2+(y-y')^2]^2}{d^3} + \ldots
\] (2.6)
When looking at the expanded equation above, the fourth term on the RHS may be neglected if it is much smaller than the wavelength:

\[
\frac{1}{8} \left[ \frac{(x-x')^2+(y-y')^2}{d^3} \right] \ll \lambda, \quad (2.7)
\]

or

\[
d \gg \sqrt{\frac{1}{8} \left[ \frac{(x-x')^2+(y-y')^2}{d^3} \right]}. \quad (2.8)
\]

With the fourth term neglected, Equation (2.6) can be simplified as:

\[
\rho = d + \frac{(x-x')^2}{2d} + \frac{(y-y')^2}{2d} \quad (2.9)
\]

Using Equation (2.9), further approximations can be made to the overall Fresnel-Kirchhoff integral in Equation (2.4). This results in the final expression, known as the Fresnel approximation or the Fresnel transform, expressed as [2,7]

\[
E'(x',y') = \frac{j}{\lambda d} \exp \left(-j \frac{2\pi}{\lambda} d \right) \exp \left[-j \frac{\pi}{\lambda d} (x'^2 + y'^2) \right] \\
\times \iint_{-\infty}^{\infty} E^*_R(x,y) h(x,y) \exp \left[-j \frac{\pi}{\lambda d} (x^2 + y^2) \right] \exp \left[j \frac{2\pi}{\lambda d} (xx' + yy') \right] dx dy \quad (2.10)
\]

The intensity can be found by taking the absolute value and squaring it:

\[
I(x',y') = |E'(x',y')|^2. \quad (2.11)
\]

The phase of the reconstructed optical field can be obtained using

\[
\phi(x',y') = \arctan \frac{\text{Im}[E'(x',y')]}{\text{Re}[E'(x',y')]} \quad (2.12)
\]
In DH, the hologram function \( h(x, y) \) is recorded onto a rectangular CCD camera that contains \( N \times N \) pixels with pixel sizes \( \Delta x \) and \( \Delta y \) in the horizontal and vertical direction respectively. Recognizing that the integral in Equation (2.10) resembles a Fourier transform integral, the function \( E' \) can be converted into a discrete form using spatial frequency coordinates as:

\[
E'(m, n) = \frac{j}{\lambda d} \exp \left( -j \frac{2\pi}{\lambda} d \right) \exp \left[ -j\pi\lambda d (m^2 \Delta v^2 + n^2 \Delta \mu^2) \right] \times 
\sum_{k=0}^{N-1} \sum_{l=0}^{N-1} E_R^*(k, l) \exp \left[ -j \frac{\pi}{\lambda d} (k^2 \Delta x^2 + l^2 \Delta y^2) \right] \exp \left[ j2\pi (km \Delta x \Delta v + ln \Delta y \Delta \mu) \right]
\] (2.13)

for \( m = 0,1, ..., N-1; n = 0,1, ..., N-1 \). Using the theory of discrete Fourier transformation, the following relationship exists between \( \Delta x, \Delta y \) and the \( \Delta \nu, \Delta \mu \) [2]:

\[
\Delta \nu = \frac{1}{N_x \Delta x}; \quad \Delta \mu = \frac{1}{N_y \Delta y}
\] (2.14)

To return to spatial coordinates as required in Equation (2.10), the following substitutions are needed:

\[
\Delta x' = \frac{\lambda d}{N_x \Delta x}; \quad \Delta y' = \frac{\lambda d}{N_y \Delta y}.
\] (2.15)

Using Equation (2.15),

\[
E'(m, n) = \frac{j}{\lambda d} \exp \left( -j \frac{2\pi}{\lambda} d \right) \exp \left[ -j\pi\lambda d \left( \frac{m^2}{N_x^2 \Delta x^2} + \frac{n^2}{N_y^2 \Delta y^2} \right) \right] \times 
\sum_{k=0}^{N-1} \sum_{l=0}^{N-1} E_R^*(k, l) h(k, l) \exp \left[ -j \frac{\pi}{\lambda d} (k^2 \Delta x^2 + l^2 \Delta y^2) \right] \exp \left[ j2\pi \left( \frac{km}{N_x} + \frac{ln}{N_y} \right) \right]
\] (2.16)

where \( m \) and \( n \) represent the coordinates for the pixels in the reconstruction plane, and \( k \) and \( l \) are the coordinates of the pixels for the hologram plane. The components within the sum represents the discrete Fresnel transform, this function can easily be processed using
a computer. The factor located in front of the sum is only prevalent when the phase of the reconstruction is taken into account, otherwise it can be ignored.

2.4 Non-Paraxial Reconstruction

The Fresnel technique discussed above is not the only method for reconstruction in DH. Another method for the reconstruction of holograms is to use the non-paraxial transfer function technique, also known as the angular spectrum. Consider that the complex field is nominally traveling in the z-direction, and the hologram is located at $z = 0$ [3,7]. Upon uniform illumination, the complex field immediately behind the hologram is proportional to $h(x, y; 0)$. The angular spectrum of this field is found by taking its spatial Fourier transform:

$$H(k_x, k_y; 0) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(x, y; 0) \exp[-j(k_x x + k_y y)] dx dy, \quad (2.17)$$

where $k_x$ and $k_y$ denote spatial frequencies. Upon multiplying this spectrum with the non-paraxial transfer function

$$\exp[-jz\sqrt{k_0^2 - k_x^2 - k_y^2}], \quad (2.18)$$

the angular spectrum after a distance $z$ of propagation is found. The last step is to take the inverse Fourier transform to find the complex reconstructed field in the spatial domain:

$$E(x', y'; z) = \mathcal{F}^{-1}\{H(k_x, k_y; 0) \exp[-jz\sqrt{k_0^2 - k_x^2 - k_y^2}]\}. \quad (2.19)$$

Propagation by positive and negative values of $z$ can reconstruct the real and virtual images, respectively.
2.5 Phase Unwrapping

Once a hologram has been numerically reconstructed, the phase (or depth information) can be retrieved from the reconstructed optical field [5]. However, all phases computed from the complex field using the \( \arctan \) function can only vary by a modulo of \( 2\pi \) [5]. Therefore, phase unwrapping is needed to determine the absolute phase or the depth information. There are two common methods for finding the unwrapped phase of a reconstruction, the discrete cosine transform (DCT) or PUMA.

When using the DCT method, it is required that a rectangular aperture is recorded with the boundaries parallel to the CCD’s pixel coordinates [9]. This allows the DCT to be taken within selected region [9]. The “reconstruction region” is determined based on the propagation distance and the wavelength that is used to generate the hologram. Since in this method the DCT is applied to the entire hologram without cropping any region, this process is straightforward and easy to use [9]. Unfortunately, because this method prefers the whole hologram, it becomes memory intensive and requires a fair amount of processing power depending on the amount of detail within the reconstruction.

Another algorithm used to find the unwrapped phase of the reconstructed field is known as PUMA. PUMA, with the combination of MATLAB® software, has been developed by Bioucas-Dias and Valadao [5]. PUMA casts the unwrapping as an energy minimization via graph mincut calculations [5]. Unlike DCT, PUMA does not have to be applied on the entire reconstructed field or image. The image can be cropped to the desired size, as along as the modified image is square. If after modifying the image, the pixel values along the horizontal and vertical axes do not match, the process will fail. Since the images can be modified/cropped, the memory usage is not as intense as with using the DCT.
method, thereby allowing the process to be ran at a faster rate. For the purpose of this thesis, PUMA was used to unwrap the phase of the reconstructions to save on memory usage.

2.6 SHOT-MT for Tomographic Reconstruction

In DH, the interference between a reference and the diffracted light from the object is picked up by a CCD camera [4]. If the recording is done in a Mach-Zehnder setup, the reconstruction can only reveal information about the surface being illuminated. Such surface reconstructions can be considered to be pseudo-3D at best. To generate a truer 3D image of an object, multangular tomography is used. With multangular tomography, several different holograms are constructed of the object, with each hologram containing information of the illuminated face.

One method used for multangular tomography is known as single-beam holographic tomography multiplicative technique or SHOT-MT. In SHOT-MT experiments done earlier, in-line digital holograms of small objects, e.g., droplets, have been recorded, with each hologram corresponding to a specific angular orientation of the object [8]. The angles can be chosen as $\theta_m = 0, 15^\circ, 30^\circ, 45^\circ, ..., \theta_m = 1, 2, 3, 4, ...$ identifying the angles. Usually higher number of angles yield higher accuracy, but a minimum of four different angles in a plane have been used to generate a tomographic reconstruction [8]. The images taken at each specific angle are reconstructed at a distance $z$ from the hologram plane, equal to the recording distance. This method of tomographic reconstruction can be used both for the Gabor (in-line) and Mach-Zehnder setups described in Section 2.2. The object to be recorded is placed on a rotating stage. An in-line hologram is captured at each specified angle. Each hologram is numerically reconstructed to the corresponding angle. Sometimes, depending on the situation, rotating the object to gather
the multiple images can have an undesirable effect. By rotating the object, a time delay between acquiring the holograms can result in error in the reconstruction, particularly for moving objects. To avoid this, multiple CCD cameras have been used, and set at the desired angles, and a single beam is engineered to simultaneously illuminate the object at different angles, and the diffraction patterns are picked up by multiple CCD cameras [8].

There are some technical details that need to be followed for SHOT-MT to work successfully, this is described below. First, the pseudo-3D reconstruction needs to be represented in a binary format, where the threshold represents the profile at the specific angles. Once the reconstructions are represented as a binary format, the pseudo-3D images are lofted into a 3D volume and rotated by the corresponding angle. The lofted and rotated 3D reconstructions are then multiplied together using the function

\[ I_{3D} = \prod_{m=1}^{N} I_m \]  

where \( N \) is the number of all recorded angles, \( m \) is the order of the reconstruction to the corresponding angle, and \( I_m \) is the lofted and rotated pseudo-3D reconstructions.

2.7 Conclusion

In this Chapter, holographic construction and reconstruction techniques have been discussed. The two common reconstruction techniques that were discussed, the Fresnel reconstruction technique and the transfer function approach. When discussing the fundamentals of tomographic reconstruction, multiple images taken at different angles are used to form a 3D reconstruction of the object. In Chapter 3, different reconstruction techniques will be applied and compared experimentally.
3.1 Introduction

In the previous Chapter, different holographic construction and reconstruction techniques have been discussed. Holographic construction techniques include the Gabor and the Mach-Zehnder setups. Reconstruction of the holograms can be done using the Fresnel method and the non-paraxial transfer function approach. The purpose of this Chapter is to report on preliminary experiments done to gain familiarity with holographic construction and reconstruction and tomography. To this end, the Gabor set up is used to generate a hologram. Multiple holograms are taken at different angles. The reconstruction of the holographic images is done by the use of the Fresnel and the transfer function techniques. The results are then compared to one another. For evaluating the final tomographic reconstruction, the quality of the final 3D reconstruction as a result of the number of angles/holograms used is studied. The comparison results show that for a
specific range of recording distances, the Fresnel technique is identical to the non-paraxial technique, in agreement with published results [8].

3.2 Experimental Setup

In our experiments, the Gabor in-line set up is first used, as shown in Figure 3.1. To generate the holograms, a 514.5 nm Lexel 2 – 7 Watt TEM\textsubscript{00} Ar- ion laser is used as the source. The light is reflected from mirror M1 and introduced into the spatial filter. The spatial filter consists of a 40x microscope objective and a 25-nm pin hole. The emerging light is then collimated using a collimating lens, and illuminates the object after reflection through mirror M2 and passing through a neutral density (ND) filter. The object is placed on a rotating stage with variable degree increments. The object is placed in the center of the rotating stage, and is set a distance, \(d\), away from the CCD camera behind the object. During numerical reconstruction, the reconstruction distance, ideally equal to \(d\), can be incrementally changed to find the ideal location for the best resolution of the reconstructed hologram.
Figure 3.1: Gabor setup used to generate the holograms using a spring as the object. The holograms are produced using a 514.5 nm Lexel 2 – 7 Watt TEM$_{00}$ Ar- ion laser. The beam is collimated using a 50cm collimating lens. $M_1$ and $M_2$ are mirrors used to reflect the beam in the appropriate direction. The object beam and the reference beam travel along the same optical path to the CCD.

3.3 Experimental Results

Figure 3.2 is a typical hologram recording when using the Gabor setup. When looking at the recorded hologram, it is clear that there is unwanted noise in the image. Some of the noise, can be the result of reflections within the glass of the CCD camera. Speckle and effects from spurious vibrations can also make their way into recorded hologram, and there are no modifications that can be made to the setup that will eliminate some of these issues including eliminating the $dc$ noise. However, by recording the reference without the object, and subtracting this from the hologram, the $dc$ and some of
the noise can be removed before the final reconstruction of the object. For best results, the reference should be recorded immediately after the hologram has been recorded.

![Image of hologram]

*Figure 3.2: Recorded holograms using the Gabor setup.*

The spring that has been used has a thickness $t \approx 450 \mu m$ (corresponding to a “Rayleigh range” of $z_R = \frac{\pi t^2}{\lambda} \sim 0.618 m$). Recording distances of $d = 10 \, cm$, $20 \, cm$ and $30 \, cm$ have been used. The object used for this experiment is rotated at 10-degree increments about the vertical axis through 360 degrees, giving a total of 36 different recorded holograms. Figure 3.2 above is a recorded hologram of the spring used in this experiment with zero-degree rotation, and a recording distance of 10 cm. The hologram has been reconstructed using the Fresnel and the non-paraxial transfer function techniques; the results are shown in Figures 3.3(a,b), respectively.
3.3.1 Fresnel and Non-Paraxial Reconstruction Results

An ideal tomographic reconstruction shows clear and sharp edges that clearly define the object. It has been visually ensured that before using the tomography algorithm that each reconstruction is in focus. When reconstructing the various holograms, the actual distance needed for the highest clarity may differ from the measured value. This results from the inaccuracy of measuring the actual optical path length for each hologram during recording. Therefore, the reconstruction distance for each hologram has to be manually adjusted to ensure that each reconstruction is in focus. To increase processing speed and preventing memory overrun, the scale of each of the holograms has been reduced to 35% of the original hologram. Figures 3.4(a-f) show the 3D tomographic reconstructions of the spring using the Fresnel approach and the non-paraxial transfer function approach. The reconstruction distances have been nominally set at 9.5 cm, 19.5 cm and 29.5 cm respectively.
Figure 3.4: Tomographic reconstruction of a pen spring for a recording distance of 9.5 cm, using (a) the Fresnel reconstruction approach, (b) non-paraxial approach. Tomographic reconstruction for a recording distance of 19.5 cm, using (c) the Fresnel reconstruction approach, (d) non-paraxial approach. Tomographic reconstruction for a recording distance of 29.5 cm, using (e) the Fresnel reconstruction approach, (f) non-paraxial approach.
The first set of holograms that have been used to generate the tomographic reconstructions in Figure 3.4(a) and 3.4(b) have been recorded at a distance of 9.5 cm (the corresponding reconstructions at 0 degrees are shown in Figures 3.3 (a,b)). A total of 19 rotations have been used to generate the final tomographic picture; more on the number of angles needed is in the next sub-section. Comparing these tomographic reconstructions to the ones for distances of 19.5 cm and 29.5 cm, as shown in Figures 3.4(c,d) and 3.4(e,f), respectively, it is clear that at 9.5 cm, the tomographic reconstructions have more noise, for both the Fresnel and the non-paraxial approaches. While the lack of clarity for the Fresnel reconstructions could be attributed to the fact that the recording distance is closer to the Rayleigh range for the object, the visible lack of clarity in the non-paraxial approach points to the fact that for both cases, the quality of the holograms recorded may have suffered more from speckle, dc noise, and extraneous fringe patterns from the recording optics. These anomalies would carry into the final reconstruction and degrade the overall results.

The next pair of images have been reconstructed for a recording distance of 19.5 cm. Figure 3.4(c) is the reconstruction of the spring using the Fresnel approach. There is a drastic change between Figure 3.4(a) and Figure 3.4(c): there is nearly accurate reconstruction when using the Fresnel approach with clearly defined edges, and the results are comparable with the non-paraxial approach, shown in Figure 3.4(d). The reconstructions at 0 degrees are shown in Figures 3.5 (a,b).
Figure 3.5: Reconstructed hologram of a pen spring. Distance is 19.5 cm @ 0 degrees of rotation using the (a) Fresnel approximation. (b) non-paraxial approach.

Looking at the final pair of images, the reconstruction distance is now set at 29.5 cm. Figure 3.4(e) is the tomographic reconstruction using the Fresnel approach, and Figure 3.4(f) is an image of the tomographic reconstruction using the non-paraxial approach. When comparing these images, the resolution is identical. The corresponding reconstructions at 0 degree rotation are shown in Figures 3.6 (a,b).

Figure 3.6: Reconstructed hologram of a pen spring. Distance is 29.5 cm @ 0 degrees of rotation using (a) the Fresnel approximation. (b) non-paraxial approach.
3.3.2 Multiangular Tomographic Resolution Results

When building a 3D tomographic reconstruction, the number of recording angles are important in determining the resolution of the object. In some instances, a minimum of four images are needed to reconstruct the object [5]. Yet, depending on the object being reconstructed, more holograms may be needed to insure a proper, high quality reconstruction. To maintain processing speed and power, the reconstructions have been scaled to 35% of their original sizes. The reconstructions that are discussed in this subsection have been reconstructed using the Fresnel approach, at 29.5 cm.

Figures 3.8(a)-(e) show the reconstructions of the pen spring at 29.5 cm. These have been reconstructed using the Fresnel approach. Each tomographic picture has various numbers of holograms incorporated into the final reconstruction, which affects the quality of the final tomographic reconstruction.
Figure 3.7: Fresnel reconstruction of a pen spring. Recording distance of 29.5 cm, using (a) 5 holograms at 40-degree increments, (b) 9 holograms at 20 degree increments, (c) 19 holograms at 10-degree increments, (d) 18 holograms at 20 degree increments. (e) 36 holograms at 10-degree increments.
For the reconstruction of the pen spring, it is clear that more angles generate a sharper reconstruction. In Figure 3.7(a), the reconstruction uses 5 different holograms, and for each hologram the spring has been rotated 40 degrees. Clearly, there is no evidence of the spring in this reconstruction. Therefore, for this object, more images are going to be needed. As shown in Figure 3.7(b), the quality of the reconstruction starts to improve by increasing the number of images to nine. The holograms that were used, were taken at 20-degree intervals up to 180 degrees. The body of the spring has begun to show, but there is plenty of noise in the reconstruction that is degrading the overall results. By increasing the number of holograms further, the final reconstruction improves. Using 18 images at 10-degree increments up to 180 and 20 degree increments up 360 degrees generate similar results. The noise seen in Figure 3.7(b) has been eliminated, as shown in Figure 3.7(c) and Figure 3.7(d). The reconstruction shown in Figure 3.7(e) uses 36 holograms at 10 degree increments up to 360 degrees. By covering the entire rotation of the object, the overall image is similar to that of Figure 3.7(c) and Figure 3.7(d). The only noticeable difference is that the color is darker, and the radius of the spring core has decreased. Ideally, the reconstructions using 18 holograms, at 10-degree increments up to 180-degrees and 20-degrees up to 360 degrees, represent the best quality. These images have the least error and highest resolution.

3.4 Conclusion

In this Chapter, different holographic reconstruction techniques have been demonstrated and compared experimentally using a pen spring as the recorded object. For recording distances within the Fresnel zone, it is more appropriate to use the non-paraxial approach. Beyond the Fresnel zone, both reconstruction techniques, Fresnel and non-
paraxial, produce similar results. Also included in this Chapter, is evaluation of the resolution quality of the tomographic reconstruction, based on the number of recorded holograms used in the final reconstruction. Experimentally it has been found that even for sparse objects like the spring, the use of 19 holograms at either 10-degree increments up to 180-degrees or 20-degree increments up to 360-degrees generate the best results.
CHAPTER 4

TRUE 3D SINGLE WAVELENGTH HOLOGRAPHIC TOMOGRAPHY USING
POINT CLOUD RECONSTRUCTION TECHNIQUES AND HOLOLENS
IMPLEMENTATION

4.1 Introduction

In the previous Chapter, we discussed the use of a simple in-line (Gabor) recording
scheme to generate holograms and perform pseudo-3D tomographic reconstruction using
myFresnel() and myLonfAndRotate() [5]. Using this method focuses on the edge
information gathered from recording the holograms on the CCD camera from various
angles, albeit restricted to rotations in one plane. Since only edge information is used,
when the holograms are reconstructed and lofted, any depth information on the illuminating
surface is lost in the final reconstruction. This Chapter discusses a Mach-Zehnder setup to
record multiple holograms from various angles in multiple planes, and reconstruction using
the point cloud image processing technique to maintain the depth information in the true
3D tomographic reconstruction. Subsequently, the point cloud reconstruction is displayed
on a Microsoft® HoloLens which allows for spatial mapping and gesture manipulation of
the true 3D holographic tomographic reconstruction to give the user an interactive experie
4.2 Experimental Setup

For recording holograms, a Mach-Zehnder setup (similar to Figure 2.2) is used as shown in Figure 4.1. Holograms are recorded for each side of the die by rotating the die along both $x$ and $y$ axis. The intensity profiles are recorded by a CCD camera. To generate the holograms, a 514.5 nm Lexel 2−7 Watt TEM$_{00}$ Ar- ion laser is used [9]. The laser beam is first sent through a microscope (40x) objective and then a pinhole ($25 \mu m$) which simulates a point source [9]. Both microscope and the pinhole can be considered as a spatial filter unit. After the spatial filter, an adjustable iris is placed for further beam cleanup. Next, the beam is collimated with a 50 cm convex lens and the collimation is verified with a collimation checker. The collimated beam is then equally split into two beams using a beam splitter (BS1 in Figure 4.1) for the Mach-Zehnder interferometric setup. The object is placed approximately at a $45^\circ$ angle. The interference of the reference beam and object beam are recorded on the CCD camera, generating a digital hologram that can be reconstructed to obtain both phase and intensity information, and used in the final tomographic reconstruction. Also for each hologram, the reference beam is recorded separately and subtracted from the hologram when reconstructing the hologram in order to remove the background which generates from laser speckle, $dc$ noise, and extraneous effects from the diffraction, and reflections generated within the CCD camera [9]. All reconstructions are done using the Fresnel propagation technique. A MATLAB® GUI developed by our group is used to record all holograms [9].
Figure 4.1: Mach-Zehnder interferometer setup used to generate the holograms for the die. The holograms are produced using a 514.5 nm Lexel 2 - 7 Watt TEM00 Ar-ion laser. The beam is collimated using a 50cm collimating lens. M1, M2, and M3 are mirrors used to reflect the beam in the appropriate directions. BS1 represents the beam splitter that is used to split the main beam into a reference beam and a second beam which illuminates the object. BS2 represents the beam splitter used to project the interference of the reference beam and the object beam onto the CCD camera [9].

4.3 Experimental Results

In this Section, the hologram reconstruction process for the die is discussed. The die is first reconstructed using the methods discussed in Chapter 3, using the functions `myFresnel()` and `myLoftAndRotate()` [5] in MATLAB®. Care is taken to scale the dimensions of the reconstructions to take care of the angle of illumination, to be described in detail below. Due to the function `myLoftAndRotate()`, the process of lofting a reconstruction converts an $n \times n$ array into an $n \times n \times n$ array. However, this simple process results in loss of the depth information during display that is needed to generate a true 3D tomographic reconstruction, and effectively only shows the information of the edges, as in the example in Chapter 3 for the spring. Figure 4.2 is an image of the
tomographic reconstruction of the die using the same method in Chapter 3. To overcome
this issue, point cloud image processing has been used.

![Image](image.png)

Figure 4.2: Tomographic reconstruction of the 8.4mm die. The reconstruction process is done
using myFresnel() and myLoftAndRotate() in a fashion similar to the reconstruction process in
Chapter 3 [5].

4.3.1 Hologram Reconstruction

The Mach-Zehnder setup described in the previous Section is used to capture the
holograms of the die and of the reference beam. The recording distance is set at 34.75 cm.
The die has to be rotated around the x and y axes to capture all six sides. As before, once
the holograms are recorded, the images are imported into MATLAB® to be reconstructed.
The reference is then subtracted from the holograms of the die faces, to remove any laser
speckle, dc noise, diffraction rings, etc. From the hologram containing the die faces. The
function myFresnel() was used to reconstruct the holograms [5]. myFresnel() is a function
that allows for the reconstruction of a digital hologram using the Fresnel technique [5].

As mentioned earlier, because of the angle of the die in the Mach-Zehnder
interferometer, the horizontal aspects of the die faces were not square. To mitigate this,
the precise angle of the die has to be determined. This is done by reconstructing the
hologram and finding the angular value that would be needed to make the pixel values of
the horizontal and vertical components match. With the angle now determined, a scaling factor is generated using the equation $1/\cos(\theta)$, where $\theta$ is the angle needed to square the hologram. In our experiment, the angle needed for the scaling factor is determined to be $52.02^\circ$. Adjusting the hologram requires a $3 \times 3$ transformation matrix to be generated. The scaling factor and the transformation matrix to stretch out the sides of the reconstructed holograms are given by

$$\text{Scaling Factor} = \frac{1}{\cos(\theta)} \quad (4.1)$$

$$\text{Transformation Matrix} = \begin{bmatrix} \text{Scaling Factor} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.2)$$

Before the stretching can be applied to the hologram, the transformation matrix must be inserted into the MATLAB® function `affine2d()`. This function allows the transformation matrix to be used in conjunction with the function `imwarp()`, that should stretch the horizontal component of the reconstructed holograms, and make the die faces square. Figures 4.3 (a)-(f) are the reconstructed holograms of each die face. The scaling has been applied to make the die faces square. Only the reconstruction intensities are shown; the phase information that gives the depth is simultaneously determined during the reconstruction process and is stored for future use in the true 3D reconstruction of the die.
Figure 4.3: Fresnel reconstruction (intensities only shown) of (a) side one, (b) side two, (c) side three, (d) side four, (e) side five, (f) side six. The recording distance is set to be 34.75cm w/ the scaling factor applied.

4.3.2 Phase Unwrapping

With the sides reconstructed using the Fresnel reconstruction technique, the next process is to phase-unwrap the phase of the reconstructed holograms. This allows us to
find the true phase value of each pixel within the reconstruction. The phase can then be converted into a distance value, which should give an accurate representation of any depth characteristics of the surface of each die face. As discussed earlier in Chapter 2, IDCT is one technique that has been used for phase-unwrapping. Bioucas-Dias and Valadao [5] have developed a program in MATLAB® known as PUMA, which is another approach to accurately phase-unwrap more efficiently by casting the reconstruction as an energy minimization via graph cutting [5]. After the holograms have been imported into PUMA, the phase reconstructions are cropped, so that the only data stored are that of the die faces. The cropped data is then run through a median filter, `medfilt2()` set to $25 \times 25$ to smooth out the surface plot representations of each unwrapped die face. Figures 4.4(a)-(f) are the surface plots of the phase unwrapped die face reconstructions. The median filter has been applied to each face before constructing the surface plots. The phase information has been converted into a depth/distance values as well.
Figure 4.4: Surface plots of the phase-unwrapped reconstruction side one using PUMA: (a) side one, (b) side two, (c) side three, (d) side four, (e) side five, (f) side six. The x and y axes are measured by the pixel value, and the z axis represents the converted phase information into depth. The depth information z is measured in meters.
4.3.3 Point Cloud Generation and Construction

Before the final reconstruction, the PUMA solutions of each die face must be converted into a format that the point cloud image processing can use. As stated previously, this is done because previous methods used for tomographic reconstructions require the holograms to be lofted and rotated in virtual space [5]. This process causes the depth information acquired from each hologram to be lost. Using the point cloud method should allow the holograms to maintain their perspective depth information, while being able to rotate the holograms in virtual space.

While using the point cloud method, the information is typically stored as an $N \times 3$ array. The $N \times 3$ array is a 3D array containing three columns. Each column represents the $x$, $y$, and $z$ data for a given hologram. This gives each pixel a specific $x$, $y$, $z$ coordinate. Each pixel coordinate contains the location of the horizontal ($x$), vertical ($y$), and depth ($z$) component. While using this format, the PUMA solutions need to be converted. Originally, the PUMA solutions are stored as an $N \times N$ array. In the case of the die solutions the arrays are $346 \times 346$. The horizontal and vertical components of the PUMA solutions are separated. Each pixel is converted into a distance value by multiplying each pixel by the pixel separation of the CCD camera. The vertical and horizontal components are each converted into a 1D vector/array ($1 \times N$) by taking the horizontal and vertical components and running them through the function $meshgrid()$. Thereafter, these vectors need to be converted into a ($N \times 1$) vectors/arrays. The phase information, found after running the holograms through PUMA, are converted into a depth value by multiplying the results by $\lambda/2\pi$. These depth values are then also converted into a $N \times 1$ vector/array.
The horizontal, vertical and depth \((N \times 1)\) arrays are combined to generate the \(N \times 3\) array needed to run the data through point cloud.

After generating the point cloud 3D arrays, the data needs to be modified to smooth out any unwanted curvature in the reconstruction as a result of the angle of the Mach-Zehnder setup and the Gaussian profile of the beam that can be seen in the PUMA reconstruction. To remove the curvature, the fit function “lowess” using \(fit()\) is first applied to the point cloud data. A computer generated “best fit” is thus created and stored. Thereafter, the function \(feval()\) is applied to remove the curvature, achieved through subtracting the curve from the depth component of the point cloud reconstruction. Some residual extraneous curvature still remains and cannot be avoided. Figures 4.5(a)-(f) are the point cloud reconstructions of each die face after removal of the curvature.
Figure 4.5: Point cloud reconstruction of die surfaces after smoothing process. (a) side one, (b) side two, (c) side three, (d) side four, (e) side five, and (f) side six. Side six has been inverted for future reconstruction purposes. All die faces have been smoothed to remove any curvature. There will still be some curvature due to the accuracy of the computer.

With the die faces converted into point cloud and smoothed, the next step is to rotate the die faces and combine the faces to reconstruct the die. Before any rotations and translation can be applied to the die faces, the point cloud $N \times 3$ arrays need to be stored.
as a point cloud object. This is done by applying the function `pointCloud()` to the $N \times 3$ arrays. Once the die faces are stored as a point cloud object, multiple $4 \times 4$ transformation matrices have to be generated. The $(4 \times 4)$ transformation matrix is used in MATLAB® to generate 3D transforms that can be applied to the point cloud objects. In this case, the die faces. This allows for the die faces to rotate and translate around/along the $x$, $y$, and $z$ axes. Adding translation in addition to the rotation is a result of how MATLAB® rotates objects using the transformation matrices. The transformation matrices are then run through the function `affine3d()` to allow the transformation matrices to be applied to the point cloud objects. The transformations are applied to each die face using the point cloud function `pctransform()`. All six die faces can then be consolidated together using the function `pcmerge()`. The `pcmerge()` function allows the manipulation of the grid step of the final reconstruction. Because of the amount of points in the final reconstruction, the grid step was set to 0.25-nm. Figure 4.6 is the die reconstruction before the edges are stitched together.
Figure 4.6: Die reconstruction using the point cloud method. The reconstruction is missing the edges due to the process used to generate the holograms for this reconstruction.

As evident from Figure 4.6, the reconstruction is incomplete. Because of the method used to capture the holograms, the entire edge information could not be extracted since the edges are not properly illuminated. Therefore, the edges need to be manually stitched together. To do this, the first process is to find the edges of each die face via the PUMA surface data and convert the edges into the $N \times 3$ point cloud format. The edges need to match the profile of the final die edges; therefore, the edges need to be curve fit similar to the die faces. With the profiles matched, the separation between the edges can be found. Once the separation has been found, an edge vector is generated to be used as a place for the edge to meet during the generation process. To fill the gap between the edges,
a function known as `linspace()` is used to fill the gap point-by-point by a specified amount $m$ evenly between two specific points. The edges are then combined with the die reconstruction by converting them into a point cloud object using `pointCloud()` and adding the edges to the die by using `pcmerge()`. Figure 4.7(a)-(c) are images of the final die reconstruction, with the addition of the edges. Because of the addition of point from stitching the faces together, some points are removed from the overall image to save memory on the computer.
Figure 4.7: Final die reconstruction, including the addition of the stitched edges showing, for example, sides (a) one, five and two, (b) one, three and four, (c) three, four, and six.

4.3.4 Hololens Implementation

The final process is to import the point cloud reconstruction of the completed die into the Microsoft® HoloLens. This is done by using two different virtual reality software, Blender™ and Unity®. Blender™ is first used to import the PLY file of the die reconstruction. Within Blender™ some modifications can be made to make viewing the die within the HoloLens easier. Blender™ also allows the removal of any unwanted data that has been generated throughout the reconstruction process. With the die imported into
Blender™, the next objective is to import the die into Unity®. Unity® is a gaming/VR software partnered with Microsoft® that is used to build applications for the HoloLens. Within Unity®, Microsoft® has released several C# scripts that have access to pre-programmed spatial mapping and gesture recognition that allows the user to interact/manipulate the object in view, in this case the die reconstruction.

Before the die can be imported into Blender™, the normal to each point has to be calculated. Usually, this is done by using two functions in MATLAB®. First, *scatteredInterpolant()* should be used along with the \((x, y, z)\) components of the point cloud object. The function allows for the interpolation of the point cloud object to generate a \(N \times N\) representation. Next a mesh grid should be generated using the max and min values of either \(x\) or \(y\) axis, since the object is square. The combination of putting in the values of the mesh grid into the stored result of the \textit{scatteredInterpolant()} generates a surface plot that can be used to find the normal. The surface plot data should then be imported into the function \textit{surfnorm()} to generate the surface normal for each point. Unfortunately, the \textit{scatteredInterpolant()} function could not handle the completed die. Therefore, each die surface has been interpolated and the normal found individually. Once the surface normal is found, the surface plot and the surface normal are saved and imported into Blender™.

With the die faces imported into Blender™, the die can be rebuilt. Using some of the modeling features within Blender™, the die can be modified to add or remove features to the die. Thereafter, the die is imported into Unity®. Within Unity®, the die is placed in the hierarchy of the scene. Attached to the die is a box collider, which is used by the gaze feature of the HoloLens to indicate a collision point between the ray cast, along with the
main camera and directional lighting. By placing items in the hierarchy includes them in the main application to be used once the scene is built. An empty game object is created in the hierarchy, this game object is used as a manager object. This “manager” object is where the various scripts are added to the application. GazeManager.cs is added to the manager game object. GazeManager.cs is used to determine the location of the user’s gaze, hit position, and normals. GestureManager.cs is then added following the GazeManager.cs. GestureManager.cs is responsible for providing access to several different input gestures like tap and manipulate. HandsManager.cs is the next script to be added. HandsManager.cs is responsible for determining if the hands are detected or not. The last two scripts added to the “manager” game object are WorldAnchorManager.cs and KeywordManager.cs. These are responsible for initializing the anchors that are used in combination with spatial mapping to anchor the object to the virtual mesh and set specified key words to be used with the voice command feature of the HoloLens, respectively. The KeywordManager.cs script allows us to program the zoom in and zoom out commands in the future.

With the “manager” game object completed, the next step is to add the spatial mapping capability. This is done by searching through the HoloLens toolkit to find a prefabricated asset call Spatial Mapping. This asset is added into the hierarchy. The spatial mapping prefab contains three scripts, SpatialMappingObserver.cs, SpatialMappingManager.cs, and ObjectSurfaceObserver.cs. These scripts allow the HoloLens to spatially map a given area and generate a virtual mesh around real world objects that objects within the application can be anchored to. Following the spatial mapping prefab, one more prefab needs to be added into the hierarchy, the HoloLens
curser. By searching the HoloLens toolkit, a prefab of several cursors appear within the folder. For this application, the standard curser labeled “Cursor” has been used. Contained within this prefab is a script called CursorManager.cs. This script allows the cursor to interact with game object that contain a collider.

Before the application is complete, there are a few more scripts that needed to be added to the application. For the main camera, two scripts, ZoomInGUI.cs and ZoomOutGUI.cs, are created and added. These scripts adjust the field of view, thereby causing the object to appear closer or further away from the camera. These scripts in combination with KeywordManager.cs allows the field of view to be adjusted by using voice commands. Finally, two other scripts, TapToPlace.cs and GestureManipulator.cs are added. TapToPlace.cs allows the user to select the object that the user is gazing at and release the anchors that are holding the object on the virtual mesh. The HoloLens then begins to spatially scan and generate meshes over real world surfaces. If the object is selected again, the object anchors are reinitiated and the object is anchored to one of the new mesh surfaces. GestureManipulator.cs allows the user to tap and hold to move the object around in the virtual space, as well as scale the object, thereby increasing or decreasing the object size, similar to the zoom in and zoom out commands. With the scene completed, the scene is built and imported using Microsoft® Visual Studios 2015. Figures 4.8(a) and 4.8(b) represent the completed die that has been reconstructed in Blender™ and seen through the HoloLens.
Figure 4.8: (a) Blender™ reconstruction of the die containing the surface normals needed for the use in the Unity application. (b) Die reconstruction once the Blender™ reconstruction has been imported into Unity and run in an application on the HoloLens. The blue circle in (b) is the cursor of the HoloLens interacting with the box collider surrounding the die.

