A Real-Time Augmented Reality System Using GPU Acceleration

by

David Chi Chung Tam

Bachelor of Science, Ryerson University, 2008

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2012

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.
I understand that my thesis may be made electronically available to the public.

A Real-Time Augmented Reality System Using GPU Acceleration

Master of Science

Computer Science

Ryerson University

Abstract

Augmented Reality (AR) is the act of overlaying 3D virtual objects into a real-world scene. Using robust computer vision algorithms, it is possible to perform AR using only a single video camera. However, these algorithms are very computationally expensive, and most proposed systems have to sacrifice accuracy for speed.

Graphics Processing Units (GPUs), originally designed to power graphics-intensive 3D video games and now commonplace on most gaming PCs, can also be used for general purpose computations. We developed a computer vision-based AR system accelerated by a single GPU, allowing robust feature detection and matching to be performed in every frame.

We conducted performance evaluations in both indoor and outdoor environments, with parameters optimized for maximum possible accuracy of recovered poses. Our AR system achieves a stable 10-12 frames per second at 640×480 resolution on a laptop.

Acknowledgements

This project was gratefully supported by NSERC Discovery Grant 356072 and the ‘CUTE’ lab from the National University of Singapore.

Contents

List of Tables

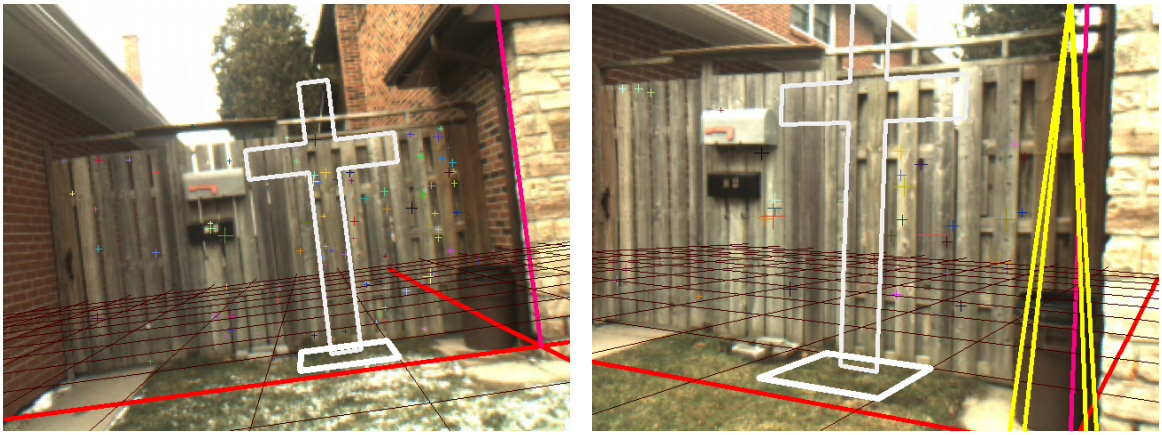
List of Figures

Chapter 1

Introduction

1.1 Augmented Reality

Augmented Reality (AR) combines computer-generated virtual objects together with live imagery in the real world. Unlike Virtual Reality (VR) where its entire environment is computer generated, the AR environment is the real world with computer-generated objects placed into it, to give the illusion that such virtual object exists in real life. For example, in Figure 1.1 a large virtual cross appears as if it were positioned in front of a fence in the real world, even as the camera moves around. AR systems need to be able to localize themselves in the real world in order to augment the virtual objects, and therefore are much more complicated than VR systems.



(a) (b)

Figure 1.1: Example of aligning computer-generated graphics in the real world with two camera angles.

1.2 Applications of AR

A simple example of AR would be a virtual 3D object to appear on top of a marker that moves along with it, when viewed on a mobile device equipped with a video camera.

AR has been demonstrated its potential for outdoor gaming. ARQuake is a modification to a first-person shooting game where the gameplay takes place outdoors in the real world. The user moves around outdoors while wearing an optical see-through head-mounted display (HMD), connected to a backpack computer equipped with a GPS receiver and a digital compass, allowing it to determine the player's location and orientation in the real world. Computer-generated monsters and other objects are rendered on the HMD [2].

AR can be deployed for medical use too. For example, in a biopsy operation, the operators would wear a video see-through HMD, which shows a Magnetic Reasonance (MR) image overlay on top of the subject, with markings to indicate the proper location, angle and depth for needle insertion [3].

AR has also been used during production of movies such as *Avatar*, where computer-generated 3D characters were superimposed on the live actors while they were being filmed in real time, allowing filmmakers to see how the end result would have looked like on the viewfinders of the cameras during shooting [4].

AR techniques have also been proposed to be used to provide guidance for technicians, engineers, or mechanics working on assembly tasks. For example, an automotive service technicians could wear an HMD that displays the names of the parts under the hood, and/or indicate what parts are to be installed or removed.

1.3 Important Components of an Augmented Reality System

Common features of an Augmented Reality system include:

- A view of the real world: the user sees the real scene in front of them - either through a transparent screen at the actual scene (optical see-through AR) [2], or at a computer display that is showing live video from a video camera (video see-through AR). The latter configuration, which this thesis currently uses, is the most widely used. With video see-through AR, the display can either be the display on mobile device (such as a gaming console, cell phone, or tablet) or the eyepiece displays in a helmet-mounted display (HMD).

- Some ways to calculate the equipment's pose, which may be accomplished using various kinds of sensors like accelerometers, compasses, or GPS, or using computer vision algorithms, or a combination thereof.

- 2D or 3D graphics added to the view. Augmenting the real world imagery by projecting virtual objects on top of the real world. For video see-through AR this is by combining the virtual objects with the real video (often simply overlaying the virtual over the real). For optical see-through AR this is simply shown on the see-through visor.

For our system is a video see-through system with a camera that calculates pose using computer vision, and displays 3D augmentations on a screen. Our system uses a single monocular camera because most consumer-grade mobile electronic devices are only equipped with one camera (if any).

1.4 Computer Vision

Computer vision refers to software algorithms that allow computers or electronic devices to extract information from images. By incorporating computer vision techniques, it is possible to compute the camera's pose from the video imagery alone without the use of any sensors, reducing it to a software process as many mobile devices are now equipped with video cameras. For AR systems using computer vision solely to achieve its localization, a typical approach might consist of the following major steps, all of which are computationally expensive:

- Using a feature detector to locate 2D interest points from the image.
- Matching these 2D interest points with 3D points.
- Using pose recovery algorithms to calculate the camera's pose using the matches in the previous step.

