Development of Multiview Image/Video Stitching Systems for Mobile Devices

by

Andrew Au

B.A.Sc. (Electronics Engineering), Simon Fraser University, 2010

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science

in the

School of Engineering Science
Faculty of Applied Sciences

©Andrew Au 2013
SIMON FRASER UNIVERSITY
Spring 2013

All rights reserved.
However, in accordance with the Copyright Act of Canada, this work may be reproduced, without authorization, under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.
Approval

Name: Andrew Au
Degree: Master of Applied Science
Title of Thesis: Development of Multiview Image/Video Stitching Systems for Mobile Devices

Examiner Committee:
Chair: Dr. Shahram Payandeh, Chair
Professor, P.Eng., School of Engineering Science

Dr. Jie Liang, Senior Supervisor
Associate Professor, P.Eng., School of Engineering Science

Dr. Sami (Hakam) Muhaidat, Supervisor
Adjunct Professor, P.Eng., School of Engineering Science

Dr. Jiangchuan Liu, Examiner
Associate Professor, School of Computing Science

Date Defended/Approved:
January 9, 2013
Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the “Institutional Repository” link of the SFU Library website (www.lib.sfu.ca) at http://summit.sfu.ca and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author’s written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2011
Abstract

The latest smartphones provide excellent platforms for constructing 3D scenes from a set of photos or videos. While there already exist many mobile applications that can stitch photos to create a single, panoramic photo, this thesis presents the design of mobile applications that can create panoramic scenes by using either photos or videos, where they are projected in a 3D space using the pinhole camera model, ultimately maintaining a realistic perspective of the scenes. Furthermore, in addition to being able to manually fine-tune each photo or video’s position via the touchscreen, automatic alignment is also developed using state-of-the-art computer vision techniques. The systems enable scenes to be easily shared to other users, and are developed using the Silverlight framework so that it can run across multiple platforms, including Windows Phone 7, the PC, and tablet devices that support Silverlight. One of the mobile applications has been downloaded over 120,000 times.
I would like to extend my sincere gratitude to all faculty members of the School of Engineering Science, Simon Fraser University, who have supported me over the years. I would like to thank especially professor Jie Liang, for his innovative project ideas, and his continuous contribution to my work. The opportunity that he has given me to work with him has become one of the highlights of my life. I would also like to warmly thank Dr. Jiangchuan Liu, Dr. Sami Muhaidat, and Dr. Shahram Payandeh for their time and willingness to review and improve my thesis.

I would also like to thank Nokia for their financial support, and extend my thanks to my family and friends who have shaped me the person I am today.
# Contents

Approval ........................................ ii  
Abstract ........................................ iii  
Acknowledgements .............................. iv  
Contents .......................................... v  
List of Figures ................................. vii  
List of Tables ................................ ix  

## 1 Introduction ................................. 1  
1.1 Introduction ...................................... 1  
1.2 Thesis Outline ................................... 6  
1.3 List of Contributions .......................... 7  

## 2 Ztitch: Immersive & Interactive Image Stitching System 9  
2.1 Image Stitching Background .................... 9  
2.2 Motivation ....................................... 10  
2.3 Scene Creation and Navigation .................. 13  
2.3.1 Coordinate Pipeline and Spherical Coordinate .... 14  
2.3.1.1 Graphics 3D Matrices ......................... 16  
2.3.1.2 Translations, Rotations, and Scaling .......... 16  
2.3.1.3 Graphics Pipeline ............................ 18  
2.3.2 Immersive Panorama Creation and Editing .... 22  
2.4 Color Balancing ................................... 24  
2.5 Multiband Blending .............................. 28  
2.6 Gap Closing ..................................... 31  
2.7 Using and Optimizing SIFT for Ztitch .......... 36  
2.7.1 Approximating the Laplacian of Gaussian ....... 37  
2.7.2 Finding the Keypoints ....................... 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7.3</td>
<td>Filtering the Keypoints</td>
<td>40</td>
</tr>
<tr>
<td>2.7.4</td>
<td>Assigning Orientation to Keypoints</td>
<td>42</td>
</tr>
<tr>
<td>2.7.5</td>
<td>Generating Descriptors</td>
<td>43</td>
</tr>
<tr>
<td>2.7.6</td>
<td>Application to Ztitch</td>
<td>44</td>
</tr>
<tr>
<td>2.7.6.1</td>
<td>Keypoint Matching and Image Realigning</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>Stitching of Multiview Videos</td>
<td>51</td>
</tr>
<tr>
<td>3.1</td>
<td>Background on Multiview Videos</td>
<td>51</td>
</tr>
<tr>
<td>3.2</td>
<td>Bundler - Structure from Motion</td>
<td>53</td>
</tr>
<tr>
<td>3.3</td>
<td>Real-Time Visual Tracking</td>
<td>57</td>
</tr>
<tr>
<td>3.4</td>
<td>Improving the First-Frame Matches</td>
<td>60</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Making SIFT Affine-Invariant</td>
<td>61</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Using Local Symmetry Features</td>
<td>62</td>
</tr>
<tr>
<td>4</td>
<td>Conclusion</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>Appendix A  Microsoft Silverlight</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>Appendix B  Bundler: Structure from Motion</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>Appendix C  Transformation Matrices</td>
<td>76</td>
</tr>
</tbody>
</table>
List of Figures

2.1 (a) A traditional photo-mosaic which warps the images onto a flat surface. (b) Our system which projects the images on a sphere in 3D space preserving the perspective view and can be navigated. ............................................. 11

2.2 A screenshot of a 3D panoramic scene comprised of multiple images. The scene was created by a user using our application, uploaded to our server, and viewed on the PC. ......................... 13

2.3 The user interface of our application as the user aligns the current frame with the previous shot. ................................................................. 13

2.4 Spherical coordinates described by $\theta_x$, $\theta_y$, $\theta_T$, and $r$. .... 15

2.5 An arbitrary matrix is shown. The first row of the 3x3 (or 4x4) affine matrix represents the up-vector in the 3D coordinate, and the second row represents the left- vector. In the 4x4 matrix, the 3rd row is the forward-vector, which is normal to the plane created by the first two rows. One can calculate a vector parallel to the 3rd vector by taking the cross product of the first two vectors, and hence, the 4x4 matrix is not needed. .................. 17

2.6 The frustum visualized. ................................................................. 20

2.7 Overview of the graphics pipeline. ............................................... 22

2.8 (a) Exposure and white balance differences among the three images create an unpleasant result. (b) Using our method to balance out the color of the images to achieve a much smoother scene. ................................................................. 26

2.9 (a) Constructing a Gaussian pyramid from a source image. (b) Building the Laplacian pyramid (bottom) from the Gaussian pyramid (top). ................................................................. 28

2.10 (a),(b) Original source images to be blended. (c) Using linear alpha blending with a narrow transition width. (d) Using linear blending with a wide transition width. Notice the blurring or "ghosting." (e) The binary mask. (f) Result of multiband blending with 6 levels/bands. ................................................................. 29

2.11 (a),(b) Linear blending applied to the boundaries of the scene in Fig. 2.8c. (c),(d) Multiband blending applied to the boundaries of the scene in Fig. 2.8c. ................................................................. 32
2.12 (a) An example with a gap between the two ends of a panorama sequence. Note that the gap has both horizontal and vertical components. (b) The view directly behind. .................................................. 35
2.13 (a) Result after correcting the radius \( r \) while maintaining the images’ alignments with each other. (b) The view directly behind, after the gap is closed. .................................................. 36
2.14 (a) Two images of a scene containing an overlap (b) The two sets of DoG images. .................................................. 39
2.15 Keypoint detection by checking whether the pixel, marked as black, is a maxima or minima among the 26 neighbors ........ 40
2.16 The SIFT keypoints found (unfiltered) from an image on the WP7 emulator. They are usually located at edges. ............ 41
2.17 (a) Detected features from the full-sized images. The different sized circles correspond to different scales: small circles were detected at a fine scale. (b) Nearest neighbor matching the full-sized images. (c) Detected features from the overlapping regions at only the fine scale. (d) Nearest neighbor matching only in the overlapping region, with many of the false matches eliminated. 47
2.18 (a) Small alignment error between the two images within Ztitch (b) The scene after alignment correction. .......................... 48
2.19 (a) Single scale extrapolation in which a rectangular region \( A \) is imposed at the left edge of the photo, and search for a rectangular region \( B \) whose right side is most similar to that of \( A \). (b) The extrapolated image after copying the pixels from the left side of \( B \) to \( A \). .................................................. 49
3.1 Mapping a set of videos into a 3D space. ................................. 53
3.2 The FAST algorithm: taking the 16 surrounding pixels around the candidate point \( p \), and testing whether at least \( n \) of these pixels’ intensity are all brighter (or dimmer) than \( p \) by a threshold, such that they form a contiguous segment. Source: E. Rosten and T. Drummond, ”Machine learning for high-speed corner detection”, European Conference on Computer Vision, 2006 .......... 58
3.3 Screenshot of the FAST-9 detector on the phone, with results shown as red dots displayed in the smaller top-left window. It took approximately 150ms to perform FAST on each frame. .... 60
3.4 Sampling of the parameters \( \theta \) and \( \phi \) can be visualized with a hemisphere. The red dot is one particular sample. ............... 62
3.5 Although the different pictures of the Eiffel Tower have varying quality and appearance, they all preserve the local symmetries of the tower. .................................................. 63
3.6 Slices through an image (shown as green lines) that produce approximately even functions mean that the image exhibits symmetry. .................................................. 64
List of Tables

2.1 The $\sigma$’s used for the Gaussian blurring .................. 38
Chapter 1

Introduction

1.1 Introduction

The number of photos and videos uploaded to the internet is exploding. The use of mobile phones for capturing multimedia materials continues to rise rapidly. In 2011, the Apple iPhone was marked as the most-used camera on Flickr. With over 6 billion photos uploaded on Flickr, over 100 billion photos on Facebook, and over 4 billion hours of videos on YouTube, it is apparent that online multimedia sharing is one of today’s most dominant social activities. It is with considerable certainty that this trend will continue. As such, the demand for photo and video applications will also continue to rise.

Software like Google Streetview and Microsoft Streetside have demonstrated how useful, and powerful, 360° immersive panoramas can be, allowing people to explore and learn. However, the images in both these systems were generated
using dedicated multi-camera systems that have to be mounted on customized cars or tricycles.

On the other hand, many smart phones are now equipped with powerful CPUs, high-resolution cameras, touchscreens, accelerometers and gyroscopes, making it possible to generate immersive panoramas on the phones anywhere and anytime. This thesis presents the design of some mobile-phone-based image and video stitching systems.