4.4 Conclusion

In this Chapter, a new reconstruction technique is used to reconstruct an 8mm die using point cloud. The reconstruction has then been imported into a modeling software known as Blender™. Blender™ is used as a bridging software to allow the PLY file to be imported and used on the HoloLens. Blender™ is also used to modify and stitch the sides of the die faces together, along with removing any unwanted artifacts that comes with the point cloud reconstruction process. With the information of the die imported into Unity®, various scripts have been added to allow the user to spatial map their surrounding and manipulate the die.

Unfortunately, there is some loss in detail when going from Blender™ to Unity®. In Blender™, the software uses the computer’s graphics process, allowing the die to be built in a resolution of 3840x2860, whereas the current HoloLens only has a resolution of 1268x720 per eye, thereby causing a loss in detail. Along with the loss in resolution of the
die reconstruction, while running the application, the processor of the HoloLens struggles to keep up with the other processes within the application because of the amount of information contained within the die, causing a noticeable lag within the application. It is hoped that future generations of the HoloLens will have more resolution and processing power to overcome these shortcomings.
5.1 Introduction

In this Chapter, the hologram recording process has been taken a step further. Thus far, the holograms have been recorded using a single wavelength set to 514.5-nm. In the example shown in this Chapter, two wavelengths, 501.7-nm and 496.5-nm, are used to generate holograms for the object, which is a coin such as a dime. The use of two wavelengths is advantageous when the feature characteristics are much larger than the wavelength [2,4,5]. Use of a single wavelength this case results in numerous fringes, which cannot always be properly recorded, and also for which phase unwrapping may sometimes give incorrect results. Using two wavelengths, the derived phases are subtracted and then unwrapped using PUMA. Similar to the last Chapter, the phase information is converted into depth, but now using the synthetic wavelength instead of the actual wavelength of the illumination [2,4,5], and converted into the point cloud format. Once in the point cloud
format, the reconstructions are again interpolated and the surface normals are found before importing the reconstruction into Blender™. While in Blender™, the edges are stiched together and imported into Unity® to generate an application to view on the HoloLens.

5.2 Experimental Setup

Similar to the last Chapter, the setup used to generate the holograms is the Mach-Zehnder interferometer shown in Figure 5.1. To illustrate the general principle, the object is rotated to record two holograms, one of each face of the coin. The holograms are recorded by a CCD camera. To generate the holograms, the Lexel 2-7 Watt TEM$_{00}$ Ar-ion laser is adjusted to 501.7-nm and 496.5-nm, generating a synthetic wavelength of 47 microns. The laser beam is first sent through a spatial filter containing a 40x MO and a 25-µm pinhole [9]. The beam then passes through an adjustable iris and a 50-cm collimating lens. The collimation is checked using a collimation checker. The collimated beam is split by the first beam-splitter (BS1). BS1 splits the beam into two arms, one containing the reference beam and the other the object beam. The reference beam is reflected into a second beam-splitter (BS2), the object beam hits the target object that is stationed at an angle of $\theta \sim 45^\circ$ [9]. The object beam reflects off the target object through BS2 and interferes with the reference beam before both beams hit the CCD camera to generate a digital hologram [9]. For each hologram that is recorded, a reference hologram is recorded separately and subtracted from the hologram during the reconstruction process in order to remove the background which generates from, laser speckle, $dc$ noise, extraneous effects from the diffraction, and reflections generated within the CCD camera [9]. All reconstructions are done using the Fresnel propagation technique. A MATLAB® GUI developed by our group is used to record all holograms.
5.3 Experimental Results

In this Section, the reconstruction process is discussed. Using the same techniques applied in the previous Chapter, the holograms are reconstructed for both wavelengths. The phase difference is unwrapped to find the depth information. Following the phase unwrapping, the dime sides is converted into point cloud and smoothened before being interpolated to find the surface normal. Thereafter the information is introduced into Blender™ and stitched together before importing the final reconstruction into Unity® to be viewed on the HoloLens.
5.3.1 Hologram Reconstruction

Using the setup described in the previous Section, the holograms for each face of the dime have been recorded for both wavelengths. The distance between the dime and the CCD camera is set at 50 cm. The dime is rotated to obtain holograms of both sides. As before, the effects of the dc and the background noise is subtracted, and the resulting holograms are reconstructed using the `myFresnel()` function [5].

Once again, because of the angle of illumination of the object in the Mach-Zehnder interferometer, the horizontal and vertical aspects of the reconstructed image are not symmetrical. The angle in this case is $\theta \sim 45^\circ$. By using the $3 \times 3$ transformation matrix defined in Chapter 4 and the MALAB® functions `affine2d()` and `imwarp()`, the dimensions are symmetrized to their correct proportions. Figure 5.1(a)-(d) are the intensity reconstructions of the dime after applying the transformation matrix to the reconstructions.
Figure 5.2: Reconstruction of the dime (a) front and (b) back using a wavelength of 505.5-nm and a recording distance of 50-cm, (b) dime back. Also shown are reconstructions of (a) front and (b) back using a wavelength of 505.5-nm and the same recording distance.

5.3.2 Phase-Unwrapping

With the holograms reconstructed using the Fresnel reconstruction technique, the next process is to phase-unwrap the holograms. Unlike single wavelength DH, the phase difference of the reconstructions at the two wavelengths is what needs to be unwrapped. To get the phase difference of the holograms, a GUI developed in MATLAB® is used to calculate the phase difference between the two wavelengths, and thereafter unwrapped using PUMA. The unwrapped phase data is converted into a depth value using the relations
\[ \Lambda = \frac{\lambda_1 \lambda_2}{\lambda_1 - \lambda_2}, \]  

(5.1)

\[ \text{depth} = \frac{\Lambda}{4\pi}, \]  

(5.2)

where \( \lambda_1 = 505.5 \text{ nm} \) and \( \lambda_2 = 496.5 \text{ nm} \) and where \( \Lambda \) is called the synthetic wavelength. Figures 5.2(a) and 5.2(b) are the PUMA surface plots of the converted phase-to-depth data from each side of the coin. Due to the amount of detail on the back side of the coin, the PUMA surface plot is not as detailed as the front side of the coin. There are still details on both sides of the coin that are missing which will be talked about in the next Chapter.

![Figure 5.3](image)

(a) (b)

*Figure 5.3: (a) PUMA surface plot of the dime front, (b) back side.*

5.3.3 Point Cloud Reconstruction

With the PUMA solutions stored, the next set is to convert the PUMA solutions into the point cloud format, as described in Chapter 4. To eliminate surface curvatures, the functions *fit()* and *feval()* have been used to generate a fit curve that is subtracted from the original point cloud depth component to flatten the overall plot. Unlike the die, the edges of the dime have not been recorded. This would need rotating the dime about an axis normal to its face several times by small angles. Only the reconstructed faces
of the coin have been interpolated and imported into Blender\textsuperscript{TM} to be processed before sending the reconstruction into Unity\textsuperscript{®}. Figures 5.3(a) and 5.3(b) are the point cloud reconstructions of both sides of the dime after using the fitted curve to remove the curvature from the overall plot.

![Figure 5.3](image)

(a)  
(b)

*Figure 5.3: (a) Front of the dime reconstructed in point cloud, (b) back side. Due to the flattening and the color mapping in MATLAB\textsuperscript{®}, some of the details seen in Figure 5.2 are not distinguishable.*

5.3.4 Hololens Implementation

Before the dime sides can be imported into Unity\textsuperscript{®}, the surface normals need to be found. As discussed in Chapter 4, the functions `scatteredInterpolant()` and `surfnorm()` are used. One of the major modifications is to remove the extra background that could not be removed from cropping the PUMA results. The process has been performed manually, thereby causing deformations along the dime edges. An automated process will be investigated in the future.

In Unity\textsuperscript{®}, the dime is taken from the assets folder and placed into the hierarchy and a box collider is added to the dime. Like the die, the spatial mapping and cursor prefabs are placed in the hierarchy along with the dime to allow the application to spatial map an area and allow the cursor to interact with the dime within the scene. All prefabs used
contain the same scripts that have been used in Chapter 4 for the die. Like the die, the TapToPlace.cs and GestureManipulator.cs scripts are added to the dime to allow the object to be anchored and unanchored from the virtual mesh generated from the spatial mapping prefab and manipulated using hand gestures. ZoomInGUI.cs and ZoomOutGUI.cs are also added to the main camera to be used in conjunction with the KeywordManager.cs script to allow for voice commands to adjust the field of view to zoom in and out on the object. The same “manager” game object is created, along with the same scripts as the “manager” game object used for the die. With the application completed, the scene is built using Visual Studios 2015 and imported onto the HoloLens. Figures 5.4(a)-(d) are images of the reconstructed dime in Blender™ and in the HoloLens. Again, due to the resolution difference in Blender™ and the HoloLens, some details are lost once the reconstruction is imported into the HoloLens.

5.4 Conclusion

In this Chapter, we have discussed about taking the tomographic reconstruction process one step further by using multi-wavelength holographic tomography. Two wavelengths, 501.7 nm and 496.5 nm, are used to generate digital holograms. The two wavelengths generate a synthetic wavelength of 47 microns, which allows us to record a significant amount of detail on the dime. Once the holograms are imported into MATLAB®, they have been reconstructed using the Fresnel technique [5]. The phase difference is found by using a GUI designed in MATLAB® by our group. The synthetic phase is then unwrapped, converted into depth information, and cropped.
Figure 5.5: Final reconstruction of dime (a) front and (b) back in Blender™. The same reconstructions in HoloLens (c) front, (d) back.

The rest of the reconstruction and visualization process is the same as that in Chapter 4. The PUMA solution has been converted into point cloud and smoothened. The final solution is interpolated and the surface normal found before importing into Blender™ and the HoloLens using Unity®. Some of the detail is lost because of the resolution drop.
between Blender™ and the HoloLens. Yet, there is still a significant amount of detail that is still observable on the HoloLens.
CHAPTER 6

CONCLUSION

In this thesis, two hologram reconstruction techniques are simulated using MATLAB® and the results are first compared. It is found that for recording distances of interest, the Fresnel reconstruction technique is almost identical to the non-paraxial technique, and is therefore used for all subsequent reconstructions. Holographic tomography of sparse objects using in-line recording is then performed, and the objects recreated in pseudo-3D from edge reconstructions from the holograms. Taking tomography to the next step, this thesis looks at generating a true 3D tomographic reconstruction by phase-unwrapping the various holograms recorded from different angles in different planes using a Mach-Zehnder setup to preserve the depth characteristics of different faces of the object, first by using a single wavelength. It is shown that by converting the unwrapped PUMA solutions into point cloud, the 3D reconstructions could be generated while maintaining the depth information. The reconstruction is then imported into Blender™ and further processed and then imported into Unity® which is compatible with the Microsoft® HoloLens to view the reconstruction in a virtual 3D format. The user can use spatial mapping, gesture manipulation, and voice commands within the HoloLens
to interact with the reconstruction. Holograms are then recorded using two wavelengths, the reconstructed phases are subtracted and unwrapped using PUMA, and converted to depth information using the synthetic wavelength. The reconstruction process is then repeated with a dime, but in this case, two wavelengths were used. The same process is used to view the object on the HoloLens. Even though the HoloLens suffers from lower resolution, many key features could still be discerned.

In the future, work should be done on improving on the recording setup by adding multiple CCD cameras. The work presented here can be improved by further investigating the reconstruction process while using the point cloud method and importing the final results into Blender™ and the HoloLens. More work needs to be done when integrating Blender™ into the reconstruction process. When importing non-symmetric or reconstructions with extra data, finding a way to remove the extra information while maintaining the integrity of the desired image is essential. Improvements to the HoloLens application and the HoloLens memory capacity can benefit the final 3D virtual reality visualization as well. Allowing the user to select a specific area on the reconstruction and take detailed measurements of the depth information would be ideal for analyzing various reconstructed objects.
REFERENCES


tomography for virtual reality applications,” paper 10396-39, SPIE Annual Meeting,
San Diego (2017).

10. L. Huang, C. Zuo, M. Idir, W. Qu, and A. Asundi, “Phase retrieval with the transport-
of-intensity equation in an arbitrarily-shaped aperture by iterative discrete cosine

holographic reconstruction of three-dimensional objects in transmissive and

12. L. Williams, Digital Holography for Three-dimensional Tomographic and


3D visualization,” paper DWE3, Digital Holography and Three-Dimensional
Imaging, OSA Technical Digest (2011).
APPENDIX A
MATLAB CODE

A.1 NON-MATLAB FUNCTIONS

A.1.1 myFresnel

function [Hr] = myFresnel(hologram,reconstruction_distance,....
   wavelength,pixel_size,in_line,phase_mask,....
   scale_factor,pad_size)%
   % %This function reconstructs a hologram using a fresnel transform
    % %and returns the reconstructed complex image matrix
    %
    % %reconstruction_distance in METERS
    % %wavelength in METERS
    % %pixel size in METERS (pixels are assumed to be SQUARE)
    %
    % %if in_line is set to 1, the hologram will rescale with distance about the
    % %zero order (usefull for in-line holograms), defaults to 0
    % %NOTE: reconstruction_dist must be positive for in_line scaling to work
    %
    % %phase_mask should be the same size as the hologram
    % %if either one is not square, they will be cropped to square
    % %if a scalar value is entered for phase_mask,
    % %a constant array of that value is used (i.e. uniform phase, plane wave)
    %
    % %scale_factor will change the size of the original hologram
    % %scale_factor should go above 1 now (updated 7 May 2013)
    % %increasing the scale factor will dramatically improve the reconstruction
    % %results when in the very near field (without extra padding)
    % %scale_factor defaults to 1
reducing the scale factor too low (<1/2 or so) will begin vignetting the image significantly, losing the higher spatial freq info due to resampling losses.

for very large CCD arrays, scale the image down
for very near field holograms, scale the image up

Begin Function
A = size(hologram);
if nargin < 8
    pad_size = 0;       %default to zero padding
end
if nargin < 7
    scale_factor = 1;   %default to original size
end
if nargin < 6
    phase_mask = 1;     %default to uniform phase (const, plane wave)
end
if nargin < 5
    in_line = 0;        %default to off-axis style hologram
end
if nargin < 4
    error('Not enough input arguments')
end
if length(phase_mask) == 1
    phase_mask = ones(A).*phase_mask;
end
B = phase_mask;
H = hologram;
d = reconstruction_distance;
w = wavelength;
dx = pixel_size;

if the image is not square, crop to be square along the smallest dimension
This cropping is done manually (i.e. without the imcrop function), and should be fairly robust
if A(1,1)<A(1,2)
    %crop the x axis of the image to match the y axis
    %crops symmetrically about the center
    H = H(:,round(A(1,2)/2+1-A(1,1)/2):round(A(1,2)/2+A(1,1)/2));
    B = B(:,round(A(1,2)/2+1-A(1,1)/2):round(A(1,2)/2+A(1,1)/2));
end
if A(1,1)>A(1,2)
    %crop the y axis of the image to match the x axis
    H = H'; %transpose
    B = B';
end
C = size(H);  %this is the line that fixes this part
%after transposing, the sizes didn't match anymore
H = H(:,round(C(1,2)/2+1-C(1,1)/2):round(C(1,2)/2+C(1,1)/2)); %crop
B = B(:,round(C(1,2)/2+1-C(1,1)/2):round(C(1,2)/2+C(1,1)/2)); %crop
H = H'; %transpose back
B = B';
%not sure if this works perfectly yet - haven't had to try it
end

%use double precision to allow for complex numbers
H = double(H);
B = double(B);

%if the image is too big to compute efficiently, scale it down.
%to scale above 1, the phase mask can cause problems because the phase
%does not scale well with bicubic resampling. An appropriate phase mask
%should be generated external to this function
if scale_factor ~= 1
    H = imresize(H, scale_factor,'bicubic');
    if length(B(:,1)) > 1  %only rescale profile if it's not a scalar
        B = imresize(B, scale_factor,'bicubic');
    end
    dx=dx/scale_factor;
end

n = length(H);   %size of hologram matrix nxn
H = double(H)-mean(mean(H));  %must be class double for imaginary #s

if pad_size > 0  %only pad array AFTER mean subtraction
    nr = length(H);  %save the orginal size
    H = padarray(H,[pad_size pad_size]);  %pad the array
    n = length(H)  %reset n after padding
    dx = ((nr/(n-pad_size*2))*pixel_size);  %account for pad in scale
    dx=dx/scale_factor;

    if length(B(:,1)) > 2
        fprintf('Cannot pad with beam profile input. /n Assuming plane wave. /n');
        B = 1;
    end
end

dy = dx;  %assumes square CCD pixels --> modify as needed for rectangle pixels
Hr = complex(zeros(n));  %reconstructed H
E = complex(zeros(n));  %exponential term

%verified correct feature size (i.e. reconstructed pixel sizes)
feature_size = abs(w*d/(length(H)*dx))

k = -n/2:1:n/2-1; %array same dimention as hologram
l = -n/2:1:n/2-1;
[XX,YY] = meshgrid(k,l);

%Calculate the Fresnel Transform
E(k+n/2+1,l+n/2+1) = B.*exp((-i*pi/(w*d)).*((XX.*dx).^2 + (YY.*dy).^2));
%Reconstruction becomes complex valued
Hr = (fftshift(fft2(H.*E)));

%If padded. . .
if pad_size >0  %cut the initial padding back out
    Hr = Hr(pad_size:nr+pad_size, pad_size:nr+pad_size);
end

%Rescaling for IN-LINE HOLOGRAMS ONLY
%scale by CCD size/object feature size = # of pixels to keep
%rescales based upon the zero order size only
if in_line == 1
    N = min(A);  %smallest dimension of original hologram
    obj_f = w*d/(N*dx);
    P = round(N*dx/obj_f);  %should be size of center region
    %if P is odd, make it even
    if (mod(P,2) == 1)  %if first dimension has odd number of rows
        P = P+1;
    end
    %end rescaling of fresnel transform
    if (P<N)
        [Hr] = Hr(round(N/2+1-P/2:N/2+P/2),round(N/2+1-P/2:N/2+P/2));
    end
    else if in_line == 0
        [Hr] = Hr;
    end
end  %end function

A.1.2 myLoftAndRotate

function [Output_Volume] = ...
    myLoftAndRotate(input_matrix,scale_factor,thx,thy,thz,tx,ty,tz)
%written by Logan Williams, 5/10/2012
%this function will accept a 2D or 3D input matrix
% and loft it along the z axis to make it a cube.
% If angle is set to a non-zero value, the matrix
%will also be rotated using the rigid3D function.

%THIS FUNCTION ONLY WORKS RIGHT FOR MATRICES THAT
% ARE SQUARE IN X AND Y there is no error checking
% to enforce this requirement

%if the matrix is already a 3D volume, but not a cube,
%this function will loft it along the z axis to make
% it a cube by lofting the space between the original
% planes of the matrix. WRITTEN, BUT NOT TESTED YET

%input_matrix should already be square, or it will not loft to a cube

%scale_factor should be less than 1, to avoid memory
% over runs. Defaults to 1 if no scale factor is given

%angle is in degrees, defaults to zero if no angle is
% given. The rotation will only be in the x-z plane.
% If some other rotation plane is desired,
% call this function to loft with an angle of zero,
% then call rigid3D afterwards for arbitrary rotation

%input matrix is converted to single to minimize
% memory use

if nargin < 3
    angle = 0;
end
if nargin < 2
    scale_factor = 1;
end

A = size(input_matrix);

if A(1) ~< A(2)
    warning('Input matrix is not square in x & y, will not loft properly')
end

%Resize the matrix to be more memory efficient
if scale_factor ~< 1

    str = sprintf('Recommended scale factor is (%s) for 100x100x100 cube', 1/(A(1)/100));
input_matrix = imresize(input_matrix, scale_factor,'bilinear');

if scale_factor > 0.5
    warning('Scaling factor > 0.5 could result in memory overrun, proceeding anyway.'
)
end

if (A(1)*scale_factor) > 100
    warning('Matrix size > 100x100x100 could result in memory overrun, proceeding anyway.'
)
end

A = size(input_matrix);  %get resized input matrix
n = A(1);  %length of resized y dimension, chosen arbitrarily

%if input matrix is 2D, loft using repmat
if length(A) == 2
    S=zeros(n,n,n);
    S=single(S);
    S=repmat(input_matrix,[1 1 n]); %loft the image by n layers
end

%if input matrix is 3D, loft between existing layers
if length(A) == 3
    nz=length(input_matrix(1,1,:));
    S = zeros(n,n,n);
    S=single(S);
    for c=1:1:n
        S(:,:,c) = single(abs(input_matrix(:,:,ceil(c/(n/nz)))));
    end
end

if thx ~= 0
    if thx ~=90
        S=rigid3D(S,thx,thy,thz,tx,ty,tz);
    end
    if thx == 90
        S = permute(S,[1 3 2]);  %not sure if this is the right permutation
        % not 1 3 2, 3 2 1, 3 1 2
    end
end
if thy ~= 0
    if thy ~= 90
        S = rigid3D(S,thx,thy,thz,tx,ty,tz);
    end
end

if thy == 90
    S = permute(S,[1 3 2]); %not sure if this is the right permutation
    % not 1 3 2, 3 2 1, 3 1 2
end
end

if thz ~= 0
    if thz ~= 90
        S = rigid3D(S,thx,thy,thz,tx,ty,tz);
    end
end

if thz == 90
    S = permute(S,[1 3 2]); %not sure if this is the right permutation
    % not 1 3 2, 3 2 1, 3 1 2
end
end

Output_Volume = S;

end  %function end

A.1.3 myDisplay3d

function [data_out] = Display3d(data_in, threshold,...
    smoothing, alpha_value)
%Written by Logan Williams, 5/10/2012

%this function will display a 3D volume using
%isosurface, while performing simple thresholding.

%the threshold is a percentage of the maximum value,
%and should vary between zero and 1

%smoothing will be gaussian smoothing. The value of
%smoothing must conform to the smooth3 function,

%which requires odd integer values greater than 1.
%Setting smoothing to zero will disable smoothing
%alpha_value is the percentage of transparency
%between 0 and 1, with 1 = opaque and 0 = transparent

if nargin < 4
    alpha_value = 1; % 1 = opaque
end

if nargin < 3
    smoothing = 0; % no smoothing
end

if nargin < 2
    threshold = 0.1; % default at 10% of max value
end

A = data_in;
S = size(A);
b = threshold; % Threshold as a percentage of the maximum value

if smoothing ~= 0
    % Hs = smooth3(A,'box',smoothing);
    A = smooth3(A,'box',smoothing);
end

% This section unnecessary maybe -----------------
% Threshold by finding all values less than the threshold value and setting
% them equal to zero
A(find(abs(A) < b.*abs(max(max(max(A))))) = 0;
%--------------------------------------------------

isovalue = b*abs(max(max(max(A))))
V = abs(A); % must use absolute magnitude (since A could be imaginary)
[m,n,p] = size(V);
[x,y,z] = meshgrid(1:m,1:n,1:p);
fv = isosurface(x,y,z,V,isovalue);
p = patch(isosurface(x,y,z,V,isovalue));
% p = reducepatch(p,0.15) % doesn't work quite yet

alpha(alpha_value)
% figure(a+4)
isonormals(x,y,z,V,p);
set(p,'FaceColor','red','EdgeColor','none');
%daspect([1 1 1])
box on
% view(-16,-83); axis tight
view(20,80); % axis tight
camlight headlight
%lighting gouraud
lighting phong
xlabel('x')
ylabel('y')
zlabel('z')
title('')
axis([1 S(1) 1 S(2) 1 S(3)]);

A.1.4 saveobjmesh

function saveobjmesh(name,x,y,z,nx,ny,nz)
% SAVEOBJMESH Save a x,y,z mesh as a Wavefront/Alias Obj file
% SAVEOBJMESH(name,x,y,z,nx,ny,nz)
% Saves the mesh to the file named in the string name
% x,y,z are equally sized matrices with coordinates.
% nx,ny,nz are normal directions (optional)

normals=1;
if (nargin<5) normals=0; end
l=size(x,1); h=size(x,2);

n=zeros(1,h);

fid=fopen(name,'w');
nn=1;
for i=1:l
  for j=1:h
    n(i,j)=nn;
    fprintf(fid, 'v %f %f %f
',x(i,j),y(i,j),z(i,j));
    fprintf(fid, 'vt %f %f
',(i-1)/(l-1),(j-1)/(h-1));
    if (normals) fprintf(fid, 'vn %f %f %f
',nx(i,j),ny(i,j),nz(i,j)); end
    nn=nn+1;
  end
end
fprintf(fid,'g mesh
');
for i=1:(l-1)
  for j=1:(h-1)
    if (normals)
      fprintf(fid,'f %d/%d/%d %d/%d/%d %d/%d/%d %d/%d/%d
',n(i,j),n(i,j),n(i,j),n(i+1,j),n(i+1,j),n(i+1,j+1),n(i+1,j+1),n(i+1,j+1),n(i+1,j+1),n(i+1,j+1));
    else

end

end

end

71
fprintf(fid,'f %d/%d %d/%d %d/%d %d/%d
',n(i,j),n(i,j),n(i+1,j),n(i+1,j),n(i+1,j+1),n(i+1,j+1),n(i,j+1),n(i,j+1));
end
end
end
fprintf(fid,'g\n\n');
close(fid);

A.1.5 puma_ho

function [unwph,iter,erglist] = puma_ho(psi,p,varargin)
%puma_ho   Graph optimization based phase unwrapping algorithm.
%   [unwph,iter,erglist] = puma_ho(psi,p,'potential',potential,'cliques',cliques,
%   'qualitymaps', qualitymaps,
%   'schedule',schedule)
% Unwraps the observed modulo-2pi phase, casting the problem as an energy
% minimization via graph mincut
% calculation. The algorithm is described in "Phase Unwrapping via Graph Cuts"
% submitted to IEEE IP, October, 2005.
% Herein we make a generalization, by allowing cliques of any order (not just 1st order).
% Authors: Jose Bioucas-Dias and Gonçalo Valadão
% %
% % ==----------------------------- REQUIRED INPUT PARAMETERS
% %==-------------------------------
% % Parameter          Values
% % name                and description
% %----------------------
% % psi          (double) The wrapped phase image.
% % p           (double) It defines the clique potential exponent (>0).
% %
% %==----------------------------- OPTIONAL INPUT PARAMETERS
% %==-------------------------------
% % Parameter          Values
% % name                and description
% %----------------------
% % potential  (1x1 struct array) This struct array has 2 fields:
% % potential.quantized ('yes','no') by default it is chosen a quantized potential.
% % potential.threshold (double) it defines a region over which the
% % potential grows quadratically. By default is 0.
% cliques (nx2 double matrix) Each row defines the "displacement vector" corresponding to each clique.
% The first and the second columns correspond to cliques along rows and columns in the image, respectively.
% By default is [1 0;0 1] (first order cliques).

% qualitymaps (size(psi) x n (nocliques) double array). The quality matrices may take values between 0 and 1 (value 1: discontinuity presence; value 0: discontinuity absence).
% There is one quality matrix per clique type. By default there is discontinuity absence. A quality value corresponding to a certain clique must be signalled in the pixel corresponding to the end of the clique displacement vector (for each pair of pixels).

% schedule (double vector) This vector contains a schedule of jump sizes.

% Parameter Values
% name and description
%
% unwph (double array) This is the unwrapped phase image.
% iter (double) This is the number of iterations the algorithm runs through
% erglist (double vector) This is a sequence of the energies of the unwrapped phases along the algorithm run.

% EXAMPLES
%
% Note: the optional arguments must be provided in pairs string+value; those pairs may be entered in any order.
% potential.quantized = 'no'; potential.threshold = 0.5;
% [unwph,iter,erglist] = puma_ho(psi,2,'potential',potential)
% potential.quantized = 'yes'; potential.threshold = 2;
% cliques = [1 0; 0 1; -1 1];
% [unwph,iter,erglist] = puma_ho(psi,1,'cliques',cliques,'potential',potential)
% potential.quantized = 'no';
% potential.threshold = 0.1; cliques = [1 1];
% qualitymaps = ones(size(psi,1),size(psi,2))
% [unwph,iter,erglist] = puma_ho(psi,p,'potential',potential,'cliques',cliques,'qualitymaps',qualitymaps)
%  For reference see:
%  J. Bioucas-Dias and G. Valadão, "Phase Unwrapping via Graph Cuts"
%  The algorithm here coded corresponds to a generalization for any
%  cliques set (not only vertical and horizontal).
%  J. Bioucas-Dias and J. Leitão, "The ZpiM Algorithm for Interferometric Image
%  The technique here employed is also based on the article:
%  V. kolmogorov and R. Zabih, "What Energy Functions can be Minimized via Graph
%  Cuts?".
%  European Conference on Computer Vision, May 2002.

% Default values
potential.quantized     = 'yes';
potential.threshold     = 0;
cliques                 = [1 0;0 1];
qualitymaps             = repmat(zeros(size(psi,1),size(psi,2)),[1,1,2]); qual=0;
schedule               = 1;

% test for number of required parameters

% Error out if there are not at least the two required input arguments
if (nargin-length(varargin)) ~= 2
    error('Wrong number of required parameters');
end

% Read the optional parameters
if (rem(length(varargin),2)==1)
    error('Optional parameters should always go by pairs');
elseif length(varargin)==0
    for i=1:2:(length(varargin)-1)
        % change the value of parameter
        switch varargin{i}
          case 'potential'    % potential definition
              potential = varargin{i+1};
          case 'cliques'     % cliques to consider
              cliques = varargin{i+1};
        end
    end
end
case 'qualitymaps'  % the quality maps
    qualitymaps = varargin{i+1};
    qual = 1;
    case 'schedule'  % jump size schedule
    schedule = varargin{i+1};
    otherwise
    % Hmmm, something wrong with the parameter string
    error(['Unrecognized parameter: "' varargin{i} '"']);
    end;
end;
end;
end;
end;

if (qual==1)&&(size(qualitymaps,3)~=size(cliques,1))
    error('qualitymaps must be a 3D matrix whos 3D size is equal to no. cliques. Each plane on qualitymaps corresponds to a clique.');
end

% INPUT AND INITIALIZING  %%%%%%%%%%%%%%%%%%%%%%%%
th = getfield(potential,'threshold');
quant = getfield(potential,'quantized');

[m,n] = size(psi);  % Size of input
kappa = zeros(m,n);  % Initial labeling
kappa = round(rand(m,n)*40);
iter = 0;

[cliquesm,cliquesn] = size(cliques);  % Size of input cliques
if qual ==0
    qualitymaps = repmat(zeros(size(psi,1),size(psi,2)),[1,1,size(cliques,1)]);
end
disc_bar = 1 - qualitymaps;
maxdesl = max(max(abs(cliques)));
base = zeros(2*maxdesl+2+m,2*maxdesl+2+n);

% PROCESSING  %%%%%%%%%%%%%%%%%%%%%%%%
possible_improvement = 1;
erg_previous = energy_ho(kappa,psi,base,p,cliques,disc_bar,th,quant);

while possible_improvement
    iter = iter + 1;
    erglist = [erglist erg_previous];
    remain = [];
    % Here we put a passe-partout (constant length = maxdesl+1) in the images kappa and psi
    base_kappa = zeros(2*maxdesl+2+m,2*maxdesl+2+n);
    base_kappa(maxdesl+2:maxdesl+2+m-1,maxdesl+2:maxdesl+2+n-1) = kappa;
    psi_base = zeros(2*maxdesl+2+m,2*maxdesl+2+n);
    psi_base(maxdesl+2:maxdesl+2+m-1,maxdesl+2:maxdesl+2+n-1) = psi;

    for t = 1:cliquesm
        % The allowed start and end pixels of the "interpixel" directed edge
        base_start(:,:,t) = circshift(base,[-cliques(t,1),-cliques(t,2)]).*base;
        base_end(:,:,t) = circshift(base,[cliques(t,1),cliques(t,2)]).*base;

        % By convention the difference images have the same size as the
        % original ones; the difference information is retrieved in the
        % pixel of the image that is subtracted (end of the diff vector)
        auxili = circshift(base_kappa,[cliques(t,1),cliques(t,2)]);
        t_dkappa(:,:,t) = (base_kappa - auxili);
        auxili2 = circshift(psi_base,[cliques(t,1),cliques(t,2)]);
        dpsi = auxili2 - psi_base;

        % Beyond base, we must multiply by
        % circshift(base,[cliques(t,1),cliques(t,2)]) in order to
        % account for frontier pixels that can't have links outside ROI
        a(:,:,t) = (2*pi*t_dkappa(:,:,t)-dpsi).*base.*circshift(base,[cliques(t,1),cliques(t,2)]);
        A(:,:,t) = clique_energy_ho(abs(a(:,:,t)),p,th,quant).*base.*circshift(base,[cliques(t,1),cliques(t,2)]);
        D(:,:,t) = A(:,:,t);
        C(:,:,t) = clique_energy_ho(abs(2*pi*jump_size + a(:,:,t)),p,th,quant).*base.*circshift(base,[cliques(t,1),cliques(t,2)]);
        B(:,:,t) = clique_energy_ho(abs(-2*pi*jump_size + a(:,:,t)),p,th,quant).*base.*circshift(base,[cliques(t,1),cliques(t,2)]);

        % The circshift by [-cliques(t,1),-cliques(t,2)] is due to the fact that differences are
        % retrieved in the
        % "second=end" pixel. Both "start" and "end" pixels can have source and sink
        % connections.
        source(:,:,t) = circshift((C(:,:,t)-A(:,:,t)).*((C(:,:,t)-A(:,:,t))>0),[-cliques(t,1),-cliques(t,2)]).*base_start(:,:,t);
sink(:,:,t) = 
	circshift((A(:,:,t)-C(:,:,t)).*((A(:,:,t)-C(:,:,t))>0),[-cliques(t,1),-cliques(t,2)]).*base_start(:,:,t);

source(:,:,t) = source(:,:,t) + ((D(:,:,t)-C(:,:,t)).*((D(:,:,t)-C(:,:,t))>0)).*base_end(:,:,t);

sink(:,:,t) = sink(:,:,t) + ((C(:,:,t)-D(:,:,t)).*((C(:,:,t)-D(:,:,t))>0)).*base_end(:,:,t);

end

% We get rid of the "pass-partous"
source(1:maxdesl+1,:,:)=[]; source(m+1:m+maxdesl+1,:,:)=[];
source(:,1:maxdesl+1,:)=[]; source(:,n+1:n+maxdesl+1,:)=[];
sink(1:maxdesl+1,:,:)=[]; sink(m+1:m+maxdesl+1,:,:)=[];
sink(:,1:maxdesl+1,:)=[]; sink(:,n+1:n+maxdesl+1,:)=[];
auxiliar1 = B + C - A - D;
auxiliar1(1:maxdesl+1,:,:)=[]; auxiliar1(m+1:m+maxdesl+1,:,:)=[];
auxiliar1(:,1:maxdesl+1,:)=[]; auxiliar1(:,n+1:n+maxdesl+1,:)=[];
base_start(1:maxdesl+1,:,:)=[]; base_start(m+1:m+maxdesl+1,:,:)=[];
base_start(:,1:maxdesl+1,:)=[]; base_start(:,n+1:n+maxdesl+1,:)=[];
base_end(1:maxdesl+1,:,:)=[]; base_end(m+1:m+maxdesl+1,:,:)=[];
base_end(:,1:maxdesl+1,:)=[]; base_end(:,n+1:n+maxdesl+1,:)=[];

% We construct the "remain" and the "sourcesink" matrices
for t=1:cliquesm
    start = find(base_start(:,:,t)~=0); endd = find(base_end(:,:,t)~=0);
    auxiliar2 = auxiliar1(:,:,t);
    auxiliar3 = [start endd auxiliar2(endd) zeros(size(endd,1),1)];
    remain = [remain; auxiliar3];
end

%remain = sortrows(remain,[1 2]);
sourcefinal = sum(source,3);
sinkfinal = sum(sink,3);
sourcesink = [(1:m*n)' sourcefinal(:) sinkfinal(:)];

% KAPPA RELABELING
[flow,cutside] = mincut(sourcesink,remain);
% The relabeling of each pixel relies on the cutside of that pixel:
%   if the cutside = 0 (source) then we increment the label by jump_size
%   if the cutside = 1 (sink) then the label remains unchanged
kappa_aux(cutside(:,1)) = kappa(cutside(:,1)) + (1 - cutside(:,2))*jump_size;