In most AR systems, interest point detectors are only used in occasional frames with optical flow tracking performed in order to reduce the computation cost and to allow the system to run in real time. A prominent optical flow tracker is Lucas Kanade (LK) tracker [5], which is designed for fast *estimation*, not accuracy of the optical flow [6]. However, noise is a major problem in optical flow tracking [7], and optic flow is said to produce multiple flow tracks that are often broken [8]. Additionally, whenever the tracked features leave the camera's field of view, matching with them is no longer possible unless feature detection is performed again. The same applies when these features are lost due to occlusion too.

Our AR system utilizes a single monocular RGB camera without the use of any other sensors, operating solely through computer vision. As most of the existing smartphones, tablets, or other handheld electronic devices have only one camera, the results of our single-camera computer vision-based AR can be potentially useful for these applications.

Our system differs from most AR systems in that interest point processing is performed for every frame, made possible with GPU acceleration (Section 1.8.1). This provides a higher level of robust performance compared with other AR systems that only find features occasionally and use simpler algorithms to track from frame to frame.

1.5 Marker-based Systems

Traditionally, Computer Vision-based AR systems rely on markers to calculate poses, such as in Figure 1.2 where virtual objects appear on top of markers.

Examples of marker-based AR systems include ArToolKit [9] and ARTag [1]. ARToolKit markers consist of a square of black and white image surrounded by a black border. As of time of writing, it is currently available online under a GPL license.

ARTag was another marker-based system developed to eliminate the false positives that ARToolKit suffered, by using markers that resemble two-dimensional bar codes. Goblin XNA [10] is a 3D AR (and VR) platform with an emphasis on gaming that incorporate ARTag.

Recently marker-based AR have been employed on commercially released handheld gaming consoles, such as Nintendo 3DS [11] and PlayStation Vita [12]. Both systems employ augmented reality cards using markers similar to ARTag.

Marker-based AR systems are well suited for tabletop usage, where markers can be conveniently placed and virtual objects easily manipulated by moving the markers around.

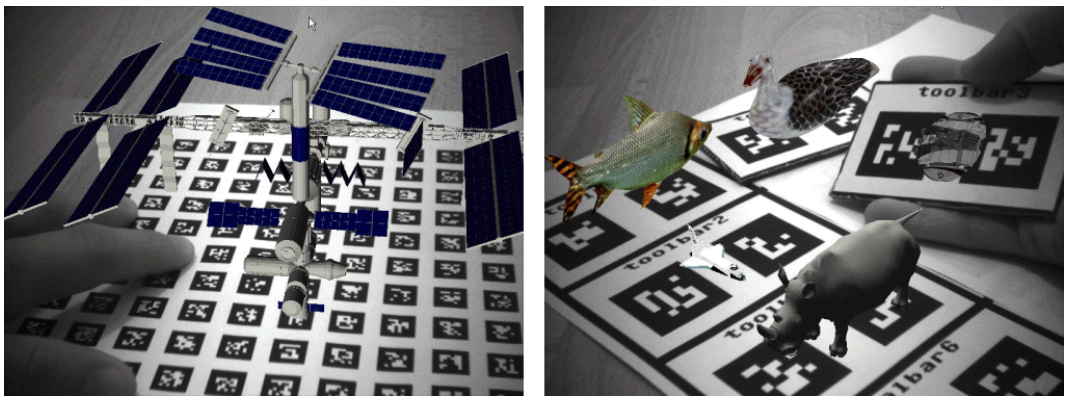


Figure 1.2: Examples of marker-based AR with ARTag [1].

1.6 Markerless Systems

While pose can be reliably determined with markers, mounting markers in the scene on most occasions is not possible or convenient. Ideally we should be able to use the features of objects already occurring in the scene. Achieving markerless AR is much more difficult than marker-based AR, but there have been many different approaches to it. Feature detection is computationally expensive, and this goes particularly to the most robust ones such as SIFT and SURF. A Graphics Processing Unit (GPU) can be used to accelerate most of such operations in order to become useful for real-time AR applications.

1.7 OpenCV

OpenCV [13] is a free and open-source library of many different classes of computer vision algorithms such as corner and edge detection, face detection, optical flow tracking, pose recovery, and others. It also includes many image and video processing functions, such as image resizing, color-grayscale conversion, and image filters such as blur or smoothing. It also has the useful matrix data structure and its manipulation functions.

Performance-minded developers do sometimes rewrite certain OpenCV algorithms when they are not satisfied with such particular function's performance. The way we utilize OpenCV functions in our AR system is detailed in Section 3.2.

1.8 The Graphics Processing Unit (GPU)

A GPU is traditionally used for processing graphical information, especially 3D graphics in games that have very high computational demands, where a very large number of textured polygons have to be rendered in every frame in real time, in high resolutions of 1920×1080 or greater. As newer games are released, they usually demand even higher polygon counts than before, driving up the development of the GPU's processing power. Graphics processing is highly parallel in nature, and GPUs are designed as such.

1.8.1 General Purpose Computing on the GPU (GPGPU)

The regular x86 CPU in each computer contains between 1 to 6 cores, each operating at about 2-3 GHz but heavily depending on the CPU's architecture. In certain Intel CPUs, hyperthreading permits each CPU core to execute two concurrent threads.

In contrast, GPUs are massively parallel processing units capable of thousands of threads executed concurrently, but the exact number of parallel threads varies between different models. One of NVIDIA's later GPU architectures, Fermi [14], has a high emphasis on general purpose computation.

One Fermi microprocessor is capable of executing 48 warps concurrently, where a warp is a group of 32 threads capable of executing a common instruction concurrently. An entry-level NVIDIA mobile GPU, say a GeForce GT 520M with just a single microprocessor, is capable of executing 1,536 threads in parallel [1][15]. On the other hand, the top-end consumer mobile GPU in the same product series, a GeForce GTX 580M with eight microprocessors allows 12,288 concurrent threads to be executed at the same time [2][15]. The more recent NVIDIA GPU architecture, Kepler [16], emphasis is on reduced power consumption which is essential for mobile applications.