The first part of the thesis focuses on Ztitch and Ztitch+ (referred to in this thesis collectively as Ztitch), which are mobile phone applications that can create immersive panoramic scenes in real time, where multiple photos are arranged, either manually or automatically, in the 3D space according to their camera poses to maintain a realistic perspective of the scene. Ztitch was released on November 2010, and was partly motivated by the desktop application PhotoSynth developed by Microsoft [5], which uses various computer vision algorithms such as bundle adjustment (bundler) to recreate the 3D scenes from multiple photos [8]. However, these algorithms are time-consuming, and are difficult to be ported to mobile phones. As a result, the mobile PhotoSynth was not released until April 2011 (for iOS) and May 2012 (for Windows Phone), more than six months after Ztitch.

By taking advantage of the touchscreen, accelerometer, and gyroscope in the phone, the applications allow users to easily fine-tune each photo’s position, navigate the scene, and edit the scenes created by others. Therefore, compared to Photosynth, our system adopts a more user-aided approach rather than relying solely on the computer vision algorithms and the processing power of
the phones. This allows it to achieve similar results and arrive at the market much earlier. Ztitch shows that mobile application development is different from desktop development. By exploiting the rich sensors of the phone, the constraints imposed by the limited computational power of the phone can be circumvented, and sometimes with even better user experience.

Users can also share the scenes with others via the Ztitch website and various social networks. Ztitch also enjoys other advantages over PhotoSynth. For example, PhotoSynth only outputs the final panorama as a whole. The result is irreversible with no way to manually readjust the position of each individual image; it does not have the gap filling feature; and it does not have the sensor-based navigation. Moreover, images in Photosynth are captured in video mode, leading to low resolution and quality. The app has been downloaded over 120,000 times and is one of the top photo apps on Windows Phone.

A typical problem when creating a 360° immersive panorama is that there could be a gap or too much overlap between the first and the last photos, due to accumulated errors or unknown camera focal length. Two techniques were developed in the past to address this problem. The first one, as discussed in [3], can estimate the focal length for pure panning motion and cylindrical images, but it only works when the camera is continuously turning in the same direction, say horizontally. Therefore it cannot be used in many user-generated panoramas with relatively arbitrary camera poses. Another approach is to use the bundle adjustment method [4], which simultaneously aligns all the images under a least-square framework to correctly distribute the misregistration errors. While it works for arbitrary camera motions, this method is time-consuming when running on the phone.
In Ztitch, a user-aided algorithm is developed, where instead of using automatic but computationally demanding algorithm to estimate and adjust the focal length and the scene radius, we allow the user to drag one of the boundary photo on the touchscreen, and move it to the desired position with respect to the other boundary photo. The radius of the panorama is updated based on the amount of the dragging, and all other images will be adjusted automatically, as if they are glued together. The method is very fast and accurate. It is also more flexible than the method in [3], since we allow spherical camera poses, not just cylindrical.

Another common problem for image stitching is that different images could have different quality. After photos have been aligned together, there usually is a seam line is visible at the intersection of two neighboring photos, due to factors such as exposures differences, camera vignetting, and radial lens distortion. Although techniques have been developed in the past to try and correct this problem for panorama photos, there is often a trade-off with computational cost. In this thesis, we propose an efficient algorithm by first performing a simple color-balancing between neighboring images, then apply a low-cost linear alpha blend. We also implemented multiband blending and compared the results with linear alpha blending. Our method is faster than that found in competing systems such, as the popular AutoStitch [12].

Scale-Invariant Features Transform (SIFT) is a popular technique for finding local points of interests (features) from an image, and then assigning a unique descriptor to each feature [15]. This allows one image to be matched to another based on the existence of similarly described features across a pair of images. Although the technique was published in 1999, the technique is still proven to
be very robust by today’s standards amidst a number of new feature detection techniques. Unfortunately, the disadvantage to SIFT is high computational cost, attributed mainly from having to construct a Gaussian pyramid for each image. Therefore, we propose a few modifications to SIFT to optimize it for Ztitch, and reduce its computation time by almost 80%.

The limitation of Ztitch is that it only stitches multiple images. A more desired feature is to stitch multiple videos. The second major project covered in this thesis is a mobile multiview video system called Mobile Veaver, based on our previously developed Veaver project for the desktop. This thesis discusses the principles, design, and implementation of small-scale scene reconstruction using multi-view videos that allows it to be delivered across multiple platforms via application frameworks such as Microsoft’s Silverlight and Adobe’s Flash. One difference between Veaver and Ztitch is that the videos are not limited to being in a panoramic setting, allowing the reconstruction of scenes from videos taken at different positions. The proposed system assumes that the fundamental matrix of each video is already computed, likely to have been estimated by a bundle adjustment algorithm. It then performs a perspective projection of each video to reconstruct the scene. In the proposed system, the user will also have manual control over the position and orientation of the videos during runtime.

While there already exist systems like Immersive Media’s spherical video player for multiview videos [6], our Mobile Veaver system built in the project is the first of its kind, from our knowledge, which reconstructs a 3D scene from multiple user videos. The Immersive Media system, like other similar systems, requires a special format of video which was pre-processed from a set of videos that were captured using special equipment, but our Mobile Veaver can take standard
videos created by the user, or streamed from an online source. Our system can be used for applications such as surveillance, advertisement, and social network. It can also be used as a test bed for other research, such as multiview video streaming, multiview object tracking, and video retrieval.

Bundler [7] is a structure-from-motion system that accepts a set of unordered images, image features, and image matches to reconstruct the scene. We use this system to first position and orient the videos in a 3D environment based on the first frames. We implemented 9-point segment Features from Accelerated Segment Test (FAST-9) [29] for visual tracking. FAST is used to detect corners in real-time, and we discuss ways to improve its computational time for the mobile phone. Combining FAST corners with a tracking system, (*i.e* Parallel Tracking and Mapping (PTAM)), subsequent frames of a video during its playback can be readjusted in its local coordinate space. Then, as part of future works, we studied ways in which we can dramatically improve matching performances of images by eliminating the affine variation, and using local symmetry features. This dramatically improves SIFT’s reliability of the first frame matching.

1.2 Thesis Outline

Chapter 2 begins with the background of image stitching and the motivation for the thesis. The chapter details the graphics pipeline for a pinhole camera model. The coordinate pipeline combined with the spherical coordinate system are used to construct the 3D framework in Ztitch. Next, we explain how the manual stitching process is achieved, as well as the automatic stitching method.
Afterwards, the novel algorithms for our color balancing and gap closing are given. The multiband blending, also known as Laplacian pyramid blending, is discussed in this section. Results are clearly presented. Finally, we discuss the Scale-Invariant Features Transform, and how to optimize it specifically for our application.

Chapter 3 provides the background of multiview videos, and details two distinct parts, bundle adjustment, and visual tracking, both of which are used in arranging the videos in the 3D space.

Finally, the thesis is concluded in chapter 4.

1.3 List of Contributions

The following publications and demos have been produced during this project [32–34].


The Ztitch app developed in this thesis can be downloaded from Windows Phone Market Place. The accompanying website is www.ztitch.com.
Chapter 2

Ztitch: Immersive & Interactive Image Stitching System

2.1 Image Stitching Background

Techniques for stitching images into a large photo-mosaic are some of the oldest and most used in the field of computer vision. These techniques have a wide range of application, from producing high-resolution maps collected from satellite imageries, to creating ultra-wide-angle panoramas for the casual user equipped with only a digital camera. Manual intensive methods were first used to stitch images, where ”tie points” would have to be manually registered [14] in order to create the photo-mosaics. However, later development of algorithms known as bundle adjustment allowed for the solving for the locations of all of the camera positions simultaneously, yielding globally consistent results [13]. Furthermore, more recent algorithms use feature-based approaches, where a sparse set of features is extracted from images and used to match them to each
other. Such approach is used in [12] to automatically stitch panoramas taken by the casual user. Image stitching can be divided into four main steps:

1. Choose the mathematical model relating pixel coordinates in one image to pixel coordinates in another image
2. Align the images by estimating the relationship between various pairs
3. Choose a final compositing surface for warping the aligned images
4. Seamlessly cut and blend overlapping images

Our application differs from the many existing photo stitching algorithms [2], where the output is simply an unfolded 2D static image (Fig. 2.1a), which loses the 3D perception and occurs at step 3 above. We aim to display all the images collectively inside a 3D space (Fig. 2.1b), rather than warp them onto a flat projection.

2.2 Motivation

In the last few years, Google Streetview and Microsoft Streetside have become very useful tools for people to explore many streets in the world in a 360° panoramic setting. With the massive success of iPhone and Android-based phones, the mobile phone market has become bigger than the desktop. Since many smart phones are equipped with high performance cameras and advanced hardware, one of our objectives in this thesis is to develop a mobile phone application for 3D scene creation, navigation and sharing.
Chapter 2. Ztitch

Figure 2.1: (a) A traditional photo-mosaic which warps the images onto a flat surface. (b) Our system which projects the images on a sphere in 3D space preserving the perspective view and can be navigated.

Note that a naive way is to take some photos using the phones and upload them to web servers providing 3D scene creation services, such as Photosynth [8], but this approach has some limitations. First, users are not able to view the results on the phones. Second, it could be quite expensive to upload the photos via cellular networks, especially when the users travel out of their own networks. Although the users can wait until a free Wi-Fi network is available, the long delay is not desired. Such a naive approach essentially only uses mobile phones as regular cameras, but fails to leverage the powerful capability of the smart
Our application, Ztitch, runs on the recently released Windows Phone 7 operating system, and is developed using the Microsoft Silverlight framework (see Appendix A), due to its native support for 3D projective transform. By taking advantage of various sensors and the touchscreen available on all Windows Phone 7 devices, we develop an automatic approach to let users create panoramic scenes in real-time. Also, a simple drag-and-drop approach lets users manually re-position photos in the 3D setting if the default position is incorrect. This yields good results and avoids running the complicated SIFT and bundle adjustment algorithms on the phone.

Our application also has seamless integration with email, Flickr, Facebook, and Bing Map for users to share the results with friends. An accompanying Silverlight-based website (www.ztitch.com) is also developed to recreate the 3D scenes using the uploaded photos from Flickr or Facebook, so that other mobile or desktop users can explore the same results. Fig. 2.2 shows a 3D scene created by a user and viewed on our website in fullscreen.

Ztitch is publicly available for the Windows Phone 7 Marketplace. It has become one of the top photo apps among about hundreds of other photo applications. Note that our application differs from many existing photo stitching algorithms [2], where the output is simply a unfolded 2D static image, which loses the 3D perception.
Chapter 2. Ztitch

Figure 2.2: A screenshot of a 3D panoramic scene comprised of multiple images. The scene was created by a user using our application, uploaded to our server, and viewed on the PC.