% CHECK ENERGY IMPROVEMENT
erg_actual = energy_ho(kappa_aux,psi,base,p,cliques,disc_bar,th,quant);
% if (erg_actual < erg_previous)
%    erg_previous = erg_actual;

77
kappa = kappa_aux;
else
    possible_improvement = 0;
    unwph = 2*pi*kappa + psi;
end

%mesh(2*pi*kappa + psi)
%surfl(2*pi*kappa + psi);shading interp; colormap(copper);
colormap(gray);
imagesc(2*pi*kappa + psi);
drawnow;

iter
end

erg_actual
jump_size

clear base_start base_end source sink auxiliar1 auxiliar2 A B C D;
end % while

set(gca,'FontSize',12);
%     title('Puma solution');
end %for

A.2 Spring Reconstruction Code

A.2.1 Fresnel Reconstruction

%% Small Spring Reconstruction - 30cm Distance
%%
% Experimental Values
d = 0.295;% meters
dx = 6.8e-6;
lambda = 514.5e-9;% meters
threshold = 0.0004;
padsize = 0;
scale = 0.35;

% Reconstruction using Fresnel Transform
H0 = abs(myFresnel(rot90(imread('0deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH0 = myLoftAndRotate(H0, scale, 0,0,0,0,0,0);

H10 = abs(myFresnel(rot90(imread('10deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH10 = myLoftAndRotate(H10, scale, 10,0,0,0,0,0);

H20 = abs(myFresnel(rot90(imread('20deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH20 = myLoftAndRotate(H20, scale, 20,0,0,0,0,0);
H30 = abs(myFresnel(rot90(imread('30deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH30 = myLoftAndRotate(H30, scale, 30, 0, 0, 0, 0, 0);

H40 = abs(myFresnel(rot90(imread('40deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH40 = myLoftAndRotate(H40, scale, 40, 0, 0, 0, 0, 0);

H50 = abs(myFresnel(rot90(imread('50deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH50 = myLoftAndRotate(H50, scale, 50, 0, 0, 0, 0, 0);

H60 = abs(myFresnel(rot90(imread('60deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH60 = myLoftAndRotate(H60, scale, 60, 0, 0, 0, 0, 0);

H70 = abs(myFresnel(rot90(imread('70deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH70 = myLoftAndRotate(H70, scale, 70, 0, 0, 0, 0, 0);

H80 = abs(myFresnel(rot90(imread('80deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH80 = myLoftAndRotate(H80, scale, 80, 0, 0, 0, 0, 0);

H90 = abs(myFresnel(rot90(imread('90deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH90 = myLoftAndRotate(H90, scale, 90, 0, 0, 0, 0, 0);

H100 = abs(myFresnel(rot90(imread('100deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH100 = myLoftAndRotate(H100, scale, 100, 0, 0, 0, 0, 0);

H110 = abs(myFresnel(rot90(imread('110deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH110 = myLoftAndRotate(H110, scale, 110, 0, 0, 0, 0, 0);

H120 = abs(myFresnel(rot90(imread('120deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH120 = myLoftAndRotate(H120, scale, 120, 0, 0, 0, 0, 0);

H130 = abs(myFresnel(rot90(imread('130deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH130 = myLoftAndRotate(H130, scale, 130, 0, 0, 0, 0, 0);

H140 = abs(myFresnel(rot90(imread('140deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH140 = myLoftAndRotate(H140, scale, 140, 0, 0, 0, 0, 0);

H150 = abs(myFresnel(rot90(imread('150deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH150 = myLoftAndRotate(H150, scale, 150, 0, 0, 0, 0, 0);
H160 = abs(myFresnel(rot90(imread('160deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH160 = myLoftAndRotate(H160,scale,160,0,0,0,0);

H170 = abs(myFresnel(rot90(imread('170deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH170 = myLoftAndRotate(H170,scale,170,0,0,0,0,0);

H180 = abs(myFresnel(rot90(imread('180deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH180 = myLoftAndRotate(H180,scale,180,0,0,0,0,0);

H190 = abs(myFresnel(rot90(imread('190deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH190 = myLoftAndRotate(H190,scale,190,0,0,0,0,0);

H200 = abs(myFresnel(rot90(imread('200deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH200 = myLoftAndRotate(H200,scale,200,0,0,0,0,0);

H210 = abs(myFresnel(rot90(imread('210deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH210 = myLoftAndRotate(H210,scale,210,0,0,0,0,0);

H220 = abs(myFresnel(rot90(imread('220deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH220 = myLoftAndRotate(H220,scale,220,0,0,0,0,0);

H230 = abs(myFresnel(rot90(imread('230deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH230 = myLoftAndRotate(H230,scale,230,0,0,0,0,0);

H240 = abs(myFresnel(rot90(imread('240deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH240 = myLoftAndRotate(H240,scale,240,0,0,0,0,0);

H250 = abs(myFresnel(rot90(imread('250deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH250 = myLoftAndRotate(H250,scale,250,0,0,0,0,0);

H260 = abs(myFresnel(rot90(imread('260deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH260 = myLoftAndRotate(H260,scale,260,0,0,0,0,0);

H270 = abs(myFresnel(rot90(imread('270deg.bmp','bmp'),2),d,lambda,dx,true,1,1,padsize));
RH270 = myLoftAndRotate(H270, scale, 270, 0, 0, 0, 0, 0);

H280 = abs(myFresnel(rot90(imread('280deg.bmp', 'bmp'), 2), d, lambda, dx, true, 1, 1, padsize));
RH280 = myLoftAndRotate(H280, scale, 280, 0, 0, 0, 0, 0);

H290 = abs(myFresnel(rot90(imread('290deg.bmp', 'bmp'), 2), d, lambda, dx, true, 1, 1, padsize));
RH290 = myLoftAndRotate(H290, scale, 290, 0, 0, 0, 0, 0);

H300 = abs(myFresnel(rot90(imread('300deg.bmp', 'bmp'), 2), d, lambda, dx, true, 1, 1, padsize));
RH300 = myLoftAndRotate(H300, scale, 300, 0, 0, 0, 0, 0);

H310 = abs(myFresnel(rot90(imread('310deg.bmp', 'bmp'), 2), d, lambda, dx, true, 1, 1, padsize));
RH310 = myLoftAndRotate(H310, scale, 310, 0, 0, 0, 0, 0);

H320 = abs(myFresnel(rot90(imread('320deg.bmp', 'bmp'), 2), d, lambda, dx, true, 1, 1, padsize));
RH320 = myLoftAndRotate(H320, scale, 320, 0, 0, 0, 0, 0);

H330 = abs(myFresnel(rot90(imread('330deg.bmp', 'bmp'), 2), d, lambda, dx, true, 1, 1, padsize));
RH330 = myLoftAndRotate(H330, scale, 330, 0, 0, 0, 0, 0);

H340 = abs(myFresnel(rot90(imread('340deg.bmp', 'bmp'), 2), d, lambda, dx, true, 1, 1, padsize));
RH340 = myLoftAndRotate(H340, scale, 340, 0, 0, 0, 0, 0);

H350 = abs(myFresnel(rot90(imread('350deg.bmp', 'bmp'), 2), d, lambda, dx, true, 1, 1, padsize));
RH350 = myLoftAndRotate(H350, scale, 350, 0, 0, 0, 0, 0);

% Display recorded hologram
figure;
imshow('0deg.bmp');

% Display reconstructed hologram
figure;
imagesc(H0);
colormap gray;

% Display the 3D Rendered Image
Htotal4 = RH0.*RH20.*RH40.*RH60.*RH80.*RH100.*RH120.*RH140.*RH160.*RH180;
Htotal5 = RH0.*RH40.*RH80.*RH120.*RH160;

figure;
myDisplay3d(Htotal,threshold)
view(-68,20)
grid on
title('18 Images (10 deg) Tomography: Fresnel Reconstruction')
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% Set tick marks on plots
length = length(RH0);
Inc = (lambda*d/((325+2*padsize)*(dx)))/scale;
Inc = Inc.*1000; % Convert to milimeters

tickmarksx = 1:floor(length/6):length;
tickmarksy = 1:floor(length/6):length;
tickmarksz = 1:floor(length/6):length;

scalex = 0:floor(length/6)*Inc:(Inc*length);
scaley = 0:floor(length/6)*Inc:(Inc*length);
scalez = 0:floor(length/6)*Inc:(Inc*length);

set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

figure;
myDisplay3d(Htotal2,threshold)
view(-68,20)
grid on
title('36 Image (10 deg) Tomography: Fresnel Reconstruction')
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% Set tick marks on plots
set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

figure;
myDisplay3d(Htotal3,threshold)
view(-68,20)
grid on
title('Tomography: Fresnel Reconstruction')
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% Set tick marks on plots
set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

figure;
myDisplay3d(Htotal4,threshold)
view(-68,20)
grid on
title('9 Image (20 Deg) Tomography: Fresnel Reconstruction')
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')
% Set tick marks on plots
set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')
figure;
myDisplay3d(Htotal5,threshold)
view(-68,20)
grid on
title('5 Image (40 Deg) Tomography: Fresnel Reconstruction')
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')
% Set tick marks on plots
set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

A.2.2 Non-Paraxial Reconstruction

%% Small Spring Reconstruction - 30cm Distance
%%
clear all
close all
%%
% Experimental Values
d = 0.295; % meters
dx = 6.8e-6;
lambda = 514.5e-9; % meters
threshold = 0.0004;
padsize = 0;
scale = 0.35;

% Reconstruction using Fresnel Transform
H0 = abs(myPropagation(rot90(imread('0deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH0 = myLoftAndRotate(H0,scale,0,0,0,0,0,0);

H10 = abs(myPropagation(rot90(imread('10deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH10 = myLoftAndRotate(H10,scale,10,0,0,0,0,0);

H20 = abs(myPropagation(rot90(imread('20deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH20 = myLoftAndRotate(H20,scale,20,0,0,0,0,0);

H30 = abs(myPropagation(rot90(imread('30deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH30 = myLoftAndRotate(H30,scale,30,0,0,0,0,0);

H40 = abs(myPropagation(rot90(imread('40deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH40 = myLoftAndRotate(H40,scale,40,0,0,0,0,0);

H50 = abs(myPropagation(rot90(imread('50deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH50 = myLoftAndRotate(H50,scale,50,0,0,0,0,0);

H60 = abs(myPropagation(rot90(imread('60deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH60 = myLoftAndRotate(H60,scale,60,0,0,0,0,0);

H70 = abs(myPropagation(rot90(imread('70deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH70 = myLoftAndRotate(H70,scale,70,0,0,0,0,0);

H80 = abs(myPropagation(rot90(imread('80deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH80 = myLoftAndRotate(H80,scale,80,0,0,0,0,0);
H90 = abs(myPropagation(rot90(imread('90deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH90 = myLoftAndRotate(H90,scale,90,0,0,0,0,0);

H100 = abs(myPropagation(rot90(imread('100deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH100 = myLoftAndRotate(H100,scale,100,0,0,0,0,0);

H110 = abs(myPropagation(rot90(imread('110deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH110 = myLoftAndRotate(H110,scale,110,0,0,0,0,0);

H120 = abs(myPropagation(rot90(imread('120deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH120 = myLoftAndRotate(H120,scale,120,0,0,0,0,0);

H130 = abs(myPropagation(rot90(imread('130deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH130 = myLoftAndRotate(H130,scale,130,0,0,0,0,0);

H140 = abs(myPropagation(rot90(imread('140deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH140 = myLoftAndRotate(H140,scale,140,0,0,0,0,0);

H150 = abs(myPropagation(rot90(imread('150deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH150 = myLoftAndRotate(H150,scale,150,0,0,0,0,0);

H160 = abs(myPropagation(rot90(imread('160deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH160 = myLoftAndRotate(H160,scale,160,0,0,0,0,0);

H170 = abs(myPropagation(rot90(imread('170deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH170 = myLoftAndRotate(H170,scale,170,0,0,0,0,0);

H180 = abs(myPropagation(rot90(imread('180deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH180 = myLoftAndRotate(H180,scale,180,0,0,0,0,0);

H190 = abs(myPropagation(rot90(imread('190deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH190 = myLoftAndRotate(H190,scale,190,0,0,0,0,0);

H200 = abs(myPropagation(rot90(imread('200deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH200 = myLoftAndRotate(H200,scale,200,0,0,0,0,0);

H210 = abs(myPropagation(rot90(imread('210deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH210 = myLoftAndRotate(H210,scale,210,0,0,0,0,0);

H220 = abs(myPropagation(rot90(imread('220deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH220 = myLoftAndRotate(H220,scale,220,0,0,0,0,0);

H230 = abs(myPropagation(rot90(imread('230deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH230 = myLoftAndRotate(H230,scale,230,0,0,0,0,0);

H240 = abs(myPropagation(rot90(imread('240deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH240 = myLoftAndRotate(H240,scale,240,0,0,0,0,0);

H250 = abs(myPropagation(rot90(imread('250deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH250 = myLoftAndRotate(H250,scale,250,0,0,0,0,0);

H260 = abs(myPropagation(rot90(imread('260deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH260 = myLoftAndRotate(H260,scale,260,0,0,0,0,0);

H270 = abs(myPropagation(rot90(imread('270deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH270 = myLoftAndRotate(H270,scale,270,0,0,0,0,0);

H280 = abs(myPropagation(rot90(imread('280deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH280 = myLoftAndRotate(H280,scale,280,0,0,0,0,0);

H290 = abs(myPropagation(rot90(imread('290deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH290 = myLoftAndRotate(H290,scale,290,0,0,0,0,0);

H300 = abs(myPropagation(rot90(imread('300deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH300 = myLoftAndRotate(H300,scale,300,0,0,0,0,0);

H310 = abs(myPropagation(rot90(imread('310deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH310 = myLoftAndRotate(H310,scale,310,0,0,0,0,0);
H320 = abs(myPropagation(rot90(imread('320deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH320 = myLoftAndRotate(H320, scale, 320, 0, 0, 0, 0, 0);

H330 = abs(myPropagation(rot90(imread('330deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH330 = myLoftAndRotate(H330, scale, 330, 0, 0, 0, 0, 0);

H340 = abs(myPropagation(rot90(imread('340deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH340 = myLoftAndRotate(H340, scale, 340, 0, 0, 0, 0, 0);

H350 = abs(myPropagation(rot90(imread('350deg.bmp','bmp'),0),d,lambda,dx,false,1,1,padsize));
RH350 = myLoftAndRotate(H350, scale, 350, 0, 0, 0, 0, 0);

% Display recorded hologram
figure;
imshow('0deg.bmp')

% Display reconstructed hologram
figure;
imagesc(H0)
colormap gray

% Display the 3D Rendered Image

figure;
myDisplay3d(Htotal, threshold)
view(-68,20)
grid on
title('Tomography: Non-paraxial Reconstruction')
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% Set tick marks on plots
length = length(RH0);
Inc = (lambda*d/((325+2*padsize)*(dx)))/scale;
Inc = Inc.*1000; % Convert to milimeters
tickmarksx = 1:floor(length/6):length;
tickmarksy = 1:floor(length/6):length;
tickmarksz = 1:floor(length/6):length;
scalex = 0:floor(length/6)*Inc:(Inc*length);
scaley = 0:floor(length/6)*Inc:(Inc*length);
scalez = 0:floor(length/6)*Inc:(Inc*length);

set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

figure (4)
myDisplay3d(Htotal2,threshold)
view(-68,20)
grid on
title('36 Image Tomography: Non-paraxial Reconstruction')
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% Set tick marks on plots
set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')
figure (5)
myDisplay3d(Htotal3,threshold)
view(-68,20)
grid on
title('Tomography: Non-paraxial Reconstruction')
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% Set tick marks on plots
set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

A.3 Die Reconstruction

A.3.1 Hologram Reconstruction

%%% Small Die Reconstruction 34.75cm
%%% %
%%% clear all
close all

%%% % Experimental Settings
clear 'length'
lambda = 514.5e-9; % in meters
d = 34.75e-2; % in meters
dx = 6.8e-6;
dy = 6.8e-6;
scale = 1.0;
padsize = 0.0;
threshold = 0.0004;

% Reconstructing Side One of Small Die
SideOne_1 = imread('SideOneHolo_1.bmp','bmp') - imread('SideOneRef_1.bmp','bmp');
SideOne_2 = imread('SideOneHolo_2.bmp','bmp') - imread('SideOneRef_2.bmp','bmp');
SideOne_3 = imread('SideOneHolo_3.bmp','bmp') - imread('SideOneRef_3.bmp','bmp');
SideOneRecon_1 = abs(myFresnel(SideOne_1(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideOneRecon_2 = abs(myFresnel(SideOne_2(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideOneRecon_3 = abs(myFresnel(SideOne_3(:,:,1),d,lambda,dx,false,1,scale,padsize));

% Resizing the x-axes to compensate for the tilting of the object
scaling = 1/cosd(52.02);
scaling_matrix = [scaling 0 0
                  0 1 0
                  0 0 1];
scaling_tform = affine2d(scaling_matrix);

SideOneRecon_1 = imwarp(SideOneRecon_1,scaling_tform);
SideOneRecon_2 = imwarp(SideOneRecon_2,scaling_tform);
SideOneRecon_3 = imwarp(SideOneRecon_3,scaling_tform);

figure;
subplot(1,3,1)
imagesc(SideOneRecon_1)
title('Side One Image #1')
subplot(1,3,2)
imagesc(SideOneRecon_2)
title('Side One Image #2')
subplot(1,3,3)
imagesc(SideOneRecon_3)
title('Side One Image #3')
colormap gray
grid on

% Set tick marks on plots
length = length(SideOneRecon_3);
Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;

Inc = Inc.*1000; % Convert to milimeters

tickmarksx = 1:floor(length/24):length;
tickmarksy = 1:floor(length/24):length;
tickmarksz = 1:floor(length/24):length;

scalex = 0:floor(length/24)*Inc:(Inc*length);
scaley = 0:floor(length/24)*Inc:(Inc*length);
scalez = 0:floor(length/24)*Inc:(Inc*length);

set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% % Reconstructing Side Two of Small Die
clear 'length'
SideTwo_1 = imread('SideTwoHolo_1.bmp','bmp') - imread('SideTwoRef_1.bmp','bmp');
SideTwo_2 = imread('SideTwoHolo_2.bmp','bmp') - imread('SideTwoRef_2.bmp','bmp');
SideTwo_3 = imread('SideTwoHolo_3.bmp','bmp') - imread('SideTwoRef_3.bmp','bmp');

SideTwoRecon_1 = abs(myFresnel(SideTwo_1(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideTwoRecon_2 = abs(myFresnel(SideTwo_2(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideTwoRecon_3 = abs(myFresnel(SideTwo_3(:,:,1),d,lambda,dx,false,1,scale,padsize));

SideTwoRecon_1 = imwarp(SideTwoRecon_1,scaling_tform);
SideTwoRecon_2 = imwarp(SideTwoRecon_2,scaling_tform);
SideTwoRecon_3 = imwarp(SideTwoRecon_3,scaling_tform);

figure;
subplot(1,3,1)
imagesc(SideTwoRecon_1)
title('Side Two Image #1')
subplot(1,3,2)
imagesc(SideTwoRecon_2)
title('Side Two Image #2')
subplot(1,3,3)
imagesc(SideTwoRecon_3)
title('Side Two Image #3')
colormap gray

% Set tick marks on plots
length = length(SideTwoRecon_3);
Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;

Inc = Inc.*1000; % Convert to milimeters

tickmarksx = 1:floor(length/24):length;
tickmarksy = 1:floor(length/24):length;
tickmarksz = 1:floor(length/24):length;

scalex = 0:floor(length/24)*Inc:(Inc*length);
scaley = 0:floor(length/24)*Inc:(Inc*length);
scalez = 0:floor(length/24)*Inc:(Inc*length);

set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmark.sy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmark.sz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% Reconstructing Side Three of Small Die
clear 'length'
SideThree_1 = imread('SideThreeHolo_1.bmp','bmp');
SideThree_2 = imread('SideThreeHolo_2.bmp','bmp');
SideThree_3 = imread('SideThreeHolo_3.bmp','bmp');
SideThreeRecon_1 = abs(myFresnel(SideThree_1(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideThreeRecon_2 = abs(myFresnel(SideThree_2(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideThreeRecon_3 = abs(myFresnel(SideThree_3(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideThreeRecon_1 = imwarp(SideThreeRecon_1,scaling_tform);
SideThreeRecon_2 = imwarp(SideThreeRecon_2,scaling_tform);
SideThreeRecon_3 = imwarp(SideThreeRecon_3,scaling_tform);

figure;
subplot(1,3,1)
imagesc(SideThreeRecon_1)
title('Side Three Image #1')
subplot(1,3,2)
imagesc(SideThreeRecon_2)
title('Side Three Image #2')
subplot(1,3,3)
imagesc(SideThreeRecon_3)
title('Side Three Image #3')
colormap gray

% Set tick marks on plots
length = length(SideThreeRecon_3);
Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;
Inc = Inc.*1000; % Convert to milimeters
% Reconstructing Side Four of Small Die

clear 'length'
SideFour_1 = imread('SideFourHolo_1.bmp','bmp') - imread('SideFourRef_1.bmp','bmp');
SideFour_2 = imread('SideFourHolo_2.bmp','bmp') - imread('SideFourRef_2.bmp','bmp');
SideFour_3 = imread('SideFourHolo_3.bmp','bmp') - imread('SideFourRef_3.bmp','bmp');

SideFourRecon_1 = abs(myFresnel(SideFour_1(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideFourRecon_2 = abs(myFresnel(SideFour_2(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideFourRecon_3 = abs(myFresnel(SideFour_3(:,:,1),d,lambda,dx,false,1,scale,padsize));

SideFourRecon_1 = imwarp(SideFourRecon_1,scaling_tform);
SideFourRecon_2 = imwarp(SideFourRecon_2,scaling_tform);
SideFourRecon_3 = imwarp(SideFourRecon_3,scaling_tform);

figure;
subplot(1,3,1)
imagesc(SideFourRecon_1)
title('Side Four Image #1')
subplot(1,3,2)
imagesc(SideFourRecon_2)
title('Side Four Image #2')
subplot(1,3,3)
imagesc(SideFourRecon_3)
title('Side Four Image #3')
colormap gray

% Set tick marks on plots
length = length(SideFourRecon_3);
Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;

Inc = Inc.*1000; % Convert to milimeters

tickmarksx = 1:floor(length/24):length;
tickmarksy = 1:floor(length/24):length;
tickmarksz = 1:floor(length/24):length;

scalex = 0:floor(length/24)*Inc:(Inc*length);
scaley = 0:floor(length/24)*Inc:(Inc*length);
scalez = 0:floor(length/24)*Inc:(Inc*length);

set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% Reconstructing Side Five of Small Die
clear 'length'
SideFive_1 = imread('SideFiveHolo_1.bmp','bmp') - imread('SideFiveRef_1.bmp','bmp');
SideFive_2 = imread('SideFiveHolo_2.bmp','bmp') - imread('SideFiveRef_2.bmp','bmp');
SideFive_3 = imread('SideFiveHolo_3.bmp','bmp') - imread('SideFiveRef_3.bmp','bmp');

SideFiveRecon_1 = abs(myFresnel(SideFive_1(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideFiveRecon_2 = abs(myFresnel(SideFive_2(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideFiveRecon_3 = abs(myFresnel(SideFive_3(:,:,1),d,lambda,dx,false,1,scale,padsize));

SideFiveRecon_1 = imwarp(SideFiveRecon_1,scaling_tform);
SideFiveRecon_2 = imwarp(SideFiveRecon_2,scaling_tform);
SideFiveRecon_3 = imwarp(SideFiveRecon_3,scaling_tform);

figure;
subplot(1,3,1)
imagesc(SideFiveRecon_1)
title('Side Five Image #1')
subplot(1,3,2)
imagesc(SideFiveRecon_2)
title('Side Five Image #2')
subplot(1,3,3)
imagesc(SideFiveRecon_3)
title('Side Five Image #3')
colormap gray
% Set tick marks on plots
length = length(SideFiveRecon_3);
Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;

Inc = Inc.*1000; % Convert to milimeters

tickmarksx = 1:floor(length/24):length;
tickmarksy = 1:floor(length/24):length;
tickmarksz = 1:floor(length/24):length;

scalex = 0:floor(length/24)*Inc:(Inc*length);
scaley = 0:floor(length/24)*Inc:(Inc*length);
scalez = 0:floor(length/24)*Inc:(Inc*length);

set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% Reconstructing Side Six of Small Die
clear 'length'
SideSix_1 = imread('SideSixHolo_1.bmp','bmp') - imread('SideSixRef_1.bmp','bmp');
SideSix_2 = imread('SideSixHolo_2.bmp','bmp') - imread('SideSixRef_2.bmp','bmp');
SideSix_3 = imread('SideSixHolo_3.bmp','bmp') - imread('SideSixRef_3.bmp','bmp');

SideSixRecon_1 = abs(myFresnel(SideSix_1(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideSixRecon_2 = abs(myFresnel(SideSix_2(:,:,1),d,lambda,dx,false,1,scale,padsize));
SideSixRecon_3 = abs(myFresnel(SideSix_3(:,:,1),d,lambda,dx,false,1,scale,padsize));

SideSixRecon_1 = imwarp(SideSixRecon_1,scaling_tform);
SideSixRecon_2 = imwarp(SideSixRecon_2,scaling_tform);
SideSixRecon_3 = imwarp(SideSixRecon_3,scaling_tform);

figure;
subplot(1,3,1)
imagesc(SideSixRecon_1)
title('Side Six Image #1')

subplot(1,3,2)
imagesc(SideSixRecon_2)
title('Side Six Image #2')

subplot(1,3,3)
imagesc(SideSixRecon_3)
title('Side Six Image #3')
colormap gray

% Set tick marks on plots
length = length(SideSixRecon_3);
Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;

Inc = Inc.*1000; % Convert to milimeters

tickmarksx = 1:floor(length/24):length;
tickmarksy = 1:floor(length/24):length;
tickmarksz = 1:floor(length/24):length;

scalex = 0:floor(length/24)*Inc:(Inc*length);
scaley = 0:floor(length/24)*Inc:(Inc*length);
scalez = 0:floor(length/24)*Inc:(Inc*length);

set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

A.3.2 Phase Unwrapping

%% Phase-Unwrapping of Die

clear 'length'

% PUMA processing Side One
p = 0.25;
figure;
cliques = [1 0; 1 1; -1 1];
potential.quantized = 'yes';
potential.threshold = 0.5;
[unwph_SO1,iter_SO1,erglist_SO1] = puma_ho(SideOneRecon_1,p,'potential','potential','cliques','cliques');
[unwph_SO2,iter_SO2,erglist_SO2] = puma_ho(SideOneRecon_2,p,'potential','potential','cliques','cliques');
[unwph_SO3,iter_SO3,erglist_SO3] = puma_ho(SideOneRecon_3,p,'potential','potential','cliques','cliques');

unwph_SideOneCrop_1 = medfilt2(unwph_SO1, [25 25]);
unwph_SideOneCrop_2 = medfilt2(unwph_SO2, [25 25]);
unwph_SideOneCrop_3 = medfilt2(unwph_SO3, [25 25]);

unwph_SideOneCrop_1 = unwph_SideOneCrop_1(440:785, 790:1135)* lambda/(2*pi);
unwph_SideOneCrop_2 = unwph_SideOneCrop_2(440:785, 790:1135)* lambda/(2*pi);
unwph_SideOneCrop_3 = unwph_SideOneCrop_3(440:785, 790:1135)* lambda/(2*pi);

figure;
subplot(1,3,1)
surfl(unwph_SideOneCrop_1);shading interp; colormap(gray);
title('Puma solution One');
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,2)
surfl(unwph_SideOneCrop_2);shading interp; colormap(gray);
title('Puma solution Two');
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,3)
surfl(unwph_SideOneCrop_3);shading interp; colormap(gray);

% % Set tick marks on plots
% title('Puma solution Three');
% length = length(unwph_SideOneCrop_3);
% Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;
% Inc = Inc.*1000; % Convert to millimeters
% tickmarksx = 1:floor(length/24):length;
% tickmarksy = 1:floor(length/24):length;
% tickmarksz = 1:floor(length/24):length;
% scalex = 0:floor(length/24)*Inc:(Inc*length);
% scaley = 0:floor(length/24)*Inc:(Inc*length);
% scalez = 0:floor(length/24)*Inc:(Inc*length);
% set(gca,'XTick',tickmarksx)
% set(gca,'XTickLabel',scalex)
% set(gca,'YTick',tickmarksy)
% set(gca,'YTickLabel',scaley)
% set(gca,'ZTick',tickmarksz)
% set(gca,'ZTickLabel',scalez)
% title('Puma solution Three');
set(gca,'FontSize',14)
xlabel('X')
ylabel('Y')
zlabel('Z')

% PUMA processing Side Two
clear 'length'
p = 0.25;
figure;
cliques = [1 0; 1 1; -1 1];
potential.quantized = 'yes';
potential.threshold = 0.5;

[unwph_ST1,iter_ST1,erglist_ST1]=
puma_ho(SideTwoRecon_1,p,'potential',potential,'cliques',cliques);
[unwph_ST2,iter_ST2,erglist_ST2]=
puma_ho(SideTwoRecon_2,p,'potential',potential,'cliques',cliques);
[unwph_ST3,iter_ST3,erglist_ST3]=
puma_ho(SideTwoRecon_3,p,'potential',potential,'cliques',cliques);

unwph_SideTwoCrop_1 = medfilt2(unwph_ST1, [25 25]);
unwph_SideTwoCrop_2 = medfilt2(unwph_ST2, [25 25]);
unwph_SideTwoCrop_3 = medfilt2(unwph_ST3, [25 25]);

unwph_SideTwoCrop_1 = unwph_SideTwoCrop_1(440:785, 760:1105)* lambda/(2*pi);
unwph_SideTwoCrop_2 = unwph_SideTwoCrop_2(440:785, 760:1105)* lambda/(2*pi);
unwph_SideTwoCrop_3 = unwph_SideTwoCrop_3(440:785, 760:1105)* lambda/(2*pi);
%

figure;
subplot(1,3,1)
surfl(unwph_SideTwoCrop_1);shading interp; colormap(gray);
title('Puma solution One');
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,2)
surfl(unwph_SideTwoCrop_2);shading interp; colormap(gray);
title('Puma solution Two');
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

subplot(1,3,3)
surfl(unwph_SideTwoCrop_3);shading interp; colormap(gray);
title('Puma solution Three');

% % Set tick marks on plots
% length = length(unwph_SideTwoCrop_3);
% Inc = (lambda*d/((1024+2*padsize)*(dx))/scale;
%
% Inc = Inc.*1000; % Convert to milimeters
%
% tickmarksx = 1:floor(length/24):length;
% tickmarksy = 1:floor(length/24):length;
% tickmarksz = 1:floor(length/24):length;
%
% scalex = 0:floor(length/24)*Inc:(Inc*length);
% scaley = 0:floor(length/24)*Inc:(Inc*length);
% scalez = 0:floor(length/24)*Inc:(Inc*length);
%
% set(gca,'XTick',tickmarksx)
% set(gca,'XTickLabel',scalex)
% set(gca,'YTick',tickmarksy)
% set(gca,'YTickLabel',scaley)
% set(gca,'ZTick',tickmarksz)
% set(gca,'ZTickLabel',scalez)
set(gca,'FontSize',14)
xlabel('X')
ylabel('Y')
zlabel('Z')

% PUMA processing Side Three
clear 'length'
p = 0.25;
figure;
cliques = [1 0; 1 1; -1 1];
potential.quantized = 'yes';
potential.threshold = 0.5;

[unwph_STh1,iter_STh1,erglist_STh1]=
puma_ho(SideThreeRecon_1,p,'potential',potential,'cliques',cliques);
[unwph_STh2,iter_STh2,erglist_STh2]=
puma_ho(SideThreeRecon_2,p,'potential',potential,'cliques',cliques);
[unwph_STh3,iter_STh3,erglist_STh3]=
puma_ho(SideThreeRecon_3,p,'potential',potential,'cliques',cliques);
unwph_SideThreeCrop_1 = medfilt2(unwph_STh1, [25 25]);
unwph_SideThreeCrop_2 = medfilt2(unwph_STh2, [25 25]);
unwph_SideThreeCrop_3 = medfilt2(unwph_STh3, [25 25]);

unwph_SideThreeCrop_1 = unwph_SideThreeCrop_1(440:785, 770:1115)*lambda/(2*pi);
unwph_SideThreeCrop_2 = unwph_SideThreeCrop_2(440:785, 770:1115)*lambda/(2*pi);
unwph_SideThreeCrop_3 = unwph_SideThreeCrop_3(440:785, 770:1115)*lambda/(2*pi);

figure;
subplot(1,3,1)
surfl(unwph_SideThreeCrop_1);shading interp; colormap(gray);title('Puma solution One');xlabel('X(m)');ylabel('Y(m)');zlabel('Z(m)');

subplot(1,3,2)
surfl(unwph_SideThreeCrop_2);shading interp; colormap(gray);title('Puma solution Two');xlabel('X(m)');ylabel('Y(m)');zlabel('Z(m)');

subplot(1,3,3)
surfl(unwph_SideThreeCrop_3);shading interp; colormap(gray);title('Puma solution Three');

% % Set tick marks on plots
% length = length(unwph_SideThreeCrop_3);
% Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;
% Inc = Inc.*1000; % Convert to milimeters
% tickmarksx = 1:floor(length/24):length;
% tickmarksy = 1:floor(length/24):length;
% tickmarksz = 1:floor(length/24):length;
% scalex = 0:floor(length/24)*Inc:(Inc*length);
% scaley = 0:floor(length/24)*Inc:(Inc*length);
% scalez = 0:floor(length/24)*Inc:(Inc*length);
% set(gca,'XTick',tickmarksx)
% set(gca,'XTickLabel',scalex)
% set(gca,'YTick',tickmarksy)
% set(gca,'YTickLabel',scaley)
% set(gca,'ZTick',tickmarksz)
% set(gca,'ZTickLabel','scalez')
set(gca,'FontSize',14)
xlabel('X')
ylabel('Y')
zlabel('Z')

% PUMA processing Side Four
clear 'length'
p = 0.25;
figure;
cliques = [1 0; 1 1; -1 1];
potential.quantized = 'yes';
potential.threshold = 0.5;

[unwph_SFo1,iter_SFo1,erglist_SFo1]=
puma_ho(SideFourRecon_1,p,'potential',potential,'cliques',cliques);
[unwph_SFo2,iter_SFo2,erglist_SFo2]=
puma_ho(SideFourRecon_2,p,'potential',potential,'cliques',cliques);
[unwph_SFo3,iter_SFo3,erglist_SFo3]=
puma_ho(SideFourRecon_3,p,'potential',potential,'cliques',cliques);

unwph_SideFourCrop_1 = medfilt2(unwph_SFo1, [25 25]);
unwph_SideFourCrop_2 = medfilt2(unwph_SFo2, [25 25]);
unwph_SideFourCrop_3 = medfilt2(unwph_SFo3, [25 25]);

unwph_SideFourCrop_1 = unwph_SideFourCrop_1(440:785, 790:1135)* lambda/(2*pi);
unwph_SideFourCrop_2 = unwph_SideFourCrop_2(440:785, 790:1135)* lambda/(2*pi);
unwph_SideFourCrop_3 = unwph_SideFourCrop_3(440:785, 790:1135)* lambda/(2*pi);

figure;
subplot(1,3,1)
surfl(unwph_SideFourCrop_1);shading interp; colormap(gray);
title('Puma solution One');
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,2)
surfl(unwph_SideFourCrop_2);shading interp; colormap(gray);
title('Puma solution Two');
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,3)
surfl(unwph_SideFourCrop_3);shading interp; colormap(gray);
title('Puma solution Three');
% % Set tick marks on plots
% length = length(unwph_SideFourCrop_3);
% Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;
%
% Inc = Inc.*1000; % Convert to milimeters
%
% tickmarksx = 1:floor(length/24):length;
% tickmarksy = 1:floor(length/24):length;
% tickmarksz = 1:floor(length/24):length;
%
% scalex = 0:floor(length/24)*Inc:(Inc*length);
% scaley = 0:floor(length/24)*Inc:(Inc*length);
% scalez = 0:floor(length/24)*Inc:(Inc*length);
%
% set(gca,'XTick',tickmarksx)
% set(gca,'XTickLabel',scalex)
% set(gca,'YTick',tickmarksy)
% set(gca,'YTickLabel',scaley)
% set(gca,'ZTick',tickmarksz)
% set(gca,'ZTickLabel',scalez)
% set(gca,'FontSize',14)
xlabel('X')
ylabel('Y')
zlabel('Z')

% PUMA processing Side Five
clear 'length'
p = 0.25;
figure;
cliques = [1 0; 1 1; -1 1];
potential.quantized = 'yes';
potential.threshold = 0.5;