Limitations

However, GPUs operate at a much lower clock speed than CPUs, for example, Fermi GPUs run at approximately 1 GHz for the units that perform general purpose computing, and Kepler GPUs run at approximately half of Fermi's clock speeds.

The transfer of memory between the computer's main memory (host memory) and the GPU's dedicated memory (device memory) have limited bandwidth and high latency. This can result in performance bottlenecks in poorly optimized GPU-acceleration applications Therefore, GPGPU programmers must strive to minimize memory transfers between host memory and device memory as much as possible.

GPUs are not ideal for algorithms that require a large amount of execution divergence [15]. As mentioned earlier, each warp consisting of 32 threads can only execute the same instruction concurrently. Whenever the execution flow within a warp branches into two different paths, they have to be executed separately in two sequential steps, resulting in wasted computational capabilities. For example, if the 32 threads in a warp run into an `if / else` statement, N threads get branched into the `if` branch and $32 - N$ threads get branched into the `else`

[1] `http://www.geforce.com/hardware/notebook-gpus/geforce-gt-520m/specifications`
[2] `http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-580m/specifications`

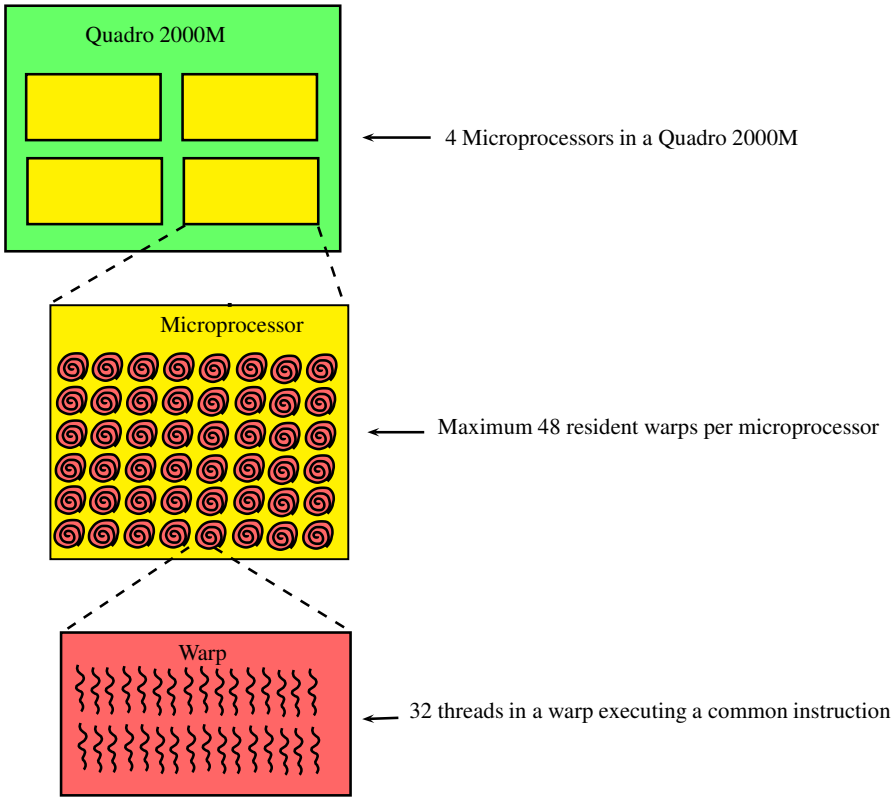


Figure 1.3: NVIDIA Quadro 2000M is a Fermi GPU with four microprocessors. Each microprocessor of a Fermi GPU can execute 48 concurrent warps. A warp consists of 32 parallel threads. Thus, an NVIDIA Quadro 2000M is capable of running 6,144 concurrent threads.

branch. The N threads for the `if` branch are executed in parallel with the other $32-N$ threads doing nothing. Then, the $32-N$ threads for the `else` branch are then executed in parallel and the other N threads doing nothing.

Programming on the GPU

There are several GPU programming frameworks available to write GPU-accelerated programs using programming languages similar to C:

- CUDA [17] is NVIDIA's parallel programming architecture on their GPUs. The CUDA framework allows GPU-accelerated program to be written in CUDA C, CUDA FORTRAN, and OpenCL. GPU cards designed solely for general purpose applications without capability to display graphics have been manufactured as of this writing. NVIDIA Tesla-branded GPUs are an example of such, designed for scientific applications that demand very large parallel computation workloads. Since version 2.2, the OpenCV library includes GPU-accelerated versions of various algorithms and supporting functions for NVIDIA GPUs through CUDA, allowing programmers with no GPU programming experience to benefit from GPU acceleration.

- ATI Stream [18] is AMD's equivalent to CUDA, launched in late 2008. However GPU code written for ATI Stream is not compatible with NVIDIA hardware and vice versa. Since its launch, ATI Stream by itself received almost no attention [19], but it also supports OpenCL just like NVIDIA.
- OpenCL is an open standard for parallel computing, including GPUs, that is supported by both NVIDIA and ATI, as well as other hardware manufacturers [20].

According to Karimi et al. an NVIDIA CUDA GPU operating through OpenCL does suffer performance penalty in the range from 16 to 67 percent increased end-to-end computation time, when compared with the same GPU operating on the native CUDA framework [21]. In addition, they also found that OpenCL code designed for NVIDIA CUDA GPU will not always compile on an ATI GPU due to differences within their OpenCL implementations. Also, executables compiled for NVIDIA GPUs using OpenCL are not compatible with ATI GPUs, and vice versa. OpenCL is useful when developing consumer-oriented GPU-accelerated applications that require compatibility with multiple brands of GPUs.

According to SimplyHired as of this writing [19], there have always been more CUDA-related jobs than OpenCL-related jobs, though the difference is not pronounced. The number of job opportunities related to OpenCL is catching up with CUDA.

As of version 2.3.1 of OpenCV, the GPU-accelerated portion is available exclusively for NVIDIA's CUDA GPUs. However, an OpenCL implementation of SURF is available [22] as of this writing.