Figure 2.3: The user interface of our application as the user aligns the current frame with the previous shot.

2.3 Scene Creation and Navigation

In this section, we describe the detailed algorithms for immersive scene creation, editing, and navigation.
Chapter 2. Ztitch

### 2.3.1 Coordinate Pipeline and Spherical Coordinate

A key step in the immersive scene creation and navigation is to apply various perspective projection transforms to a photo. Homogeneous coordinate is usually used in for this purpose. Our system uses the pinhole camera model. The corresponding graphics pipeline workflow consists of a sequence of transformations. Each transformation is represented by a $4 \times 4$ matrix. Specifically, a 3D point is first augmented to its projective equivalent $s$, which then undergoes the following matrix multiplications that yields a 2D Euclidean point $p$ in the output device coordinates [9]:

$$\begin{align*}
M_V C_P T_v M_W s & \rightarrow p,
\end{align*}$$

(2.1)

where $M_W$ is the positioning transformation, $T_v$ the viewing transformation, $C_P$ the perspective projection transformation, and $M_V$ the viewport transformation[9].

Finally, the right-hand side of the equation is projected as a 2D point.

Ztitch runs on the Windows Phone operating system, and is developed using the Microsoft Silverlight framework, which provides the necessary APIs to implement various 3D projective transforms to images/videos. For example, the Matrix3D structure defines the $4 \times 4$ matrix required by various 3D perspective projections. The Matrix3DProjection class makes it easier to apply arbitrary transformations to Silverlight elements, such as translate, scale, rotate, and perspective transforms. However, significant efforts are still needed in order to create user-friendly 3D and immersive multimedia applications from these basic APIs.
In this thesis, we only consider panoramic scenes, so all photos will be placed on the surface of a sphere. The spherical coordinate is thus very handy, which maps the center of each photo to a point \((x, y, z)\) on a sphere with radius \(r\) using the following equations, as shown in Fig. 2.4,

\[
\begin{align*}
    x &= r \sin \theta_y \cos \theta_x, \\
    y &= r \cos \theta_y, \\
    z &= r \sin \theta_y \sin \theta_x,
\end{align*}
\]  

(2.2)

where \(\theta_x\) is the horizontal angle of the photo in radians, and \(\theta_y\) is the vertical angle of the photo. An additional tangent rotation \(\theta_T\) is also needed in Ztitch, which denotes the rotation of the image in the plane tangent to the sphere at \((x, y, z)\).

The radius \(r\) is directly related to the focal length of the camera. In our system \(r\) is in units of pixels. Usually for a digital camera, the focal length can be extracted from the EXIF tags in the original photo file. The focal length also varies a lot between different devices. For example, the Nokia Lumia 800 phone
uses a camera with focal length of 28 mm. The corresponding value of $r$ is 530. The camera focal length of the Samsung Focus phone is 4 mm, with $r = 680$. All other Windows Phone 7 devices we tested have $r$ between these two values. However, some manufacturers do not disclose the camera focal length. An unknown focal length could lead to an undesired gap or overlap when creating panoramas. In Sec. 2.6, we present a fast method to resolve this problem.

2.3.1.1 Graphics 3D Matrices

The previous section discussed the derivation of the camera matrix $F$. This section will take a look at this matrix, which will be used to transform the videos. The problem of scene reconstruction will be dependent on this matrix.

Matrices are best represented using transformations mapping coordinate systems. It enables one to handle a unified 3x3 (or 4x4) matrix with linear algebra. Unification is extremely important in programming, particularly in object-oriented programming. Also, matrix concatenations (multiplication) can be computed efficiently in programs. What do the numbers in this affine matrix mean? Fig. 2.5 gives an overview.

2.3.1.2 Translations, Rotations, and Scaling

Matrices allow arbitrary linear transformations to be represented in a consistent format, suitable for computation. This also allows transformations to be concatenated easily (by multiplying their matrices). Linear transformations are not the only ones that can be represented by matrices. Using homogeneous coordinates, both affine transformations and perspective projections on $R^n$ can
Figure 2.5: An arbitrary matrix is shown. The first row of the 3x3 (or 4x4) affine matrix represents the up-vector in the 3D coordinate, and the second row represents the left-vector. In the 4x4 matrix, the 3rd row is the forward-vector, which is normal to the plane created by the first two rows. One can calculate a vector parallel to the 3rd vector by taking the cross product of the first two vectors, and hence, the 4x4 matrix is not needed.

be represented as linear transformations on $\mathbb{RP}^{n+1}$ (that is, $n+1$-dimensional real projective space). For this reason, 4x4 transformation matrices are widely used in 3D computer graphics.

For instance, for rotation angle anticlockwise about the origin, the functional form is $x' = x\cos\theta y\sin\theta$ and $y' = x\sin\theta + y\cos\theta$. Multiplying a matrix corresponds to applying that transformation to the corresponding point(s), represented by a vector. The space of 2D matrix transformations is called linear space and includes the so-called linear transformations: identity, scaling, rotation, shear, and mirror. Manipulation of the 2x2 matrix is great for image stitching where only a 2D environment is involved, but it is limited and cannot obtain the ubiquitous transformations. We resolve this issue by adding the final coordinate, the z-coordinate, and use the 3x3 matrix instead.
2.3.1.3 Graphics Pipeline

Modeling

At the first stage, objects undergo positioning and modeling transformations, and are converted from the object coordinate systems to a common world coordinate system.

Transformations are processes for positioning and modeling the objects in 3D scenes. Basically, transformations operating on these objects are classified into two types: modeling transformation, and positioning transformation. The modeling transformations are used for deforming the object by changing its fundamental intrinsic, such as forming complex models. The positioning transformation is what we are primarily interested for the purpose of this thesis, as it changes the position and orientation of the objects without modifying intrinsic properties such as the shapes. This will be used to place the videos in the 3D space.

Any object $S$ is defined by a set of points, which can be described by their coordinates. As such, the world coordinate system is defined by the origin point, a null vector, associated with the standard basis $x$, $y$, and $z$ of perpendicular vectors acting as the principle axis. Any positioning transformation on the object $S$ is simply translating and/or rotating the object, and is interpreted as a single coordinate system operation. In this system positioning transformations are handled by linear functions, i.e. $f(u+v) = f(u) + f(v)$ for any two vectors $u$ and $v$, and $f(\lambda u) = \lambda f(u)$. Matrices and their linear algebra offer a convenient methodology for processing positioning transformations: $p = [p_xp_yp_z]^T = p_xx + p_yy + p_zz$ of object $S$ is mapped to $f(p) = p_xf(x) = p_yf(y) + p_zf(z)$. 
The original standard basis $x$, $y$, and $z$ is mapped to a new basis $f(x)$, $f(y)$, and $f(z)$ while the points $p_x$, $p_y$, and $p_z$ are preserved by positioning transformations. 3D objects are defined according to their local object coordinates defined by an origin and a standard basis, and are put together in a 3D scene using individual mappings to a global world coordinate system.

**Viewing**

At the second stage, the camera eye position and direction defining the projection image plane are initialized. Object coordinates are now expressed in the view coordinate by setting the world coordinate origin at the camera eye position with positional vector $(e_x e_y e_z)^T$. The positional vector $e$ sets virtually where the user’s eyes and gaze direction are with respect to the objects in the 3D space, to create a first person view of the scene. The $x$, $y$, and $z$ principal axes are mapped to the right, up, and back camera vectors, respectively.

In the Silverlight framework, the library for transforming objects in the 3D space is limited and we must set the viewing transformation manually, as opposed to OpenGL or other 3D utilities where the viewing transformation is conveniently set for the developer. In our system, the positional vector starts at the origin, and users can navigate freely with six degrees of freedom.

The viewing transform matrix $T_v$ is as a 4x4 matrix (Eq. ??) [17]. When the lookat point is at the origin the twist angle is zero, and the viewing reduces to a simple form (Eq. C.1). $T_v$ transforms the world coordinates to the viewing coordinates.
Clipping

A clipping volume is defined to select the portion of the 3D scene to project onto the 2D image. The portion of space to project and the camera image dimensions are defined by the clipping volume, also known as the frustum. Characterizing the clipping volume defines the camera image boundaries and clip objects to render. Fig. 2.6, from the Microsoft Developer Network Library, shows the front clipping plane which acts as the image boundary.

The front plane and the back plane are the far and near clipping planes. Therefore, the front plane and the back plane, combined with the field of view defines the frustum.

Perspective Projection

A projection transforms the selected 3D element in the clipped volume and projects them onto the 2D camera image plane with depth information. The projection stage is the central part of the graphics pipeline, and it transforms 3D coordinates to 2D coordinates. There are two main types of projections - orthographic and perspective.

In our multiview system, the perspective projection model is primarily what we are interested in using. For perspective views, we define a 3D clipping volume...
that is constrained by the field of view (in radian, and describes how wide the viewing angle is), the near plane, and the far plane. Fig. 2.6 shows this clipping volume bounded by the truncated pyramid. The transformation matrix for this operation is given by Eq. C.5.

**Viewport**

The final stage outputs the camera image elements to the screen. The normalized screen coordinates are converted to the device coordinate system, and this conversion process is called viewport mapping. Viewport mapping consists of 2D translation and 2D scaling operations, and the transformation matrix is given by Eq. C.13.

As discussed previously, the transformation workflow of the coordinate pipeline in graphics is summarized by the following matrix concatenation. The 3D point \( s \) is first in its object local coordinate, and then it is augmented to its projective equivalent \( s \). The point \( s \) undergoes a series of matrix multiplications that yields a 2D Euclidean point \( p \) in the output device coordinates.

\[
M_V C_P T_v M_W s \rightarrow p, 
\]

(2.3)

As a reminder, \( M_W \) is the positioning transformation, \( T_v \) the viewing transformation, \( C_P \) the perspective projection transformation, \( M_V \) is the viewport transformation, and \( s \) is the homogeneous point that went from being a 3D point in the object local coordinate to its projective equivalent. The transformation matrices can be found in Appendix C.
2.3.2 Immersive Panorama Creation and Editing

The earlier Windows Phone 7.0 did not allow taking photo in the app. To create panorama scenes using Ztitch, users need to take all photos first, then bring them into Ztitch. Using the touchscreen, users can drag each photo to its desired location in the 3D space. To avoid running time-consuming automatic image matching algorithms, each photo is displayed in a semi-transparent mode. This allows the user to visually identify the best overlapped position between neighboring photos. Although bundle adjustment and feature-based algorithms can be used to align the images automatically, these methods are computationally expensive on the phones. Besides, sometimes these algorithms do not work well, due to, e.g., large exposure differences between images, fast moving objects, and lack of distinctive features. The user-aided approach in Ztitch is intuitive and the result is quite satisfactory. It effectively walks around the limitation imposed by the hardware.