[unwph_SFi1,iter_SFi1,erglist_SFi1]=
puma_ho(SideFiveRecon_1,p,'potential',potential,'cliques',cliques);
[unwph_SFi2,iter_SFi2,erglist_SFi2]=
puma_ho(SideFiveRecon_2,p,'potential',potential,'cliques',cliques);
[unwph_SFi3,iter_SFi3,erglist_SFi3]=
puma_ho(SideFiveRecon_3,p,'potential',potential,'cliques',cliques);

unwph_SideFiveCrop_1 = medfilt2(unwph_SFi1, [25 25]);
unwph_SideFiveCrop_2 = medfilt2(unwph_SFi2, [25 25]);
unwph_SideFiveCrop_3 = medfilt2(unwph_SFi3, [25 25]);

unwph_SideFiveCrop_1 = unwph_SideFiveCrop_1(440:785, 765:1110)* lambda/(2*pi);
unwph_SideFiveCrop_2 = unwph_SideFiveCrop_2(440:785, 765:1110)* lambda/(2*pi);
unwph_SideFiveCrop_3 = unwph_SideFiveCrop_3(440:785, 765:1110)* lambda/(2*pi);
figure;
subplot(1,3,1)
surfl(unwph_SideFiveCrop_1);shading interp; colormap(gray);
title('Puma solution One');
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,2)
surfl(unwph_SideFiveCrop_2);shading interp; colormap(gray);
title('Puma solution Two');
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,3)
surfl(unwph_SideFiveCrop_3);shading interp; colormap(gray);
title('Puma solution Three');

% % Set tick marks on plots
% length = length(unwph_SideFiveCrop_3);
% Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;
% % Inc = Inc.*1000; % Convert to millimeters
% % tickmarksx = 1:floor(length/24):length;
% % tickmarksy = 1:floor(length/24):length;
% % tickmarksz = 1:floor(length/24):length;
% % scalex = 0:floor(length/24)*Inc:(Inc*length);
% % scaley = 0:floor(length/24)*Inc:(Inc*length);
% % scalez = 0:floor(length/24)*Inc:(Inc*length);
% % set(gca,'XTick',tickmarksx)
% set(gca,'XTickLabel',scalex)
% set(gca,'YTick',tickmarksy)
% set(gca,'YTickLabel',scaley)
% set(gca,'ZTick',tickmarksz)
% set(gca,'ZTickLabel',scalez)
set(gca,'FontSize',14)
xlabel('X')
ylabel('Y')
zlabel('Z')

% PUMA processing Side Six
clear 'length'
p = 0.25;
figure;
cliques = [1 0; 1 1; -1 1];
potential.quantized = 'yes';
potential.threshold = 0.5;

[unwph_SS1,iter_SS1,erglist_SS1]= puma_ho(SideSixRecon_1,p,'potential',potential,'cliques',cliques);
[unwph_SS2,iter_SS2,erglist_SS2]= puma_ho(SideSixRecon_2,p,'potential',potential,'cliques',cliques);
[unwph_SS3,iter_SS3,erglist_SS3]= puma_ho(SideSixRecon_3,p,'potential',potential,'cliques',cliques);

unwph_SideSixCrop_1 = medfilt2(unwph_SS1, [25 25]);
unwph_SideSixCrop_2 = medfilt2(unwph_SS2, [25 25]);
unwph_SideSixCrop_3 = medfilt2(unwph_SS3, [25 25]);

unwph_SideSixCrop_1 = unwph_SideSixCrop_1(440:785, 790:1135)* lambda/(2*pi);
unwph_SideSixCrop_2 = unwph_SideSixCrop_2(440:785, 790:1135)* lambda/(2*pi);
unwph_SideSixCrop_3 = unwph_SideSixCrop_3(440:785, 790:1135)* lambda/(2*pi);

figure;
subplot(1,3,1)
surfl(unwph_SideSixCrop_1);shading interp; colormap(gray);
title('Puma solution One');
xlabel('X(m)');
ylabel('Y(m)');
zlabel('Z(m)');
subplot(1,3,2)
surfl(unwph_SideSixCrop_2);shading interp; colormap(gray);
title('Puma solution Two');
xlabel('X(m)');
ylabel('Y(m)');
zlabel('Z(m)');
subplot(1,3,3)
surfl(unwph_SideSixCrop_3);shading interp; colormap(gray);
title('Puma solution Three');

% Set tick marks on plots
% length = length(unwph_SideSixCrop_3);
% Inc = (lambda*d/((1024+2*padsize)*(dx)))/scale;
% Inc = Inc.*1000; % Convert to milimeters
% tickmarksx = 1:floor(length/24):length;
% tickmarksy = 1:floor(length/24):length;
% tickmarksz = 1:floor(length/24):length;
% scalex = 0:floor(length/24)*Inc:(Inc*length);
% scaley = 0:floor(length/24)*Inc:(Inc*length);
% scalez = 0:floor(length/24)*Inc:(Inc*length);
%
% set(gca,'XTick',tickmarksx)
% set(gca,'XTickLabel',scalex)
% set(gca,'YTick',tickmarksy)
% set(gca,'YTickLabel',scaley)
% set(gca,'ZTick',tickmarksz)
% set(gca,'ZTickLabel',scalez)
set(gca,'FontSize',14)
xlabel('X')
ylabel('Y')
zlabel('Z')

A.3.3 Point Cloud Generation

%% Generating 3D image using point cloud algorithms
clear 'xyzPoints' 'x' 'y' 'X' 'Y' 'SO1_xyzPoints' 'SO2_xyzPoints' 'SO3_xyzPoints' 'ST1_xyzPoints' 'ST2_xyzPoints' 'ST3_xyzPoints' 'STh1_xyzPoints' 'STh2_xyzPoints' 'STh3_xyzPoints'...
'SFo1_xyzPoints' 'SFo2_xyzPoints' 'SFo3_xyzPoints' 'SFi1_xyzPoints' 'SFi2_xyzPoints' 'SFi3_xyzPoints' 'SS1_xyzPoints' 'SS2_xyzPoints' 'SS3_xyzPoints'

for m = 1:1:346
    x(1,m) = m.*  dx;
    y(1,m) = m.*  dy;
end

[X,Y] = meshgrid(x,y);

% Side One Depth Information
SO1_xyzPoints(:,1) = X(:,);
SO1_xyzPoints(:,2) = Y(:,);
SO2_xyzPoints(:,1) = X(:,);
SO2_xyzPoints(:,2) = Y(:,);
SO3_xyzPoints(:,1) = X(:,);
SO3_xyzPoints(:,2) = Y(:,);
SO1_xyzPoints(:,3) = unwph_SideOneCrop_1(:,);
SO2_xyzPoints(:,3) = unwph_SideOneCrop_1(:,);
SO3_xyzPoints(:,3) = unwph_SideOneCrop_3(:,);

% Side Two Depth Information
ST1_xyzPoints(:,1) = X(:,);
ST1_xyzPoints(:,2) = Y(:,);
ST2_xyzPoints(:,1) = X(:,);
ST2_xyzPoints(:,2) = Y(:,);
ST3_xyzPoints(:,1) = X(:,);
ST3_xyzPoints(:,2) = Y(:,);
ST1_xyzPoints(:,3) = unwph_SideTwoCrop_1(:,);
ST2_xyzPoints(:,3) = unwph_SideTwoCrop_2(:,);
ST3_xyzPoints(:,3) = unwph_SideTwoCrop_3(:,);

% Side Three Depth Information
STh1_xyzPoints(:,1) = X(:,);
STh1_xyzPoints(:,2) = Y(:,);
STh2_xyzPoints(:,1) = X(:,);
STh2_xyzPoints(:,2) = Y(:,);
STh3_xyzPoints(:,1) = X(:,);
STh3_xyzPoints(:,2) = Y(:,);
STh1_xyzPoints(:,3) = unwph_SideThreeCrop_1(:,);
STh2_xyzPoints(:,3) = unwph_SideThreeCrop_2(:,);
STh3_xyzPoints(:,3) = unwph_SideThreeCrop_3(:,);

% Side Four Depth Information
SFo1_xyzPoints(:,1) = X(:,);
SFo1_xyzPoints(:,2) = Y(:,);
SFo2_xyzPoints(:,1) = X(:,);
SFo2_xyzPoints(:,2) = Y(:,);
SFo3_xyzPoints(:,1) = X(:,);
SFo3_xyzPoints(:,2) = Y(:,);
SFo1_xyzPoints(:,3) = unwph_SideFourCrop_1(:,);
SFo2_xyzPoints(:,3) = unwph_SideFourCrop_1(:,);
SFo3_xyzPoints(:,3) = unwph_SideFourCrop_3(:,);

% Side Five Depth Information
SFi1_xyzPoints(:,1) = X(:,);
SFi1_xyzPoints(:,2) = Y(:,);
SFi2_xyzPoints(:,1) = X(:,);
SFi2_xyzPoints(:,2) = Y(:,);
SFi3_xyzPoints(:,1) = X(:,);
SFi3_xyzPoints(:,2) = Y(:,);
SFi1_xyzPoints(:,3) = unwph_SideFiveCrop_1(:,);
SFi2_xyzPoints(:,3) = unwph_SideFiveCrop_2(:,);
SFi3_xyzPoints(:,3) = unwph_SideFiveCrop_3(:,);

% Side Six Depth Information
SS1_xyzPoints(:,1) = X(:,);
SS1_xyzPoints(:,2) = Y(:,);
SS2_xyzPoints(:,1) = X(:,);
SS2_xyzPoints(:,2) = Y(:,);
SS3_xyzPoints(:,1) = X(:,);
SS3_xyzPoints(:,2) = Y(:);  
SS1_xyzPoints(:,3) = -1*unwph_SideSixCrop_1(:);  
SS2_xyzPoints(:,3) = -1*unwph_SideSixCrop_2(:);  
SS3_xyzPoints(:,3) = -1*unwph_SideSixCrop_3(:);  

figure;  
subplot(1,3,1)  
pcshow(SO1_xyzPoints)  
title('Side One Point Cloud #1')  
xlabel('X(m)')  
ylabel('Y(m)')  
zlabel('Z(m)')  
subplot(1,3,2)  
pcshow(SO2_xyzPoints)  
title('Side One Point Cloud #2')  
xlabel('X(m)')  
ylabel('Y(m)')  
zlabel('Z(m)')  
subplot(1,3,3)  
pcshow(SO3_xyzPoints)  
title('Side One Point Cloud #3')  
shading interp  
xlabel('X(m)')  
ylabel('Y(m)')  
zlabel('Z(m)')  

figure;  
subplot(1,3,1)  
pcshow(ST1_xyzPoints)  
title('Side Two Point Cloud #1')  
xlabel('X(m)')  
ylabel('Y(m)')  
zlabel('Z(m)')  
subplot(1,3,2)  
pcshow(ST2_xyzPoints)  
title('Side Two Point Cloud #2')  
xlabel('X(m)')  
ylabel('Y(m)')  
zlabel('Z(m)')  
subplot(1,3,3)  
pcshow(ST3_xyzPoints)  
title('Side Two Point Cloud #3')  
xlabel('X(m)')  
ylabel('Y(m)')  
zlabel('Z(m)')
figure;
subplot(1,3,1)
pchow(STh1_xyzPoints)
title('Side Three Point Cloud #1')
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,2)
pchow(STh2_xyzPoints)
title('Side Three Point Cloud #2')
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,3)
pchow(STh3_xyzPoints)
title('Side Three Point Cloud #3')
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
subplot(1,3,1)
pchow(SFo1_xyzPoints)
title('Side Four Point Cloud #1')
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,2)
pchow(SFo2_xyzPoints)
title('Side Four Point Cloud #2')
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
subplot(1,3,3)
pchow(SFo3_xyzPoints)
title('Side Four Point Cloud #3')
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
subplot(1,3,1)
pchow(SFi1_xyzPoints)
title('Side Five Point Cloud #1')
xlabel('X(m)')
ylabel('Y(m)')
A.3.4 Smoothing Die Faces

%% Smoothing using fit function
%% Generating the curve fit for various die sides
SO3_CurveFit = fit([SO3_xyzPoints(:,1),SO3_xyzPoints(:,2)],SO3_xyzPoints(:,3),’lowess’);
ST3_CurveFit = fit([ST3_xyzPoints(:,1),ST3_xyzPoints(:,2)],
ST3_xyzPoints(:,3),’lowess’);
STh3_CurveFit = fit([STh3_xyzPoints(:,1),STh3_xyzPoints(:,2)],
STh3_xyzPoints(:,3),’lowess’);
SFo3_CurveFit = fit([SFo3_xyzPoints(:,1),SFo3_xyzPoints(:,2)],
SFo3_xyzPoints(:,3),’lowess’);
SFi3_CurveFit = fit([SFi3_xyzPoints(:,1),SFi3_xyzPoints(:,2)],
SFi3_xyzPoints(:,3),'lowess');
SS3_CurveFit = fit([SS3_xyzPoints(:,1),SS3_xyzPoints(:,2)],
SS3_xyzPoints(:,3),'lowess');

% Evaluate the generated curve fit to subtract from the xyz data
SO3_z = feval(SO3_CurveFit,[SO3_xyzPoints(:,1),SO3_xyzPoints(:,2)]);
ST3_z = feval(ST3_CurveFit,[ST3_xyzPoints(:,1),ST3_xyzPoints(:,2)]);
STh3_z = feval(STh3_CurveFit,[STh3_xyzPoints(:,1),STh3_xyzPoints(:,2)]);
SFo3_z = feval(SFo3_CurveFit,[SFo3_xyzPoints(:,1),SFo3_xyzPoints(:,2)]);
SFi3_z = feval(SFi3_CurveFit,[SFi3_xyzPoints(:,1),SFi3_xyzPoints(:,2)]);
SS3_z = feval(SS3_CurveFit,[SS3_xyzPoints(:,1),SS3_xyzPoints(:,2)]);

figure;
plot(SO3_CurveFit,[SO3_xyzPoints(:,1),SO3_xyzPoints(:,2)],unwph_SideOneCrop_3(:))
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
plot(ST3_CurveFit,[ST3_xyzPoints(:,1),ST3_xyzPoints(:,2)],ST3_xyzPoints(:,3));
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
plot(STh3_CurveFit,[STh3_xyzPoints(:,1),STh3_xyzPoints(:,2)],STh3_xyzPoints(:,3));
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
plot(SFo3_CurveFit,[SFo3_xyzPoints(:,1),SFo3_xyzPoints(:,2)],SFo3_xyzPoints(:,3));
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
plot(SFi3_CurveFit,[SFi3_xyzPoints(:,1),SFi3_xyzPoints(:,2)],SFi3_xyzPoints(:,3));
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
plot(SS3_CurveFit,[SS3_xyzPoints(:,1),SS3_xyzPoints(:,2)],SS3_xyzPoints(:,3));
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

% Generate new z-coordinate values for point cloud model.
% By subtracting the curve fit z values from the original values will
% generate a smoothed model

SO3_xyzPoints(:,3) = SO3_xyzPoints(:,3) - SO3_z;
ST3_xyzPoints(:,3) = ST3_xyzPoints(:,3) - ST3_z;
STh3_xyzPoints(:,3) = STh3_xyzPoints(:,3) - STh3_z;
SFo3_xyzPoints(:,3) = SFo3_xyzPoints(:,3) - SFo3_z;
SFi3_xyzPoints(:,3) = SFi3_xyzPoints(:,3) - SFi3_z;
SS3_xyzPoints(:,3) = SS3_xyzPoints(:,3) - SS3_z;

figure;
pcshow(SO3_xyzPoints)
% title('Side One Post Smoothing')
set(gca,'FontSize',14)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
pcshow(ST3_xyzPoints);
% title('Side Two Post Smoothing')
set(gca,'FontSize',14)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
pcshow(STh3_xyzPoints);
% title('Side Three Post Smoothing')
set(gca,'FontSize',14)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
pcshow(SFo3_xyzPoints);
% title('Side Four Post Smoothing')
set(gca,'FontSize',14)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
figure;
pcshow(SFi3_xyzPoints);
% title('Side Five Post Smoothing')
set(gca,'FontSize',14)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

figure;
pcshow(SS3_xyzPoints);
% title('Side Six Post Smoothing')
set(gca,'FontSize',14)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')

A.3.5 Storing Die Faces as Point Cloud Objects

%%% 3-D Die Reconstruction using pctransform(ptCloudIn,tform)

% Need to store all the xyzPoint data as a PointCloud object before
% transformations can be applied

% Storing all the xyzPoints data for the die as a ptCloud object
% This data will later be used in conjunction with the transform function
% to add the rotation and translation needed to generate the complete die.
SO1_ptCloud = pointCloud(SO1_xyzPoints);
SO2_ptCloud = pointCloud(SO2_xyzPoints);
SO3_ptCloud = pointCloud(SO3_xyzPoints);

ST1_ptCloud = pointCloud(ST1_xyzPoints);
ST2_ptCloud = pointCloud(ST2_xyzPoints);
ST3_ptCloud = pointCloud(ST3_xyzPoints);

STh1_ptCloud = pointCloud(STh1_xyzPoints);
STh2_ptCloud = pointCloud(STh2_xyzPoints);
STh3_ptCloud = pointCloud(STh3_xyzPoints);

SFo1_ptCloud = pointCloud(SFo1_xyzPoints);
SFo2_ptCloud = pointCloud(SFo2_xyzPoints);
SFo3_ptCloud = pointCloud(SFo3_xyzPoints);

SFi1_ptCloud = pointCloud(SFi1_xyzPoints);
SFi2_ptCloud = pointCloud(SFi2_xyzPoints);
SFi3_ptCloud = pointCloud(SFi3_xyzPoints);
SS1_ptCloud = pointCloud(SS1_xyzPoints);
SS2_ptCloud = pointCloud(SS2_xyzPoints);
SS3_ptCloud = pointCloud(SS3_xyzPoints);

% Applying the transformation for each given side
% Side One Transformation
SO1_ptCloudOut = pctransform(SO1_ptCloud,SideOne_tform);
SO2_ptCloudOut = pctransform(SO2_ptCloud,SideOne_tform);
SO3_ptCloudOut = pctransform(SO3_ptCloud,SideOne_tform);

% Side Two Transformation
ST1_ptCloudOut = pctransform(ST1_ptCloud,SideTwo_tform);
ST2_ptCloudOut = pctransform(ST2_ptCloud,SideTwo_tform);
ST3_ptCloudOut = pctransform(ST3_ptCloud,SideTwo_tform);

% Side Three Transformation
STh1_ptCloudOut = pctransform(STh1_ptCloud,SideThree_tform);
STh2_ptCloudOut = pctransform(STh2_ptCloud,SideThree_tform);
STh3_ptCloudOut = pctransform(STh3_ptCloud,SideThree_tform);

% Side Four Transformation
SFo1_ptCloudOut = pctransform(SFo1_ptCloud,SideFour_tform);
SFo2_ptCloudOut = pctransform(SFo2_ptCloud,SideFour_tform);
SFo3_ptCloudOut = pctransform(SFo3_ptCloud,SideFour_tform);
% Additional transformations
SFo1_ptCloudOut = pctransform(SFo1_ptCloudOut,SideFour_tform_b);
SFo2_ptCloudOut = pctransform(SFo2_ptCloudOut,SideFour_tform_b);
SFo3_ptCloudOut = pctransform(SFo3_ptCloudOut,SideFour_tform_b);

% Side Five Transformation
SFi1_ptCloudOut = pctransform(SFi1_ptCloud,SideFive_tform);
SFi2_ptCloudOut = pctransform(SFi2_ptCloud,SideFive_tform);
SFi3_ptCloudOut = pctransform(SFi3_ptCloud,SideFive_tform);
% Additional Transformation
SFi1_ptCloudOut = pctransform(SFi1_ptCloudOut,SideFive_tform_b);
SFi2_ptCloudOut = pctransform(SFi2_ptCloudOut,SideFive_tform_b);
SFi3_ptCloudOut = pctransform(SFi3_ptCloudOut,SideFive_tform_b);

% Side Six Transformation
SS1_ptCloudOut = pctransform(SS1_ptCloud,SideSix_tform);
SS2_ptCloudOut = pctransform(SS2_ptCloud,SideSix_tform);
SS3_ptCloudOut = pctransform(SS3_ptCloud,SideSix_tform);
A.3.6 Transformation Matrix Generation

%%% Transforming each figure to combine into one plot
%%% Transformation matrices will be used to apply rotation and translations
%%% to die faces during the reconstruction process.
clear 'SideTwo_tform'

theta = pi/2;

scale_factor = double(1/cosd(52.2));

% Side One
SO_Transform = [1 0 0 0
                0 1 0 0
                0 0 1 0
                0 0 0 1];

SideOne_tform = affine3d(SO_Transform);

% Side Two
% Rotating the coordinates for Side Three with respect to the x axis
ST_Transform = [1 0 0 0
                0 cos(theta) sin(theta) 0
                0 -sin(theta) cos(theta) 0
                0 -0.09e-3 -2.5e-3 1];

SideTwo_tform = affine3d(ST_Transform);

% Side Three
% Rotating the coordinates for Side Three with respect to the z axis
% The rotation value is 90 degrees
STh_Transform = [cos(theta) 0 -sin(theta) 0
                 0 1 0 0
                 sin(theta) 0 cos(theta) 0
                 2.5e-3 0 -1.4040e-04 1];

SideThree_tform = affine3d(STh_Transform);

% Side Four
% Rotating the coordinates for Side Four with respect to the z axis
% The rotation value is 180 degrees
SFo_Transform = [1 0 0 0
                0 cos(3*theta) sin(3*theta) 0
                0 1 0 0
                0 0 1 0]
0 -sin(3*theta) cos(3*theta) 0
4.7200e-5 2.5e-3 -0.1e-3 1];
SFo_Transform_b = [cos(pi) 0 -sin(pi) 0
0 1 0 0
sin(pi) 0 cos(pi) 0
0.0024 0 -0.0026 1];

SideFour_tform = affine3d(SFo_Transform);
SideFour_tform_b = affine3d(SFo_Transform_b);

% Side Five
% Rotating with respect to the z axis and translating around the z axis
% The rotation value is 270/-90 degrees
SFi_Transform = [cos(3*theta) 0 -sin(3*theta) 0
0 1 0 0
sin(3*theta) 0 cos(3*theta) 0
-0.16e-3 0 -2.5e-3 1];
SFi_Transform_b = [1 0 0 0
0 cos(pi/2) sin(pi/2) 0
0 -sin(pi/2) cos(pi/2) 0
0 -1.4040e-04 -0.0025 1];

SideFive_tform = affine3d(SFi_Transform);
SideFive_tform_b = affine3d(SFi_Transform_b);

% Side Six
SS_Transform = [1 0 0 0
0 1 0 0
0 0 1 0
0 0 -2.65e-3 1];
SideSix_tform = affine3d(SS_Transform);

A.3.7 Stitching Die Faces Together

%% Stitch the edges of each side together
clear 'Vec_SOrow1pc' 'Vec_SOrow346pc' 'Vec_SOcol1pc' 'Vec_SOcol346pc' ...
'Vec_STrow1pc' 'Vec_STrow346pc' 'Vec_STcol1pc' 'Vec_STcol346pc' ...
'Vec_STrow1pc' 'Vec_STrow346pc' 'Vec_STcol1pc' 'Vec_STcol346pc' ...
'Vec_SForow1pc' 'Vec_SForow346pc' 'Vec_SFcol1pc' 'Vec_SFcol346pc' ...
'Vec_SForow1pc' 'Vec_SForow346pc' 'Vec_SFcol1pc' 'Vec_SFcol346pc' ...
'Vec_SFirow1pc' 'Vec_SFirow346pc' 'Vec_SFicol1pc' 'Vec_SFicol346pc' ...
'Vec_SFirow1pc' 'Vec_SFirow346pc' 'Vec_SFicol1pc' 'Vec_SFicol346pc' ...
'Vec_x1' 'Vec_y1' 'Vec_z1' 'Edge1zpc' 'Edge1ypc' 'Edge2xpc' 'Edge2zpc' ...
'Corner2pc' 'Corner3pc' 'Edge3zpc' 'Edge3zpc' 'Corner4pc' 'Edge4zpc' ...
'Edge4zpc' 'Vec_SFirow1x' 'Vec_SFirow1y' 'Vec_SFirow1z' 'Corner5pc' ...
'Edge5xpc' 'Edge5xpc' 'Corner6pc' 'Corner7pc' 'Vec_SSrow346x' 'Vec_SSrow346y' ...
'Vec_SSrow346z' 'Edge5ypc' 'Edge6xpc' 'Edge6ypc' 'Edge7xpc' 'Edge7ypc' ...
% Generate an edge vectors for Side #1 and the adjacent edge of Side #2
% Find the max and min values for all components x, y, and z
% Generate a linspace function between the max and min values total points = 346
% Create a for loop to make two points(array) points adjacent to another
% Average each component(x, y, z) of the points generated in the four loop
% Create three vec variables for x, y, z

% Max and min values for the xyz points of the point cloud objects
% Side One
SO_maxx = max(SO3_ptCloudOut.Location(:,1));
SO_minx = min(SO3_ptCloudOut.Location(:,1));
SO_maxy = max(SO3_ptCloudOut.Location(:,2));
SO_miny = min(SO3_ptCloudOut.Location(:,2));
SO_maxz = max(SO3_ptCloudOut.Location(:,3));
SO_minz = min(SO3_ptCloudOut.Location(:,3));
% Side Two
ST_maxx = max(ST3_ptCloudOut.Location(:,1));
ST_minx = min(ST3_ptCloudOut.Location(:,1));
ST_maxy = max(ST3_ptCloudOut.Location(:,2));
ST_miny = min(ST3_ptCloudOut.Location(:,2));
ST_maxz = max(ST3_ptCloudOut.Location(:,3));
ST_minz = min(ST3_ptCloudOut.Location(:,3));
% Side Three
STh_maxx = max(STh3_ptCloudOut.Location(:,1));
STh_minx = min(STh3_ptCloudOut.Location(:,1));
STh_maxy = max(STh3_ptCloudOut.Location(:,2));
STh_miny = min(STh3_ptCloudOut.Location(:,2));
STh_maxz = max(STh3_ptCloudOut.Location(:,3));
STh_minz = min(STh3_ptCloudOut.Location(:,3));
% Side Four
SFO_maxx = max(SFo3_ptCloudOut.Location(:,1));
SFO_minx = min(SFo3_ptCloudOut.Location(:,1));
SFO_maxy = max(SFo3_ptCloudOut.Location(:,2));
SFO_miny = min(SFo3_ptCloudOut.Location(:,2));
SFO_maxz = max(SFo3_ptCloudOut.Location(:,3));
SFO_minz = min(SFo3_ptCloudOut.Location(:,3));
% Side Five
SFi_maxx = max(SFi3_ptCloudOut.Location(:,1));
SFi_minx = min(SFi3_ptCloudOut.Location(:,1));
SFi_maxy = max(SFi3_ptCloudOut.Location(:,2));
SFi_miny = min(SFi3_ptCloudOut.Location(:,2));
SFi_maxz = max(SFi3_ptCloudOut.Location(:,3));
SFi_minz = min(SFi3_ptCloudOut.Location(:,3));
SF_i maxz = max(SF_i3_ptCloudOut.Location(:,3));

% Side Six
SS_minx = min(SS3_ptCloudOut.Location(:,1));
SS_maxx = max(SS3_ptCloudOut.Location(:,1));
SS_miny = min(SS3_ptCloudOut.Location(:,2));
SS_maxy = max(SS3_ptCloudOut.Location(:,2));
SS_minz = min(SS3_ptCloudOut.Location(:,3));
SS_maxz = max(SS3_ptCloudOut.Location(:,3));
Npoints = 25;
z = x;

% Edge 1 for side 1 and 2
Vec_SOrow346 = unwph_SideOneCrop_3(346,:);
Vec_STrow1 = unwph_SideTwoCrop_3(1,:);
Vec_SOrow346x = [];
Vec_SOrow346y = [];
Vec_SOrow346z = [];
Vec_STrow1x = [];
Vec_STrow1y = [];
Vec_STrow1z = [];

% Creating delta values for edge 1
for m = 1:1:346
    Vec_SOrow346x = [Vec_SOrow346x x(m)];
    Vec_SOrow346y = [Vec_SOrow346y SO_miny];
    Vec_SOrow346z = [Vec_SOrow346z Vec_SOrow346z(m)];
    Vec_STrow1x = [Vec_STrow1x x(m)];
    Vec_STrow1y = [Vec_STrow1y y(346)];
    Vec_STrow1z = [Vec_STrow1z Vec_STrow1z(m)];
end
Vec_SOrow346pc(:,1) = Vec_SOrow346x(:);
Vec_SOrow346pc(:,2) = Vec_SOrow346y(:);
Vec_SOrow346pc(:,3) = Vec_SOrow346z(:);
Vec_STrow1pc(:,1) = Vec_STrow1x(:);
Vec_STrow1pc(:,2) = Vec_STrow1y(:);
Vec_STrow1pc(:,3) = Vec_STrow1z(:);

% Apply curve fit to match edge profiles to die face
% Side 1 Row 346
CFSOrow346 = fit([Vec_SOrow346pc(:,1),Vec_SOrow346pc(:,2)],Vec_SOrow346pc(:,3),'lowess');
SOrow346_z = feval(CFSOrow346,[Vec_SOrow346pc(:,1),Vec_SOrow346pc(:,2)]);
Vec_SOrow346pc(:,3) = Vec_SOrow346z(:) - SOrow346_z + SO_minz/4;

% Side 2 Row 1
CurveFit = fit([Vec_STrow1pc(:,1),Vec_STrow1pc(:,2)],Vec_STrow1pc(:,3),'lowess');
STrow1_z = feval(CurveFit,[Vec_STrow1pc(:,1),Vec_STrow1pc(:,2)]);
Vec_STrow1pc(:,3) = Vec_STrow1z(:) - STrow1_z + ST_maxz/4;
Vec_STrow1pc = pctransform(pointCloud(Vec_STrow1pc),SideTwo_tform);

% Create delta vectors from edge 1 data
delta_y1 = [];
delta_z1 = [];
for m = 1:1:346
    delta_y1 = [delta_y1 Vec_STrow1pc.Location(m,2) - Vec_SOrow346pc(m,2)];
    delta_z1 = [delta_z1 Vec_SOrow346pc(m,3) - Vec_STrow1pc.Location(m,3)];
end

% Stiching code to fill gap between 1 and 2
% Generate corner vector using delta y, delta z and die edge vectors
Edge1_x = [];
Edge1_y = [];
Edge1_z = [];
for xx = 1:1:346
    Edge1_x = [Edge1_x x(xx)];
    Edge1_y = [Edge1_y delta_y1(xx) + Vec_SOrow346pc(xx,2)];
    Edge1_z = [Edge1_z delta_z1(xx) + Vec_STrow1pc.Location(xx,3)];
end

% Convert corner into Point Cloud
Edge1pc(:,1) = Edge1_x(:);
Edge1pc(:,2) = Edge1_y(:);
Edge1pc(:,3) = Edge1_z(:);

% Generate stitching components for y and z to fill in the gap between 1
% and 2
Edge1y_x = [];
Edge1y_y = [];
Edge1y_z = [];
Edge1z_x = [];
Edge1z_y = [];
Edge1z_z = [];
for xx = 1:1:346
    % Y Component
    Edge1y_x = [Edge1y_x ones(1,Npoints).* x(xx)];
    Edge1y_y = [Edge1y_y linspace(Vec_SOrow346pc(xx,2),Edge1pc(xx,2),Npoints)];
    Edge1y_z = [Edge1y_z ones(1,Npoints).*Vec_SOrow346pc(xx,3)];
    % Z Component
    Edge1z_x = [Edge1z_x ones(1,Npoints).*x(xx)];
    Edge1z_y = [Edge1z_y ones(1,Npoints).*Vec_STrow1pc.Location(xx,2)];

\[ \text{Edge1z}_z = \text{lin} \text{space(Edge1pc(xx,3),Vec_STrow1pc.Location(xx,3),Npoints)}; \]
\end

% Convert the stitching components into Point Cloud
% Y Component
Edge1ypc(:,1) = Edge1y_x(:);
Edge1ypc(:,2) = Edge1y_y(:);
Edge1ypc(:,3) = Edge1y_z(:);
Edge1ypc = pointCloud(Edge1ypc);
% Z Component
Edge1zpc(:,1) = Edge1z_x(:);
Edge1zpc(:,2) = Edge1z_y(:);
Edge1zpc(:,3) = Edge1z_z(:);
Edge1zpc = pointCloud(Edge1zpc);

% Edge 2 "Sides 1 and 3"
Vec_SOcol346 = unwph_SideOneCrop_3(:,346);
Vec_SThcol11 = unwph_SideThreeCrop_3(:,1);
Vec_SOcol346x = [];
Vec_SOcol346y = [];
Vec_SOcol346z = [];
Vec_SThcol11x = [];
Vec_SThcol11y = [];
Vec_SThcol11z = [];

% Generate edge vector for delta values
for m = 1:1:346
    Vec_SOcol346x = [Vec_SOcol346x x(346)];
    Vec_SOcol346y = [Vec_SOcol346y y(m)];
    Vec_SOcol346z = [Vec_SOcol346z vec_SOcol346(m)];
    Vec_SThcol11x = [Vec_SThcol11x x(1)];
    Vec_SThcol11y = [Vec_SThcol11y y(m)];
    Vec_SThcol11z = [Vec_SThcol11z Vec_SThcol11(m)];
end
% Convert edge vector for delta values
Vec_SOcol346pc(:,1) = Vec_SOcol346x(:);
Vec_SOcol346pc(:,2) = Vec_SOcol346y(:);
Vec_SOcol346pc(:,3) = Vec_SOcol346z(:);
Vec_SThcol11pc(:,1) = Vec_SThcol11x(:);
Vec_SThcol11pc(:,2) = Vec_SThcol11y(:);
Vec_SThcol11pc(:,3) = Vec_SThcol11z(:);

% Apply curve fit to match edge profile to die face
% Side 1 Col 346
CFSOcol346 = fit([Vec_SOcol346pc(:,1),Vec_SOcol346pc(:,2)],Vec_SOcol346pc(:,3),'lowess');
SOcol346_z = feval(CFSOcol346,[Vec_SOcol346pc(:,1),Vec_SOcol346pc(:,2)]);
Vec_SOcol346pc(:,3) = Vec_SOcol346z(:) - SOcol346_z + SO_minz/4;

% Side 3 Row 1
CFSThcol1 = fit([Vec_SThcol1pc(:,1),Vec_SThcol1pc(:,2)],Vec_SThcol1pc(:,3),'lowess');
SThcol1_z = feval(CFSThcol1,[Vec_SThcol1pc(:,1),Vec_SThcol1pc(:,2)]);
Vec_SThcol1pc(:,3) = Vec_SThcol1z(:) - SThcol1_z + STh_maxz/4;
Vec_SThcol1pc = pctransform(pointCloud(Vec_SThcol1pc),SideThree_tform);

% Create delta vectors from edge 2 data
delta_x1 = [];
delta_z2 = [];
for m = 1:1:346
    delta_x1 = [delta_x1 Vec_SThcol1pc.Location(m,1) - Vec_SOcol346pc(m,1)];
    delta_z2 = [delta_z2 Vec_SOcol346pc(m,3) - Vec_SThcol1pc.Location(m,3)];
end

% Stitch code to fill gap between 1 and 3
% Generate corner vectors using delta x, delta z and die edge vectors
Edge2_x = [];
Edge2_y = [];
Edge2_z = [];

for m = 1:1:346
    Edge2_x = [Edge2_x Vec_SOcol346pc(m,1) + delta_x1(m)];
    Edge2_y = [Edge2_y y(m)];
    Edge2_z = [Edge2_z Vec_SThcol1pc.Location(m,3) + delta_z2(m)];
end

% Convert corner vectors into Point Cloud
Edge2pc(:,1) = Edge2_x(:,1);
Edge2pc(:,2) = Edge2_y(:,1);
Edge2pc(:,3) = Edge2_z(:,1);

% Generate stitching in x and z to connect die faces 1 and 3
Edge2x_x = [];
Edge2x_y = [];
Edge2x_z = [];
Edge2z_x = [];
Edge2z_y = [];
Edge2z_z = [];

for yy = 1:1:346
    % X Components
    Edge2x_x = [Edge2x_x linspace(Edge2pc(yy,1),Vec_SOcol346pc(yy,1),Npoints)];
Edge2x\_y = [Edge2x\_y \times \text{ones}(1,Npoints) \times \text{Vec\_SOcol346pc(yy,2)]};

Edge2x\_z = [Edge2x\_z \times \text{ones}(1,Npoints) \times \text{Vec\_SOcol346pc(yy,3)];

% Z Components
Edge2z\_x = [Edge2z\_x \times \text{ones}(1,Npoints) \times \text{Vec\_SThcol1pc.Location(yy,1)];

Edge2z\_y = [Edge2z\_y \times \text{ones}(1,Npoints) \times \text{Vec\_SThcol1pc.Location(yy,2)];