1.9 Summary of Contributions

- Implemented the architecture of a markerless computer vision-based augmented reality system, which utilizes a single monocular RGB camera without any other sensors.
- Modified the source code of Speeded Up SURF (GPUSURF) [23], a GPU-accelerated CUDA implementation of SURF (Speeded Up Robust Features), to gain compatibility with Fermi and later GPUs (Section 3.7).
- Implemented GPU-accelerated brute force interest point matching, where CUBLAS library used for calculating dot products through matrix multiplication, and written new CUDA code to implement maximum dot product selector.
- Added GPU acceleration to front end image undistortion.
- Consistent frame rates of 10.5-12 frames per second with 640×480 pixels resolution, using only robust algorithms without resorting to problematic optical-flow tracking as most other systems do.

We also developed the following additional programs to support our AR system:

- A camera calibration program, using OpenCV's checkerboard detection functions, to find necessary camera parameters.

- A 3D map generation program, where the maps are created from one or more planar surfaces from the real world. The 3D points are captured using GPUSURF interest points from a camera image, assuming all points to lie on a plane, and their 3D world coordinates are calculated using pixel coordinates and two manually-measured reference world points at either the camera's bottom corners or midpoint of vertical edges.

We have written approximately 1505 lines of code for the main AR system, 480 lines for the 3D map generator, and 178 lines for camera calibration.

In the main AR system, we have also written an additional 336 lines of code to implement GPU-accelerated RANSAC reprojection, for the ePnP pose recovery algorithm [24], which was not used for our experiment results.

Chapter 2

Literature Review

2.1 Feature Detectors

The most important component of our markerless computer vision-based AR system is robust feature detection, where interest points are detected in every video frame, and their image coordinates are matched to the corresponding 3D points in the real world, and these matches are used to calculate the camera's position and orientation.

To understand feature detectors, we need to first describe corner detectors which predated modern feature detectors.

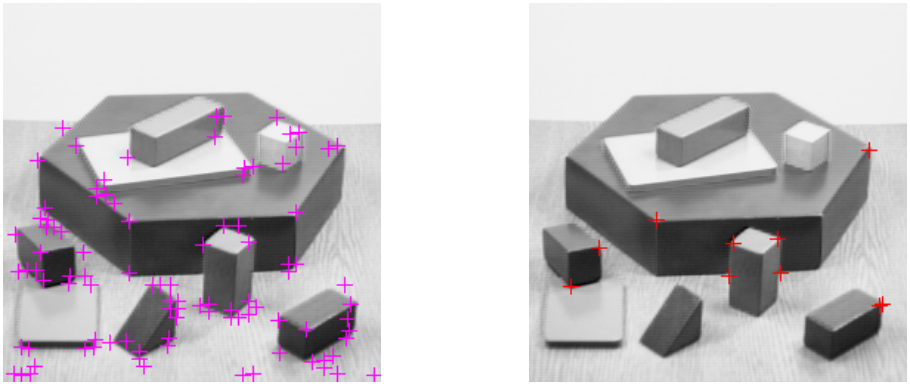
2.1.1 Corner Detection

One major class of computer vision algorithms is corner detection, which finds a set of 2D interest points from an image. The two prominent basic corner detectors include the Harris / KLT detector [25] and the FAST detector [26, 27]. Figure 2.1 shows an example of interest points detected by Harris / KLT and FAST corner detectors respectively. These points are also called *interest points* as they mark potentially useful image coordinates that are useful for other computer vision operations.

However, these detected 2D image points are not unique as they contain only their x and y coordinates within the image.

Harris / KLT

The Harris corner detector, originally proposed by Harris and Stephens in 1988 [25], is one of the most widely known corner detectors. It measures the "cornerness" of each pixel using the partial derivatives of the pixel intensity values, in the horizontal and vertical directions; Pixels that are corners have the largest cornerness values. As a simple corner detector, the details of the algorithms are very often included in university course notes in Computer Vision courses. It was rather computationally expensive for what it did [28], but there were many variants built on top of it to increase its ability to be matched, such as adding scale invariance or affine-invariance [29], illumination invariance [30], or reducing computation cost by incorporating techniques employed by SURF [28].



(a) (b)

Figure 2.1: Interest points (purple crosses) detected using corner detectors. Left: Harris / KLT. Right: FAST.

FAST

FAST detector is a much simpler corner detection algorithm, intended to reduce the computational cost of corner detection when compared to Harris/KLT. For each point in the image, FAST checks the pixels in a circle of a certain radius (typically 3 pixels) surrounding it. If at least a number of them are either all brighter or all darker than the pixel by a certain threshold (typically 12 out of 16), then it is a feature [26, 27].

2.1.2 Interest Points Matching

Interest points detected from two different images using plain corner detectors such as above, can be matched together using square patch correlation. That is, by comparing intensities of pixels around the interest point in a 5×5 or a 11×11 square patch, for example. Larger comparison windows provide better matching result than smaller windows, but increase computational complexity. Comparison criteria include Sum of Absolute Differences (SAD), Sum of Squared Differences (SSD), or Normalized Cross Correlation (NCC) [31].

However, the possibility of matching without unique identification for each interest point is very limited. Matching by image patch comparison alone will work only if objects in both images have almost the same scale, orientation as shown in the top image of Figure 2.2, and a few false matches were made due to illumination changes. Otherwise the matching will fail, as shown in bottom of Figure 2.2 in a failed attempt to match two images have different scales. This was the primary method for matching interest points before robust descriptor-based detectors such as SIFT [32] was invented in 1999.

2.1.3 Detectors with Descriptors

To uniquely identify each interest point in an image, descriptors consisting of many dimensions are added, where each dimension is represented by a number. The two most prominent interest point detectors with descriptors are SIFT [32] and SURF [33]. For example, a 64-dimensional SURF descriptor consists of 64 numbers. The detected interest points can have descriptors attached to allow matching while under changes in rotation, scale, viewpoint