With the latest Windows Phone 7.5 Mango, photos can be taken directly in the app. Therefore, the latest Ztitch allows the users to rotate the camera, take new pictures at constant rotation intervals, and add them to the scene directly.
photos are still displayed in semi-transparent mode to help the user identifying the best position to take the next photo.

After the scene is created, Ztitch still allows the user to manually drag each image using the touchscreen to fine-tune its position. The coordinate of the center of the photo is updated during the dragging. That is, when the user drags the photo horizontally or vertically, we adjust $\theta_x$ or $\theta_y$ in Fig. 2.4. The tangent rotation $\theta_T$ is obtained by a traditional button-based user interface, since this action is not frequently used.

Let $\theta_x(t-1)$ be the photo’s horizontal angle when the photo was refreshed last time, and let $\Delta_x(t-1)$ be the $x$-axis change of the user’s finger position on the touchscreen from the last refreshing time, the photo’s new X angle will be

$$\theta_x(t) = \theta_x(t-1) + \frac{\alpha}{f} \Delta_x(t-1),$$

where $\alpha$ is the field of view of the camera, and $f$ is the focal length. The photo’s vertical angle $\theta_Y$ can be updated similarly.

We then obtain the rotations $R_x(t)$ and $R_y(t)$ from Eq. (2.5) and (2.6), and their combination $R_x(t)R_y(t)$ is applied to matrix $M_V$ in Eq. (2.3). The new location of the photo can be calculated accordingly using the spherical coordinate.
\[ R_y(t) = \begin{bmatrix}
\cos \theta_y(t) & 0 & \sin \theta_y(t) & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta_y(t) & 0 & \cos \theta_y(t) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}. \quad (2.6) \]

The application will attempt to refresh the screen as quickly as possible when the users drag the photo. The refresh frequency is about 18-25 times per second.

### 2.4 Color Balancing

In photo panoramas, after stitching photos, a seam line is usually visible at the intersection of two neighboring photos, due to their different exposures and/or white-balance levels, as shown in the example of Fig. 2.8 (a).

For applications on iPhone/iPad such as Photosynth, developers have an ample amount of controls over the camera’s functionalities through the provided APIs, such as locking the exposure and white-balance when the user takes different pictures in the scene. This allows the panorama to have a more homogeneous appearance. However, it also creates another problem, as some images could be too bright or too dark, especially for 360° scene taken in sunny days.

Windows Phone 7 does not provide such APIs to the developers, and the exposure and white-balance are handled automatically by the camera. To produce a smoother looking final result, a color correction method has to be used. The challenge is that it should have low complexity in order to run on mobile phones.

Various algorithms have been developed in the past to blend images together and create a more seamless result. One of the most successful algorithms is the
Laplacian pyramid blending (or multi-band blending) [10]. First, it decomposes each source image into its own Laplacian pyramid, consisting of high, medium, and low frequency parts. It then tries to smoothly blend the low-frequency variations (i.e. color), but quickly blends the high-frequency variations (i.e. texture). While it has good result, the method is very slow. A more recent method is developed in [11], but it requires knowing the exposure value and uses high dynamic range images.

In this thesis, an efficient algorithm is developed. Assume that the sequence of $N$ images is arranged from left to right, and the first image is numbered $i = 0$, then our algorithm works as follows:

**Step 1:** Find the region of overlap between an image pair. For a pair of images projected on a cylinder that are set apart horizontally by $\theta_i$, we can approximate the amount of overlap $O$ (in pixels) using $O \approx w - \theta_i r$, where $w$ is the width of the image and $r$ is the radius of the cylinder. In the $i^{th}$ image, extract the region with width $O$ from the right end. Denote it $R_{\text{right}}^i$. In the $(i + 1)^{th}$ image, extract the region with width $O$ from the left end. Denote it $R_{\text{left}}^{i+1}$.

**Step 2:** Since RGB is readily available from the image, we compare $R_{\text{right}}^i$ and $R_{\text{left}}^{i+1}$ in terms of their RGB components and spread the difference across the image pair. To do this, we first calculate the sum of the pixel values in each region for the red, green, and blue channels. Then, the difference between the two sums for each channel is calculated.

$$D_k^i = \frac{\sum_{x=0}^{X} \sum_{y=0}^{Y} R_{\text{left},k}^{i+1}(x,y) - \sum_{x=0}^{X} \sum_{y=0}^{Y} R_{\text{right},k}^i(x,y))}{XY} \quad (2.7)$$
Figure 2.8: (a) Exposure and white balance differences among the three images create an unpleasant result. (b) Using our method to balance out the color of the images to achieve a much smoother scene.

where $k = \{R, G, B\}$ represents the color channel, $(x, y)$ is the pixel location over width $X$ and height $Y$. Then for all pixels in the $i^{th}$ image, add $\sigma_{i,k}D_{i,k}^i$ to the corresponding channel $k$. Similarly for all pixels in the next image, add $(\sigma_{i,k} - 1)D_{i,k}^i$. The weight $\sigma_{i,k}$ is initialized to 0.5, but is adjusted in Step 5. This weighting function dictates how much of the difference should be spread to one image, and how much should be spread to the other image.

**Step 3:** What we have done in the previous step is augment the pixels by
addition. Visually, this can lead to a contrast difference between the image pair. We compensate for this by applying a contrast factor, \( f^i \). First, we normalize \( D_k^i \) between 0 and 1, then calculate:

\[
    f^i = \sqrt{1 + (D_R^i + D_G^i + D_B^i)/3}. \tag{2.8}
\]

For all pixels in the \( i^{th} \) image, we then adjust their contrast by \( \sigma_{i,k} f^i \) in channel \( k \). Similarly, the contrast in the next image is adjusted by \( (\sigma_{i,k} - 1) f^i \). Adjusting the contrast of a pixel is achieved by transforming its range to \([-128, 127]\), multiplying by the contrast factor, then transforming the range back to \([0, 255]\), with necessary clip.

**Step 4:** Increment \( i \) and repeat steps 1-3 for the next image pair.

**Step 5:** Repeat steps 1-4 multiple times for all image pairs in the sequence, including the pair of the first and last image. We find that doing it \( N/2 \) times is sufficient. In the first loop, all the weights \( \sigma \) were initialized to 0.5. When we run a new loop, we calculate new weights using \( \sigma_{i,k} \propto D_k^{i+2}/D_k^i \) for each color channel.

The final result of our algorithm is shown in Fig. 2.8 (b), which has significant improvement over Fig. 2.8 (a). For 6 images, it took about 5.5 seconds to run our algorithm on a single-core Samsung Focus S phone. Note that we can use YUV to save the complexity of this method, but the need to convert RGB to the YUV color space and then back to RGB may offset the benefit. Later, in 3.3, we mention that we use YCrCb on videos, because it is already provided by the system.
Even with color balancing, seam lines between neighboring images are still visible due to vignetting, misalignments, radial distortion, and other unmodeled effects. Therefore, blending is required.

The most simple form of blending is linear alpha blending. Images in Windows Phone 7 are in the ARGB format, where A is the alpha or transparency channel.
Chapter 2. Ztitch

Figure 2.10: (a), (b) Original source images to be blended. (c) Using linear alpha blending with a narrow transition width. (d) Using linear blending with a wide transition width. Notice the blurring or "ghosting." (e) The binary mask. (f) Result of multiband blending with 6 levels/bands.
To implement linear blending, we can adjust transparency of each pixel, and achieve smooth transition between images by applying a linear gradient to each image. This gradient becomes transparent as it nears the edge of the photo. We take the region $R_{\text{right}}^i$ and decrease its alpha channel over its width $X$. Similarly, this is applied to $R_{\text{left}}^{i+1}$ in the opposite direction.

Suppose we want to blend two objects from two different images together to create a single hybrid object (Fig. 2.10a,b). We can use linear blending to combine the two (Fig. 2.10b). However, if the transition width is too wide, this can cause blurring or "ghosting" (Fig. 2.10c). A solution is to use the multiband blending developed in [10]. The idea behind this is instead of using a single transition width, a frequency-adaptive width is used by creating a band-pass (Laplacian) pyramid and making the transition width within each level dependent on the level. The low-frequency color variations are blended smoothly over a wide transition width, while the higher frequency textures are blended more quickly over a narrow transition width to avoid blur.

To form a Laplacian pyramid, we first need to form a Gaussian pyramid. To do so, we convolve the original image with a Gaussian (low-pass) and downsample it, essentially creating a smaller, blurred version of the original image. Then, we do this multiple times for each newly formed image, to create each new level of the Gaussian pyramid (Fig. 2.9a). Now, we take the difference (band-pass) between successive levels of the Gaussian pyramid (Fig. 2.9b) to form the Laplacian pyramid.

To create the blended image, each source image is decomposed into its own
Laplacian pyramid. Next, we create a binary mask (Fig. 2.10e) which determines how the two images are to be blended, then create a Gaussian pyramid from this mask. Next, we multiply each band on the Laplacian pyramid with a corresponding level of the Gaussian pyramid. Finally, The sum of the two weighted pyramids are used to construct the final image (Fig. 2.10f).

The result of applying multiband blending to the scene of Fig. 2.8c is shown in Fig. 2.11. If looked carefully, it can be seen that there is no blurring or ghosting compared to linear alpha blending.

### 2.6 Gap Closing

When the focal length of the camera is unknown, Ztitch sets 620 as the default scene radius (an average from several tested devices). If the default value differs too much from the true value, the users might not be able to create a 360° panoramic view, due to a gap or excess overlap between the last image and the first image. On the other hand, sometimes even if the focal length is known, various accumulated errors can still lead to an incorrect panorama. This problem exists in many panorama software, including the mobile Photosynth on iPhone/iPad.

In [3], a technique was proposed to eliminate the gap or overlap for a pure panning panorama, where the inter-frame rotations are constant. First, the amount of misregistration is calculated by taking the difference between the rotation matrices of the first and the last images. Next, this misregistration is converted into an error quaternion, which is then divided by the number
Figure 2.11: (a),(b) Linear blending applied to the boundaries of the scene in Fig. 2.8c. (c),(d) Multiband blending applied to the boundaries of the scene in Fig. 2.8c.
of images in the sequence. This divided value is distributed evenly across the entire sequence. Finally, the error quaternion is converted into a gap angle $\theta_g$, and the focal length is estimated by $f' = f(1 - \theta_g/360^\circ)$. Unfortunately, this technique works only for pure panning motion and when the images are perfectly orthogonal to the ground ($\theta_y = 90^\circ$), which is difficult to achieve in practice.

In Ztitch, we develop an efficient user-aided algorithm to solve this problem. Our solution is more flexible than [3], because our method works even if the first image and the last image have different vertical positions, and when the images do not have constant inter-frame rotations.