Edge2z\_z = \text{linspace(Edge2pc(yy,3),Vec\_SThcol1pc.Location(yy,3),Npoints)];

end

% Convert stitching vectors into Point Cloud
% X Components
Edge2xpc(:,1) = Edge2x\_x(:);

Edge2xpc(:,2) = Edge2x\_y(:);

Edge2xpc(:,3) = Edge2x\_z(:);

Edge2xpc = pointCloud(Edge2xpc);

% Z Components
Edge2zpc(:,1) = Edge2z\_x(:);

Edge2zpc(:,2) = Edge2z\_y(:);

Edge2zpc(:,3) = Edge2z\_z(:);

Edge2zpc = pointCloud(Edge2zpc);

% Edge 3 "Sides 1 and 4"
Vec\_SOrow1 = unwph\_SideOneCrop\_3(1,:);

Vec\_SForow1 = unwph\_SideFourCrop\_3(1,:);

Vec\_SOrow1x = [];

Vec\_SOrow1y = [];

Vec\_SOrow1z = [];

Vec\_SForow1x = [];

Vec\_SForow1y = [];

Vec\_SForow1z = [];

% Creating delta values for edge 3

for m = 1:1:346

Vec\_SOrow1x = [Vec\_SOrow1x x(m)];

Vec\_SOrow1y = [Vec\_SOrow1y y(346)];

Vec\_SOrow1z = [Vec\_SOrow1z Vec\_SOrow1(m)];

Vec\_SForow1x = [Vec\_SForow1x x(m)];

Vec\_SForow1y = [Vec\_SForow1y y(346)];

Vec\_SForow1z = [Vec\_SForow1z Vec\_SForow1(m)];

end

% Convert edge data into point cloud
Vec\_SOrow1pc(:,1) = Vec\_SOrow1x(:,);

Vec\_SOrow1pc(:,2) = Vec\_SOrow1y(:,);

Vec\_SOrow1pc(:,3) = Vec\_SOrow1z(:,);

Vec\_SForow1pc(:,1) = Vec\_SForow1x(:,);

Vec\_SForow1pc(:,2) = Vec\_SForow1y(:,);
Vec_SForow1pc(:,3) = Vec_SForow1z(:);

% Apply curve fit to match edge profile to die face
% Side 1 Row 1
CFSOrow1 = fit([Vec_SOrow1pc(:,1),Vec_SOrow1pc(:,2)],Vec_SOrow1pc(:,3),'lowess');
SOrow1_z = feval(CFSOrow1,[Vec_SOrow1pc(:,1),Vec_SOrow1pc(:,2)]);
Vec_SOrow1pc(:,3) = Vec_SOrow1z(:) - SOrow1_z + SO_minz/4;
% Side 4 Row 1
CFSForow1 = fit([Vec_SForow1pc(:,1),Vec_SForow1pc(:,2)],Vec_SForow1pc(:,3),'lowess');
SForow1_z = feval(CFSForow1,[Vec_SForow1pc(:,1),Vec_SForow1pc(:,2)]);
Vec_SForow1pc(:,3) = Vec_SForow1z(:) - SForow1_z + SFo_maxz/4;
Vec_SForow1pc = pctransform(pointCloud(Vec_SForow1pc),SideFour_tform);
Vec_SForow1pc = pctransform(Vec_SForow1pc,SideFour_tform_b);

% Create delta vectors from edge 3 data
delta_y2 = [];
delta_z3 = [];
for m = 1:1:346
    delta_y2 = [delta_y2 Vec_SForow1pc.Location(m,2) - Vec_SOrow1pc(m,2)];
    delta_z3 = [delta_z3 Vec_SOrow1pc(m,3) - Vec_SForow1pc.Location(m,3)];
end

% Stitching code to fill gap between 1 and 4
% Generate corner vectors using delta y, delta z, and die edge vectors
Edge3_x = [];
Edge3_y = [];
Edge3_z = [];

for m = 1:1:346
    Edge3_x = [Edge3_x x(m)];
    Edge3_y = [Edge3_y Vec_SOrow1pc(m,2) + delta_y2(m)];
    Edge3_z = [Edge3_z Vec_SForow1pc.Location(m,3) + delta_z3(m)];
end

% Convert the corner vectors into Point Cloud
Edge3pc(:,1) = Edge3_x(:);
Edge3pc(:,2) = Edge3_y(:);
Edge3pc(:,3) = Edge3_z(:);

% Generate Stitching in y and z to connect die faces 1 and 4
Edge3y_x = [];
Edge3y_y = [];
Edge3y_z = [];
Edge3z_x = [];

123
Edge3z_y = [];
Edge3z_z = [];

for xx = 1:1:346
% Y Components
    Edge3y_x = [Edge3y_x ones(1,Npoints) .* x(347-xx)];
    Edge3y_y = [Edge3y_y linspace(Edge3pc(347-xx,2),Vec_SOrow1pc(347-xx,2),Npoints)];
    Edge3y_z = [Edge3y_z ones(1,Npoints) .* Vec_SOrow1pc(347-xx,3)];
% Z Components
    Edge3z_x = [Edge3z_x ones(1,Npoints) .* x(347-xx)];
    Edge3z_y = [Edge3z_y ones(1,Npoints) .* Vec_SForow1pc.Location(347-xx,2)];
    Edge3z_z = [Edge3z_z linspace(Edge3pc(347-xx,3),Vec_SForow1pc.Location(347-xx,3),Npoints)];
end

% onvert stitching vectors into Point Cloud
% Y Components
    Edge3ypc(:,1) = Edge3y_x(:);
    Edge3ypc(:,2) = Edge3y_y(:);
    Edge3ypc(:,3) = Edge3y_z(:);
    Edge3ypc = pointCloud(Edge3ypc);
% Z Components
    Edge3zpc(:,1) = Edge3z_x(:);
    Edge3zpc(:,2) = Edge3z_y(:);
    Edge3zpc(:,3) = Edge3z_z(:);
    Edge3zpc = pointCloud(Edge3zpc);

% Edge 4 "Sides 1 and 5"
Vec_SOcol1 = unwph_SideOneCrop_3(:,1);
Vec_SFirow1 = unwph_SideFiveCrop_3(1,:);
Vec_SOcol1x = [];
Vec_SOcol1y = [];
Vec_SOcol1z = [];
Vec_SFirow1x = [];
Vec_SFirow1y = [];
Vec_SFirow1z = [];

% Generate edge vectors for delta values
for m = 1:1:346
    Vec_SOcol1x = [Vec_SOcol1x x(1)];
    Vec_SOcol1y = [Vec_SOcol1y y(m)];
    Vec_SOcol1z = [Vec_SOcol1z Vec_SOcol1z(m)];
    Vec_SFirow1x = [Vec_SFirow1x x(m)];
    Vec_SFirow1y = [Vec_SFirow1y y(346)];
    Vec_SFirow1z = [Vec_SFirow1z Vec_SFirow1z(m)];
end

% Convert edge vectors into Point Cloud
Vec_SOcol1pc(:,1) = Vec_SOcol1x(:);
Vec_SOcol1pc(:,2) = Vec_SOcol1y(:);
Vec_SOcol1pc(:,3) = Vec_SOcol1z(:);
Vec_SFirow1pc(:,1) = Vec_SFirow1x(:);
Vec_SFirow1pc(:,2) = Vec_SFirow1y(:);
Vec_SFirow1pc(:,3) = Vec_SFirow1z(:);

% Apply curve fit to match edge profile to die face
% Side 1 Col 1
CFSOcol1 = fit([Vec_SOcol1pc(:,1),Vec_SOcol1pc(:,2)],Vec_SOcol1pc(:,3),'lowess');
SOcol1_z = feval(CFSOcol1,[Vec_SOcol1pc(:,1),Vec_SOcol1pc(:,2)]);
Vec_SOcol1pc(:,3) = Vec_SOcol1z(:) - SOcol1_z + SO_minz/4;
% Side 5 Row 1
CFSFirow1 = fit([Vec_SFirow1pc(:,1),Vec_SFirow1pc(:,2)],Vec_SFirow1pc(:,3),'lowess');
SFirow1_z = feval(CFSFirow1,[Vec_SFirow1pc(:,1),Vec_SFirow1pc(:,2)]);
Vec_SFirow1pc(:,3) = Vec_SFirow1z(:) - SFirow1_z + SFi_maxz/4;
Vec_SFirow1pc = pctransform(pointCloud(Vec_SFirow1pc),SideFive_tform);
Vec_SFirow1pc = pctransform(Vec_SFirow1pc,SideFive_tform_b);

% Create delta vectors from edge 4 data
delta_x2 = [];
delta_z4 = [];
for m = 1:1:346
    delta_x2 = [delta_x2 Vec_SFirow1pc.Location(m,1) - Vec_SOcol1pc(m,1)];
    delta_z4 = [delta_z4 Vec_SOcol1pc(m,3) - Vec_SFirow1pc.Location(m,3)];
end

% Stitching code to fill in gap between 1 and 5
% Generate corner vectors using delta x, delta z and die edge vectors
Edge4_x = [];
Edge4_y = [];
Edge4_z = [];
for m = 1:1:346
    Edge4_x = [Edge4_x Vec_SOcol1pc(m,1) + delta_x2(m)];
    Edge4_y = [Edge4_y y(m)];
    Edge4_z = [Edge4_z Vec_SFirow1pc.Location(m,3) + delta_z4(m)];
end

% Convert corner vectors into Point Cloud
Edge4pc(:,1) = Edge4_x(:);
Edge4pc(:,2) = Edge4_y(:);
Edge4pc(:,3) = Edge4_z(:,);

% Generate stitching vectors in x and z to connect die faces 1 and 5
Edge4x_x = [];
Edge4x_y = [];
Edge4x_z = [];
Edge4z_x = [];
Edge4z_y = [];
Edge4z_z = [];

for yy = 1:1:346
    % X Components
    Edge4x_x = [Edge4x_x linspace(Edge4pc(yy,1),Vec_SOcol1pc(yy,1),Npoints)];
    Edge4x_y = [Edge4x_y ones(1,Npoints) .* y(yy)];
    Edge4x_z = [Edge4x_z ones(1,Npoints) .* Vec_SOcol1pc(yy,3)];
    % Z Components
    Edge4z_x = [Edge4z_x ones(1,Npoints) .* Vec_SFirow1pc.Location(yy,1)];
    Edge4z_y = [Edge4z_y ones(1,Npoints) .* y(yy)];
    Edge4z_z = linspace(Edge4pc(yy,3),Vec_SFirow1pc.Location(yy,3),Npoints)];
end

% Convert stitching vectors into Point Cloud
% X Components
Edge4xpc(:,1) = Edge4x_x(:);
Edge4xpc(:,2) = Edge4x_y(:);
Edge4xpc(:,3) = Edge4x_z(:);
Edge4xpc = pointCloud(Edge4xpc);
% Z Components
Edge4zpc(:,1) = Edge4z_x(:);
Edge4zpc(:,2) = Edge4z_y(:);
Edge4zpc(:,3) = Edge4z_z(:);
Edge4zpc = pointCloud(Edge4zpc);

% Edge 5 "Sides 2 and 3"
Vec_STcol346 = unwph_SideTwoCrop_3(:,346);
Vec_STThrow346 = unwph_SideThreeCrop_3(346,:);
Vec_STcol346x = [];
Vec_STcol346y = [];
Vec_STcol346z = [];
Vec_STThrow346x = [];
Vec_STThrow346y = [];
Vec_STThrow346z = [];

% Generate edge vectors for delta values
for m = 1:1:346
Vec_STcol346x = [Vec_STcol346x x(346)];
Vec_STcol346y = [Vec_STcol346y y(m)];
Vec_STcol346z = [Vec_STcol346z Vec_STcol346(m)];
Vec_SThrow346x = [Vec_SThrow346x x(m)];
Vec_SThrow346y = [Vec_SThrow346y y(1)];
Vec_SThrow346z = [Vec_SThrow346z Vec_SThrow346(m)];
end

% Convert edge vectors into Point Cloud
Vec_STcol346pc(:,1) = Vec_STcol346x(:);
Vec_STcol346pc(:,2) = Vec_STcol346y(:);
Vec_STcol346pc(:,3) = Vec_STcol346z(:);
Vec_SThrow346pc(:,1) = Vec_SThrow346x(:);
Vec_SThrow346pc(:,2) = Vec_SThrow346y(:);
Vec_SThrow346pc(:,3) = Vec_SThrow346z(:);

% Apply curve fit to match edge profile to die face
% Side 2 Col 346
CFSTcol346 = fit([Vec_STcol346pc(:,1),Vec_STcol346pc(:,2)],Vec_STcol346pc(:,3),'lowess');
STcol346_z = feval(CFSTcol346,[Vec_STcol346pc(:,1),Vec_STcol346pc(:,2)]);
Vec_STcol346pc(:,3) = Vec_STcol346z(:) - STcol346_z + ST_maxz/6;
Vec_STcol346pc = pctransform(pointCloud(Vec_STcol346pc),SideTwo_tform);
% Side 3 Col 1
CFSThrow346 = fit([Vec_SThrow346pc(:,1),Vec_SThrow346pc(:,2)],Vec_SThrow346pc(:,3),'lowess');
SThrow346_z = feval(CFSThrow346,[Vec_SThrow346pc(:,1),Vec_SThrow346pc(:,2)]);
Vec_SThrow346pc(:,3) = Vec_SThrow346z(:) - SThrow346_z + STh_maxz/6;
Vec_SThrow346pc = pctransform(pointCloud(Vec_SThrow346pc),SideThree_tform);

% Create delta vectors with edge 5 data
delta_x3 = [];
delta_y3 = [];
for m = 1:1:346
delta_x3 = [delta_x3 Vec_SThrow346pc.Location(m,1) - Vec_STcol346pc.Location(m,1)];
delta_y3 = [delta_y3 Vec_STcol346pc.Location(m,1) - Vec_SThrow346pc.Location(m,2)];
end

% Stitching code to fill gap between 2 and 3
% Generate corner vectors using delta x, delta y, and die edge vectors
Edge5_x = [];
Edge5_y = [];
Edge5_z = [];

127
for m = 1:1:346
    Edge5_x = [Edge5_x Vec_STcol346pc.Location(347-m,1) + delta_x3(347-m)];
    Edge5_y = [Edge5_y Vec_SThrow346pc.Location(347-m,2) + delta_y3(347-m)];
    Edge5_z = [Edge5_z Vec_STcol346pc.Location(347-m,3)];
end

% Convert corner vectors into Point Cloud
Edge5pc(:,1) = Edge5_x(:);
Edge5pc(:,2) = Edge5_y(:);
Edge5pc(:,3) = Edge5_z(:);

% Generate stitching vectors in x and y to connect die faces 2 and 3
Edge5x_x = [];
Edge5x_y = [];
Edge5x_z = [];
Edge5y_x = [];
Edge5y_y = [];
Edge5y_z = [];
for zz = 1:1:346
    % X Components
    Edge5x_x = [Edge5x_x linspace(Edge5pc(zz,1),Vec_STcol346pc.Location(zz,1),Npoints)];
    Edge5x_y = [Edge5x_y ones(1,Npoints) .* Vec_STcol346pc.Location(zz,2)];
    Edge5x_z = [Edge5x_z ones(1,Npoints) .* Vec_STcol346pc.Location(zz,3)];
    % Y Components
    Edge5y_x = [Edge5y_x ones(1,Npoints) .* Vec_SThrow346pc.Location(zz,1)];
    Edge5y_y = [Edge5y_y linspace(Edge5pc(zz,2),Vec_SThrow346pc.Location(zz,2),Npoints)];
    Edge5y_z = [Edge5y_z ones(1,Npoints) .* Vec_SThrow346pc.Location(zz,3)];
end

% Convert stitching vectors into Point Cloud
% X Components
Edge5xpc(:,1) = Edge5x_x(:);
Edge5xpc(:,2) = Edge5x_y(:);
Edge5xpc(:,3) = Edge5x_z(:);
Edge5xpc = pointCloud(Edge5xpc);
% Y Components
Edge5ypc(:,1) = Edge5y_x(:);
Edge5ypc(:,2) = Edge5y_y(:);
Edge5ypc(:,3) = Edge5y_z(:);
Edge5ypc = pointCloud(Edge5ypc);

% Edge 6 "Sides 3 and 4"
Vec_SThrow1 = unwph_SideThreeCrop_3(1,:);
Vec_SFocol1 = unwph_SideFourCrop_3(:,1);
Vec_STSthrow1x = [];
Vec_STSthrow1y = [];
Vec_STSthrow1z = [];
Vec_SFocol1x = [];
Vec_SFocol1y = [];
Vec_SFocol1z = [];

% Generate edge vectors for delta values
for m = 1:1:346
    Vec_STSthrow1x = [Vec_STSthrow1x x(m)];
    Vec_STSthrow1y = [Vec_STSthrow1y y(346)];
    Vec_STSthrow1z = [Vec_STSthrow1z Vec_STSthrow1(m)];
    Vec_SFocol1x = [Vec_SFocol1x x(1)];
    Vec_SFocol1y = [Vec_SFocol1y y(m)];
    Vec_SFocol1z = [Vec_SFocol1z Vec_SFocol1(m)];
end

% Convert edge vectors into Point Cloud
Vec_STSthrow1pc(:,1) = Vec_STSthrow1x(:);
Vec_STSthrow1pc(:,2) = Vec_STSthrow1y(:);
Vec_STSthrow1pc(:,3) = Vec_STSthrow1z(:);
Vec_SFocol1pc(:,1) = Vec_SFocol1x(:);
Vec_SFocol1pc(:,2) = Vec_SFocol1y(:);
Vec_SFocol1pc(:,3) = Vec_SFocol1z(:);

% Apply curve fit to match edge profile to die face
% Side 3 Col 346
CFSTSthrow1 = fit([Vec_STSthrow1pc(:,1),Vec_STSthrow1pc(:,2)],Vec_STSthrow1pc(:,3),'lowess');
SThrow1_z = feval(CFSTSthrow1,[Vec_STSthrow1pc(:,1),Vec_STSthrow1pc(:,2)]);
Vec_STSthrow1pc(:,3) = Vec_STSthrow1z(:) - SThrow1_z + STh_maxz/6;
Vec_STSthrow1pc = pctransform(pointCloud(Vec_STSthrow1pc),SideThree_tform);
% Side 4 Col 1
CFSFocol1 = fit([Vec_SFocol1pc(:,1),Vec_SFocol1pc(:,2)],Vec_SFocol1pc(:,3),'lowess');
SFocol1_z = feval(CFSFocol1,[Vec_SFocol1pc(:,1),Vec_SFocol1pc(:,2)]);
Vec_SFocol1pc(:,3) = Vec_SFocol1z(:) - SFocol1_z + SFo_maxz/6;
Vec_SFocol1pc = pctransform(pointCloud(Vec_SFocol1pc),SideFour_tform);
Vec_SFocol1pc = pctransform(Vec_SFocol1pc,SideFour_tform_b);

% Create delta vectors with edge 6 data
delta_x4 = [];
delta_y4 = [];
for m = 1:1:346
    delta_x4 = [delta_x4 Vec_STSthrow1pc.Location(m,1) - Vec_SFocol1pc.Location(m,1)];
delta_y4 = [delta_y4 Vec_SFocol1pc.Location(m,2) - Vec_SThrow1pc.Location(m,2)];
end

% Stitching code to fill gap between 3 and 4
% Generate corner vectors using delta x, delta y and die face edge vectors
Edge6_x = [];
Edge6_y = [];
Edge6_z = [];

for m = 1:1:346
    Edge6_x = [Edge6_x Vec_SFocol1pc.Location(347-m,1) + delta_x4(347-m)];
    Edge6_y = [Edge6_y Vec_SThrow1pc.Location(347-m,2) + delta_y4(347-m)];
    Edge6_z = [Edge6_z Vec_SThrow1pc.Location(347-m,3)];
end

% Convert corner vectors into Point Cloud
Edge6pc(:,1) = Edge6_x(:);
Edge6pc(:,2) = Edge6_y(:);
Edge6pc(:,3) = Edge6_z(:);

% Generate stitching vectors in x and y to connect die faces 3 and 4
Edge6x_x = [];
Edge6x_y = [];
Edge6x_z = [];

for zz = 1:1:346
    % X Components
    Edge6x_x = [Edge6x_x linspace(Edge6pc(347-zz,1),Vec_SFocol1pc.Location(347-zz,1),Npoints)];
    Edge6x_y = [Edge6x_y ones(1,Npoints) .* Vec_SFocol1pc.Location(347-zz,2)];
    Edge6x_z = [Edge6x_z ones(1,Npoints) .* Vec_SFocol1pc.Location(347-zz,3)];
% Y Components
    Edge6y_x = [Edge6y_x ones(1,Npoints) .* Vec_SThrow1pc.Location(347-zz,1)];
    Edge6y_y = [Edge6y_y linspace(Edge6pc(347-zz,2),Vec_SThrow1pc.Location(347-zz,2),Npoints)];
    Edge6y_z = [Edge6y_z ones(1,Npoints) .* Vec_SThrow1pc.Location(347-zz,3)];
end

% Convert stitching vectors into Point Cloud
% X Components
Edge6xpc(:,1) = Edge6x_x(:);
Edge6xpc(:,2) = Edge6x_y(:);
Edge6xpc(:,3) = Edge6x_z(:);
Edge6xpc = pointCloud(Edge6xpc);
% Y Components
Edge6ypc(:,1) = Edge6y_x(:);
Edge6ypc(:,2) = Edge6y_y(:);
Edge6ypc(:,3) = Edge6y_z(:);
Edge6ypc = pointCloud(Edge6ypc);

% Edge 7 "Side 4 and 5"
Vec_SFocol346 = unwph_SideFourCrop_3(:,346);
Vec_SFicol1 = unwph_SideFiveCrop_3(:,1);
Vec_SFocol346x = [];
Vec_SFocol346y = [];
Vec_SFocol346z = [];
Vec_SFicol1x = [];
Vec_SFicol1y = [];
Vec_SFicol1z = [];

% Genrate edge vectors for delta values
for m = 1:1:346
    Vec_SFocol346x = [Vec_SFocol346x x(346)];
    Vec_SFocol346y = [Vec_SFocol346y y(m)];
    Vec_SFocol346z = [Vec_SFocol346z Vec_SFocol346(m)];
    Vec_SFicol1x = [Vec_SFicol1x x(1)];
    Vec_SFicol1y = [Vec_SFicol1y y(m)];
    Vec_SFicol1z = [Vec_SFicol1z Vec_SFicol1(m)];
end

% Convert edge vectors into Point Cloud
Vec_SFocol346pc(:,1) = Vec_SFocol346x(:);
Vec_SFocol346pc(:,2) = Vec_SFocol346y(:);
Vec_SFocol346pc(:,3) = Vec_SFocol346z(:);
Vec_SFicol1pc(:,1) = Vec_SFicol1x(:);
Vec_SFicol1pc(:,2) = Vec_SFicol1y(:);
Vec_SFicol1pc(:,3) = Vec_SFicol1z(:);

% Apply curve fit to match edge profile to die face
% Side 4 Col 346
CFSFocol346 = fit([Vec_SFocol346pc(:,1),Vec_SFocol346pc(:,2)],Vec_SFocol346pc(:,3),'lowess');
SFocol346_z = feval(CFSFocol346,[Vec_SFocol346pc(:,1),Vec_SFocol346pc(:,2)]);
Vec_SFocol346pc(:,3) = Vec_SFocol346z(:) - SFocol346_z + SFo_maxz/6;
Vec_SFocol346pc = pctransform(pointCloud(Vec_SFocol346pc),SideFour_tform);
Vec_SFocol346pc = pctransform(Vec_SFocol346pc,SideFour_tform_b);

% Side 5 Col 1
CFSFicol1 = fit([Vec_SFicol1pc(:,1),Vec_SFicol1pc(:,2)],Vec_SFicol1pc(:,3),'lowess');
SFicol1_z = feval(CFSFicol1,[Vec_SFicol1pc(:,1),Vec_SFicol1pc(:,2)]);
Vec_SFicol1pc(:,3) = Vec_SFicol1lz(:,3) - SFicol1_z + SFi_maxz/6;
Vec_SFicol1pc = pctransform(pointCloud(Vec_SFicol1pc),SideFive_tform);
Vec_SFicol1pc = pctransform(Vec_SFicol1pc,SideFive_tform_b);

% Create delta vectors from edge 7 data
delta_x5 = [];
delta_y5 = [];
for m = 1:1:346
    delta_x5 = [delta_x5 Vec_SFicol1pc.Location(m,1) - Vec_SFocol346pc.Location(m,1)];
    delta_y5 = [delta_y5 Vec_SFicol1pc.Location(m,2) - Vec_SFocol346pc.Location(m,2)];
end

% Stitching code to fill gap between 4 and 5
% Generate corner vector using delta x, delta y and die face edge vectors
Edge7_x = [];
Edge7_y = [];
Edge7_z = [];
for m = 1:1:346
    Edge7_x = [Edge7_x Vec_SFocol346pc.Location(347-m,1) + delta_x5(347-m)];
    Edge7_y = [Edge7_y Vec_SFicol1pc.Location(347-m,2) + delta_y5(347-m)];
    Edge7_z = [Edge7_z Vec_SFocol346pc.Location(347-m,3)];
end

% Convert corner vectors into Point Cloud
Edge7pc(:,1) = Edge7_x(:);
Edge7pc(:,2) = Edge7_y(:);
Edge7pc(:,3) = Edge7_z(:);

% Generate stitching vectors in x and y to connect die faces 4 and 5
Edge7x_x = [];
Edge7x_y = [];
Edge7x_z = [];
Edge7y_x = [];
Edge7y_y = [];
Edge7y_z = [];
for zz = 1:1:346
    % X Components
    Edge7x_x = [Edge7x_x linspace(Edge7pc(347-zz,1),Vec_SFocol346pc.Location(347-zz,1),Npoints)];
    Edge7x_y = [Edge7x_y ones(1,Npoints) .* Vec_SFocol346pc.Location(347-zz,2)];
    Edge7x_z = [Edge7x_z ones(1,Npoints) .* Vec_SFocol346pc.Location(347-zz,3)];
    % Y Components

    % X Components
    Edge7y_x = [Edge7y_x linspace(Edge7pc(347-zz,1),Vec_SFocol346pc.Location(347-zz,1),Npoints)];
    Edge7y_y = [Edge7y_y ones(1,Npoints) .* Vec_SFocol346pc.Location(347-zz,2)];
    Edge7y_z = [Edge7y_z ones(1,Npoints) .* Vec_SFocol346pc.Location(347-zz,3)];
% Y Components
Edge7y_x = [Edge7y_x ones(1,Npoints) .* Vec_SFicol1pc.Location(347-zz,1)];
Edge7y_y = [Edge7y_y linspace(Edge7pc(347-zz,2),Vec_SFicol1pc.Location(347-
zz,2),Npoints)];
Edge7y_z = [Edge7y_z ones(1,Npoints) .* Vec_SFicol1pc.Location(347-zz,3)];
end

% Convert stitching vectors into Point Cloud
% X Components
Edge7xpc(:,1) = Edge7x_x(:);
Edge7xpc(:,2) = Edge7x_y(:);
Edge7xpc(:,3) = Edge7x_z(:);
Edge7xpc = pointCloud(Edge7xpc);
% Y Component
Edge7ypc(:,1) = Edge7y_x(:);
Edge7ypc(:,2) = Edge7y_y(:);
Edge7ypc(:,3) = Edge7y_z(:);
Edge7ypc = pointCloud(Edge7ypc);

% Edge 8 "Sides 5 and 2"
Vec_SFicol346 = unwph_SideFiveCrop_3(:,346);
Vec_STcol1 = unwph_SideTwoCrop_3(:,1);
Vec_SFicol346x = [];
Vec_SFicol346y = [];
Vec_SFicol346z = [];
Vec_STcol1x = [];
Vec_STcol1y = [];
Vec_STcol1z = [];

% Generate edge vectors for delta values
for m = 1:1:346
    Vec_SFicol346x = [Vec_SFicol346x x(346)];
    Vec_SFicol346y = [Vec_SFicol346y y(m)];
    Vec_SFicol346z = [Vec_SFicol346z Vec_SFicol346(m)];
    Vec_STcol1x = [Vec_STcol1x x(1)];
    Vec_STcol1y = [Vec_STcol1y y(m)];
    Vec_STcol1z = [Vec_STcol1z Vec_STcol1(m)];
end

% Convert edge vector into Point Cloud
Vec_SFicol346pc(:,1) = Vec_SFicol346x(:);
Vec_SFicol346pc(:,2) = Vec_SFicol346y(:);
Vec_SFicol346pc(:,3) = Vec_SFicol346z(:);
Vec_STcol1pc(:,1) = Vec_STcol1x(:);
Vec_STcol1pc(:,2) = Vec_STcol1y(:);
Vec_STcol1pc(:,3) = Vec_STcol1z(:);
% Apply curve fit to match edge profile to die face
% Side 2 Col 1
CFSTcol1 = fit([Vec_STcol1pc(:,1),Vec_STcol1pc(:,2)],Vec_STcol1pc(:,3),'lowess');
STcol1_z = feval(CFSTcol1,[Vec_STcol1pc(:,1),Vec_STcol1pc(:,2)]);
Vec_STcol1pc(:,3) = Vec_STcol1z(:) - STcol1_z + ST_maxz/6;
Vec_STcol1pc = pctransform(pointCloud(Vec_STcol1pc),SideTwo_tform);
% Side 5 Col 346
CFSFicol346 = fit([Vec_SFicol346pc(:,1),Vec_SFicol346pc(:,2)],Vec_SFicol346pc(:,3),'lowess');
SFicol346_z = feval(CFSFicol346,[Vec_SFicol346pc(:,1),Vec_SFicol346pc(:,2)]);
Vec_SFicol346pc(:,3) = Vec_SFicol346z(:) - SFicol346_z + SFi_maxz/6;
Vec_SFicol346pc = pctransform(pointCloud(Vec_SFicol346pc),SideFive_tform);

% Create delta vector from edge 8 data
delta_x6 = [];
delta_y6 = [];
for m = 1:1:346
    delta_x6 = [delta_x6 Vec_STcol1pc.Location(m,1) - Vec_SFicol346pc.Location(m,1)];
    delta_y6 = [delta_y6 Vec_SFicol346pc.Location(m,2) - Vec_STcol1pc.Location(m,2)];
end

% Stitching code to fill gap between 5 and 2
% Generate corner vectors using delta x, delta y and die edge vectors
Edge8_x = [];
Edge8_y = [];
Edge8_z = [];
for m = 1:1:346
    Edge8_x = [Edge8_x Vec_STcol1pc.Location(m,1) - delta_x6(m)];
    Edge8_y = [Edge8_y Vec_SFicol346pc.Location(m,2) - delta_y6(m)];
    Edge8_z = [Edge8_z Vec_SFicol346pc.Location(m,3)];
end

% Convert corner vectors into Point Cloud
Edge8pc(:,1) = Edge8_x(:);
Edge8pc(:,2) = Edge8_y(:);
Edge8pc(:,3) = Edge8_z(:);

% Generate stitching vectors in x and y to connect die faces 5 and 2
Edge8x_x = [];
Edge8x_y = [];
Edge8z_x = [];
Edge8z_y = [];
Edge8y_x = [];
Edge8y_y = [];
Edge8y_z = [];

for zz = 1:1:346
% X Components
    Edge8x_x = [Edge8x_x linspace(Edge8pc(347-zz,1),Vec_STcol1pc.Location(347-zz,1),Npoints)];
    Edge8x_y = [Edge8x_y ones(1,Npoints) .* Vec_STcol1pc.Location(347-zz,2)];
    Edge8x_z = [Edge8x_z ones(1,Npoints) .* Vec_STcol1pc.Location(347-zz,3)];
% Y Components
    Edge8y_x = [Edge8y_x ones(1,Npoints) .* Vec_SFicol346pc.Location(347-zz,1)];
    Edge8y_y = [Edge8y_y linspace(Edge8pc(347-zz,2),Vec_SFicol346pc.Location(347-zz,2),Npoints)];
    Edge8y_z = [Edge8y_z ones(1,Npoints) .* Vec_SFicol346pc.Location(347-zz,3)];
end

% Convert stitching vectors into Point Cloud
% X Components
    Edge8xpc(:,1) = Edge8x_x(:);
    Edge8xpc(:,2) = Edge8x_y(:);
    Edge8xpc(:,3) = Edge8x_z(:);
    Edge8xpc = pointCloud(Edge8xpc);
% Y Components
    Edge8ypc(:,1) = Edge8y_x(:);
    Edge8ypc(:,2) = Edge8y_y(:);
    Edge8ypc(:,3) = Edge8y_z(:);
    Edge8ypc = pointCloud(Edge8ypc);

% Edge 9 "Side 6 and 2"
Vec_SSrow346 = unwph_SideSixCrop_3(346,:);
Vec_STrow346 = unwph_SideTwoCrop_3(346,:);
Vec_SSrow346x = []; 
Vec_SSrow346y = []; 
Vec_SSrow346z = [];
Vec_STrow346x = [];  
Vec_STrow346y = [];  
Vec_STrow346z = [];

% Generate edge vectors for delta values
for m = 1:1:346
    Vec_STrow346x = [Vec_STrow346x x(m)];
    Vec_STrow346y = [Vec_STrow346y y(1)];
    Vec_STrow346z = [Vec_STrow346z Vec_STrow346(m)];
    Vec_SSrow346x = [Vec_SSrow346x x(m)];
    Vec_SSrow346y = [Vec_SSrow346y y(1)];
    Vec_SSrow346z = [Vec_SSrow346z Vec_SSrow346(m)];
end
% Convert edge vector into Point Cloud
Vec_STrow346pc(:,1) = Vec_STrow346x(:);
Vec_STrow346pc(:,2) = Vec_STrow346y(:);
Vec_STrow346pc(:,3) = Vec_STrow346z(:);
Vec_SSrow346pc(:,1) = Vec_SSrow346x(:);
Vec_SSrow346pc(:,2) = Vec_SSrow346y(:);
Vec_SSrow346pc(:,3) = Vec_SSrow346z(:);

% Apply curve fit to match edge profile to die face
% Side 2 Row 346
CFSTrow346 = fit([Vec_STrow346pc(:,1),Vec_STrow346pc(:,2)],Vec_STrow346pc(:,3),'lowess');
STrow346_z = feval(CFSTrow346,[Vec_STrow346pc(:,1),Vec_STrow346pc(:,2)]);
Vec_STrow346pc(:,3) = Vec_STrow346z(:) - STrow346_z + ST_maxz/4;
Vec_STrow346pc = pctransform(pointCloud(Vec_STrow346pc),SideTwo_tform);

% Side 6 Row 346
CFSSrow346 = fit([Vec_SSrow346pc(:,1),Vec_SSrow346pc(:,2)],Vec_SSrow346pc(:,3),'lowess');
SSrow346_z = feval(CFSSrow346,[Vec_SSrow346pc(:,1),Vec_SSrow346pc(:,2)]);
Vec_SSrow346pc(:,3) = Vec_SSrow346z(:) - SSrow346_z - SO_minz/2;
Vec_SSrow346pc = pctransform(pointCloud(Vec_SSrow346pc),SideSix_tform);

% Create delta vectors from edge 9 data
delta_y7 = [];
delta_z5 = [];
for m = 1:1:346
    delta_y7 = [delta_y7 Vec_SSrow346pc.Location(m,2) - Vec_STrow346pc.Location(m,2)];
    delta_z5 = [delta_z5 Vec_STrow346pc.Location(m,3) - Vec_SSrow346pc.Location(m,3)];
end

% Stitching code to fill gap between 6 and 2
% Generate corner vectors using delta y, delta z and die edge vectors
Edge9_x = [];
Edge9_y = [];
Edge9_z = [];
for m = 1:1:346
    Edge9_x = [Edge9_x x(m)];
    Edge9_y = [Edge9_y Vec_SSrow346pc.Location(m,2) - delta_y7(m)];
    Edge9_z = [Edge9_z Vec_STrow346pc.Location(m,3) - delta_z5(m)];
end

% Convert corner vectors into Point Cloud
% Generate stitching vectors in y and z to connect die faces 6 and 2
Edge9y_x = [];  
Edge9y_y = [];  
Edge9y_z = [];  
Edge9z_x = [];  
Edge9z_y = [];  
Edge9z_z = [];

for xx = 1:1:346
  % Y Components
  Edge9y_x = [Edge9y_x ones(1,Npoints) .* x(xx)];  
  Edge9y_y = linspace(Edge9pc(xx,2),Vec_SSrow346pc.Location(xx,2),Npoints)];  
  Edge9y_z = [Edge9y_z ones(1,Npoints) .* Vec_SSrow346pc.Location(xx,3)];  
  % Z Components
  Edge9z_x = [Edge9z_x ones(1,Npoints) .* x(xx)];  
  Edge9z_y = linspace(Edge9pc(xx,3),Vec_STrow346pc.Location(xx,3),Npoints)];  
  Edge9z_z = [Edge9z_z ones(1,Npoints) .* Vec_STrow346pc.Location(xx,3)];
end