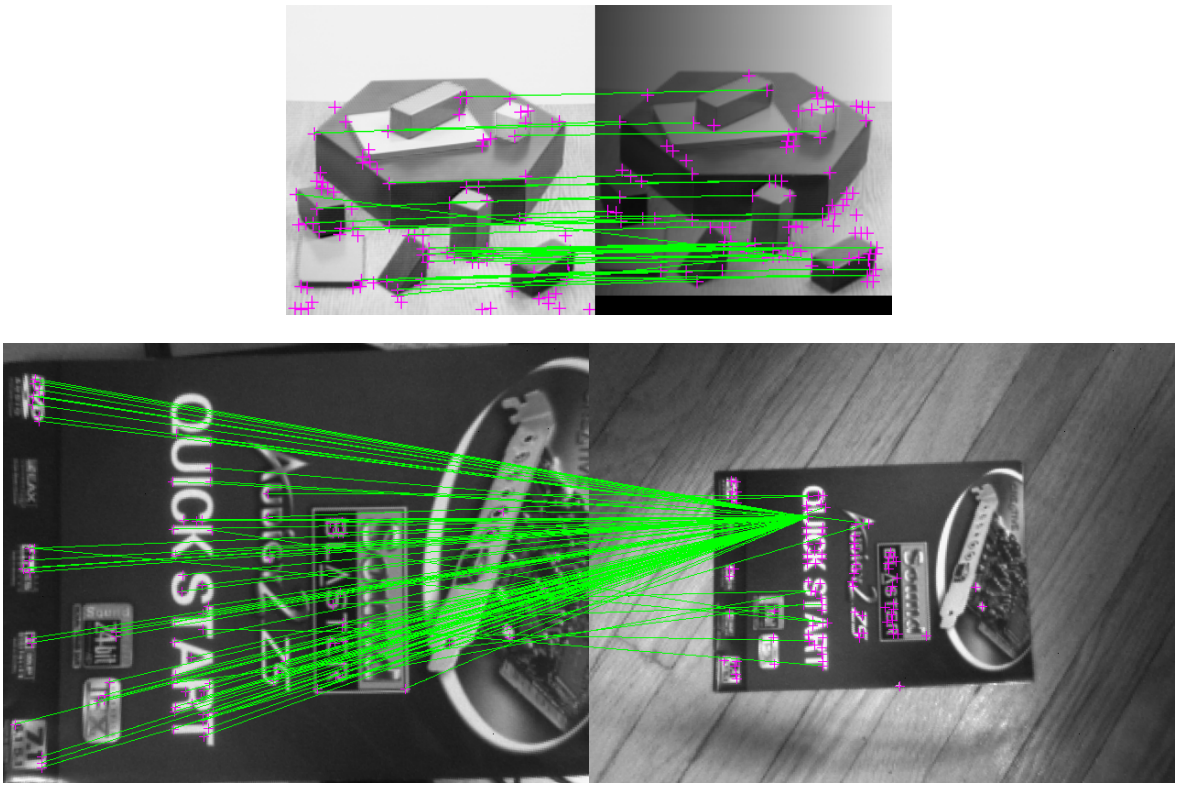


Figure 2.2: Limitations of matching corners using square patches with NCC comparison criteria. In the bottom image, change of scale resulted in complete matching failure.

(Figure 2.3), illumination, motion blur, and the combination of the first four conditions (Figure 2.4).

To allow the robust matching of interest points in such scenarios, detection is performed at multiple image scales, where the scale and orientation of each interest points is computed. A descriptor composed of many dimensions is then computed around each interest point. The descriptors are rotation invariant if the orientation of the respective interest point is computed [32].

Descriptor Matching

To calculate the matching score for a particular pair of descriptors, a typical approach includes calculating the Sum of Squared Difference (SSD) or the dot product of all dimensions between the two descriptor sets of their respective interest points.

Early descriptors used simply the pixel values around the interest point to describe the interest point, known as ”patch comparisons”. More modern descriptors use a more abstract description of the pixels around a corner or interest point detector instead of using the actual pixel values.

The most basic method of descriptor matching is the brute force approach, where every single possible combination of interest points between the source and destination is checked to find the best matches. This method

Figure 2.3: GPUSURF feature matching with rotation, scale, and viewpoint changes.

yields the maximum number of correct matches, but is extremely computationally expensive to the point where matching a pair of images with approximately 2,000 interest points can take over two seconds for 128-dimensional SIFT descriptors, or over one second for 64-dimensonal SURF descriptors.

Thus, typical approaches to descriptor matching would involve binary tree-based methods [34] and Approximate Nearest Neighbor (ANN) [35, 36] searches, which greatly speed up computation but with a drawback of returning fewer matches.

With GPU acceleration, brute force matching can now be performed in real-time. Our AR system employs