Consider a panoramic sequence of $N$ images, where neighboring images are well aligned, but there is a gap or overlap between the first and the last image that has both horizontal and vertical components, as shown in Fig. 2.12 (a). Let $\theta^i_x$, $\theta^i_y$, and $\theta^i_T$ denote the rotations that define the location of the $i$-th image on the sphere with radius $r$, as in Fig. 2.4. We use two angles $\theta_{gx}$ and $\theta_{gy}$ to describe the undesired gap or overlap, i.e., if Point A at $(r, \theta_x(A), \theta_y(A))$ is in the first image, and its matching point B in the last image is at $(r, \theta_x(B), \theta_y(B))$, then the gap or overlap is defined as

$$\theta_{gx} = \theta_x(B) - \theta_x(A),$$
$$\theta_{gy} = \theta_y(B) - \theta_y(A).$$ (2.9)

Suppose the values of the two angles above have been measured by a method, which will be described in the end of this section. In order to eliminate the gap or overlap, we adjust the rotations of all images uniformly, i.e., the updated
orientation of each image will be:

\[
\begin{align*}
\theta_x' &= \theta_x^i + \theta_g x^i / N, \\
\theta_y' &= \theta_y^i + \theta_g y^i / N, \\
\theta_T' &= \theta_T^i - \theta_g y^i / N.
\end{align*}
\] (2.10)

Note that since the vertical position of each image is changed, to maintain the alignment between neighboring images, the last line in Eq. (2.10) is used to slightly adjust the tangent angle of each image. This empirical formula works well if the 360° scene contains sufficient images \((N \geq 10)\), which is usually the case.

We also need to update the radius, for which the following empirical formula is used:

\[
r' = r + C\theta_g x,
\] (2.11)

where \(C\) is a constant dependent on several factors. We found that by setting \(C = 126\), we can get good results even when \(r\) needs to be updated by 90 pixels, which is the worst case (the default radius is 620. The Nokia Lumia 800 has the smallest radius of 530, and the Samsung Focus has the largest radius of 680).

With this method, the gap or overlap can be eliminated satisfactorily while all the images in the sequence would remain relatively aligned with each other.

As described above, the key to the method is to measure \(\theta_g x\) and \(\theta_g y\), the values of the gap or the overlap. This could be achieved by running featured-based alignment algorithms to calculate the amount of misregistration between the first and the last images (Sec. 2.7 describes how to do this). However, this increases the complexity of the algorithm, which means the user has to wait...
Figure 2.12: (a) An example with a gap between the two ends of a panorama sequence. Note that the gap has both horizontal and vertical components. (b) The view directly behind.

During the computation, leading to unpleasant user experience. In addition, these algorithms do not always work.

In Ztitch, a user-aided approach is used, where the user can manually drag one of the boundary images towards the other on the touchscreen, and visually identifies the best alignment position. This operation is intuitive, fast, and accurate because both images are displayed semi-transparently. The dragging can include both horizontal and vertical components; hence our method is more flexible. The required gap/overlap values can thus be easily measured, and all other images and the radius will be jointly updated according to Eq. (2.10) and (2.11).

Fig. 2.13 shows the result after manually closing the gap of the scene in Fig. 2.12. As can be seen, the gap can be satisfactorily closed, and all other images
Chapter 2. Ztitch

2.7 Using and Optimizing SIFT for Ztitch

SIFT (Scale-Invariant Features Transform) matches features across two images even when the images have different orientation and scale. This section covers the implementation of the algorithm, as well as how we use it to optimize it for our application to minimize small alignment errors after the manual or sensor-based automatic stitching. The algorithm can be divided into five components: approximating the Laplacian of Gaussian (LoG), finding keypoints, filtering the keypoints, assigning orientation to each keypoint, and defining a descriptor to each keypoint.

Figure 2.13: (a) Result after correcting the radius $r$ while maintaining the images’ alignments with each other. (b) The view directly behind, after the gap is closed.

remain perfectly aligned.
2.7.1 Approximating the Laplacian of Gaussian

To approximate the Laplacian of Gaussian, we first progressively apply Gaussian blur on the original image. A Gaussian blur is the convolution of the image and the Gaussian operator:

\[ L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \]  

(2.12)

where \( G \) is the Gaussian operator, with \( \sigma \) parameter determining the amount of blur. \( I \) is the input image, and \( L \) is the output blurred image. The Gaussian operator is given by:

\[ G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}} \]  

(2.13)

We blur the original progressively five times. This creates the first "octave" with five "scales." Next, we reduce the original image to half the size by down sampling, and again progressively create the Gaussian blurred images to generate the next octave. Lowe [15] recommends creating four octaves with five scales, although different numbers may be used, as we will discuss later.

The \( \sigma \) parameter determining the amount of blurring is chosen specifically. For each subsequent blurred image in the same octave, \( \sigma \) is multiplied by a constant factor (Lowe uses \( \sqrt{2} \), as do we). Table 2.1 shows popular chosen \( \sigma \) values. The first image is blurred with \( \sigma = \sqrt{2}/2 \).

After generating the set of blurred images, we use it to generate another set of images that are the approximation of the Laplacian of Gaussian. The problem
with computing the actual Laplacian of Gaussian is that it is computationally expensive. To generate the approximation, we can simply subtract each pair of consecutive scales for all the octaves, and produce a set images known as the Difference of Gaussian (DoG).

In addition to being computationally cheap, computing the DoG also has the advantage that it is scale-invariant. The LoG is not scale-invariant because the Gaussian Operator (Eq. 2.13), has a scale parameter $\sigma^2$ in the denominator. Since the LoG is given by $\Delta^2 G$, a scale-invariant version must then be $\sigma^2 \Delta^2 G$ in order to cancel out the scale in the denominator. Fortunately, the DoG provides a close approximation to the scale-normalized LoG. Lowe [15] showed that when the scales $\sigma$ differ by a constant factor, then it incorporates $\sigma^2$ scale normalization needed for scale-invariant Laplacian.

In Ztitch, if we assume that all the images in the same scene were taken using the same camera under the same zoom level, then scale-invariance is not a necessary. As a result, we do not need to downsample the original into more octaves, and simply generate three DoG images for the first octave (Fig. 2.14). This drastically reduces the runtime compared to running the original SIFT with four octaves and five scales.
2.7.2 Finding the Keypoints

The DoG images generated from the previous steps are useful for finding keypoints. Keypoints can first be roughly located by detecting any maxima or minima in a local region. We check each pixel and all its neighbors, including pixels from the same local region in the scales above and below. If the pixel is greater than or less than all 26 neighbor pixels, then it is predetermined as a candidate keypoint. Fig. 2.15 summarizes this process. Since there are 3 DoG
images in the octave, this rough keypoint detection process is done only once, to produce one extrema image.

The keypoints found are only candidates; they are not the actual keypoints that we want because the real maxima/minima usually lie between these keypoints. Therefore, the next step is to interpolate the detected keypoints to approximate the real location of the maxima/minima in the image. For this approach, Brown [18] suggests the Taylor expansion of the scale-space function \( D(x, y, \sigma) \). After solving, we have the approximate locations of the actual extrema.

2.7.3 Filtering the Keypoints

Some of the keypoints detected in the previous step may have low contrast and need to be eliminated. Simply set a minimum contrast threshold to determine if a keypoint should be rejected. Furthermore, the DoG tends to have a strong response along edges, generating unwanted keypoints along edges that also need to be rejected. Fig. 2.16 shows the original image and the unfiltered keypoints from one octave, running on the Windows Phone 7 emulator. The building in this image is a famous cathedral in Florence, which contains many edges and small details, where most of the SIFT keypoints are detected.
Figure 2.16: The SIFT keypoints found (unfiltered) from an image on the WP7 emulator. They are usually located at edges.

If we compute two gradients (principle curvatures), perpendicular to each other, based on the image around the keypoint, then it is known that the keypoint is a corner if both gradients are large. Conversely, it is an unwanted edge if one gradient is large and the other small. This process is achieved efficiently using a 2x2 Hessian matrix, \( H \) at the location and scale of the keypoint.

\[
H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}, \tag{2.14}
\]

The eigenvalues of the Hessian matrix are proportional to the principle curvatures. Therefore, only the ratio of the two eigenvalues is needed to determine if the keypoint is an edge or a corner, and explicitly calculating the eigenvalues
can be avoided. It is shown in [20] that we only need to compute the trace and determinant of $H$ and check:

$$\frac{Tr(H)^2}{Det(H)} < \frac{(r + 1)^2}{r}$$

(2.15)

where $r$ is the ratio of the largest eigenvalue over the smaller eigenvalue. In practice, we set $r = 10$, rejecting keypoints that have ratio between principle curvatures greater than 10.

### 2.7.4 Assigning Orientation to Keypoints

The keypoints produced thus far are not rotation-invariant; meaning that they cannot be used to match common features across an image if one of the images is rotated relative to the other. Therefore, the next step is to assign an orientation to each keypoint. The idea is very simple: compute the gradients around each keypoint, and collect their directions and magnitude. Then assign an orientation to the keypoint based on the most significant directions.

First, take the Gaussian blurred image $L$ at the same scale as the keypoint, and use pixel differences to find the magnitude at the sample point $L(x, y)$:

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - L(x, y-1))^2}$$

(2.16)

Again we use pixel differences to find the orientation at the sample point $L(x, y)$:

$$\theta(x, y) = \tan^{-1}((L(x, y+1) - L(x, y-1))/(L(x+1, y) - L(-1x, y)))$$

(2.17)
The region around the keypoint’s location is used to compute the gradient magnitudes and orientations. After computing these, the gradients are further weighted by a circular Gaussian-window whose kernel is equal to the keypoint’s scale times 1.5 ($\sigma \times 1.5$). As a result, the magnitude of each sample is proportional to the magnitude of that sample point and the Gaussian-weighted window.

Next, each gradient orientation is placed into a histogram with 36 bins covering the $360^\circ$ range, and the amount added into the bin is proportional to the weighted magnitude. Doing this for all the sample points, there will be a peak in one of the 36 bins. The orientation of the keypoint is assigned by that particular bin. Furthermore, any other peaks in the histogram that are greater than 80% of the highest peak is assigned into a new keypoint, whose location and scale are the same as the original keypoint.

2.7.5 Generating Descriptors

So far, image location, and orientation have been assigned to each keypoint. The final step is to assign a descriptor that acts like a fingerprint to uniquely identify one keypoint from any other keypoint. The descriptor must be distinctive, but ideally must still be recognizable across two images even if they have different illumination or viewpoint.