% Convert stitching vectors into Point Cloud
% Y Components
Edge9ypc(:,1) = Edge9y_x(:);  
Edge9ypc(:,2) = Edge9y_y(:);  
Edge9ypc(:,3) = Edge9y_z(:);  
Edge9ypc = pointCloud(Edge9ypc);
% Z Components
Edge9zpc(:,1) = Edge9z_x(:);  
Edge9zpc(:,2) = Edge9z_y(:);  
Edge9zpc(:,3) = Edge9z_z(:);  
Edge9zpc = pointCloud(Edge9zpc);

% Edge 10 "Sides 6 and 3"
Vec_SScol346 = unwph_SideSixCrop_3(:,346);  
Vec_SThcol346 = unwph_SideThreeCrop_3(:,346);  
Vec_SScol346x = [];  
Vec_SScol346y = [];  
Vec_SScol346z = [];  
Vec_SThcol346x = [];  
Vec_SThcol346y = [];  
Vec_SThcol346z = [];
% Generate edge vectors for delta values
for m = 1:1:346
    Vec_SThcol346x = [Vec_SThcol346x x(346)];
    Vec_SThcol346y = [Vec_SThcol346y y(m)];
    Vec_SThcol346z = [Vec_SThcol346z Vec_SThcol346(m)];
    Vec_SScol346x = [Vec_SScol346x x(346)];
    Vec_SScol346y = [Vec_SScol346y y(m)];
    Vec_SScol346z = [Vec_SScol346z Vec_SScol346(m)];
end

% Convert edge vectors into Point Cloud
Vec_SThcol346pc(:,1) = Vec_SThcol346x(:);
Vec_SThcol346pc(:,2) = Vec_SThcol346y(:);
Vec_SThcol346pc(:,3) = Vec_SThcol346z(:);
Vec_SScol346pc(:,1) = Vec_SScol346x(:);
Vec_SScol346pc(:,2) = Vec_SScol346y(:);
Vec_SScol346pc(:,3) = Vec_SScol346z(:);

% Apply curve fit to match edge profile to die face
% Side 3 Row 346
CFSThcol346 = fit([Vec_SThcol346pc(:,1),Vec_SThcol346pc(:,2)],Vec_SThcol346pc(:,3),'lowess');
SThcol346_z = feval(CFSThcol346,[Vec_SThcol346pc(:,1),Vec_SThcol346pc(:,2)]);
Vec_SThcol346pc(:,3) = Vec_SThcol346z(:) - SThcol346_z + STh_maxz/6;
Vec_SThcol346pc = pctransform(pointCloud(Vec_SThcol346pc),SideThree_tform);

% Side 6 Col 346
CFSScol346 = fit([Vec_SScol346pc(:,1),Vec_SScol346pc(:,2)],Vec_SScol346pc(:,3),'lowess');
SScol346_z = feval(CFSScol346,[Vec_SScol346pc(:,1),Vec_SScol346pc(:,2)]);
Vec_SScol346pc(:,3) = Vec_SScol346z(:) - SScol346_z - SO_minz/4;
Vec_SScol346pc = pctransform(pointCloud(Vec_SScol346pc),SideSix_tform);

% Create delta vectors from edge 10 data
delta_x7 = [];
delta_z6 = [];
for m = 1:1:346
delta_x7 = [delta_x7 Vec_SThcol346pc.Location(m,1) - Vec_SScol346pc.Location(m,1)];
Vec_SScol346pc.Location(m,1)];
delta_z6 = [delta_z6 Vec_SThcol346pc.Location(m,3) - Vec_SScol346pc.Location(m,3)];
end

% Stitching code to fill gap between 6 and 3
% Generate corner vectors using delta x, delta z, and die edge vectors
Edge10_x = [];

138
Edge10_y = []; 
Edge10_z = []; 

for m = 1:1:346 
    Edge10_x = [Edge10_x Vec_SScol346pc.Location(m,1) + delta_x7(m)]; 
    Edge10_y = [Edge10_y y(m)]; 
    Edge10_z = [Edge10_z Vec_SThcol346pc.Location(m,3) - delta_z6(m)]; 
end 

% Convert corner vectors into Point Cloud 
Edge10pc(:,1) = Edge10_x(:); 
Edge10pc(:,2) = Edge10_y(:); 
Edge10pc(:,3) = Edge10_z(:); 

% Generate stitching vectors in x and z to connect die faces 6 and 3 
Edge10x_x = []; 
Edge10x_y = []; 
Edge10x_z = []; 
Edge10z_x = []; 
Edge10z_y = []; 
Edge10z_z = []; 

for yy = 1:1:346 
    % X Components 
    Edge10x_x = [Edge10x_x linspace(Edge10pc(yy,1),Vec_SScol346pc.Location(yy,1),Npoints)]; 
    Edge10x_y = [Edge10x_y ones(1,Npoints) .* y(yy)]; 
    Edge10x_z = [Edge10x_z ones(1,Npoints) .* Vec_SScol346pc.Location(yy,3)]; 
    % Z Components 
    Edge10z_x = [Edge10z_x ones(1,Npoints) .* Vec_SThcol346pc.Location(yy,1)]; 
    Edge10z_y = [Edge10z_y ones(1,Npoints) .* y(yy)]; 
    Edge10z_z = [Edge10z_z linspace(Edge10pc(yy,3),Vec_SThcol346pc.Location(yy,3),Npoints)]; 
end 

% Convert stitching vectors into Point Cloud 
% X Components 
Edge10xpc(:,1) = Edge10x_x; 
Edge10xpc(:,2) = Edge10x_y; 
Edge10xpc(:,3) = Edge10x_z; 
Edge10xpc = pointCloud(Edge10xpc); 

% Z Components 
Edge10zpc(:,1) = Edge10z_x; 
Edge10zpc(:,2) = Edge10z_y; 
Edge10zpc(:,3) = Edge10z_z; 
Edge10zpc = pointCloud(Edge10zpc);
% Edge 11 "Sides 6 and 4"
Vec_SSrow1 = unwph_SideSixCrop_3(1,:);
Vec_SForow346 = unwph_SideFourCrop_3(346,:);
Vec_SSrow1x = [];
Vec_SSrow1y = [];
Vec_SSrow1z = [];
Vec_SForow346x = [];
Vec_SForow346y = [];
Vec_SForow346z = [];

% Generate edge vector for delta values
for m = 1:1:346
    Vec_SForow346x = [Vec_SForow346x x(m)];
    Vec_SForow346y = [Vec_SForow346y y(1)];
    Vec_SForow346z = [Vec_SForow346z Vec_SForow346(m)];
    Vec_SSrow1x = [Vec_SSrow1x x(m)];
    Vec_SSrow1y = [Vec_SSrow1y y(346)];
    Vec_SSrow1z = [Vec_SSrow1z Vec_SScol346(m)];
end

% Convert edge vectors into Point Cloud
Vec_SForow346pc(:,1) = Vec_SForow346x(:);
Vec_SForow346pc(:,2) = Vec_SForow346y(:);
Vec_SForow346pc(:,3) = Vec_SForow346z(:);
Vec_SSrow1pc(:,1) = Vec_SSrow1x(:);
Vec_SSrow1pc(:,2) = Vec_SSrow1y(:);
Vec_SSrow1pc(:,3) = Vec_SSrow1z(:);

% Apply curve fit to match edge profile to die face
% Side 4 Row 346
CFSForow346 = fit([Vec_SForow346pc(:,1),Vec_SForow346pc(:,2)],Vec_SForow346pc(:,3),'lowess');
SForow346_z = feval(CFSForow346,[Vec_SForow346pc(:,1),Vec_SForow346pc(:,2)]);
Vec_SForow346pc(:,3) = Vec_SForow346z(:,1) - SForow346_z + SFo_maxz/2;
Vec_SForow346pc = pctransform(pointCloud(Vec_SForow346pc),SideFour_tform);

% Side 6 Row 1
CFSSrow1 = fit([Vec_SSrow1pc(:,1),Vec_SSrow1pc(:,2)],Vec_SSrow1pc(:,3),'lowess');
SSrow1_z = feval(CFSSrow1,[Vec_SSrow1pc(:,1),Vec_SSrow1pc(:,2)]);
Vec_SSrow1pc(:,3) = Vec_SSrow1z(:,1) - SSrow1_z - SO_minz/2;
Vec_SSrow1pc = pctransform(pointCloud(Vec_SSrow1pc),SideSix_tform);

% Create delta vectors from edge 11 data
delta_y8 = [];
delta_z7 = [];

140
for m = 1:1:346
    delta_y8 = [delta_y8 Vec_SForow346pc.Location(m,2) - Vec_SSSrow1pc.Location(m,2)];
    delta_z7 = [delta_z7 Vec_SForow346pc.Location(m,3) - Vec_SSSrow1pc.Location(m,3)];
end

% Stitching code to fill gap between 6 and 4
% Generate corner vector using delta y, delta z and die edge vectors
Edge11_x = [];
Edge11_y = [];
Edge11_z = [];
for m = 1:1:346
    Edge11_x = [Edge11_x x(m)];
    Edge11_y = [Edge11_y Vec_SSrow1pc.Location(m,2) + delta_y8(m)];
    Edge11_z = [Edge11_z Vec_SForow346pc.Location(m,3) - delta_z7(m)];
end

% Convert corner vectors into Point Cloud
Edge11pc(:,1) = Edge11_x(:);
Edge11pc(:,2) = Edge11_y(:);
Edge11pc(:,3) = Edge11_z(:);

% Generate stitching vectors in y and z to connect die faces 6 and 4
Edge11y_x = [];
Edge11y_y = [];
Edge11y_z = [];
Edge11z_x = [];
Edge11z_y = [];
Edge11z_z = [];
for xx = 1:1:346
    % Y Components
    Edge11y_x = [Edge11y_x ones(1,Npoints).*x(xx)];
    Edge11y_y = [Edge11y_y linspace(Edge11pc(xx,2),Vec_SSrow1pc.Location(xx,2),Npoints)];
    Edge11y_z = [Edge11y_z ones(1,Npoints).*Vec_SSrow1pc.Location(xx,3)];
    % Z Components
    Edge11z_x = [Edge11z_x ones(1,Npoints).*x(xx)];
    Edge11z_y = [Edge11z_y ones(1,Npoints).*Vec_SForow346pc.Location(xx,2)];
    Edge11z_z = [Edge11z_z linspace(Edge11pc(xx,3),Vec_SForow346pc.Location(xx,3),Npoints)];
end

% Convert stitching vectors into Point Cloud
% Y Components
Edge11ypc(:,1) = Edge11y_x(:);
Edge11ypc(:,2) = Edge11y_y(:);
Edge11ypc(:,3) = Edge11y_z(:);
Edge11ypc = pointCloud(Edge11ypc);
% Z Components
Edge11zpc(:,1) = Edge11z_x(:);
Edge11zpc(:,2) = Edge11z_y(:);
Edge11zpc(:,3) = Edge11z_z(:);
Edge11zpc = pointCloud(Edge11zpc);

% Edge 12 "Sides 6 and 5"
Vec_SFirow346 = unwph_SideFiveCrop_3(346,:);
Vec_SScol1 = unwph_SideSixCrop_3(:,1);
Vec_SFirow346x = [];
Vec_SFirow346y = [];
Vec_SFirow346z = [];
Vec_SScol1x = [];
Vec_SScol1y = [];
Vec_SScol1z = [];

% Generate edge vectors for delta values
for m = 1:1:346
    Vec_SFirow346x = [Vec_SFirow346x x(m)];
    Vec_SFirow346y = [Vec_SFirow346y y(1)];
    Vec_SFirow346z = [Vec_SFirow346z Vec_SFirow346(m)];
    Vec_SScol1x = [Vec_SScol1x x(1)];
    Vec_SScol1y = [Vec_SScol1y y(m)];
    Vec_SScol1z = [Vec_SScol1z Vec_SScol1(m)];
end

% Convert edge vector into Point Cloud
Vec_SFirow346pc(:,1) = Vec_SFirow346x(:);
Vec_SFirow346pc(:,2) = Vec_SFirow346y(:);
Vec_SFirow346pc(:,3) = Vec_SFirow346z(:);
Vec_SScol1pc(:,1) = Vec_SScol1x(:);
Vec_SScol1pc(:,2) = Vec_SScol1y(:);
Vec_SScol1pc(:,3) = Vec_SScol1z(:);

% Apply curve fit to match edge profile to die face
% Side 5 Row 346
CFSFirow346 = fit([Vec_SFirow346pc(:,1),Vec_SFirow346pc(:,2)],Vec_SFirow346pc(:,3),'lowess');
SFirow346_z = feval(CFSFirow346,[Vec_SFirow346pc(:,1),Vec_SFirow346pc(:,2)]);
Vec_SFirow346pc(:,3) = Vec_SFirow346z(:) - SFirow346_z + SFi_maxz/6;
Vec_SFirow346pc = pctransform(pointCloud(Vec_SFirow346pc),SideFive_tform);
Vec_SFirow346pc = pctransform(Vec_SFirow346pc,SideFive_tform_b);
% Side 6 Row 346
CFSScol1 = fit([Vec_SScol1pc(:,1),Vec_SScol1pc(:,2)],Vec_SScol1pc(:,3),'lowess');
SScol1_z = feval(CFSScol1,[Vec_SScol1pc(:,1),Vec_SScol1pc(:,2)]);
Vec_SScol1pc(:,3) = Vec_SScol1z(:) - SScol1_z - SO_minz/6;
Vec_SScol1pc = pctransform(pointCloud(Vec_SScol1pc),SideSix_tform);

% Create delta vectors from edge 12 data
delta_x8 = [];
delta_z8 = [];
for m = 1:1:346
    delta_x8 = [delta_x8 Vec_SScol1pc.Location(m,1) - Vec_SFirow346pc.Location(m,1)];
    delta_z8 = [delta_z8 Vec_SFirow346pc.Location(m,3) - Vec_SScol1pc.Location(m,3)];
end

% Stitching code to fill gap between 6 and 5
% Generate corner vector using delta x, delta z and die edge vectors
Edge12_x = [];
Edge12_y = [];
Edge12_z = [];
for m = 1:1:346
    Edge12_x = [Edge12_x Vec_SScol1pc.Location(m,1) - delta_x8(m)];
    Edge12_y = [Edge12_y ones(1,Npoints) .* y(m)];
    Edge12_z = [Edge12_z Vec_SFirow346pc.Location(m,3) - delta_z8(m)];
end

% Convert corner vectors into Point Cloud
Edge12pc(:,1) = Edge12_x(:);
Edge12pc(:,2) = Edge12_y(:);
Edge12pc(:,3) = Edge12_z(:);

% Generate stitching vectors in x and z to connect die faces 6 and 5
Edge12x_x = [];
Edge12x_y = [];
Edge12x_z = [];
Edge12z_x = [];
Edge12z_y = [];
Edge12z_z = [];
for yy = 1:1:346
    % X Components
    Edge12x_x = [Edge12x_x linspace(Edge12pc(yy,1),Vec_SScol1pc.Location(yy,1),Npoints)];
    Edge12x_y = [Edge12x_y ones(1,Npoints) .* y(yy)];
    Edge12x_z = [Edge12x_z Vec_SFirow346pc.Location(yy,3) - delta_z8(m)];
end
Edge12x_z = [Edge12x_z ones(1,Npoints) .* Vec_SScol1pc.Location(yy,3)];
% Z Components
Edge12z_x = [Edge12z_x ones(1,Npoints) .* Vec_SFirow346pc.Location(yy,1)];
Edge12z_y = [Edge12z_y ones(1,Npoints) .* y(yy)];
Edge12z_z = linspace(Edge12pc(yy,3),Vec_SFirow346pc.Location(yy,3),Npoints)];
end

% Convert stitching vectors into Point Cloud

% X Components
Edge12xpc(:,1) = Edge12x_x(:);
Edge12xpc(:,2) = Edge12x_y(:);
Edge12xpc(:,3) = Edge12x_z(:);
Edge12xpc = pointCloud(Edge12xpc);

% Z Components
Edge12zpc(:,1) = Edge12z_x(:);
Edge12zpc(:,2) = Edge12z_y(:);
Edge12zpc(:,3) = Edge12z_z(:);
Edge12zpc = pointCloud(Edge12zpc);

A.3.8 Final Die Reconstruction

gridstep = 0.25e-9;
DieRecon_1 = pcmerge(SO3_ptCloudOut,ST3_ptCloudOut,gridstep);
DieRecon_2 = pcmerge(DieRecon_1,Edge1ypc,gridstep);
DieRecon_3 = pcmerge(DieRecon_2,Edge1zpc,gridstep);
DieRecon_4 = pcmerge(DieRecon_3,STh3_ptCloudOut,gridstep);
DieRecon_5 = pcmerge(DieRecon_4,Edge2xpc,gridstep);
DieRecon_6 = pcmerge(DieRecon_5,Edge2zpc,gridstep);
DieRecon_7 = pcmerge(DieRecon_6,SSFi3_ptCloudOut,gridstep);
DieRecon_8 = pcmerge(DieRecon_7,Edge3ypc,gridstep);
DieRecon_9 = pcmerge(DieRecon_8,Edge3zpc,gridstep);
DieRecon_10 = pcmerge(DieRecon_9,SSFi3_ptCloudOut,gridstep);
DieRecon_11 = pcmerge(DieRecon_10,Edge4xpc,gridstep);
DieRecon_12 = pcmerge(DieRecon_11,Edge4zpc,gridstep);
DieRecon_13 = pcmerge(DieRecon_12,Edge5xpc,gridstep);
DieRecon_14 = pcmerge(DieRecon_13,Edge5ypc,gridstep);
DieRecon_15 = pcmerge(DieRecon_14,Edge6xpc,gridstep);
DieRecon_16 = pcmerge(DieRecon_15,Edge6ypc,gridstep);
DieRecon_17 = pcmerge(DieRecon_16,Edge7xpc,gridstep);
DieRecon_18 = pcmerge(DieRecon_17,Edge7ypc,gridstep);
DieRecon_19 = pcmerge(DieRecon_18,Edge8xpc,gridstep);
DieRecon_20 = pcmerge(DieRecon_19,Edge8ypc,gridstep);
DieRecon_21 = pcmerge(DieRecon_20,SS3_ptCloudOut,gridstep);
DieRecon_22 = pcmerge(DieRecon_21,Edge9ypc,gridstep);
DieRecon_23 = pcmerge(DieRecon_22,Edge9zpc,gridstep);
DieRecon_24 = pcmerge(DieRecon_23,Edge10xpc,gridstep);
DieRecon_25 = pcmerge(DieRecon_24,Edge10zpc,gridstep);
DieRecon_26 = pcmerge(DieRecon_25,Edge11ypc,gridstep);
DieRecon_27 = pcmerge(DieRecon_26,Edge11zpc,gridstep);
DieRecon_28 = pcmerge(DieRecon_27,Edge12xpc,gridstep);
DieRecon_29 = pcmerge(DieRecon_28,Edge12zpc,gridstep);

figure;
pcshow(DieRecon_29);
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

A.3.9 Interpolating Die Faces and Finding Surface Normals

%% Generating normals for die reconstruction

% Side One
SO_F = scatteredInterpolant(SO3_xyzPoints(:,1),SO3_xyzPoints(:,2),SO3_xyzPoints(:,3),'natural');

SOmax_x = max(SO3_xyzPoints(:,1)); SOmin_x = min(SO3_xyzPoints(:,1));
SOmax_y = max(SO3_xyzPoints(:,2)); SOmin_y = min(SO3_xyzPoints(:,2));
SOmax_z = max(SO3_xyzPoints(:,3)); SOmin_z = min(SO3_xyzPoints(:,3));

step = 346;
stepsize = (abs(SOmin_x) - abs(SOmax_x))/(step - 1);
SO_xi = SOmin_x:abs(stepsize):SOmax_x;
SO_yi = SOmin_y:abs(stepsize):SOmax_y;
SO_zi = SOmin_z:abs(stepsize):SOmax_z;

[SO_qx,SO_qy] = meshgrid(SO_xi,SO_yi);
SO_qz = SO_F(SO_qx,SO_qy);
SO_C = SO_F(SO_qx,SO_qy);

figure;
surfl(SO_qz)
% surf(qx,qy,qz,C);
shading interp;
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

[SO_nx,SO_ny,SO_nz] = surfnorm(SO_qz);
saveobjmesh('SideOne_2.obj',SO_qx,SO_qy,SO_qz,SO nx,SO ny,SO nz);

%%
% Side Two
ST_F = scatteredInterpolant(ST3_xyzPoints(:,1),ST3_xyzPoints(:,2),ST3_xyzPoints(:,3),'natural');

STmax_x = max(ST3_xyzPoints(:,1)); STmin_x = min(ST3_xyzPoints(:,1));
STmax_y = max(ST3_xyzPoints(:,2)); STmin_y = min(ST3_xyzPoints(:,2));
STmax_z = max(ST3_xyzPoints(:,3)); STmin_z = min(ST3_xyzPoints(:,3));

step = 346;
stepsize = (abs(STmin_x) - abs(STmax_x)) / (step - 1);
ST_xi = STmin_x:abs(stepsize):STmax_x;
ST_yi = STmin_y:abs(stepsize):STmax_y;
ST_zi = STmin_z:abs(stepsize):STmax_z;

[ST_qx,ST_qy] = meshgrid(ST_xi,ST_yi);
ST_qz = ST_F(ST_qx,ST_qy);
ST_C = ST_F(ST_qx,ST_qy);

figure;
surfl(ST_qz);
shading interp;
xlabel('x-distance, mm');
ylabel('y-distance, mm');
zlabel('z-distance, mm');

[ST_nx,ST_ny,ST_nz] = surfnorm(ST_qz);
saveobjmesh('SideTwo.obj',ST_qx,ST_qy,ST_qz,ST_nx,ST_ny,ST_nz);

%%
% Side Three
STh_F = scatteredInterpolant(STh3_xyzPoints(:,1),STh3_xyzPoints(:,2),STh3_xyzPoints(:,3),'natural');

SThmax_x = max(STh3_xyzPoints(:,1)); SThmin_x = min(STh3_xyzPoints(:,1));
SThmax_y = max(STh3_xyzPoints(:,2)); SThmin_y = min(STh3_xyzPoints(:,2));
SThmax_z = max(STh3_xyzPoints(:,3)); SThmin_z = min(STh3_xyzPoints(:,3));

step = 346;
stepsize = (abs(SThmin_x) - abs(SThmax_x)) / (step-1);
STh_xi = SThmin_x:abs(stepsize):SThmax_x;
STh_yi = SThmin_y:abs(stepsize):SThmax_y;
STh_zi = SThmin_z:abs(stepsize):SThmax_z;

[STh_qx,STh_qy] = meshgrid(STh_xi,STh_yi);
STh_qz = STh_F(STh_qx,STh_qy);
STh_C = STh_F(STh_qx,STh_qy);

figure;
surfl(STh_qz);
shading interp;
xlabel('x-distance, mm');
ylabel('y-distance, mm');
zlabel('z-distance, mm');

[STh_nx,STh_ny,STh_nz] = surfnorm(STh_qz);
saveobjmesh('SideThree.obj',STh_qx,STh_qy,STh_qz,STh_nx,STh_ny,STh_nz);

%%
% Side Four
SFo_F = scatteredInterpolant(SFo3_xyzPoints(:,1),SFo3_xyzPoints(:,2),SFo3_xyzPoints(:,3),'natural');

SFomax_x = max(SFo3_xyzPoints(:,1)); SFomin_x = min(SFo3_xyzPoints(:,1));
SFomax_y = max(SFo3_xyzPoints(:,2)); SFomin_y = min(SFo3_xyzPoints(:,2));
SFomax_z = max(SFo3_xyzPoints(:,3)); SFomin_z = min(SFo3_xyzPoints(:,3));

step = 346;
stepsize = (abs(SFomin_x) - abs(SFomax_x))/(step - 1);
SFo_xi = SFomin_x:abs(stepsize):SFomax_x;
SFo_yi = SFomin_y:abs(stepsize):SFomax_y;
SFo_zi = SFomin_z:abs(stepsize):SFomax_z;

[SFo_qx,SFo_qy] = meshgrid(SFo_xi,SFo_yi);
SFo_qz = SFo_F(SFo_qx, SFo_qy);
SFo_C = SFo_F(SFo_qx, SFo_qy);

figure;
surfl(SFo_qz);
shading interp;
xlabel('x-distance, mm');
ylabel('y-distance, mm');
zlabel('z-distance, mm');

[SFo_nx,SFo_ny,SFo_nz] = surfnorm(SFo_qz);
saveobjmesh('SideFour.obj',SFo_qx, SFo_qy, SFo_qz, SFo_nx, SFo_ny, SFo_nz);
%%
% Side Five
SFi_F = scatteredInterpolant(SFi3_xyzPoints(:,1),SFi3_xyzPoints(:,2),SFi3_xyzPoints(:,3),'natural');

SFimax_x = max(SFi3_xyzPoints(:,1)); SFimin_x = min(SFi3_xyzPoints(:,1));
SFimax_y = max(SFi3_xyzPoints(:,2)); SFimin_y = min(SFi3_xyzPoints(:,2));
SFimax_z = max(SFi3_xyzPoints(:,3)); SFimin_z = min(SFi3_xyzPoints(:,3));

step = 346;
stepsize = (abs(SFimin_x) - abs(SFimax_x))/(step - 1);
SFi_xi = SFimin_x:abs(stepsize):SFimax_x;
SFi_yi = SFimin_y:abs(stepsize):SFimax_y;
SFi_zi = SFimin_z:abs(stepsize):SFimax_z;

[SFi_qx,SFi_qy] = meshgrid(SFi_xi,SFi_yi);
SFi_qz = SFi_F(SFi_qx,SFi_qy);
SFi_C = SFi_F(SFi_qx,SFi_qy);

figure;
surfl(SFi_qz);
shading interp;
xlabel('x-distance, mm');
ylabel('y-distance, mm');
zlabel('z-distance, mm');

[SFi_nx,SFi_ny,SFi_nz] = surfnorm(SFi_qz);
saveobjmesh('SideFive.obj',SFi_qx,SFi_qy,SFi_qz,SFi_nx,SFi_ny,SFi_nz);

%%
% Side Six
SS_F = scatteredInterpolant(SS3_xyzPoints(:,1),SS3_xyzPoints(:,2),SS3_xyzPoints(:,3),'natural');

SSmax_x = max(SS3_xyzPoints(:,1)); SSmin_x = min(SS3_xyzPoints(:,1));
SSmax_y = max(SS3_xyzPoints(:,2)); SSmin_y = min(SS3_xyzPoints(:,2));
SSmax_z = max(SS3_xyzPoints(:,3)); SSmin_z = min(SS3_xyzPoints(:,3));

step = 346;
stepsize = (abs(SSmin_x) - abs(SSmax_x))/(step - 1);
SS_xi = SSmin_x:abs(stepsize):SSmax_x;
SS_yi = SSmin_y:abs(stepsize):SSmax_y;
SS_zi = SSmin_z:abs(stepsize):SSmax_z;
[SS_qx,SS_qy] = meshgrid(SS_xi,SS_yi);
SS_qz = SS_F(SS_qx,SS_qy);
SS_C = SS_F(SS_qx,SS_qy);

figure;
surfl(SS_qz);
shading interp;
xlabel('x-distance, mm');
ylabel('y-distance, mm');
zlabel('z-distance, mm');

[SS_nx,SS_ny,SS_nz] = surfnorm(SS_qz);
saveobjmesh('SideSix.obj',SS_qx,SS_qy,SS_qz,SS_nx,SS_ny,SS_nz);

A.4 Dime Reconstruction

A.4.1 Dime Hologram Reconstruction

%%% Dime Multiwavelength Reconstruction
%%% clear all
close all
%%% % Experimental Settings
clear 'length'
lambda1 = 505.5e-9; % in meters
lambda2 = 496.5e-9; % in meters
d = 50e-2; % in meters
dx = 6.8e-6;
dy = 6.8e-6;
scale = 1.0;
padsize = 0.0;
threshold = 0.0004;

%%% Reconstructing Dime Face

DimeFace_1 = imread('Face_Holo1.bmp','bmp') - imread('Face_Ref1.bmp','bmp');
DimeFace_2 = imread('Face_Holo2.bmp','bmp') - imread('Face_Ref2.bmp','bmp');

DF_Recon_1 = abs(myFresnel(DimeFace_1(:,:,1),d,lambda1,dx,false,1,scale,padsize));
DF_Recon_2 = abs(myFresnel(DimeFace_2(:,:,1),d,lambda2,dx,false,1,scale,padsize));

%%% Resizing the x-axes to compensate for the tilting of the object
scaling = 1/cosd(45.0);
scaling_matrix = [scaling 0 0
0 1 0
0 0 1];
scaling_tform = affine2d(scaling_matrix);

DF_Recon_1 = imwarp(DF_Recon_1,scaling_tform);
DF_Recon_2 = imwarp(DF_Recon_2,scaling_tform);
% (215:715, 800:1300)

figure;
imagesc(DF_Recon_1(215:715, 800:1300))
title('Dime Face @ 505.5nm')
colormap gray
grid on
% Set tick marks on plots
length = length(DF_Recon_1);
Inc = (lambda1*d/((1024+2*padsize)*(dx)))/scale;
Inc = Inc.*1000; % Convert to milimeters

taxmarksx = 1:floor(length/24):length;
tickmarksy = 1:floor(length/24):length;
tickmarksz = 1:floor(length/24):

scalex = 0:floor(length/24)*Inc:(Inc*length);
scaley = 0:floor(length/24)*Inc:(Inc*length);
scalez = 0:floor(length/24)*Inc:(Inc*length);

set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')
clear 'length'
figure;
imagesc(DF_Recon_2(215:715, 800:1300))
title('Dime Face @ 496.5nm')
colormap gray
grid on
% Set tick marks on plots
length = length(DF_Recon_2);
Inc = (lambda1*d/((1024+2*padsize)*(dx)))/scale;

Inc = Inc.*1000; % Convert to milimeters

tickmarksx = 1:floor(length/24):length;
tickmarksy = 1:floor(length/24):length;
tickmarksz = 1:floor(length/24):length;

scalex = 0:floor(length/24)*Inc:(Inc*length);
scaley = 0:floor(length/24)*Inc:(Inc*length);
scalez = 0:floor(length/24)*Inc:(Inc*length);

set(gca,'XTick',tickmarksx)
set(gca,'XTickLabel',scalex)
set(gca,'YTick',tickmarksy)
set(gca,'YTickLabel',scaley)
set(gca,'ZTick',tickmarksz)
set(gca,'ZTickLabel',scalez)
xlabel('x-distance, mm')
ylabel('y-distance, mm')
zlabel('z-distance, mm')

% Reconstructing Dime Tail
DimeTail_1 = imread('Tail_Holo1.bmp','bmp') - imread('Tail_Ref1.bmp','bmp');
DimeTail_2 = imread('Tail_Holo2.bmp','bmp') - imread('Tail_Ref2.bmp','bmp');

DT_Recon_1 = abs(myFresnel(DimeTail_1(:,:,1),d,lambda1,dx,false,1,scale,padsize));
DT_Recon_2 = abs(myFresnel(DimeTail_2(:,:,1),d,lambda2,dx,false,1,scale,padsize));

% Resizing the x-axes to compensate for the tilting of the object
scaling = 1/cosd(45.0);
scaling_matrix = 
[scaling 0 0
 0 1 0
0 0 1];
scaling_tform = affine2d(scaling_matrix);

DT_Recon_1 = imwarp(DT_Recon_1,scaling_tform);
DT_Recon_2 = imwarp(DT_Recon_2,scaling_tform);

% (245:745, 875:1375)
clear 'length'
figure;
imagesc(DT_Recon_1(245:745, 870:1370))
title('Dime Face @ 505.5nm')
colormap gray
grid on
% Set tick marks on plots
% length = length(DT_Recon_1);
% Inc = (lambda1*d/((1024+2*padsize)*(dx)))/scale;
% Inc = Inc.*1000; % Convert to milimeters
% tickmarksx = 1:floor(length/24):length;
% tickmarksy = 1:floor(length/24):length;
% tickmarksz = 1:floor(length/24):length;
% scalex = 0:floor(length/24)*Inc:(Inc*length);
% scaley = 0:floor(length/24)*Inc:(Inc*length);
% scalez = 0:floor(length/24)*Inc:(Inc*length);
% set(gca,'XTick',tickmarksx)
% set(gca,'XTickLabel',scalex)
% set(gca,'YTick',tickmarksy)
% set(gca,'YTickLabel',scaley)
% set(gca,'ZTick',tickmarksz)
% set(gca,'ZTickLabel',scalez)
xlabel('X')
ylabel('Y')
zlabel('Z')
clear 'length'

figure;
imagesc(DT_Recon_2(245:745, 875:1375))
title('Dime Face @ 496.5nm')
colormap gray
grid on
% Set tick marks on plots
% length = length(DT_Recon_2);
% Inc = (lambda1*d/((1024+2*padsize)*(dx)))/scale;
% Inc = Inc.*1000; % Convert to milimeters
% tickmarksx = 1:floor(length/24):length;
% tickmarksy = 1:floor(length/24):length;
% tickmarksz = 1:floor(length/24):length;
% scalex = 0:floor(length/24)*Inc:(Inc*length);
% scaley = 0:floor(length/24)*Inc:(Inc*length);
% scalez = 0:floor(length/24)*Inc:(Inc*length);
% set(gca,'XTick',tickmarksx)
% set(gca,'XTickLabel',scalex)
% set(gca,'YTick',tickmarksy)
% set(gca,'YTickLabel',scaley)
% set(gca,'ZTick',tickmarksz)
% set(gca,'ZTickLabel',scalez)
xlabel('X')
ylabel('Y')
zlabel('Z')

A.4.2 Phase Unwrapping

%%% Phase-Unwrapping of Dime Face

clear 'length'

% PUMA processing Dime Face
p = 0.25;
figure;
cliques = [1 0; 1 1; -1 1];
potential.quantized = 'yes';
potential.threshold = 0.5;
L = lambda1*lambda2/(lambda1-lambda2);

DimeFace_Phase = load('DF_PhaseDiff.mat');

% DimeFace_Phase = imwarp(DimeFace_Phase.dp,scaling_tform);

[unwph_DF1,iter_DF1,erglist_DF1] = puma_ho(DimeFace_Phase.dp(215:715,500:1000),p,'potential',potential,'cliques',cliques);

unwph_DF_Recon = medfilt2(unwph_DF1, [3 3]);

unwph_DF_Recon = unwph_DF_Recon * L/(4*pi);

figure;
surfl(unwph_DF_Recon);shading interp; colormap(gray);
set(gca,'FontSize',14)
xlabel('X')
ylabel('Y')
zlabel('Z')

%%% Phase-Unwrapping of Dime Tail

% PUMA processing Dime Tail
p = 0.25;
figure;
cliques = [1 0; 1 1; -1 1];
potential.quantized = 'yes';
potential.threshold = 0.5;
L = lambda1*lambda2/(lambda1-lambda2);

DimeTail_Phase = load('DT PhaseDiff.mat');
% DimeTail_Phase = imwarp(DimeTail_Phase.dp,scaling_tform);

[unwph_DT1,iter_DT1,erglist_DT1] = puma_ho(-DimeTail_Phase.dp(240:740,500:1000),
  p,'potential',potential,'cliques',cliques);

unwph_DT_Recon = medfilt2(unwph_DT1, [3 3]);
unwph_DT_Recon = unwph_DT_Recon * L/(4*pi);

figure;
surf(unwph_DT_Recon);shading interp; colormap(gray);
set(gca,'FontSize',14)
xlabel('X')
ylabel('Y')
zlabel('Z')

A.4.3 Point Cloud Reconstruction

%% Point Cloud Reconstruction of Dime Face/Tail

for m = 1:1:501
    x(1,m) = m.*dx;
    y(1,m) = m.*dy;
end

[X,Y] = meshgrid(x,y);
% Dime Face xyzPoints
DF_xyzPoints(:,1) = X(:);
DF_xyzPoints(:,2) = Y(:);
DF_xyzPoints(:,3) = unwph_DF_Recon(:);
% Dime Tail xyzPoints
DT_xyzPoints(:,1) = X(:);
DT_xyzPoints(:,2) = Y(:);
DT_xyzPoints(:,3) = unwph_DT_Recon(:);

figure;
pcshow(DF_xyzPoints)
colormap gray
set(gca,'FontSize',14)
xlabel('X')
ylabel('Y')
zlabel('Z')
figure;
pcshow(DT_xyzPoints)
colormap gray
set(gca,'FontSize',14)
xlabel('X')
ylabel('Y')
zlabel('Z')