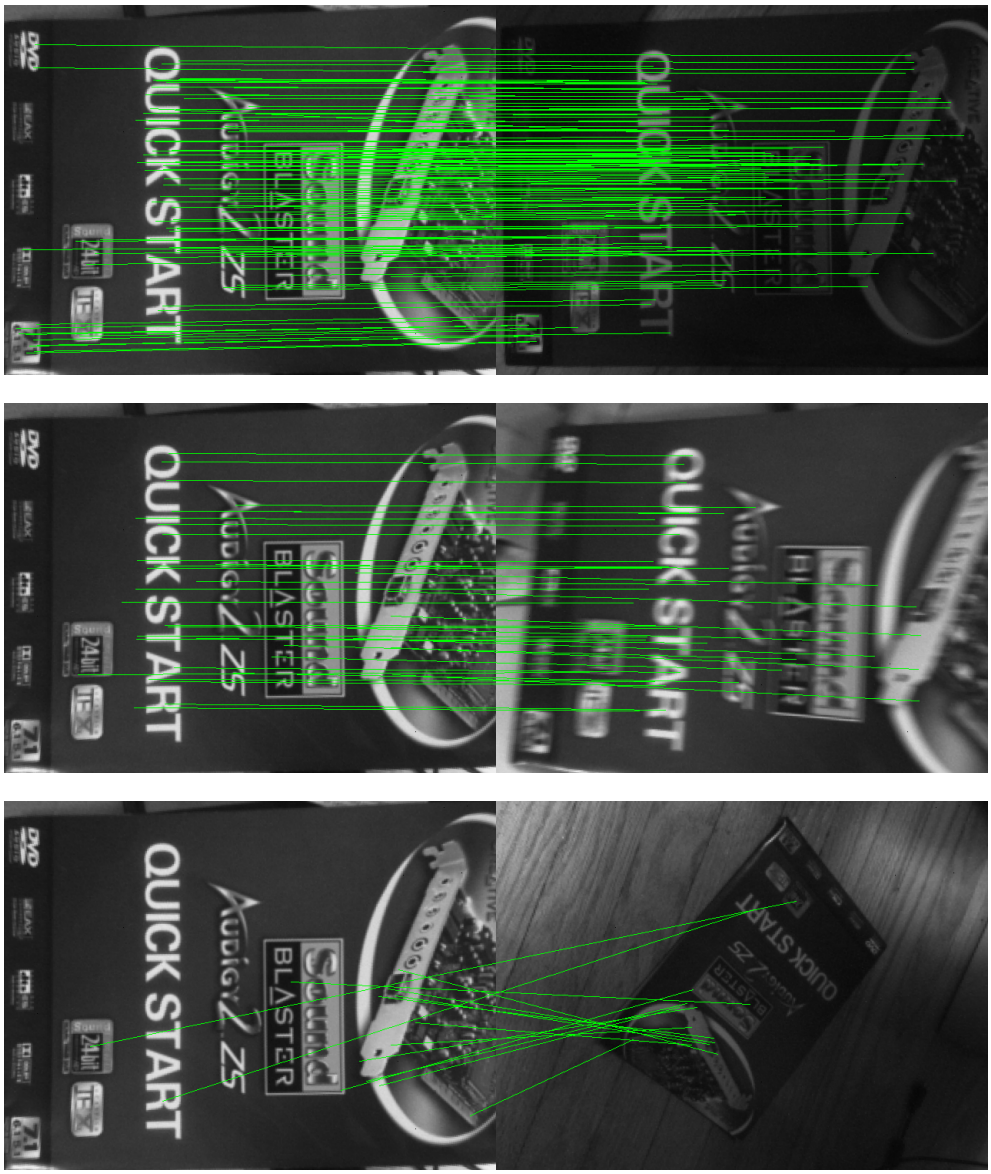


Figure 2.4: GPUSURF feature matching with illumination, motion blur, and combination changes.

brute force matching to achieve as many correct matches as possible between the camera's images and the 3D point databases.

SIFT

The original Scale-invariant Feature Transform (SIFT) algorithm by David Lowe [32] introduced the approach of combining feature detection along with descriptors, which enabled detection of keypoints that can be matched

even when subjected to changes in scale, rotation, or illumination. The image coordinates of detected interest points have sub-pixel precision. It became very popular, at least until SURF became available. Many different implementations of this algorithm by various authors are now available today. This is achieved by inserting additional attributes to the interest points detected: scale, orientation, and a 128-dimension descriptor.

The biggest drawback of this difference-of-Gaussian (DoG) feature detector is that its computational time is very slow, particularly the scale-space computations necessary to achieve scale invariance [37]. A GPU-accelerated implementation of SIFT exists [38], but according to the author of *Speeded Up SURF* [23] real time performance is not achievable even with GPU acceleration [23].

The descriptor has 128 dimensions by default, and the number of dimensions can be decreased to improve performance, with a small sacrifice in matching accuracy [39].

SURF

Speeded Up Robust Features (SURF), first published in 2007 by Herbert Bay et al. [33] draws upon the idea of SIFT and makes several approaches to speed up the computation. Its feature detector is based on the Hessian matrix, and detected interest points have sub-pixel precision.

Its use of integral images allows calculation of sum of image intensities in any rectangular area to be done with only three additions and four memory accesses. The integral images, combined with box filters, are used for approximating the second order Gaussian derivatives without actually calculating them. The descriptor has been reduced to 64 dimensions, compared with SIFT's default of 128. This allows significant speedup to be achieved when compared with SIFT, but comes with a decrease of matching capability (repeatability) when subjected to rotations of approximately 45 or 135 degrees.

But even then SURF by itself does not have real-time computational speed. By employing GPU acceleration, real-time performance has finally been achieved [23]. Our AR system employs UTIAS's *Speeded Up SURF* (GPUSURF) [23], a GPU-accelerated version of SURF for feature detection.

Figure 2.5 illustrates the process of matching an image pair using SURF (specifically, GPUSURF).

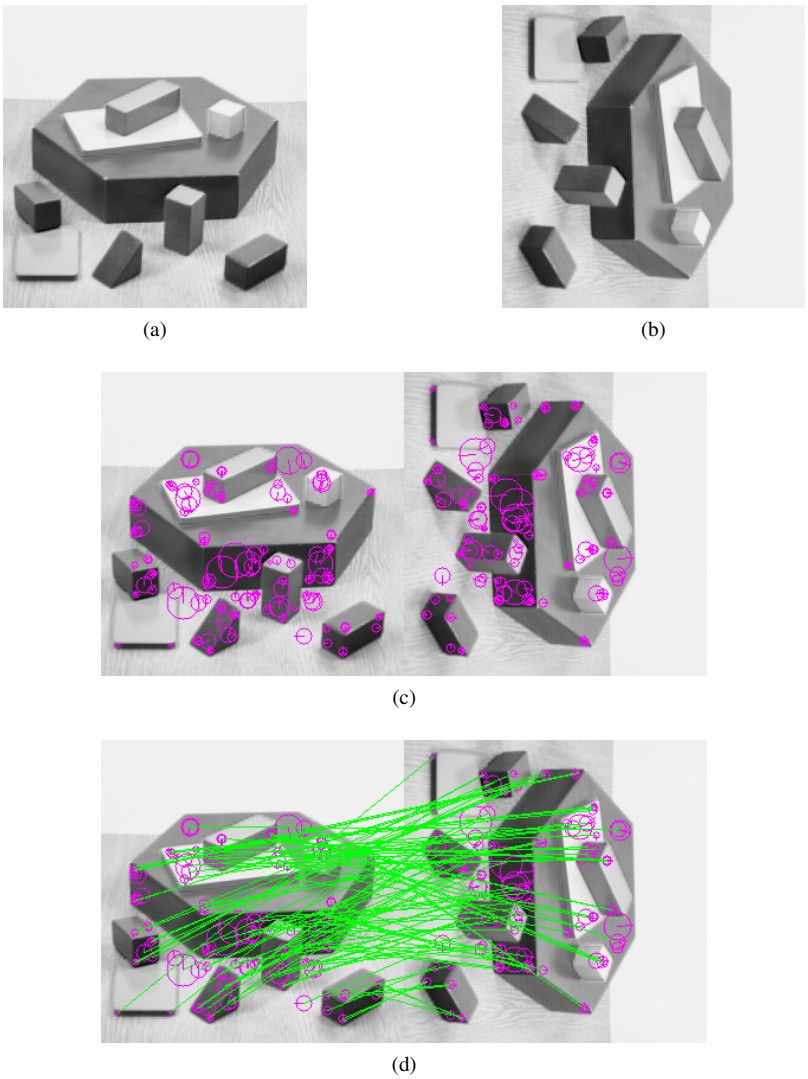
ORB

Both SIFT and SURF are patented. Oriented FAST and Rotated BRIEF (ORB) [40] is designed to be a patent-free alternative to these detectors. It builds upon the FAST corner detector, but adds orientation computation and rotation-invariant version of BRIEF binary descriptors to allow robust matching. Its major drawback is the lack of scale invariance.