First, a 16x16 window is formed over the keypoint location. The 16x16 windows are divided into 4x4 windows (each one containing a 4x4 block). Inside each 4x4 window, the gradient magnitudes and orientations are computed like before, and they are further weighted by a Gaussian weighting function whose
kernel is equal to 0.5 times the width of the descriptor window (the purpose of
applying this Gaussian window is to give less emphasis to gradients far from the
center of the region, and also remove some small sudden changes). Then, the
orientations are placed into a histogram with 8 bins covering the 360° range.
As such, each bin can be visualized as an arrow whose direction and length
represent the orientation and magnitude, respectively. The arrow can be one
of eight directions because there are eight bins, the amount that Lowe found
experimentally to yield the most keypoint matches.

Doing this for all sixteen 4x4 blocks, each block yielding 8 bins, we end up with
4x4x8 = 128 elements. Then, these 128 elements are normalized, have their
magnitudes capped at 0.2, and normalized again. Capping the magnitude can
help achieve illumination invariance, and the 0.2 value was found experimentally.
Finally, the modified 128 vectors form a 128-element feature vector for
the keypoint, which is the desired descriptor.

2.7.6 Application to Ztitch

In Ztitch, running SIFT over the entire image (Fig. 2.17(a),(b)) is unnecessary,
since the images have already been coarsely aligned using the methods afore-
mentioned in this chapter. Therefore, we only need to find the SIFT features in
subregions at the overlapping areas. The obvious advantage of this is reduced
computation time because less keypoints are needed to be found, but also for
each keypoint, the number of candidate keypoint matches in the second image
are reduced, leading to increased reliability as false matches are reduced. For
two overlapping images that are spaced apart horizontally by $\theta_x$, we can approximate the overlapping region width $O$ (in pixels) using $O \approx w - \theta_x r$, where $w$ is the width of the image and $r$ is the radius of the cylinder.

### 2.7.6.1 Keypoint Matching and Image Realigning

To find the best match for each keypoint, we find the keypoint from the other image with minimum Euclidean distance for the 128-element descriptor (Sec. 2.7.5):

$$||q - p|| = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots + (q_{128} - p_{128})^2}$$  \hspace{1cm} (2.18)

where $q$ and $p$ are the two descriptors being tested for best match, and $q_n, p_n$ the magnitude of the $n^{th}$ element. For the purpose of our experiment, we do a simple exhaustive search to find the best candidate keypoint whose Euclidean distance is minimal, and the ratio of this distance compared to the next-closest neighbor is less than 0.8.

Even when performing keypoint matching in only the overlapping regions, false matches still exist, as shown in Fig. 2.17(c),(d). In our experiment, we use a voting scheme to discard these outliers. Denoting $(x_k, y_k)$ as the pixel location of the keypoint that has been matched, we know its corresponding angular position in the 3D coordinate system in which the image is projected using the approximations:

$$\theta_{k_x} \approx \theta_x + (x_k - (w/2))/r$$  \hspace{1cm} (2.19)
\[ \theta_{by} \approx \theta_y + \frac{y_k - (h/2)}{r} \]  

(2.20)

where \( \theta_x \) and \( \theta_y \) are the angles used to project the image on the sphere with radius \( r \) (the notations as used in Sec. 2.3.1), \( w \) is the width of the image, and \( h \) is the height of the image. Eq. 2.19 and 2.20 are approximations, because in our system the images remain flat rather than curved; the keypoint location \((x_k, y_k)\) does not lie on the sphere. However, they have worked well in our experiments because the alignment correction is always assumed to be small.

After computing all the angular position difference for all keypoint matches, we compute the averages. Any pair of matches which has an angular position difference greater than the average by 20% is discarded, and the average is re-computed. Finally, we correct the alignment using the averaged angular difference. Fig. 2.18 shows the result after using our technique to successfully correct the small alignment error in Ztitch.

Compared to running the original SIFT algorithm, our modifications as shown in this section reduces the runtime by about 81%, making it more suitable to run on a mobile device. However, it still takes more than 2 seconds on average to match a pair of images. We believe that further optimization can be achieved by running SIFT over a window within the overlapping region that exhibits high frequency, ignoring the low frequency such as an empty sky, wall, or ground that do not contain many features. The most time-consuming stage is the Gaussian blurring, so reducing the image over which to run SIFT is the key to reducing computation time.

Up until now, we assume that sufficient overlapping regions exist between the images, but situations may arise where there is not enough or no overlap. A
Figure 2.17: (a) Detected features from the full-sized images. The different sized circles correspond to different scales: small circles were detected at a fine scale. (b) Nearest neighbor matching the full-sized images. (c) Detected features from the overlapping regions at only the fine scale. (d) Nearest neighbor matching only in the overlapping region, with many of the false matches eliminated.
Figure 2.18: (a) Small alignment error between the two images within Ztitch (b) The scene after alignment correction.
method for aligning non-overlapping images was proposed in [28] by extrapolating the image beyond the boundaries. There are two stages involved in doing this, a single scale extrapolation, and a multi-scale extrapolation.

Single scale extrapolation is a patch based technique shown in Fig. 2.19. Let $A$ be a rectangular region centered at the left edge of an image. The left side is outside of the image and the right side is within the image. We can extrapolate the left side of $A$ by looking for a rectangular region $B$ whose right side is the most similar to the right side of $A$, and then copy over the pixels. The search for region $B$ is done over all images in the collection, not only within the same image. Patch similarity is based on the differences of the pixel’s color components.

The single scale extrapolation is good only for extrapolating some small amount of $k$ pixels in each direction (the authors use $k = 5$). Intuitively, high frequencies in the extrapolated regions should diminish as we move further away from the original image, so a multi-scale approach is adopted. First, we create an upsampled Gaussian image above the original image. Then, we take the extrapolated region with width $k$ from the single scale extrapolation step, stretch

**Figure 2.19:** (a) Single scale extrapolation in which a rectangular region $A$ is imposed at the left edge of the photo, and search for a rectangular region $B$ whose right side is most similar to that of $A$. (b) The extrapolated image after copying the pixels from the left side of $B$ to $A$. 
it, and attach it around the upsampled Gaussian image (the extrapolated strips preserve their original width of $k$). Now, we repeat the previous extrapolation step, and extrapolate $k$ pixels around this higher resolution image again. We repeat this process for $l$ levels above the original image until a desired width $k \times 2^l$ is extrapolated.

Results from [28] show that this method works fine for $k = 5$ and $l = 2$. After this, the images can be aligned as usual.
3.1 Background on Multiview Videos

Multiview videos are encountered in many traditional applications such as video surveillance. Recently, it also emerges as a new social media format, as many user-generated videos are available online, and many of them are taken at the same place, but with different camera orientations. Therefore, the means of effectively organizing and presenting these videos has become an important topic.

The work involved in this thesis aims to contribute to the field of computer vision and visualization. It seeks to provide an efficient approach for implementing an interface that renders reconstructed scenes from videos. Ultimately, the system will provide visual context for videos, enabling videos to be placed in a three dimensional space according to their actual geometric position of the capture location. Furthermore, browsing and searching for videos become
simplified, as videos captured in the same region can be grouped together based on their location.

The work in this section is the foundation and gateway to many research topics. For instance, multiple view videos can aid in security surveillance systems. In the traditional scenario, an observer has to watch many screens arranged like a matrix, but if powered by the proposed technology, one may only need to take care of one big screen displaying live mosaic video. If one wants to change the orientation of a camera or zoom in/out, the geometrical organization of the live mosaic video will be changed correspondingly. In short, it gives people a more intuitive way to watch what happens in an important place. In addition, this technology may be integrated with the ongoing research on object tracking in videos, so that when an object on the screen is selected, that object can be tracked across all the screens that have view of the same object.

Algorithms already exist which take common features between images, then compute precisely where the images were taken (not limited to only panoramic scenes, as in Ztitch). One particularly powerful algorithm of this nature is called Bundler. More details of this algorithm can be found in the paper [8] by N. Snavely. This thesis will assume that the required 3D affine matrix is already given. Therefore, the focus of this thesis is on the application of the computed affine matrix to transform videos into three dimensional projections. Imagine that a user captures several videos in the same environment. Then, the system will simultaneously display all the videos, each oriented accordingly in the 3D space, as demonstrated in Fig. 3.1.
3.2 Bundler - Structure from Motion

Perhaps one of the latest software that extensively utilizes computer vision to solve the problem of scene reconstruction is Microsoft’s Photosynth. Our system leverages the same algorithm used by Photosynth to automatically place the videos in their corresponding locations in 3D space. However, one important advantage of our system over Photosynth is that Photosynth can only deal with images, whereas our system works with videos, which is therefore much more powerful. This algorithm is available as a software package called Bundler [7] which was developed by Dr. Noah Snavely, and it uses the established Scale-Invariant Features Transform (SIFT) [15] by David Lowe to extract features
within an image.

The primary goal of Snavely’s project is computing the parameters of the camera that captured the image. Snavely proposes a method to compute a 3x4 matrix which describes the orientation as well as its position (translation) relative to another camera within the same scene. Other parameters of the camera are extracted as well, such as the focal length (zoom), and the sensor size (CCD width) which can affect how incoming light is projected onto the film. These parameters are extracted directly from the EXIF tag of the images. As one can see, Snavely’s method is precise and detailed, taking into account all the variables that matter.

Bundler will also output a set of points in the 3D space, also known as "points cloud". These points are used primarily for the structure from motion system, which is the process for finding the 3D structure of an object, and are not used for the purpose of our system. Only the camera parameters are of interest to this thesis. In general, the steps involved, as described in the Bundler homepage, are:

1. Supply a set of input images

2. Compute correspondences:
   - Detect features in each image.
   - Match key-points between each pair of images.
   - For each pair, estimate the fundamental matrix $F$ and refine matches, chain pair-wise matches into tracks.

3. Structure-from-motion:
• Select good initial pair of images to seed reconstruction.

• Add new images to reconstruction and triangulate new points.

• Bundle adjust

4. Output camera and scene reconstruction

The system collects a dataset of images of the three dimensional space to be re-modeled. Bundler not only needs the images to extract the features from, but it also needs the EXIF tag containing the focal length information, just like in Ztitch. Since the system of this thesis uses videos instead of images, we are presented with an obvious problem the user must supply the focal length information manually, because video files do not contain the focal length information as it is a dynamic parameter for video sequences.

In the next stage, sets of matching 2D pixels among the set of images are found, and each set of matching pixels across multiple images should correspond to a single point in the 3D space. SIFT features are extracted from the images, and a points cloud is created. A points cloud is a set of points on a two dimensional plane which are extracted features of the image, such as corners.