A.4.4 Smoothing Dime

% Curve fit Dime face and tail to flatten
DF_CurveFit = fit([DF_xyzPoints(:,1),DF_xyzPoints(:,2)],DF_xyzPoints(:,3),'lowess');
DT_CurveFit = fit([DT_xyzPoints(:,1),DT_xyzPoints(:,2)],DT_xyzPoints(:,3),'lowess');

DF_z = feval(DF_CurveFit,[DF_xyzPoints(:,1),DF_xyzPoints(:,2)]);
DT_z = feval(DT_CurveFit,[DT_xyzPoints(:,1),DT_xyzPoints(:,2)]);

DF_xyzPoints(:,3) = DF_xyzPoints(:,3) - DF_z;
DT_xyzPoints(:,3) = DT_xyzPoints(:,3) - DT_z;

figure;
pcshow(DF_xyzPoints)
set(gca,'FontSize',14)
colormap gray
xlabel('X')
ylabel('Y')
zlabel('Z')

figure;
pcshow(DT_xyzPoints)
set(gca,'FontSize',14)
colormap gray
xlabel('X')
ylabel('Y')
zlabel('Z')

A.4.5 Interpolate Surfaces and Find Surface Normals

%% Interpolate point cloud reconstruction to find surface normals
% Dime Face

DF_F = scatteredInterpolant(DF_xyzPoints(:,1),DF_xyzPoints(:,2),DF_xyzPoints(:,3),'natural');
DFmax_x = max(DF_xyzPoints(:,1)); DFmin_x = min(DF_xyzPoints(:,1));
DFmax_y = max(DF_xyzPoints(:,2)); DFmin_y = min(DF_xyzPoints(:,2));
DFmax_z = max(DF_xyzPoints(:,3)); DFmin_z = min(DF_xyzPoints(:,3));

step = 501;
stepsize = (abs(DFmin_x) - abs(DFmax_x))/(step - 1);
DF_xi = DFmin_x:abs(stepsize):DFmax_x;
DF_yi = DFmin_y:abs(stepsize):DFmax_y;
DF_zi = DFmin_z:abs(stepsize):DFmax_z;

[DF_qx,DF_qy] = meshgrid(DF_xi,DF_yi);
DF_qz = DF_F(DF_qx,DF_qy);
DF_C = DF_F(DF_qx,DF_qy);

figure;
surfl(DF_qz);
shading interp;
xlabel('x');
ylabel('y');
zlabel('z');

[DF_nx,DF_ny,DF_nz] = surfnorm(DF_qz);
saveobjmesh('DimeFace3.obj',DF_qx,DF_qy,DF_qz,DF_nx,DF_ny,DF_nz);

%% Interpolate point cloud reconstruction to find surface normals
%% Dime Tail

DT_F = scatteredInterpolant(DT_xyzPoints(:,1),DT_xyzPoints(:,2),DT_xyzPoints(:,3),'natural');

DTmax_x = max(DT_xyzPoints(:,1)); DTmin_x = min(DT_xyzPoints(:,1));
DTmax_y = max(DT_xyzPoints(:,2)); DTmin_y = min(DT_xyzPoints(:,2));
DTmax_z = max(DT_xyzPoints(:,3)); DTmin_z = min(DT_xyzPoints(:,3));

step = 501;
stepsize = (abs(DTmin_x) - abs(DTmax_x))/(step - 1);
DT_xi = DTmin_x:abs(stepsize):DTmax_x;
DT_yi = DTmin_y:abs(stepsize):DTmax_y;
DT_zi = DTmin_z:abs(stepsize):DTmax_z;

[DT_qx,DT_qy] = meshgrid(DT_xi,DT_yi);
DT_qz = DT_F(DT_qx,DT_qy);
DT_C = DT_F(DT_qx,DT_qy);

figure;
surfl(DT_qz);
shading interp;
xlabel('x');
ylabel('y');
zlabel('z');

[DT_nx,DT_ny,DT_nz] = surfnorm(DT_qz);
saveobjmesh('DimeTails3.obj',DT_qx,DT_qy,DT_qz,DT_nx,DT_ny,DT_nz);
APPENDIX B

HOLOLENS CODES

B.1 MANAGER SCRIPTS

B.1.1 GazeManager.cs

// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for license information.

using UnityEngine;

namespace HoloToolkit.Unity
{
    /// <summary>
    /// GazeManager determines the location of the user's gaze, hit position and normals.
    /// </summary>
    public partial class GazeManager : Singleton<GazeManager>
    {
        /// <summary>
        /// Maximum gaze distance, in meters, for calculating a hit from a GameObject's Collider.
        /// </summary>
        [Tooltip("Maximum gaze distance, in meters, for calculating a hit.")]
        public float MaxGazeDistance = 15.0f;

        /// <summary>
        /// Select the layers raycast should target.
        /// </summary>
        [Tooltip("Select the layers raycast should target.")]
        public LayerMask RaycastLayerMask = Physics.DefaultRaycastLayers;

        /// <summary>
        /// Checking enables SetFocusPointForFrame to set the stabilization plane
        /// </summary>
public bool SetStabilizationPlane = true;

public float LerpStabilizationPlanePowerCloser = 4.0f;

public float LerpStabilizationPlanePowerFarther = 7.0f;

public bool UseBuiltInGazeStabilization = true;

public bool Hit { get; private set; }

public RaycastHit HitInfo { get; private set; }

public Vector3 Position { get; private set; }

public Vector3 Normal { get; private set; }

/// Object currently being focused on.
/// <summary>
public GameObject FocusedObject { get; private set; }

/// <summary>
/// Helper class that stabilizes gaze using gravity wells
/// <summary>
public GazeStabilizer GazeStabilization { get; private set; }

private Vector3 gazeOrigin;
private Vector3 gazeDirection;
private Quaternion gazeRotation;
private float lastHitDistance = 15.0f;

private void Awake()
{
    if (UseBuiltInGazeStabilization)
    {
        GazeStabilization = gameObject.GetComponent<GazeStabilizer>() ??
        gameObject.AddComponent<GazeStabilizer>();
    }
}

private void Update()
{
    gazeOrigin = Camera.main.transform.position;
    gazeDirection = Camera.main.transform.forward;
    gazeRotation = Camera.main.transform.rotation;

    if (GazeStabilization != null)
    {
        GazeStabilization.UpdateHeadStability(gazeOrigin, gazeRotation);
    }

    UpdateRaycast();
    UpdateStabilizationPlane();
}

/// <summary>
/// Calculates the Raycast hit position and normal.
/// <summary>
private void UpdateRaycast()
{
    // Get the raycast hit information from Unity's physics system.
    RaycastHit hitInfo;

    if (GazeStabilization != null)
Hit = Physics.Raycast(GazeStabilization.StableHeadRay, out hitInfo, MaxGazeDistance, RaycastLayerMask);
}
else
{
    Hit = Physics.Raycast(gazeOrigin, gazeDirection, out hitInfo, MaxGazeDistance, RaycastLayerMask);
}

GameObject oldFocusedObject = FocusedObject;

// Update the HitInfo property so other classes can use this hit information.
HitInfo = hitInfo;

if (Hit)
{
    // If the raycast hits a hologram, set the position and normal to match the
    // intersection point.
    Position = hitInfo.point;
    Normal = hitInfo.normal;
    lastHitDistance = hitInfo.distance;
    FocusedObject = hitInfo.collider.gameObject;
}
else
{
    // If the raycast does not hit a hologram, default the position to last hit distance
    // in front of the user,
    // and the normal to face the user.
    Position = gazeOrigin + (gazeDirection * lastHitDistance);
    Normal = -gazeDirection;
    FocusedObject = null;
}

// Check if the currently hit object has changed
if (oldFocusedObject != FocusedObject)
{
    if (oldFocusedObject != null)
    {
        oldFocusedObject.SendMessage("OnGazeLeave", SendMessageOptions.DontRequireReceiver);
    }
    if (FocusedObject != null)
    {
        FocusedObject.SendMessage("OnGazeEnter", SendMessageOptions.DontRequireReceiver);
    }
}
/// <summary>
/// Adds the stabilization plane modifier if it's enabled and if it doesn't exist yet.
/// </summary>
private void UpdateStabilizationPlane()
{
    // We want to use the stabilization logic.
    if (SetStabilizationPlane)
    {
        // Check if it exists in the scene.
        if (StabilizationPlaneModifier.Instance == null)
        {
            // If not, add it to us.
            gameObject.AddComponent<StabilizationPlaneModifier>();
        }
    }

    if (StabilizationPlaneModifier.Instance != null)
    {
    }
}

B.1.2 GestureManager.cs

// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for license information.

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.VR.WSA.Input;

namespace HoloToolkit.Unity
{
    /// <summary>
    /// GestureManager provides access to several different input gestures, including Tap and Manipulation.
    /// </summary>

    /// <summary>
    /// GestureManager provides access to several different input gestures, including Tap and Manipulation.
    /// </summary>
}
/// <summary>
/// When a tap gesture is detected, GestureManager uses GazeManager to find the
currently focused object.
/// GestureManager then sends a message to that game object.
///
/// /// Using Manipulation requires subscribing the the ManipulationStarted events and then
querying
/// information about the manipulation gesture via ManipulationOffset and
ManipulationHandPosition.
///
/// /// Editor and Companion App Input can also be used by assigning a keyboard select key
and
/// using the right mouse button to select the currently focused object.
///
/// /// Using Gestures with mouse is currently not supported.
/// </summary>
[RequireComponent(typeof(GazeManager))]
public partial class GestureManager : Singleton<GestureManager>
{
    #region Delegate Events

    /// <summary>
    /// Occurs when a manipulation gesture has started.
    /// </summary>
    /// <param name="sourceKind">The Interaction Source Kind that started the
event.</param>
    public delegate void ManipulationStartedDelegate(InteractionSourceKind sourceKind);

    public event ManipulationStartedDelegate OnManipulationStarted;

    /// <summary>
    /// Occurs when a manipulation gesture ended as a result of user input.
    /// </summary>
    /// <param name="sourceKind">The Interaction Source Kind that completed the
event.</param>
    public delegate void ManipulationCompletedDelegate(InteractionSourceKind sourceKind);

    public event ManipulationCompletedDelegate OnManipulationCompleted;

    /// <summary>

/// Occurs when a manipulated gesture ended as a result of some other condition.
/// (e.g. the hand being used for the gesture is no longer visible).
/// </summary>
/// <param name="sourceKind">The Interaction Source Kind that cancelled the event.</param>
public delegate void ManipulationCanceledDelegate(InteractionSourceKind sourceKind);
public event ManipulationCanceledDelegate OnManipulationCanceled;

#region Public Properties

/// <summary>
/// To select even when a hologram is not being gazed at,
/// set the override focused object.
/// If its null, then the gazed at object will be selected.
/// </summary>
public GameObject OverrideFocusedObject { get; set; }

/// <summary>
/// Gets the currently focused object, or null if none.
/// </summary>
public GameObject FocusedObject { get; private set; }

/// <summary>
/// Whether or not a manipulation gesture is currently in progress
/// </summary>
public bool ManipulationInProgress { get; private set; }

/// <summary>
/// The offset of the input source from its position at the beginning of
/// the currently active manipulation gesture, in world space.
/// Valid if a manipulation gesture is in progress.
/// </summary>
public Vector3 ManipulationOffset { get; private set; }

/// <summary>
/// The world space position of manipulation source being used for the current
/// manipulation gesture.
/// Valid if a manipulation gesture is in progress.
/// </summary>
public Vector3 ManipulationPosition
{
    get
    {
        Vector3 position;
        if (!currentInteractionSourceState.properties.location.TryGetPosition(out position))
        {
            position = Vector3.zero;
        }
        return position;
    }
}

/// <summary>
/// InteractionSourceDetected tracks the interaction detected state.
/// Returns true if the list of tracked interactions is not empty.
/// </summary>
public bool InteractionSourceDetected
{
    get { return trackedInteractionSource.Count > 0; } 
}

/// <summary>
/// InteractionSourcePressed track the interaction pressed state.
/// Returns true if the list of pressed interactions is not empty.
/// </summary>
public bool InteractionSourcePressed
{
    get { return pressedInteractionSource.Count > 0; } 
}

#endregion

#if UNITY_EDITOR || UNITY_STANDALONE
/// <summary>
/// Key to press that will select the currently focused object.
/// </summary>
#endif
public KeyCode keyboardSelectKey = KeyCode.Space;

/// <summary>
/// Enumerated Mouse Buttons.
/// </summary>
public enum MouseButton
{
    Left = 0,
    Right = 1,
    Middle = 2
}

/// <summary>
/// Mouse button to press that will select the current focused object.
/// </summary>
public MouseButton MouseSelectButton = MouseButton.Right;

private GestureRecognizer gestureRecognizer;

/// <summary> We use a separate manipulation recognizer here because the tap gesture recognizer cancels capturing gestures whenever the GazeManager focus changes, which is not the behavior we want for manipulation /// </summary>
private GestureRecognizer manipulationRecognizer;

private InteractionSourceState currentInteractionSourceState;

private HashSet<uint> trackedInteractionSource = new HashSet<uint>();

private HashSet<uint> pressedInteractionSource = new HashSet<uint>();

private bool hasRecognitionStarted;

private GameObject lastFocusedObject;

private void Awake()
{
}
InteractionManager.SourceDetected += InteractionManager_SourceDetected;
InteractionManager.SourcePressed += InteractionManager_SourcePressed;
InteractionManager.SourceReleased += InteractionManager_SourceReleased;
InteractionManager.SourceUpdated += InteractionManager_SourceUpdated;
InteractionManager.SourceLost += InteractionManager_SourceLost;

// Create a new GestureRecognizer. Sign up for tapped events.
// Will register Taps for both Hand and Clicker.
gestureRecognizer = new GestureRecognizer();
gestureRecognizer.SetRecognizableGestures(GestureSettings.Tap);
gestureRecognizer.TappedEvent += GestureRecognizer_TappedEvent;

// We need to send pressed and released events to UI so they can provide visual feedback
// of the current state of the UI based on user input.
gestureRecognizer.RecognitionStartedEvent += GestureRecognizer_RecognitionStartedEvent;
gestureRecognizer.RecognitionEndedEvent += GestureRecognizer_RecognitionEndedEvent;

// Start looking for gestures.
gestureRecognizer.StartCapturingGestures();

// Create a new GestureRecognizer.
// Sign up for manipulation events.
manipulationRecognizer = new GestureRecognizer();

// ManipulationTranslate only recognizes hand gesture translations, but not Clicker translations.
manipulationRecognizer.SetRecognizableGestures(GestureSettings.ManipulationTranslate);

// Subscribe to our manipulation events.
manipulationRecognizer.ManipulationStartedEvent += ManipulationRecognizer_ManipulationStartedEvent;
manipulationRecognizer.ManipulationUpdatedEvent += ManipulationRecognizer_ManipulationUpdatedEvent;
manipulationRecognizer.ManipulationCompletedEvent += ManipulationRecognizer_ManipulationCompletedEvent;
manipulationRecognizer.ManipulationCanceledEvent += ManipulationRecognizer_ManipulationCanceledEvent;

// Start looking for manipulation gestures.
manipulationRecognizer.StartCapturingGestures();
}

#region Interaction Management

/// <summary>
/// Raised when we detect an interaction source.
/// </summary>
/// <param name="state"></param>
private void InteractionManager_SourceDetected(InteractionSourceState state)
{
    trackedInteractionSource.Add(state.source.id);
}

/// <summary>
/// Raised when the interaction source is pressed.
/// </summary>
/// <param name="state">The current state of the Interaction source.</param>
private void InteractionManager_SourcePressed(InteractionSourceState state)
{
    // Make sure we're using a tracked interaction source.
    if (trackedInteractionSource.Contains(state.source.id))
    {
        // Add it to the list of pressed states.
        if (!pressedInteractionSource.Contains(state.source.id))
        {
            pressedInteractionSource.Add(state.source.id);
        }

        // Don't start another manipulation gesture if one is already underway.
        if (!ManipulationInProgress)
        {
            // Cache our current source state for use later.
            currentInteractionSourceState = state;

            // Gesture Support for Controllers: (i.e. Clicker, Xbox Controller, etc.)
if (state.source.kind == InteractionSourceKind.Controller)
{
    OnManipulation(inProgress: true, offset: ManipulationPosition);

    if (OnManipulationStarted != null)
    {
        OnManipulationStarted(state.source.kind);
    }
}

/// <summary>
/// Raised when the interaction source is updated.
/// </summary>
/// <param name="state">The current state of the Interaction source.</param>
private void InteractionManager_SourceUpdated(InteractionSourceState state)
{
    // if we currently in a manipulation, update our data.
    // Check the current interaction source matches our cached value.
    if (ManipulationInProgress && state.source.id == currentInteractionSourceState.source.id)
    {
        currentInteractionSourceState = state;

        // Gesture Support for Controllers: (i.e. Clicker, Xbox Controller, etc.)
        if (state.source.kind == InteractionSourceKind.Controller)
        {
            Vector3 cumulativeDelta = ManipulationOffset - ManipulationPosition;
            OnManipulation(inProgress: true, offset: cumulativeDelta);
        }
    }
}

/// <summary>
/// Raised when the interaction source is released.
/// </summary>
/// <param name="state">The current state of the Interaction source.</param>
private void InteractionManager_SourceReleased(InteractionSourceState state)
if we currently in a manipulation then stop.
// Check the current interaction source matches our cached value.
if (ManipulationInProgress && state.source.id == currentInteractionSourceState.source.id)
{
    // Gesture Support for Controllers: (i.e. Clicker, Xbox Controller, etc.)
    if (state.source.kind == InteractionSourceKind.Controller)
    {
        Vector3 cumulativeDelta = ManipulationOffset - ManipulationPosition;
        OnManipulation(inProgress: false, offset: cumulativeDelta);
        
        if (OnManipulationCompleted != null)
        {
            OnManipulationCompleted(state.source.kind);
        }
    }
}

// Removed our pressed state.
if (pressedInteractionSource.Contains(state.source.id))
{
    pressedInteractionSource.Remove(state.source.id);
}

/// <summary>
/// Raised when the interaction source is no longer available.
/// </summary>
/// <param name="state">The current state of the Interaction source.</param>
private void InteractionManager_SourceLost(InteractionSourceState state)
{
    // If we currently in a manipulation then stop.
    // Check the current interaction source matches our cached value.
    if (ManipulationInProgress && state.source.id == currentInteractionSourceState.source.id)
    {
        // Gesture Support for Controllers: (i.e. Clicker, Xbox Controller, etc.)
        if (state.source.kind == InteractionSourceKind.Controller)
        {
            