Combining plain corner detectors with descriptors

As mentioned before, interest points detected using plain corner detectors have to rely on square patch correlation, which work only if both images are oriented exactly the same way without any differences in scale or illumination.

Fortunately, all interest point detectors including the traditional corner detectors, such as Harris or FAST, can be combined with any descriptor generators, including but not limited to SIFT or SURF's descriptors. For instance, the Harris corner detector can be combined with either SIFT [37] or SURF [41] descriptors to gain



(a) (b) (c) (d)

Figure 2.5: Matching two images using GPUSURF. 2.5a, 2.5b: Two source images to be matched. 2.5c: GPUSURF interest points detected for each image, and their descriptors computed. 2.5d: Matching interest points using their descriptors.

rotation invariance. However it takes extra work in order to make such combinations scale-invariant. For example, Harris-SIFT performs corner detection on multiple resized copies of the image to gain scale-invariance [37].

2.1.4 Pose Recovery

Pose is a term describing both position and orientation, typically defined as translation (position) and rotation (orientation). Pose is a six degree measure and requires at least 6 numbers to define it: x, y, z coordinates for

position, and the angle of rotation in the x-y, y-z, and x-z planes.

Pose recovery algorithms take a list of 2D features / 3D points correspondences and computes the camera's pose, which contains the rotation and translation of the camera, that can be combined to form a projection matrix that is useful for AR systems to project computer-generated virtual objects onto the display.

In OpenCV, an iterative pose recovery algorithm based on Levenberg-Marquardt optimization is the default algorithm used by the `solvePnP` function. While relatively slow, it has good accuracy on the recovered pose, and works even if all 3D points lie on the same plane. Thus this is the pose recovery algorithm we use for our AR system.

EPnP [24] is a non-iterative pose recovery algorithm intended to lower computational cost. We considered incorporating this pose recovery algorithm into our AR system at some point, but it was not able to compute the correct angle of the camera when all the 2D-3D correspondences lie on a plane, as our 3D map generation captures 3D points only on planar surfaces. In the future work, with more sophisticated multi-view 3D map we can consider to use the faster ePnP, instead of the slower iterative algorithm as used by our AR system.

RANSAC

To take account of incorrect matches (outliers) between 2D features and 3D points, a RANSAC [42] loop is incorporated. RANSAC stands for RANdom SAmple Consensus, and pose recovery functions incorporating RANSAC loops are included in the OpenCV library, as the `solvePnPRansac` [43] function.

The following is an example of incorporating RANSAC in pose recovery, as used by OpenCV's `solvePnPRansac` [43] function:

1. Pick four random 2D-3D correspondences.
2. Calculate camera's pose using these four correspondences.
3. Using this camera pose, reproject the 3D points back to 2D image coordinates.
4. Calculate the reprojection error for each reprojected 2D points and their actual 2D coordinates. If the error is less than a specified threshold in pixels, it is counted as an inlier. Otherwise, it is counted as an outlier.

The above progress is repeated for a specified number of times with different sets of four random 2D-3D correspondences. Optionally the loop may end early when a satisfactory inlier ratio is reached. Finally the calculated pose that yields the highest number of inliers is returned as the final result.

2.2 Related Works

A paper from year 2000 by Simon, Fitzgibbon, and Zisserman [44] proposed a general workflow for markerless AR that tracks planar surfaces, which our AR system generally follows upon. Initialization steps include setup of camera calibration parameters, feature detection in the initial image, and manual indication of planar surface in the initial image. For the main loop, the computations are feature detection for every frame. These are matched to the initial image to create 2D-3D correspondences, and are then used to compute the homography matrices, and in turn used for computation of the camera's pose.

Skrypnyk and Lowe 2004 [45] proposed a markerless AR system that uses SIFT for robust feature detection, and k-d trees based feature matching. These processing were computed in every frame, and the high computation cost of SIFT resulted in major performance limitations.

To achieve real-time speeds using only a GPU, a typical approach for markerless approach as presented by Lee, Lee, Lee, and Choi [46], performs feature detection using a Harris-like detector [5] during initialization only, and the rest handled by tracking using a Lucas-Kanade (LK) [47] tracker. When the tracked features are lost, feature detection is performed again which momentarily slowed down the system. This approach came with major drawbacks as mentioned by their paper, as image brightness between frames has to be consistent, and only small camera movements are permitted.

Alternatively, the *PTAM* system [48] presented a two-threaded approach, where one thread detects FAST interest points [49] in every frame, while another thread performs pose recovery over multiple frames with occasional bundle adjustment for correction.

Lee and Hollerer [50] presented another multithreaded solution. The user's hand is tracked in one thread while feature detection with SIFT is running in a separate thread. Yet another thread is responsible for using optical flow to track movement of the SIFT features, along with pose recovery supported by RANSAC [42] to remove outliers and off-plane features.

Markerless AR had been demonstrated on a mobile phone, which used SIFT for feature detection but did not use descriptors matching, and instead used a classicification-based Ferns tracker to perform the tracking. They had to reduce the number of SIFT descriptors from 128 to 36 to speed up computation, and the Ferns sizes to reduce memory requirements. Orientation computations were added to compensate for the above compromises [39].

Another approach toward markerless AR on a mobile phone, was the use of panorama images combined with the use of GPS for outdoor use [51].

In the commercial field, Sony has demonstrated markerless AR running on a PlayStation Vita in a technology demo, known as Magnet [52].

Chapter 3

Augmented Reality System

3.1 System Description

Our markerless Augmented Reality (AR) system uses a single video camera to track natural features (through prepared 3D maps) to localize the position and orientation in the real world. It does not use any additional sensors such as GPS, compasses, or accelerometers. By using GPU acceleration on most of the algorithms in the workflow, computational speed can be increased to near real time levels.

As the video camera connected to the laptop moves around an area, it utilizes computer vision techniques to calculate the camera's pose and orientation within the real world, and then overlays virtual objects aligned to the real world onto the display, such as in Figure 3.2.