The desired output is the fundamental matrix $F$ which relates the points between two images - if $x$ is a point in one image and $x'$ a point in another image, then $x'Fx = 0$. Colour is another important variable but is subject to gross variations due to lighting. The Photosynth system is invariant to lighting, but at the same time, must recognize the different colors inside a set of images to aid in the matching process a very challenging problem. It needs to learn to recognize feature that are important to the job at hand. This can be
achieved through a combination of local self-organization combined with feedback from subsequent and former processing layers (temporal distance learning is an interesting topic). Bayesian theory is probably used to guide the algorithm development.

Two overlapping photos are selected, and matching sets of features within the different photos are examined. The photos would have been taken from different camera positions with different lenses. To make this achievable, the system focuses on images that were taken from camera that have similar characteristics to constrain the problem greatly.

Assuming that we have matching points in two photos, we can work out where the camera was. Then, the problem becomes an optimization problem: comparing between two images, find the exact position in the X-Y-Z coordinate in which the image belongs, and taking into considerations scale and rotation. Keep trying different positions until the error is minimized. The two photos can now be related in 3D space, and we can setup a virtual camera somewhere else and create a new photo of the scene. Once we repeat this with all of the photos we can reconstruct the scene. It should be emphasized that the Photosynth is not a photo stitcher, as not only are the calculations are much more involved, but the results are entirely different. However, just like Ztitch, the images are being rendered in real-time.

In conclusion, we use Bundler to get camera parameters. The code is openly available at Snavely’s website (please see Appendix B), and is useful for:

- F-matrix estimation
- Calibrated 5-point relative pose
• Triangulation of multiple rays

For the scope of this thesis, we assume that the captured video remain stationary throughout its entire sequence. For future works, we will tackle the problem of weaving together dynamically moving videos. This will require computations that are much faster than the one presented above, since a video sequence may contain many frames. For now, we have implemented the FAST-9 feature detection, discussed in the next section, which works in near real-time on the Windows Phone 7. Next, we will need to utilize these detected features for real-time visual tracking, and one promising solution is the Parallel Tracking and Mapping (PTAM) [22], originally designed for augmented reality applications.

### 3.3 Real-Time Visual Tracking

Visual tracking in real-time requires extremely fast feature detection, preferably one that can be done between frames. SIFT and Harris Corner detectors are too slow. From our knowledge, the fastest, yet relatively robust feature detector most suitable this regard is the Features from Accelerated Segment Test (FAST) corner detector [29].

The FAST algorithm is illustrated in Fig. 3.2. We take the 16 surrounding pixels around a candidate point $p$, and test to see if an $n$-contiguous segment of pixels, all brighter (or dimmer) than $p$ by a threshold, is formed. In the figure, $n = 9$ is used, although $n$ can range from 9 to 16. Tests have shown that $n = 9$ gives the best performance with highest repeatability for a variety of different
Chapter 3. *Stitching of Multiview Videos*

Figure 3.2: The FAST algorithm: taking the 16 surrounding pixels around the candidate point $p$, and testing whether at least $n$ of these pixels’ intensity are all brighter (or dimmer) than $p$ by a threshold, such that they form a contiguous segment.


conditions such as affine and noise variations, and is also the lowest for which edges are not detected.

We implemented the FAST corner detector on the Windows Phone 7 using $n = 9$ (a.k.a. FAST-9) to establish whether the technique is applicable for mobile devices for real-time visual tracking. Source code of this project is available in [1]. We take several steps to reduce the computation time on the phone:

1. Instead of the ARGB color space, we use the YCrCb color space. The Silverlight platform allows us to grab the YCrCb information of the live camera feed into a buffer, and we neglect the Cr and Cb data, using only the Y luminance data to perform the FAST detection. The luminance is provided as bytes (0 to 255), which is convenient for the FAST algorithm. If we use the ARGB data, we have to first extract the individual RGB components, multiply each color channel by specific weights, then sum them up. As a result, compared to using ARGB data, our method was tested to be approximately 4 times faster.
2. Although the new Windows Phone devices have dual-cores, all our test devices have only a single-core, so multithreading is applied. We break the processing into two threads: one thread for handling the user interface (UI) for handling such tasks as updating the live camera feed on the screen, and another thread for performing the image processing. More specifically, the live camera feed is drawn on the primary UI thread, and a second thread is created that runs in a while loop which handles the task of gathering the YCrCb data from the live camera feed, computing the FAST features, and rendering the results. The threading works like this: before acquiring the YCrCb data, we block the current thread until we receive a signal that the acquisition is complete. Once complete, we block both threads, but before doing so, we send an asynchronous signal to compute FAST, to render the results, and to unblock both threads after those tasks are done. Doing it this way, the two threads are synchronized and we can take advantage of unused computing resources to execute the FAST detection.

3. The method can be accelerated by testing pixel #1 and #9 first, then pixel #5 and #13 (see Fig. 3.2), and rapidly rejecting the point \( p \) as a candidate if necessary.

The total time for computing FAST on a single 640x480 frame is approximately 150ms, and as long as the camera does not move too rapidly, it is sufficient. However, we hope to improve this timing more in the future. Fig. 3.3 is a screenshot of the application running on the phone, with the detected FAST corners displayed as red dots in a smaller window in the top-left corner. After features have been detected, we can use monocular Simultaneous Localization
Figure 3.3: Screenshot of the FAST-9 detector on the phone, with results shown as red dots displayed in the smaller top-left window. It took approximately 150ms to perform FAST on each frame.

and Mapping (SLAM) [30][31], or Parallel Tracking and Mapping (PTAM) [22] to trace the movement of the camera, although the latter method is more convincing and better suited for our application. This will be left as future work.

3.4 Improving the First-Frame Matches

As previously discussed, Veaver uses SIFT to try and match the first frames of the video collection. When the images have small affine (viewpoint) changes, then the use of SIFT is fine. Otherwise, SIFT will be unable to find correct matches to register the image correctly with Bundler when there are large affine changes, an important parameter to consider in scene reconstruction. Although SIFT was published in 1999, and that a number of SIFT variants and extensions such as PCA-SIFT and GLOH (Gradient Location-Orientation Histogram) have since been proposed, it was pointed out by [23] that no substantial improvement of the original SIFT method can ever be hoped, especially regarding translation, rotation, and scale-invariance. With regards to large affine changes, it was found
[26] that all feature descriptor methods suffer from poor performance, but with SIFT being more robust compared to PCA-SIFT and the recently developed and popular SURF (Speeded-Up Robust Features) [27].

How, then, can we accommodate videos whose first frames exhibit large affine changes? Recently, a very successful modification to SIFT named affine-SIFT (ASIFT) was proposed in [24] which has been mathematically proven to be fully affine-invariant, and has been shown to yield excellent results under large affine variations. The downside of this approach is increased computation complexity; approximately 13.5x more computationally expensive for the feature detection and matching stage. In this thesis, we are not concerned with the speed of first-frame matching of Veaver. If speed is a concern, then SURF should first be chosen over SIFT.

Furthermore, we can also take advantage of local symmetries found commonly in natural and man-made scenes to improve our matching performance. This concept was recently explored in [25]. This section takes a look at these two methods as part of future works for Veaver.

3.4.1 Making SIFT Affine-Invariant

The idea behind ASIFT, a fully affine-invariant SIFT method is simple: simulate a set of affine transformations on both images to be matched, then apply SIFT. Each affine transformation is defined by two parameters, \( \phi \) the longitude angle, and \( \theta \) the latitude angle. The paper [24] suggests a set of \( \phi \) and \( \theta \) for which to apply the affine transformations, based on their empirical findings. \( \theta \) is sampled following a geometric series \([1, a, a^2, ..., a^n]\) with \( a = \sqrt{2} \), and \( \phi \) is
Figure 3.4: Sampling of the parameters \( \theta \) and \( \phi \) can be visualized with a hemisphere. The red dot is one particular sample.

sampled following an arithmetic series \([0, b/t, \ldots, kb/t]\) with \( b = 72^\circ \). Values for \( t \) is shown in Fig. 3.4 and \( k \) is the last integer such that \( kb/t < 180^\circ \).

ASIFT works because any geometric tilt can be reverted by simulating the same amount of tilt in the orthogonal direction. The price to pay is a zoomed-out scale change. In other words, ASIFT is fully affine invariant, up to sampling errors.

With respect to computational complexity, using the above proposed sampling parameters, the feature computation is approximately 13.5x that of SIFT. Thus, the complexity of feature matching is \( 13.5^2 \approx 180x \) that of SIFT. In practice, feature computation dominates the feature matching stage if the database is not too large, so total complexity is about 13.5x that of SIFT.

### 3.4.2 Using Local Symmetry Features

Symmetry is a powerful feature in the structure of our world, evident in many natural scenes and architectures. In computer vision, we can take advantage of symmetries to define a new set of features that are robust and stable; when
Although the different pictures of the Eiffel Tower have varying quality and appearance, they all preserve the local symmetries of the tower. Considered at all locations and scale, they are also quite descriptive [25]. For instance, observe the images in Fig. 3.5. Even though there are drastic changes in the image quality, it is clear that these images depict the eifel tower. Observe that many of the local symmetries are preserved across the images, like the vertical bilateral symmetry of the entire tower or the smaller symmetries within the structure.

For a given location \((x, y)\) and scale space \(s\), we want to know if a window of size \(\lambda \ast s\) exhibits rotation or bilateral symmetry. If we consider 1D slices through the image, we can understand these two symmetry types can be intuitively. Detecting symmetries can be posed as determining whether these slices produce approximately even functions. So we can check for symmetry in the image by applying the two types of slices, as shown in Fig. 3.6.
We need a local symmetry scoring system to determine how much a window of particular size within the image exhibits symmetry

\[ SD(p) = \sum_{p'} w(||p' - p||)d(p', g_{s,p}(p')) \]  \hspace{1cm} (3.1)

Ideally, the score would have a strong response to symmetric regions but robust to small asymmetric regions. To compute this score for a given window, we need 3 components. First, we need a remapping function \( g_{s,p}(x) \) that computes the symmetrically opposing point to \( x \) for a symmetry type \( s \). The type can be rotational, horizontal, or vertical. Second, we need a pairwise distance function \( d(p_1, p_2) \) that compares the intensities of the image at two locations. Third, we need a weight mask that gives the importance of each set of point pairs. If we are interested in global symmetries, the weight mask would be 1 everywhere, and to detect local symmetries, we could use a circular Gaussian mask giving more importance to points close to \( p \). We then use (3.1) by varying \( p \) over all pixel locations.

The symmetry scores can then be used to detect local features at different
scales. To do this, vary image size but fix the window size. Local maxima of
the score with respect to scale and location are used as the interest points, and
at these interest points, we compute the $SD$ score.

Next, a simple feature descriptor based on the local symmetry score is devised.
We impose a log polar grid on the image, centered at the interest point. For
each symmetry type and each cell, store the maximum value of the symmetry
score within that cell, then concatenate the results into one final descriptor.
The author of the method uses 20 angular cells with 4 radial cells, and there
are 3 symmetry types (horizontal, vertical, and rotational), so the resulting
descriptor has a total of 240 elements.