```
Vector3 cumulativeDelta = ManipulationOffset - ManipulationPosition;
OnManipulation(inProgress: false, offset: cumulativeDelta);

if (OnManipulationCanceled != null)
{
    OnManipulationCanceled(state.source.kind);
}

// Removed our pressed state.
if (pressedInteractionSource.Contains(state.source.id))
{
    pressedInteractionSource.Remove(state.source.id);
}

// Remove our tracked interaction state.
if (trackedInteractionSource.Contains(state.source.id))
{
    trackedInteractionSource.Remove(state.source.id);
}

#region Gesture Management

/// <summary>
/// Throws <see cref="OnTap"/>.
/// </summary>
/// <param name="source">Interaction Source.</param>
/// <param name="tapCount">TODO: Need clarification on what this is </param>
/// <param name="headRay">The Ray from the users forward direction.</param>
private void GestureRecognizer_TappedEvent(InteractionSourceKind source, int tapCount, Ray headRay)
{
    CalcFocusedObject();
    OnTap();
}
/// <summary>
/// Throws <see cref="OnRecognitionStarted"/>. Only used for UI states.
/// </summary>
/// <param name="source">Input Source.</param>
/// <param name="headRay">The Ray from the users forward direction.</param>
private void GestureRecognizer_RecognitionStartedEvent(InteractionSourceKind source, Ray headRay)
{
    CalcFocusedObject();
    OnRecognitionStarted();
}

/// <summary>
/// Throws <see cref="OnRecognitionEnded"/>. Only used for UI states.
/// </summary>
/// <param name="source"></param>
/// <param name="headRay"></param>
private void GestureRecogniser_RecognitionEndedEvent(InteractionSourceKind source, Ray headRay)
{
    OnRecognitionEndeded(CalcFocusedObject());
}

#endregion

#region Manipulation Management

/// <summary>
/// Raised when the gesture manager recognizes that a manipulation has begun.
/// </summary>
/// <param name="source">Input Source Kind.</param>
/// <param name="cumulativeDelta">Cumlulative Data.</param>
/// <param name="headRay">The Ray from the users forward direction.</param>
private void ManipulationRecognizer_ManipulationStartedEvent(InteractionSourceKind source, Vector3 cumulativeDelta, Ray headRay)
{
    if (!ManipulationInProgress)
    {
        // Don't start another manipulation gesture if one is already underway
        ...
    }
}

#endregion
OnManipulation(inProgress: true, offset: cumulativeDelta);
if (OnManipulationStarted != null)
{
    OnManipulationStarted(source);
}

/// <summary>
/// Raised when the gesture manager recognizes that a manipulation has been updated.
/// </summary>
/// <param name="source">Input Source Kind.</param>
/// <param name="cumulativeDelta">Cumlulative Data.</param>
/// <param name="headRay">The Ray from the users forward direction.</param>
private void ManipulationRecognizer_ManipulationUpdatedEvent(InteractionSourceKind source, Vector3 cumulativeDelta, Ray headRay)
{
    OnManipulation(inProgress: true, offset: cumulativeDelta);
}

/// <summary>
/// Raised when the gesture manager recognizes that a manipulation has completed.
/// </summary>
/// <param name="source">Input Source Kind.</param>
/// <param name="cumulativeDelta">Cumlulative Data.</param>
/// <param name="headRay">The Ray from the users forward direction.</param>
private void ManipulationRecognizer_ManipulationCompletedEvent(InteractionSourceKind source, Vector3 cumulativeDelta, Ray headRay)
{
    OnManipulation(inProgress: false, offset: cumulativeDelta);
    if (OnManipulationCompleted != null)
    {
        OnManipulationCompleted(source);
    }
}

/// <summary>
/// Raised when the gesture manager recognizes that a manipulation has been canceled.
/// /// <summary>Input Source Kind.</summary>
/// <param name="source">Input Source Kind.</param>
/// <param name="cumulativeDelta" Cumulative Delta.</param>
/// <param name="headRay" The Ray from the users forward direction.</param>
private void ManipulationRecognizer_ManipulationCanceledEvent(InteractionSourceKind source, Vector3 cumulativeDelta, Ray headRay)
{
    OnManipulation(inProgress: false, offset: cumulativeDelta);
    if (OnManipulationCanceled != null)
    {
        OnManipulationCanceled(source);
    }
}

/// /// Processes Manipulation Data.
/// /// <summary>Processes Manipulation Data.</summary>
/// <param name="inProgress">Is this manipulation in progress?</param>
/// <param name="offset">The Offset of our manipulation to calculate delta positions.</param>
private void OnManipulation(bool inProgress, Vector3 offset)
{
    ManipulationInProgress = inProgress;

    // If we're doing a manipulation set the offset, else reset it to zero.
    ManipulationOffset = inProgress ? offset : Vector3.zero;
}

#endregion

#region Event Management

/// /// Throws OnSelect.
/// /// <summary>Throws OnSelect.</summary>
private void OnTap()
{
    if (FocusedObject != null)
    {
        // If we're doing a manipulation set the offset, else reset it to zero.
        ManipulationOffset = inProgress ? offset : Vector3.zero;
    }
}

#endregion
FocusedObject.SendMessage("OnSelect",
SendMessageOptions.DontRequireReceiver);
}
}

/// <summary>
/// Throws OnPressed. Only used for determining UI states.
/// </summary>
private void OnRecognitionStarted()
{
    if (FocusedObject != null)
    {
        hasRecognitionStarted = true;
        FocusedObject.SendMessage("OnPressed",
SendMessageOptions.DontRequireReceiver);
    }
}

/// <summary>
/// Throws OnReleased. Only used for determining UI states.
/// </summary>
private void OnRecognitionEnded(bool changedFocus)
{
    GameObject focusedObject = FocusedObject;
    if (changedFocus)
    {
        focusedObject = lastFocusedObject;
    }

    if (focusedObject != null && hasRecognitionStarted)
    {
        focusedObject.SendMessage("OnReleased",
SendMessageOptions.DontRequireReceiver);
    }

    hasRecognitionStarted = false;
}

#endregion
/// <summary>
/// Calculates the current object in Focus.
/// </summary>
/// <returns>True if we've changed our focus to a new object, else false.</returns>
private bool CalcFocusedObject()
{
    // set the next focus object to see if focus has changed, but don't replace the current
    // focused object
    // until all the inputs are handled, like Unity Editor input for OnTap().
    GameObject newFocusedObject;

    if (GazeManager.Instance.Hit &&
        OverrideFocusedObject == null &&
        GazeManager.Instance.HitInfo.collider != null)
    {
        // If gaze hits a hologram, set the focused object to that game object.
        // Also if the caller has not decided to override the focused object.
        newFocusedObject = GazeManager.Instance.HitInfo.collider.gameObject;
    }
    else
    {
        // If our gaze doesn't hit a hologram, set the focused object to null or override
        // focused object.
        newFocusedObject = OverrideFocusedObject;
    }

    // Checks to see if our focus has changed.
    bool focusedChanged = FocusedObject != newFocusedObject;

    if (focusedChanged)
    {
        // If the currently focused object doesn't match the new focused object, cancel
        // the current gesture.
        // This is to prevent applying gestures from one hologram to another.
        gestureRecognizer.CancelGestures();

        // Set our last Focused object.
        lastFocusedObject = FocusedObject;
    }

    // Set our current Focused Object.
}
FocusedObject = newFocusedObject;

// Start looking for new gestures.
gestureRecognizer.StartCapturingGestures();
}

return focusedChanged;
}

private void LateUpdate()
{
    bool focusedChanged = CalcFocusedObject();

#if UNITY_EDITOR || UNITY_STANDALONE
    // Process Editor/Companion app input. Tap by pressing both right and left mouse buttons. Release Tap is on any mouse button up.

    // If we're stop pressing our mouse button, our keyboard select key, or if the focus has changed then throw recognition Ended.
    if (Input.GetMouseButtonUp((int)MouseSelectButton) || Input.GetKeyUp(keyboardSelectKey) || focusedChanged)
    {
        OnRecognitionEndeded(focusedChanged);
    }

    // If we're currently pressing our mouse button, or our keyboard select key.
    if (Input.GetMouseButtonDown((int)MouseSelectButton) || Input.GetKeyDown(keyboardSelectKey))
    {
        // If our focus has changed or we're not currently manipulating our object in focus since the last frame,
        // then throw a new Tap and start recognition.
        if (focusedChanged || !ManipulationInProgress)
        {
            OnRecognitionStarted();
            OnTap();
        }
    }
#endif
}
private void OnDestroy()
{
    gestureRecognizer.StopCapturingGestures();
    gestureRecognizer.TappedEvent -= GestureRecognizer_TappedEvent;
    gestureRecognizer.RecognitionStartedEvent -= GestureRecognizer_RecognitionStartedEvent;
    gestureRecognizer.RecognitionEndedEvent -= GestureRecognizer_RecognitionEndedEvent;

    manipulationRecognizer.StopCapturingGestures();
    manipulationRecognizer.ManipulationStartedEvent -= ManipulationRecognizer_ManipulationStartedEvent;
    manipulationRecognizer.ManipulationUpdatedEvent -= ManipulationRecognizer_ManipulationUpdatedEvent;
    manipulationRecognizer.ManipulationCompletedEvent -= ManipulationRecognizer_ManipulationCompletedEvent;
    manipulationRecognizer.ManipulationCanceledEvent -= ManipulationRecognizer_ManipulationCanceledEvent;

    InteractionManager.SourceDetected -= InteractionManager_SourceDetected;
    InteractionManager.SourcePressed -= InteractionManager_SourcePressed;
    InteractionManager.SourceReleased -= InteractionManager_SourceReleased;
    InteractionManager.SourceUpdated -= InteractionManager_SourceUpdated;
    InteractionManager.SourceLost -= InteractionManager_SourceLost;
}

B.1.3 WorldAnchorManager.cs

// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for license information.

using UnityEngine;
using UnityEngine.VR.WSA.Persistence;
using System.Collections.Generic;
using UnityEngine.VR.WSA;

namespace HoloToolkit.Unity
{
/// <summary>
/// Wrapper around world anchor store to streamline some of the
/// persistence api busy work.
/// </summary>
public class WorldAnchorManager : Singleton<WorldAnchorManager>
{
    /// <summary>
    /// To prevent initializing too many anchors at once
    /// and to allow for the WorldAnchorStore to load asynchronously
    /// without callers handling the case where the store isn't loaded yet
    /// we'll setup a queue of anchor attachment operations.
    /// The AnchorAttachmentInfo struct has the data needed to do this.
    /// </summary>
    private struct AnchorAttachmentInfo
    {
        public GameObject GameObjectToAnchor { get; set; }
        public string AnchorName { get; set; }
    }

    /// <summary>
    /// The queue mentioned above.
    /// </summary>
    private Queue<AnchorAttachmentInfo> anchorOperations = new Queue<AnchorAttachmentInfo>();

    /// <summary>
    /// The WorldAnchorStore for the current application.
    /// Can be null when the application starts.
    /// </summary>
    public WorldAnchorStore AnchorStore { get; private set; }

    /// <summary>
    /// Callback function that contains the WorldAnchorStore object.
    /// </summary>
    /// <param name="Store">The WorldAnchorStore to cache.</param>
    private void AnchorStoreReady(WorldAnchorStore Store)
    {
        AnchorStore = Store;
    }

    /// <summary>
    /// When the app starts grab the anchor store immediately.
    /// </summary>
    private void Awake()
    {
        AnchorStore = null;
    }
}
WorldAnchorStore.GetAsync(AnchorStoreReady);

/// <summary>
/// Each frame see if there is work to do and if we can do a unit, do it.
/// </summary>
private void Update()
{
    if (AnchorStore != null && anchorOperations.Count > 0)
    {
        DoAnchorOperation(anchorOperations.Dequeue());
    }
}

/// <summary>
/// Attaches an anchor to the game object. If the anchor store has
/// an anchor with the specified name it will load the anchor, otherwise
/// a new anchor will be saved under the specified name.
/// </summary>
/// <param name="gameObjectToAnchor">The GameObject to attach the anchor to.</param>
/// <param name="anchorName">Name of the anchor.</param>
public void AttachAnchor(GameObject gameObjectToAnchor, string anchorName)
{
    if (gameObjectToAnchor == null)
    {
        Debug.LogError("Must pass in a valid GameObject");
        return;
    }

    if (string.IsNullOrEmpty(anchorName))
    {
        Debug.LogError("Must supply an AnchorName.");
        return;
    }

    anchorOperations.Enqueue(
        new AnchorAttachmentInfo()
        {
            GameObjectToAnchor = gameObjectToAnchor,
            AnchorName = anchorName
        });
}

/// <summary>
/// Removes the anchor from the game object and deletes the anchor
/// from the anchor store.
/// </summary>
/// <param name="gameObjectToUnanchor">gameObject to remove the anchor from.</param>
public void RemoveAnchor(GameObject gameObjectToUnanchor)
{
    // This case is unexpected, but just in case.
    if (AnchorStore == null)
    {
        Debug.LogError("remove anchor called before anchor store is ready.");
    }

    WorldAnchor anchor = gameObjectToUnanchor.GetComponent<WorldAnchor>();

    if (anchor != null)
    {
        AnchorStore.Delete(anchor.name);
        DestroyImmediate(anchor);
    }
}

/// <summary>
/// Function that actually adds the anchor to the game object.
/// </summary>
/// <param name="anchorAttachmentInfo">Parameters for attaching the anchor.</param>
private void DoAnchorOperation(AnchorAttachmentInfo anchorAttachmentInfo)
{
    string AnchorName = anchorAttachmentInfo.AnchorName;
    GameObject gameObjectToAnchor = anchorAttachmentInfo.GameObjectToAnchor;

    if (gameObjectToAnchor == null)
    {
        Debug.Log("GameObject must have been destroyed before we got a chance to anchor it.");
        return;
    }

    // Try to load a previously saved world anchor.
    WorldAnchor savedAnchor = AnchorStore.Load(AnchorName, gameObjectToAnchor);
    if (savedAnchor == null)
    {
    }
Either world anchor was not saved / does not exist or has a different name.
Debug.Log(gameObjectToAnchor.name + " : World anchor could not be loaded for this game object. Creating a new anchor.");

Create anchor since one does not exist.
CreateAnchor(gameObjectToAnchor, AnchorName);
else

savedAnchor.name = AnchorName;
Debug.Log(gameObjectToAnchor.name + " : World anchor loaded from anchor store and updated for this game object.");

/// <summary>
/// Creates an anchor, attaches it to the gameObjectToAnchor, and saves the anchor to the anchor store.
/// </summary>
/// <param name="gameObjectToAnchor">The GameObject to attach the anchor to.</param>
/// <param name="anchorName">The name to give to the anchor.</param>
private void CreateAnchor(GameObject gameObjectToAnchor, string anchorName)
{
    WorldAnchor anchor = gameObjectToAnchor.AddComponent<WorldAnchor>();
    anchor.name = anchorName;

    // Sometimes the anchor is located immediately. In that case it can be saved immediately.
    if (anchor.isLocated)
    {
        SaveAnchor(anchor);
    }
    else
    {
        // Othertimes we must wait for the
        anchor.OnTrackingChanged += Anchor_OnTrackingChanged;
    }
}

/// <summary>
/// When an anchor isn't located immediately we subscribe to this event so we can save the anchor when it is finally located.
/// </summary>
/// <param name="self">The anchor that is reporting a tracking changed event.</param>
private void Anchor_OnTrackingChanged(WorldAnchor self, bool located)
{
    if (located)
    {
        Debug.Log(gameObject.name + " : World anchor located successfully.");
        SaveAnchor(self);
        // Once the anchor is located we can unsubscribe from this event.
        self.OnTrackingChanged -= Anchor_OnTrackingChanged;
    }
    else
    {
        Debug.LogError(gameObject.name + " : World anchor failed to locate.");
    }
}

/// <summary>
/// Saves the anchor to the anchor store.
/// </summary>
/// <param name="anchor"></param>
private void SaveAnchor(WorldAnchor anchor)
{
    // Save the anchor to persist holograms across sessions.
    if (AnchorStore.Save(anchor.name, anchor))
    {
        Debug.Log(gameObject.name + " : World anchor saved successfully.");
    }
    else
    {
        Debug.LogError(gameObject.name + " : World anchor save failed.");
    }
}

B.1.4 HandsManager.cs

// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for license information.

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.VR.WSA.Input;
namespace HoloToolkit.Unity
{
    /// <summary>
    /// HandsManager determines if the hand is currently detected or not.
    /// </summary>
    public partial class HandsManager : Singleton<HandsManager>
    {
        /// <summary>
        /// Occurs when users hand is detected or lost.
        /// </summary>
        /// <param name="handDetected">True if a hand is Detected, else false.</param>
        public delegate void HandInViewDelegate(bool handDetected);
        public event HandInViewDelegate HandInView;

        /// <summary>
        /// HandDetected tracks the hand detected state.
        /// Returns true if the list of tracked hands is not empty.
        /// </summary>
        public bool HandDetected
        {
            get { return trackedHands.Count > 0; }
        }

        /// <summary>
        /// HandPressed tracks the pressed state.
        /// Returns true if the list of pressed hands is not empty.
        /// </summary>
        public bool HandsPressed
        {
            get { return pressedHands.Count > 0; }
        }

        /// <summary>
        /// The world space position of the hand being used for the current manipulation gesture.
        /// Valid only if a manipulation gesture is in progress.
        /// </summary>
        public Vector3 ManipulationHandPosition
        {
            get
            {
                Vector3 handPosition;
                if (!currentHandState.properties.location.TryGetPosition(out handPosition))
                {
                    handPosition = Vector3.zero;
                }
            }
        }
    }
}
return handPosition;
}
}

private InteractionSourceState currentHandState;

private HashSet<uint> trackedHands = new HashSet<uint>();

private HashSet<uint> pressedHands = new HashSet<uint>();

private void OnEnable()
{
    InteractionManager.SourceDetected += InteractionManager_SourceDetected;
    InteractionManager.SourcePressed += InteractionManager_SourcePressed;
    InteractionManager.SourceReleased += InteractionManager_SourceReleased;
    InteractionManager.SourceUpdated += InteractionManager_SourceUpdated;
    InteractionManager.SourceLost += InteractionManager_SourceLost;
}

private void OnDisable()
{
    InteractionManager.SourcePressed -= InteractionManager_SourcePressed;
    InteractionManager.SourceReleased -= InteractionManager_SourceReleased;
    InteractionManager.SourceUpdated -= InteractionManager_SourceUpdated;
    InteractionManager.SourceLost -= InteractionManager_SourceLost;
}

/// <summary>
/// Thrown when we detect a hand.
/// </summary>
/// <param name="state"></param>
private void InteractionManager_SourceDetected(InteractionSourceState state)
{
    // Check to see that the source is a hand.
    if (state.source.kind == InteractionSourceKind.Hand)
    {
        trackedHands.Add(state.source.id);

        if (HandInView != null)
        {
            HandInView(HandDetected);
        }
    }
}
/// <summary>
/// Thrown when the Interaction Source is pressed.
/// </summary>
/// <param name="state">The current state of the Interaction source.</param>
private void InteractionManager_SourcePressed(InteractionSourceState state)
{
    if (state.source.kind == InteractionSourceKind.Hand)
    {
        if (trackedHands.Contains(state.source.id))
        {
            currentHandState = state;
            if (!pressedHands.Contains(state.source.id))
            {
                pressedHands.Add(state.source.id);
            }
        }
    }
}

/// <summary>
/// Thrown when the Interaction Source is updated.
/// </summary>
/// <param name="state">The current state of the Interaction source.</param>
private void InteractionManager_SourceUpdated(InteractionSourceState state)
{
    if (state.source.kind == InteractionSourceKind.Hand)
    {
        if (state.source.id == currentHandState.source.id)
        {
            currentHandState = state;
        }
    }
}

/// <summary>
/// Thrown when the Interaction Source is released.
/// </summary>
/// <param name="state">The current state of the Interaction source.</param>
private void InteractionManager_SourceReleased(InteractionSourceState state)
{
    if (state.source.kind == InteractionSourceKind.Hand)
    {
        if (pressedHands.Contains(state.source.id))
        {
            pressedHands.Remove(state.source.id);
        }
    }
}
/// <summary>
/// Thrown when the Interaction Source is no longer available.
/// </summary>
/// <param name="state">The current state of the Interaction source.</param>
private void InteractionManager_SourceLost(InteractionSourceState state)
{
    InteractionManager_SourceReleased(state);

    if (state.source.kind == InteractionSourceKind.Hand)
    {
        if (trackedHands.Contains(state.source.id))
        {
            trackedHands.Remove(state.source.id);

            if (HandInView != null)
            {
                HandInView(HandDetected);
            }
        }
    }
}

B.1.5 KeywordManager.cs

// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for license information.

using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.Events;
using UnityEngine.Windows.Speech;

namespace HoloToolkit.Unity
{
    /// <summary>
    /// KeywordManager allows you to specify keywords and methods in the Unity Inspector, instead of registering them explicitly in code.
    /// This also includes a setting to either automatically start the
/// keyword recognizer or allow your code to start it.
///
/// IMPORTANT: Please make sure to add the microphone capability in your app, in
Unity under
/// Edit -> Project Settings -> Player -> Settings for Windows Store -> Publishing
Settings -> Capabilities
/// or in your Visual Studio Package.appxmanifest capabilities.
///</summary>
public partial class KeywordManager : MonoBehaviour
{
    [System.Serializable]
    public struct KeywordAndResponse
    {
        [Tooltip("The keyword to recognize.")]
        public string Keyword;
        [Tooltip("The KeyCode to recognize.")]
        public KeyCode KeyCode;
        [Tooltip("The UnityEvent to be invoked when the keyword is recognized.")]
        public UnityEvent Response;
    }

    // This enumeration gives the manager two different ways to handle the recognizer. Both will
    // set up the recognizer and add all keywords. The first causes the recognizer to start
    // immediately. The second allows the recognizer to be manually started at a later time.
    public enum RecognizerStartBehavior { AutoStart, ManualStart };

    [Tooltip("An enumeration to set whether the recognizer should start on or off.")]
    public RecognizerStartBehavior RecognizerStart;

    [Tooltip("An array of string keywords and UnityEvents, to be set in the Inspector.")]
    public KeywordAndResponse[] KeywordsAndResponses;

    private KeywordRecognizer keywordRecognizer;
    private Dictionary<string, UnityEvent> responses;

    void Start()
    {
        if (KeywordsAndResponses.Length > 0)
        {
            // Convert the struct array into a dictionary, with the keywords and the keys and
            the methods as the values.
            // This helps easily link the keyword recognized to the UnityEvent to be invoked.
            responses = KeywordsAndResponses.ToDictionary(keywordAndResponse =>
                keywordAndResponse.Keyword,
            }
keywordAndResponse =>
keywordAndResponse.Response);

keywordRecognizer = new KeywordRecognizer(responses.Keys.ToArray());
keywordRecognizer.OnPhraseRecognized += KeywordRecognizer_OnPhraseRecognized;

if (RecognizerStart == RecognizerStartBehavior.AutoStart)
{
    keywordRecognizer.Start();
}
else
{
    Debug.LogError("Must have at least one keyword specified in the Inspector on " + gameObject.name + ",");
}

void Update()
{
    ProcessKeyBindings();
}

void OnDestroy()
{
    if (keywordRecognizer != null)
    {
        StopKeywordRecognizer();
        keywordRecognizer.OnPhraseRecognized -= KeywordRecognizer_OnPhraseRecognized;
        keywordRecognizer.Dispose();
    }
}

private void ProcessKeyBindings()
{
    foreach (var kvp in KeywordsAndResponses)
    {
        if (Input.GetKeyDown(kvp.KeyCode))
        {
            kvp.Response.Invoke();
            return;
        }
    }
}
private void KeywordRecognizer_OnPhraseRecognized(PhraseRecognizedEventArgs args)
{
    UnityEvent keywordResponse;

    // Check to make sure the recognized keyword exists in the methods dictionary, then invoke the corresponding method.
    if (responses.TryGetValue(args.text, out keywordResponse))
    {
        keywordResponse.Invoke();
    }
}

/// <summary>
/// Make sure the keyword recognizer is off, then start it.
/// Otherwise, leave it alone because it's already in the desired state.
/// </summary>
public void StartKeywordRecognizer()
{
    if (keywordRecognizer != null && !keywordRecognizer.IsRunning)
    {
        keywordRecognizer.Start();
    }
}

/// <summary>
/// Make sure the keyword recognizer is on, then stop it.
/// Otherwise, leave it alone because it's already in the desired state.
/// </summary>
public void StopKeywordRecognizer()
{
    if (keywordRecognizer != null && keywordRecognizer.IsRunning)
    {
        keywordRecognizer.Stop();
    }
}

B.2 SPATIAL MAPPING SCRIPTS

B.2.1 SpatialMappingObserver.cs

// Copyright (c) Microsoft Corporation. All rights reserved.
namespace HoloToolkit.Unity
{
    /// <summary>
    /// Spatial Mapping Observer states.
    /// </summary>
    public enum ObserverStates
    {
        /// <summary>
        /// The SurfaceObserver is currently running.
        /// </summary>
        Running = 0,
        /// <summary>
        /// The SurfaceObserver is currently idle.
        /// </summary>
        Stopped = 1
    }

    /// <summary>
    /// The SpatialMappingObserver class encapsulates the SurfaceObserver into an easy to use
    /// object that handles managing the observed surfaces and the rendering of surface geometry.
    /// </summary>
    public class SpatialMappingObserver : SpatialMappingSource
    {
        [Tooltip("The number of triangles to calculate per cubic meter.")]
        public float TrianglesPerCubicMeter = 500f;

        [Tooltip("The extents of the observation volume.")]
        public Vector3 Extents = Vector3.one * 10.0f;

        [Tooltip("How long to wait (in sec) between Spatial Mapping updates.")]
        public float TimeBetweenUpdates = 3.5f;

        /// <summary>
        /// Event for hooking when surfaces are changed.
        /// </summary>
        public event SurfaceObserver.SurfaceChangedDelegate SurfaceChanged;
    }
}
public event SurfaceObserver.SurfaceDataReadyDelegate DataReady;

private SurfaceObserver observer;

private Dictionary<int, GameObject> surfaces = new Dictionary<int, GameObject>();

private Dictionary<int, GameObject> pendingCleanup = new Dictionary<int, GameObject>();

private Queue<GameObject> availableSurfaces = new Queue<GameObject>();

private Queue<SurfaceData> surfaceWorkQueue = new Queue<SurfaceData>();

private bool surfaceWorkOutstanding = false;
/// Used to track when the Observer was last updated.
/// </summary>
private float updateTime;

/// <summary>
/// Indicates the current state of the Surface Observer.
/// </summary>
public ObserverStates ObserverState { get; private set; }

/// <summary>
/// Represents the center of the playspace.
/// </summary>
private Vector3 observerOrigin;

protected override void Awake()
{
    base.Awake();

    observer = new SurfaceObserver();
    ObserverState = ObserverStates.Stopped;
    observerOrigin = Vector3.zero;
}

/// <summary>
/// Can be called to override the default origin for the observed volume
/// </summary>
public void SetObserverOrigin(Vector3 origin)
{
    observerOrigin = origin;
}

/// <summary>
/// Called once per frame.
/// </summary>
private void Update()
{
    // Only do processing if the observer is running.
    if (ObserverState == ObserverStates.Running)
    {
        // If we don't have mesh creation in flight, but we could schedule mesh creation, do so.
        if (surfaceWorkOutstanding == false && surfaceWorkQueue.Count > 0)
        {
            // Pop the SurfaceData off the queue. A more sophisticated algorithm could prioritize
            // the queue based on distance to the user or some other metric.
SurfaceData surfaceData = surfaceWorkQueue.Dequeue();

// If RequestMeshAsync succeeds, then we have successfully scheduled mesh creation.
surfaceWorkOutstanding = observer.RequestMeshAsync(surfaceData, SurfaceObserver_OnDataReady);

// If we don't have any other work to do, and enough time has passed since the previous
// update request, request updates for the spatial mapping data.
else if (surfaceWorkOutstanding == false && (Time.time - updateTime) >= TimeBetweenUpdates)
{
    observer.SetVolumeAsAxisAlignedBox(observerOrigin, Extents);
    observer.Update(SurfaceObserver_OnSurfaceChanged);
    updateTime = Time.time;
}

/// <summary>
/// Starts the Surface Observer.
/// </summary>
public void StartObserving()
{
    if (ObserverState != ObserverStates.Running)
    {
        Debug.Log("Starting the observer.");
        // on device, this isn't necessary, but sometimes in the emulator the observer
        // won't realize that it hasn't already sent you the surfaces, and since the surfaces
        // don't really get updated in the emulator you'll end up getting no surfaces at all.
        if (surfaces.Count == 0)
        {
            if (observer != null)
            {
                observer.Dispose();
                observer = null;
            }

            observer = new SurfaceObserver();
        }

        ObserverState = ObserverStates.Running;
    }

    // We want the first update immediately.
    updateTime = 0;
/// <summary>
/// Stops the Surface Observer.
/// </summary>
/// <remarks>Sets the Surface Observer state to ObserverStates.Stopped.</remarks>
public void StopObserving()
{
    if (ObserverState == ObserverStates.Running)
    {
        Debug.Log("Stopping the observer.");
        ObserverState = ObserverStates.Stopped;
    }
}

/// <summary>
/// Handles the SurfaceObserver's OnDataReady event.
/// </summary>
/// <param name="cookedData">Struct containing output data.</param>
/// <param name="outputWritten">Set to true if output has been written.</param>
/// <param name="elapsedCookTimeSeconds">Seconds between mesh cook request and propagation of this event.</param>
private void SurfaceObserver_OnDataReady(SurfaceData cookedData, bool outputWritten, float elapsedCookTimeSeconds)
{
    // We have new visuals, so we can disable and clean up the older surface
    GameObject surfaceToCleanup;
    if (pendingCleanup.TryGetValue(cookedData.id.handle, out surfaceToCleanup))
    {
        CleanupSurface(surfaceToCleanup);
        pendingCleanup.Remove(cookedData.id.handle);
    }

    GameObject surface;
    if (surfaces.TryGetValue(cookedData.id.handle, out surface))
    {
        // Set the draw material for the renderer.
        MeshRenderer renderer = surface.GetComponent<MeshRenderer>();
        renderer.sharedMaterial = SpatialMappingManager.Instance.SurfaceMaterial;
        renderer.enabled = SpatialMappingManager.Instance.DrawVisualMeshes;

        if (SpatialMappingManager.Instance.CastShadows == false)
        {
        }
    }
}

GameObject surface;
if (surfaces.TryGetValue(cookedData.id.handle, out surface))
{
    // Set the draw material for the renderer.
    MeshRenderer renderer = surface.GetComponent<MeshRenderer>();
    renderer.sharedMaterial = SpatialMappingManager.Instance.SurfaceMaterial;
    renderer.enabled = SpatialMappingManager.Instance.DrawVisualMeshes;

    if (SpatialMappingManager.Instance.CastShadows == false)
    {
    }
}
surfaceWorkOutstanding = false;
SurfaceObserver.SurfaceDataReadyDelegate dataReady = DataReady;
if (dataReady != null)
{
    dataReady(cookedData, outputWritten, elapsedCookTimeSeconds);
}

private void CleanupSurface(GameObject surface)
{
    // Destroy the meshes, and add the surface back for reuse
    CleanupMeshes(surface.GetComponent<MeshFilter>().sharedMesh,
    surface.GetComponent<MeshCollider>().sharedMesh);
    availableSurfaces.Enqueue(surface);
    surface.name = "Unused Surface";
    surface.SetActive(false);
}

private void CleanupMeshes(Mesh visualMesh, Mesh colliderMesh)
{
    if (colliderMesh != null && colliderMesh != visualMesh)
    {
        Destroy(colliderMesh);
    }

    if (visualMesh != null)
    {
        Destroy(visualMesh);
    }
}

private GameObject GetSurfaceObject(int surfaceID, Transform parentObject)
{
    // If we have surfaces ready for reuse, use those first
    if (availableSurfaces.Count > 1)
    {
        GameObject existingSurface = availableSurfaces.Dequeue();
        existingSurface.SetActive(true);
        existingSurface.name = string.Format("Surface-{0}", surfaceID);
        UpdateSurfaceObject(existingSurface, surfaceID);
    }

    return existingSurface;
// If we are adding a new surface, construct a GameObject
to represent its state and attach some Mesh-related
components to it.
GameObject toReturn = AddSurfaceObject(null, string.Format("Surface-{0}",
surfaceID), transform, surfaceID);
toReturn.AddComponent<WorldAnchor>();
return toReturn;
}

/// <summary>
/// Handles the SurfaceObserver's OnSurfaceChanged event.
/// </summary>
/// <param name="id">The identifier assigned to the surface which has
changed.</param>
/// <param name="changeType">The type of change that occurred on the
surface.</param>
/// <param name="bounds">The bounds of the surface.</param>
/// <param name="updateTime">The date and time at which the change
occurred.</param>
private void SurfaceObserver_OnSurfaceChanged(SurfaceId id, SurfaceChange
changeType, Bounds bounds, System.DateTime updateTime)
{
    // Verify that the client of the Surface Observer is expecting updates.
    if (ObserverState != ObserverStates.Running)
    {
        return;
    }

    GameObject surface;

    switch (changeType)
    {
        // Adding and updating are nearly identical. The only difference is if a new
gameobject to contain
        // the surface needs to be created.
        case SurfaceChange.Added:
case SurfaceChange.Updated:
            // Check to see if the surface is known to the observer.  
            // If so, we want to add it for cleanup after we get new meshes
            // We do this because Unity doesn't properly cleanup baked collision data
            if (surfaces.TryGetValue(id.handle, out surface))
            {
                }
pendingCleanup.Add(id.handle, surface);
surfaces.Remove(id.handle);
}

// Get an available surface object ready to be used
surface = GetSurfaceObject(id.handle, transform);

// Add the surface to our dictionary of known surfaces so
// we can interact with it later.
surfaces.Add(id.handle, surface);

// Add the request to create the mesh for this surface to our work queue.
QueueSurfaceDataRequest(id, surface);
break;

case SurfaceChange.Removed:
// Always process surface removal events.
// This code can be made more thread safe
if (surfaces.TryGetValue(id.handle, out surface))
{
    surfaces.Remove(id.handle);
    CleanupSurface(surface);
    RemoveSurfaceObject(surface, false);
}

    break;

// Event
if (SurfaceChanged != null)
{
    SurfaceChanged(id, changeType, bounds, updateTime);
}
}

/// <summary>
/// Calls GetMeshAsync to update the SurfaceData and re-activate the surface object
/// when ready.
/// </summary>
private void QueueSurfaceDataRequest(SurfaceId id, GameObject surface)
{
    SurfaceData surfaceData = new SurfaceData(id,
        surface.GetComponent<MeshFilter>(),
        surface.GetComponent<WorldAnchor>(),
        surface.GetComponent<MeshCollider>(),
        surface.GetComponent<MeshCollider>(),
        surface.GetComponent<MeshFilter>(),
        surface.GetComponent<WorldAnchor>(),
        surface.GetComponent<MeshCollider>(),
TrianglesPerCubicMeter,
true);

surfaceWorkQueue.Enqueue(surfaceData);
}

/// <summary>
/// Called when the GameObject is unloaded.
/// </summary>
private void OnDestroy()
{
    // Stop the observer.
    StopObserving();

    observer.Dispose();
    observer = null;

    // Clear our surface mesh collection.
    surfaces.Clear();
}

B.2.2 SpatialMappingManager.cs

// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for license information.

using System.Collections.Generic;
using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;
#endif

namespace HoloToolkit.Unity
{
    /// <summary>
    /// The SpatialMappingManager class allows applications to use a SurfaceObserver or a
    /// stored Spatial Mapping mesh (loaded from a file).
    /// When an application loads a mesh file, the SurfaceObserver is stopped.
    /// Calling StartObserver() clears the stored mesh and enables real-time SpatialMapping
    /// updates.
    /// </summary>
[RequireComponent(typeof(SpatialMappingObserver))]

public partial class SpatialMappingManager : Singleton<SpatialMappingManager>
{
    [Tooltip("The physics layer for spatial mapping objects to be set to.")]
    public int PhysicsLayer = 31;

    [Tooltip("The material to use for rendering spatial mapping data.")]
    public Material surfaceMaterial;

    [Tooltip("Determines if the surface observer should be automatically started.")]
    public bool autoStartObserver = true;

    [Tooltip("Determines if spatial mapping data will be rendered.")]
    public bool drawVisualMeshes = false;

    [Tooltip("Determines if spatial mapping data will cast shadows.")]
    public bool castShadows = false;

    /// <summary>
    /// Used for gathering real-time Spatial Mapping data on the HoloLens.
    /// </summary>
    private SpatialMappingObserver surfaceObserver;

    /// <summary>
    /// Time when StartObserver() was called.
    /// </summary>
    [HideInInspector]
    public float StartTime { get; private set; }

    /// <summary>
    /// The current source of spatial mapping data.
    /// </summary>
    public SpatialMappingSource Source { get; private set; }

    // Called when the GameObject is first created.
    private void Awake()
    {
        surfaceObserver = gameObject.GetComponent<SpatialMappingObserver>();
        Source = surfaceObserver;
    }

    // Use for initialization.
    private void Start()
    {
        if (autoStartObserver)
        {
            }
StartObserver();

/// <summary>
/// Returns the layer as a bit mask.
/// </summary>
public int LayerMask
{
    get { return (1 << PhysicsLayer); }
}

/// <summary>
/// The material to use when rendering surfaces.
/// </summary>
public Material SurfaceMaterial
{
    get
    {
        return surfaceMaterial;
    }
    set
    {
        if (value != surfaceMaterial)
        {
            surfaceMaterial = value;
            SetSurfaceMaterial(surfaceMaterial);
        }
    }
}

/// <summary>
/// Specifies whether or not the SpatialMapping meshes are to be rendered.
/// </summary>
public bool DrawVisualMeshes
{
    get
    {
        return drawVisualMeshes;
    }
    set
    {
        if (value != drawVisualMeshes)
        {
            drawVisualMeshes = value;
            UpdateRendering(drawVisualMeshes);
        }
    }
}
/// <summary>
/// Specifies whether or not the SpatialMapping meshes can cast shadows.
/// </summary>
public bool CastShadows
{
    get
    {
        return castShadows;
    }
    set
    {
        if (value != castShadows)
        {
            castShadows = value;
            SetShadowCasting(castShadows);
        }
    }
}

/// <summary>
/// Sets the source of surface information.
/// </summary>
/// <param name="mappingSource">The source to switch to. Null means return to the live stream if possible.</param>
public void SetSpatialMappingSource(SpatialMappingSource mappingSource)
{
    UpdateRendering(false);

    if (mappingSource == null)
    {
        Source = surfaceObserver;
    }
    else
    {
        Source = mappingSource;
    }

    UpdateRendering(DrawVisualMeshes);
}

/// <summary>
/// Sets the material used by all Spatial Mapping meshes.
public void SetSurfaceMaterial(Material surfaceMaterial)
{
    SurfaceMaterial = surfaceMaterial;
    if (DrawVisualMeshes)
    {
        foreach (Renderer renderer in Source.GetMeshRenderers())
        {
            if (renderer != null)
            {
                renderer.sharedMaterial = surfaceMaterial;
            }
        }
    }
}

public bool IsObserverRunning()
{
    return surfaceObserver.ObserverState == ObserverStates.Running;
}

public void StartObserver()
{
    #if UNITY_EDITOR
    // Allow observering if a device is present (Holographic Remoting)
    if (!UnityEngine.VR.VRDevice.isPresent) return;
    #endif
    if (!IsObserverRunning())
    {
        surfaceObserver.StartObserving();
        StartTime = Time.time;
    }
}

public void StopObserver()
{  
#if UNITY_EDITOR
    // Allow observering if a device is present (Holographic Remoting)  
    if(!UnityEngine.VR.VRDevice.isPresent) return;  
#endif
    if (IsObserverRunning())  
    {  
        surfaceObserver.StopObserving();  
    }  
}

/// <summary>
/// Gets all meshes that are associated with the SpatialMapping mesh.  
/// </summary>
/// <returns>
/// Collection of Mesh objects representing the SpatialMapping mesh.  
/// </returns>
public List<Mesh> GetMeshes()
{
    List<Mesh> meshes = new List<Mesh>();  
    List<MeshFilter> meshFilters = GetMeshFilters();  

    // Get all valid mesh filters for observed surfaces.  
    foreach (MeshFilter filter in meshFilters)
    {
        // GetMeshFilters ensures that both filter and filter.sharedMesh are not null.  
        meshes.Add(filter.sharedMesh);  
    }

    return meshes;
}

/// <summary>
/// Gets all the surface objects associated with the Spatial Mapping mesh.  
/// </summary>
/// <returns>Collection of SurfaceObjects.</returns>
public List<SpatialMappingSource.SurfaceObject> GetSurfaceObjects()
{
    return Source.SurfaceObjects;
}

/// <summary>
/// Gets all Mesh Filter objects associated with the Spatial Mapping mesh.  
/// </summary>
/// <returns>Collection of Mesh Filter objects.</returns>
public List<MeshFilter> GetMeshFilters()
private void SetShadowCasting(bool castShadows)
{
    CastShadows = castShadows;
    foreach (Renderer renderer in Source.GetMeshRenderers())
    {
        if (renderer != null)
        {
            if (castShadows)
            {
            }
            else
            {
            }
        }
    }
}

private void UpdateRendering(bool Enable)
{
    List<MeshRenderer> renderers = Source.GetMeshRenderers();
    for (int index = 0; index < renderers.Count; index++)
    {
        if (renderers[index] != null)
        {
            renderers[index].enabled = Enable;
            if (Enable)
            {
                renderers[index].sharedMaterial = SurfaceMaterial;
            }
        }
    }
}
using UnityEngine;

namespace HoloToolkit.Unity
{
    public class ObjectSurfaceObserver : SpatialMappingSource
    {
        [Tooltip("The room model to use when loading meshes in Unity.")]
        public GameObject roomModel;

        // Use this for initialization.
        private void Start()
        {
            if (UNITY_EDITOR)
            {
                // When in the Unity editor, try loading saved meshes from a model.
                Load(roomModel);

                if (GetMeshFilters().Count > 0)
                {
                    SpatialMappingManager.Instance.SetSpatialMappingSource(this);
                }
            }
        }

        /// <summary>
        /// Loads the SpatialMapping mesh from the specified room object.
        /// </summary>
        /// <param name="roomModel">The room model to load meshes from.</param>
        public void Load(GameObject roomModel)
        {
            if (roomModel == null)
            {
                Debug.Log("No room model specified.");
                return;
            }

            GameObject roomObject = GameObject.Instantiate(roomModel);

            if (roomObject) // Ensure roomObject was instantiated.
            {
                // Continue with loading mesh...
            }
        }
    }
}
Cleanup();

try
{
    MeshFilter[] roomFilters = roomObject.GetComponentsInChildren<MeshFilter>();

    foreach (MeshFilter filter in roomFilters)
    {
        GameObject surface = AddSurfaceObject(filter.sharedMesh, "roomMesh-" + SurfaceObjects.Count, transform);
        Renderer renderer = surface.GetComponent<MeshRenderer>();

        if (SpatialMappingManager.Instance.DrawVisualMeshes == false)
        {
            renderer.enabled = false;
        }

        if (SpatialMappingManager.Instance.CastShadows == false)
        {
        }

        // Reset the surface mesh collider to fit the updated mesh.
        // Unity tribal knowledge indicates that to change the mesh assigned to a
        // mesh collider, the mesh must first be set to null. Presumably there
        // is a side effect in the setter when setting the shared mesh to null.
        MeshCollider collider = surface.GetComponent<MeshCollider>();
        collider.sharedMesh = null;
        collider.sharedMesh = surface.GetComponent<MeshFilter>().sharedMesh;
    }
}
catch
{
    Debug.Log("Failed to load object " + roomModel.name);
}
finally
{
    if (roomModel != null && roomObject != null)
    {
        GameObject.DestroyImmediate(roomObject);
    }
}
B.3 MAIN CAMERA SCRIPTS

B.3.1 ZoomOutGUI.cs

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

class ZoomOutGUI : MonoBehaviour {
    public void Update()
    {
        ZoomOut();
    }

    public void ZoomOut()
    {
        if (Camera.main.fieldOfView < 125)
            Camera.main.fieldOfView += 5.0F;
        if (Camera.main.orthographicSize <= 20)
            Camera.main.orthographicSize += 0.5F;
    }
}

B.3.2 ZoomInGUI.cs

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

class ZoomInGUI : MonoBehaviour {
    public void Update()
    {
        ZoomIn();
    }

    public void ZoomIn()
    {
        if (Camera.main.fieldOfView > 2)
            Camera.main.fieldOfView -= 5.0F;
    }
}
if (Camera.main.orthographicSize >= 1)
    Camera.main.orthographicSize -= 0.5F;
}

}

B.4 CURSOR SCRIPT

B.4.1 CursorManager.cs

// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for license
// information.

using UnityEngine;

namespace HoloToolkit.Unity {
    /// <summary>
    /// CursorManager class uses two GameObjects to render the Cursor.
    /// One is used when on Holograms and the other when off Holograms.
    /// 1. Shows the appropriate Cursor when a Hologram is hit.
    /// 2. Places the appropriate Cursor at the hit position.
    /// 3. Matches the Cursor normal to the hit surface.
    /// </summary>
    public partial class CursorManager : Singleton<CursorManager>
    {
        [Tooltip("Drag the Cursor object to show when it hits a hologram.")]
        public GameObject CursorOnHolograms;

        [Tooltip("Drag the Cursor object to show when it does not hit a hologram.")]
        public GameObject CursorOffHolograms;

        [Tooltip("Distance, in meters, to offset the cursor from the collision point.")]
        public float DistanceFromCollision = 0.01f;

        private void Awake()
        {
            // Hide the Cursors to begin with.
            if (CursorOnHolograms != null)
            {
                CursorOnHolograms.SetActive(false);
            }
            if (CursorOffHolograms != null)
            {
                CursorOffHolograms.SetActive(false);
            }
        }
    }
}
// Make sure there is a GazeManager in the scene
if (GazeManager.Instance == null)
{
    Debug.LogWarning("CursorManager requires a GazeManager in your scene.");
    enabled = false;
}

private void LateUpdate()
{
    // Enable/Disable the cursor based whether gaze hit a hologram
    if (CursorOnHolograms != null)
    {
        CursorOnHolograms.SetActive(GazeManager.Instance.Hit);
    }
    if (CursorOffHolograms != null)
    {
        CursorOffHolograms.SetActive(!GazeManager.Instance.Hit);
    }

    // Place the cursor at the calculated position.
    gameObject.transform.position = GazeManager.Instance.Position + GazeManager.Instance.Normal * DistanceFromCollision;

    // Orient the cursor to match the surface being gazed at.
    gameObject.transform.up = GazeManager.Instance.Normal;
}

B.5 OBJECT SCRIPTS

B.5.1 GestureManipulator.cs

// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for license information.

using UnityEngine;
using UnityEngine.VR.WSA.Input;

namespace HoloToolkit.Unity
{

}
/// <summary>
/// A component for moving an object via the GestureManager manipulation gesture.
/// </summary>
/// <remarks>
/// When an active GestureManipulator component is attached to a GameObject it will subscribe
/// to GestureManager's manipulation gestures, and move the GameObject when a ManipulationGesture occurs.
/// If the GestureManipulator is disabled it will not respond to any manipulation gestures.
/// This means that if multiple GestureManipulators are active in a given scene when a manipulation
/// gesture is performed, all the relevant GameObjects will be moved. If the desired behavior is that only
/// a single object be moved at a time, it is recommended that objects which should not be moved disable
/// their GestureManipulators, then re-enable them when necessary (e.g. the object is focused).
/// </remarks>
public class GestureManipulator : MonoBehaviour
{
    [Tooltip("How much to scale each axis of movement (camera relative) when manipulating the object")]
    public Vector3 PositionScale = new Vector3(2.0f, 2.0f, 4.0f); // Default tuning values, expected to be modified per application

    private Vector3 initialManipulationPosition;
    private Vector3 initialObjectPosition;
    private Interpolator targetInterpolator;
    private GestureManager gestureManager;
    private bool Manipulating { get; set; }

    private void Awake()
    {
        gestureManager = GestureManager.Instance;
    }
if (gestureManager == null)
{
    Debug.LogError(string.Format("GestureManipulator on {0} could not find GestureManager instance, manipulation will not function", name));
}
}

private void OnEnable()
{
    gestureManager.OnManipulationStarted += BeginManipulation;
    gestureManager.OnManipulationCompleted += EndManipulation;
    gestureManager.OnManipulationCanceled += EndManipulation;
}

private void OnDisable()
{
    gestureManager.OnManipulationStarted -= BeginManipulation;
    gestureManager.OnManipulationCompleted -= EndManipulation;
    gestureManager.OnManipulationCanceled -= EndManipulation;

    Manipulating = false;
}

private void BeginManipulation(InteractionSourceKind sourceKind)
{
    // Check if the gesture manager is not null, we're currently focused on this GameObject, and a current manipulation is in progress.
    if (gestureManager != null && gestureManager.FocusedObject != null && gestureManager.FocusedObject == gameObject && gestureManager.ManipulationInProgress)
    {
        Manipulating = true;

        targetInterpolator = gameObject.GetComponent<Interpolator>();

        // In order to ensure that any manipulated objects move with the user, we do all our math relative to the camera,
        // so when we save the initial manipulation position and object position we first transform them into the camera's coordinate space
initialManipulationPosition = Camera.main.transform.InverseTransformPoint(gestureManager.ManipulationPosition);
initialObjectPosition = Camera.main.transform.InverseTransformPoint(transform.position);
}
}

private void EndManipulation(InteractionSourceKind sourceKind)
{
    Manipulating = false;
}

// Update is called once per frame
private void Update()
{
    if (Manipulating)
    {
        // First step is to figure out the delta between the initial manipulation position and the current manipulation position
        Vector3 localManipulationPosition = Camera.main.transform.InverseTransformPoint(gestureManager.ManipulationPosition);
        Vector3 initialToCurrentPosition = localManipulationPosition - initialManipulationPosition;

        // When performing a manipulation gesture, the manipulation generally only translates a relatively small amount.
        // If we move the object only as much as the input source itself moves, users can only make small adjustments before
        // the source is lost and the gesture completes. To improve the usability of the gesture we scale each
        // axis of movement by some amount (camera relative). This value can be changed in the editor or
        // at runtime based on the needs of individual movement scenarios.
        Vector3 scaledLocalPositionDelta = Vector3.Scale(initialToCurrentPosition, PositionScale);

        // Once we've figured out how much the object should move relative to the camera we apply that to the initial
        // camera relative position. This ensures that the object remains in the appropriate location relative to the camera
// and the input source as the camera moves. The allows users to use both gaze
and gesture to move objects. Once they
// begin manipulating an object they can rotate their head or walk around and the
object will move with them
// as long as they maintain the gesture, while still allowing adjustment via input
movement.
Vector3 localObjectPosition = initialObjectPosition + scaledLocalPositionDelta;
Vector3 worldObjectPosition = Camera.main.transform.TransformPoint(localObjectPosition);

// If the object has an interpolator we should use it, otherwise just move the
transform directly
if (targetInterpolator != null)
{
    targetInterpolator.SetTargetPosition(worldObjectPosition);
}
else
{
    transform.position = worldObjectPosition;
}

B.5.2 TapToPlace.cs

// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for license
information.

using UnityEngine;

namespace HoloToolkit.Unity
{
    /// <summary>
    /// The TapToPlace class is a basic way to enable users to move objects
    /// and place them on real world surfaces.
    /// Put this script on the object you want to be able to move.
    /// Users will be able to tap objects, gaze elsewhere, and perform the
    ///// tap gesture again to place.
    /// This script is used in conjunction with GazeManager, GestureManager,
/// and SpatialMappingManager.
/// TapToPlace also adds a WorldAnchor component to enable persistence.
/// </summary>

public partial class TapToPlace : MonoBehaviour
{
    [Tooltip("Supply a friendly name for the anchor as the key name for the
WorldAnchorStore.")]
    public string SavedAnchorFriendlyName = "SavedAnchorFriendlyName";

    /// <summary>
    /// Manages persisted anchors.
    /// </summary>
    private WorldAnchorManager anchorManager;

    /// <summary>
    /// Controls spatial mapping. In this script we access spatialMappingManager
/// to control rendering and to access the physics layer mask.
    /// </summary>
    private SpatialMappingManager spatialMappingManager;

    /// <summary>
    /// Keeps track of if the user is moving the object or not.
    /// </summary>
    private bool placing;

    private void Start()
    {
        // Make sure we have all the components in the scene we need.
        anchorManager = WorldAnchorManager.Instance;
        if (anchorManager == null)
        {
            Debug.LogError("This script expects that you have a WorldAnchorManager
component in your scene.");
        }

        spatialMappingManager = SpatialMappingManager.Instance;
        if (spatialMappingManager == null)
        {   
            Debug.LogError("This script expects that you have a SpatialMappingManager
component in your scene.");
        }

        if (anchorManager != null && spatialMappingManager != null)
        {
            anchorManager.AttachAnchor(this.gameObject, SavedAnchorFriendlyName);
        }
    }
}
else {
    // If we don't have what we need to proceed, we may as well remove ourselves.
    Destroy(this);
}

// Called by GazeGestureManager when the user performs a tap gesture.
public void OnSelect() {
    // On each tap gesture, toggle whether the user is in placing mode.
    placing = !placing;

    // If the user is in placing mode, display the spatial mapping mesh.
    if (placing) {
        spatialMappingManager.DrawVisualMeshes = true;
        Debug.Log(gameObject.name + " : Removing existing world anchor if any.");
        anchorManager.RemoveAnchor(gameObject);
    }
    // If the user is not in placing mode, hide the spatial mapping mesh.
    else {
        spatialMappingManager.DrawVisualMeshes = false;
        // Add world anchor when object placement is done.
        anchorManager.AttachAnchor(gameObject, SavedAnchorFriendlyName);
    }
}

private void Update() {
    // If the user is in placing mode,
    // update the placement to match the user's gaze.
    if (placing) {
        // Do a raycast into the world that will only hit the Spatial Mapping mesh.
        var headPosition = Camera.main.transform.position;
        var gazeDirection = Camera.main.transform.forward;
        RaycastHit hitInfo;
        if (Physics.Raycast(headPosition, gazeDirection, out hitInfo, 30.0f, spatialMappingManager.LayerMask)) {
            RaycastHit hitInfo;
            if (Physics.Raycast(headPosition, gazeDirection, out hitInfo, 30.0f, spatialMappingManager.LayerMask)) {

        }
// Move this object to where the raycast
// hit the Spatial Mapping mesh.
// Here is where you might consider adding intelligence
// to how the object is placed. For example, consider
// placing based on the bottom of the object's
// collider so it sits properly on surfaces.
this.transform.position = hitInfo.point;

// Rotate this object to face the user.
Quaternion toQuat = Camera.main.transform.localRotation;
toQuat.x = 0;
toQuat.z = 0;
this.transform.rotation = toQuat;