In evaluating the proposed detection method, the authors found that detect-
ing local symmetries had higher repeatability than detecting SIFT features,
but in evaluating the descriptors, SIFT descriptors actually outperformed the
proposed descriptors in most cases. However, since both descriptors may pro-
vide complementary information, we can combine local symmetry features with
SIFT features together by simply concatenating the descriptors. The newly
formed descriptor, called SIFT-SYM, was shown to outperform either descrip-
tor alone. The symmetry descriptor performed particularly well when matching
image pairs that have drastic illumination changes. Therefore, the use of local
symmetry features is a promising aspect in increasing the performance of the
first-frame matching in Veaver.
Chapter 4

Conclusion

In Chapter 2 of this thesis, we presented an algorithm that can balance large color contrasts between neighboring images of a photo panorama, as well as two different techniques of blending to eliminate any remaining seam line. The motivation of behind this is to try and improve the visual results of the 3D panorama application, Ztitch, that we developed for the Windows Phone 7. Linear alpha blending is currently the method employed in Ztitch, but this thesis also evaluated the more advanced multiband blending. According to our results, multiband blending is indeed better in some cases where blurring or ghosting occurs. However, it must be noted that the difference of the two results is small in many cases, and probably unnoticed by the average user when viewing the scene from the small screen of a mobile phone. Then, we optimized SIFT to drastically reduce its computational complexity assuming that the images are ordered and have already been coarsely aligned, and discussed a simple method to make alignment possible even if there is little or no overlap between the neighboring images.
Chapter 3 presents the principles and implementation of a multiview video system, *Veaver* that parallels some of the ongoing research in the field of computer vision. We studied ways in which a set of videos from the same scene can be positioned and oriented in a 3D space (not limited to a panorama setting), by using SIFT and bundle adjustment on the first frame of each video. Next, we implemented the FAST-9 detector along with steps that reduce computation time, to find keypoints in real-time. These keypoints, if leveraged by a tracking and mapping system such as SLAM or PTAM, allows us to locally reposition the videos for the subsequent frames. The tracking and mapping system is left as a future work, along with the implementation of Affine-Invariant SIFT, which is required to drastically improve the performance of the first frame matching, since the original SIFT performs poorly when large affine changes are present. Finally, as future work, we will explore the opportunity to offload the stitching process to the cloud.

Our technology is promising in changing the way videos are browsed and displayed, enabling a more dynamic and interactive user experience for viewing multiple videos that captured the same scenery. The system can aid in scenarios such as real-time multiview video conference and video surveillance.
Bibliography


Appendix A

Microsoft Silverlight

Microsoft Silverlight is a single runtime environment and a framework for building rich web applications that support multimedia, computer graphics, animation, and interactivity. It was initially released as a plug-in for web browsers for video streaming, but subsequent versions of Silverlight brought more and more features for developers. Like the widely popular Adobe Flash runtime, Silverlight is compatible with most web browsers on both Windows and Mac OSX operating systems.

In Silverlight applications, the user interface controls are declared in the Extensible Application Markup Language (XAML) and programmed using a subset of the .NET Framework. In our multiview video system, default controls such as the play, pause, and mute buttons will be added in XAML, as well as other controls such as the video seekbar slider, and the canvas screen which contains the viewport. The videos themselves must be added programmatically to the
viewport during runtime, in the code behind, as these are variables to the system. The desired outcome of the multiview video system is a standalone media player, where users provide their own multiview video files.

The Silverlight platform has been chosen as the target development platform by my team (headed by Dr. Jie Liang). Rendering a scene of multiple-view videos, even if small-scaled, is extremely resource intensive, but prototypes developed on the Silverlight platform has so far delivered promising performance.

In order to implement scene reconstruction, we must first build a 3D engine framework. As of the time of this writing, the latest version 4 of Silverlight is provided with only the minimal APIs for modeling 3D objects. The Matrix3D structure, which houses the 4x4 matrix, is also very minimalistic. Therefore, the work from previous chapter must be incorporated. Rene Schulte [17] has compiled together a great library extension for Silverlight’s Matrix 3D structure that can be used to aid our development of the system. The language that will be used is C#. One note-worthy feature of using Silverlight with C# and the Microsoft Common Language Runtime (CLR) is the great multi-threaded programming options.

Adobe Flash is another option for building the proposed system of this thesis, as it also supports 3D rendering. Flash uses a different language, called ActionScript (also object-oriented programming), but since the principles of the graphics pipeline hold true across all platforms, it is possible to build the same system using Flash.
Appendix B

Bundler: Structure from Motion

Bundler: Structure from Motion for Unordered Image Collections Bundler is a structure-from-motion system for unordered image collections (for instance, images from the Internet) written in C and C++. An earlier version of this system was used in the Photo Tourism project.

Bundler takes a set of images, image features, and image matches as input, and produces a 3D reconstruction of camera and (sparse) scene geometry as output. The system reconstructs the scene incrementally, a few images at a time, using a modified version of the Sparse Bundle Adjustment package of Lourakis and Argyros as the underlying optimization engine. Bundler has been successfully run on many Internet photo collections, as well as more structured collections.

The Bundler source distribution also contains potentially useful implementations of several computer vision algorithms, including:

- F-matrix estimation
• Calibrated 5-point relative pose

• Triangulation of multiple rays

To reiterate, Bundler produces sparse point clouds. For denser points, Dr. Yasutaka Furukawa has written a beautiful software package called PMVS2 for running dense multi-view stereo. A typical pipeline is to run Bundler to get camera parameters, use the provided Bundle2PMVS program to convert the results into PMVS2 input, then run PMVS2. You might also be interested in Dr. Furukawa’s CMVS view clustering software, which is a helpful preprocess to running PMVS2.

Source URL: http://phototour.cs.washington.edu/bundler/
Appendix C

Transformation Matrices

Viewing

$T_v$ transforms the world coordinates to the viewing coordinates, and when the
lookat point is at the origin, the viewing is defined as

$$
T_v = \begin{bmatrix}
-\sin \theta & -\cos \theta \cos \phi & -\cos \theta \sin \phi & 0 \\
\cos \theta & -\sin \theta \cos \phi & -\sin \theta \sin \phi & 0 \\
0 & \sin \phi & -\cos \phi & 0 \\
0 & 0 & r & 1
\end{bmatrix}, \quad (C.1)
$$

Orthographic Projection

An orthographic projection essentially flattens the 3D space to the direction
perpendicular to the view plane. In computer graphics, the orthographic pro-
jection matrix is set by defining the clipping volume constrained by six bounding
planes: top, bottom, left, right, near, and far. The 3D clipping volume is then
transformed to the canonical view volume, which is a 2x2x2 cube at the origin.
For orthogonal views, the transformation matrix that maps to the canonical volume is defined as

\[
C_o = \begin{bmatrix}
\frac{2}{\text{right-left}} & 0 & 0 & -\frac{\text{right-left}}{\text{right-left}} \\
0 & \frac{2}{\text{top-bottom}} & 0 & -\frac{\text{top-bottom}}{\text{top-bottom}} \\
0 & 0 & \frac{2}{\text{far-near}} & -\frac{\text{far-near}}{\text{far-near}} \\
0 & 0 & 0 & 1
\end{bmatrix}, \quad (C.2)
\]

The view is translated via the last column, which can be divided out as a translation matrix

\[
\begin{bmatrix}
1 & 0 & 0 & -\frac{\text{left}+\text{right}}{2} \\
0 & 1 & 0 & -\frac{\text{top}+\text{bottom}}{2} \\
0 & 0 & 1 & -\frac{\text{far}+\text{near}}{2} \\
0 & 0 & 0 & 1
\end{bmatrix}, \quad (C.3)
\]

The view is also scaled to the unit cube which is defined by having a minimum corner at (-1, -1, -1) and a maximum corner at (1, 1, 1), via the remaining scaling matrix

\[
\begin{bmatrix}
\frac{2}{\text{right-left}} & 0 & 0 & 0 \\
0 & \frac{2}{\text{top-bottom}} & 0 & 0 \\
0 & 0 & \frac{2}{\text{far-near}} & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}, \quad (C.4)
\]
Appendix C. **Transformation Matrices**

**Perspective Projection**

For perspective views, the matrix, \(C_P\), for mapping to the canonical view is defined as

\[
C_P = \begin{bmatrix}
a & 0 & 0 & 0 \\
0 & b & 0 & 0 \\
0 & 0 & c & cd \\
0 & 0 & 1 & 1 \\
\end{bmatrix}, \tag{C.5}
\]

Where \(w\), \(h\), \(sz\), and \(z\) are defined as follows

\[
h = \frac{1}{\tan(0.5 \times \text{FieldofView})} \tag{C.6}
\]

\[
w = \frac{h}{\text{AspectRatio}} \tag{C.7}
\]

\[
d = \text{far} - \text{near} \tag{C.8}
\]

\[
c = \frac{\text{far}}{d} \tag{C.9}
\]

\[
\frac{cd}{d} = \frac{-\text{near} \times \text{far}}{d} \tag{C.10}
\]
The AspectRatio is the width of the viewport divided by the height, and the FieldofView (FoV) is in radian, and describes how wide the viewing angle is. It is related to the focal length $f$ by the equation

$$\tan\left(\frac{\text{FoV}}{2}\right) = \frac{\text{width}}{2f} \quad (C.11)$$

**Viewport**

Let $x_d$ and $y_d$ be the normalized device coordinates obtained after view transformation and projection transformation. Then the window coordinates $(x_w, y_w)$ are obtained as

$$\begin{pmatrix} x_w \\ y_w \end{pmatrix} = \begin{pmatrix} x_o \\ y_o \end{pmatrix} \begin{pmatrix} (x_d + 1) * \text{width}/2 \\ (y_d + 1) * \text{height}/2 \end{pmatrix} \quad (C.12)$$

The width and height are the window pixel dimensions, $[x_d \ y_d]^T$ is the coordinate of the normalized device vector, $[x_w \ y_w]^T$ is the window vector, and $[x_o \ y_o]^T$ is an offset vector. Since the normalized device coordinates range from -1 to 1 for both x and y, the window coordinates range from $[x_o \ y_o]^T$ to $[x_o + \text{width} \ y_o + \text{height}]^T$. The operation is applied using the homogeneous matrix, $M_V$. 
Appendix C. Transformation Matrices

\[ M_v = \begin{bmatrix}
\frac{\text{width}}{2} & 0 & 0 & \frac{\text{width}}{2} + x_o \\
0 & \frac{\text{height}}{2} & 0 & \frac{\text{height}}{2} + y_o \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}, \quad (C.13) \]