This AR system was developed with just a laptop and a Firefly camera, held on the user's hand while walking around, and displayed augmented results on the laptop's display. However it is envisaged that future work will turn this into a hands-free system similiar to to ARQuake [2], where a user walks around wearing a laptop on the back, and the augmented results displayed in a HMD worn on the user's head.

3.1.1 System workflow

Figure 3.1 is a flowchart that describes the working process of the AR system in each frame. The tasks performed by the AR system are as follows:

- Camera input: At the start of the loop, the program reads a video frame captured from the camera, and undistorts the captured image with GPU acceleration. Additionally a grayscale copy of the image is created as required for feature detection (Section 3.4)
- Feature detection: GPU-accelerated SURF is used for finding robust interest points from the camera image, and generating their 64-dimension descriptors.
- 3D Point-Descriptor Map: A database of 3D points in the real world space. Each of the 3D points also has a set of 64-dimension descriptors. Map file generation is an offline process implemented in a separate program. (Section 3.3)

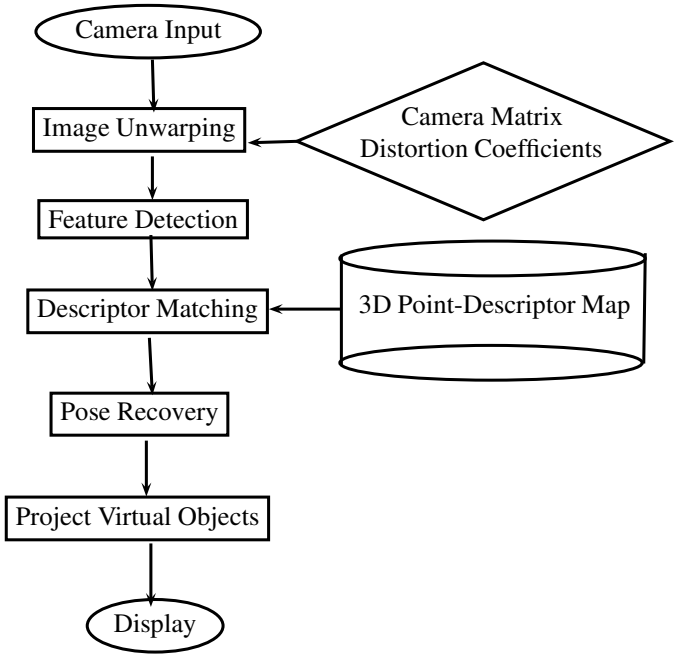


Figure 3.1: Overview of AR system's execution tasks

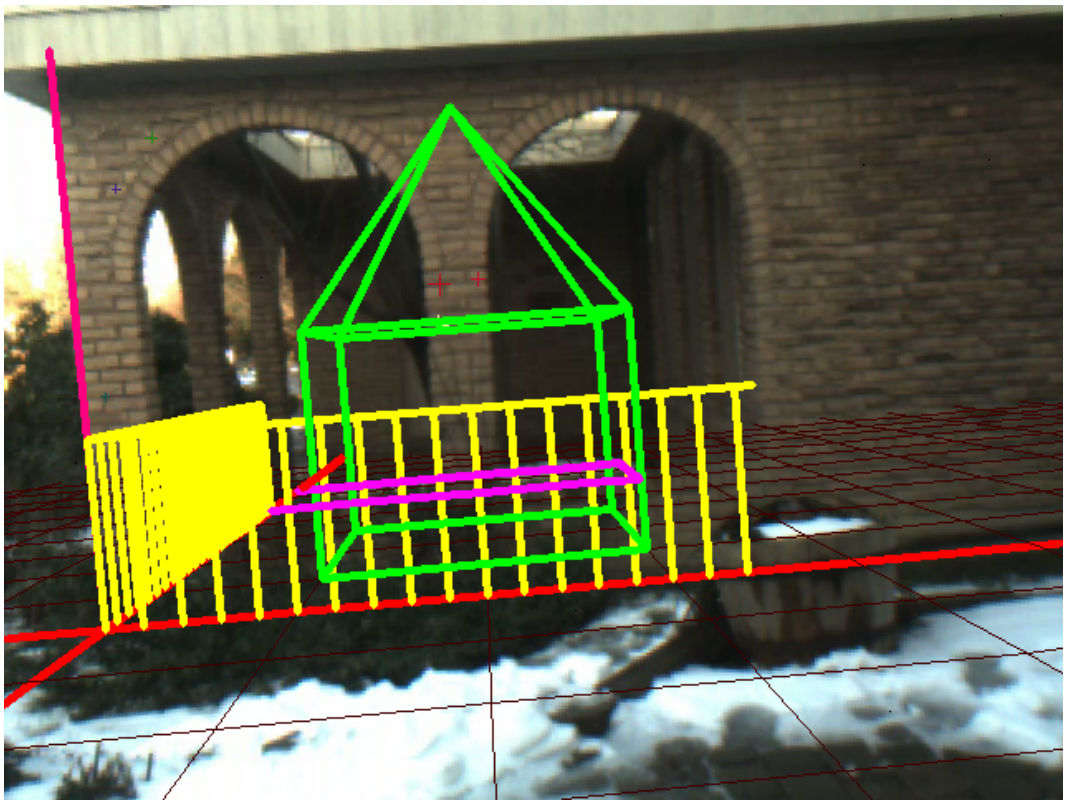
- Descriptor matching: The detected points are matched to the 3D map's world coordinate points, by matching their descriptors using GPU-accelerated brute force algorithm.
- Pose recovery: After the 2D-3D correspondences are made, the camera's position and orientation in the real world can be estimated. Only the RANSAC reprojection portion is accelerated by the GPU.
- Draw projected graphics: Virtual objects, defined in real world coordinates, are projected onto the display using the camera's position and orientation estimated in the previous step.
- Display: The projected virtual objects are displayed on the screen, as though they were part of the real world. Figure shows an example of augmented output.

3.2 Algorithms

3.2.1 Image Undistortion

The Firefly MV camera has noticeable *radial distortion*, as shown in Figure 3.3a. Correcting the camera's distortion serves three purposes. First, it reduces the effective viewpoint angle in the corner of the images. Second, it improves the accuracy of pose recovery. And finally, the user sees a more pleasant augmented image.

Image undistortion has two parts:



(a)

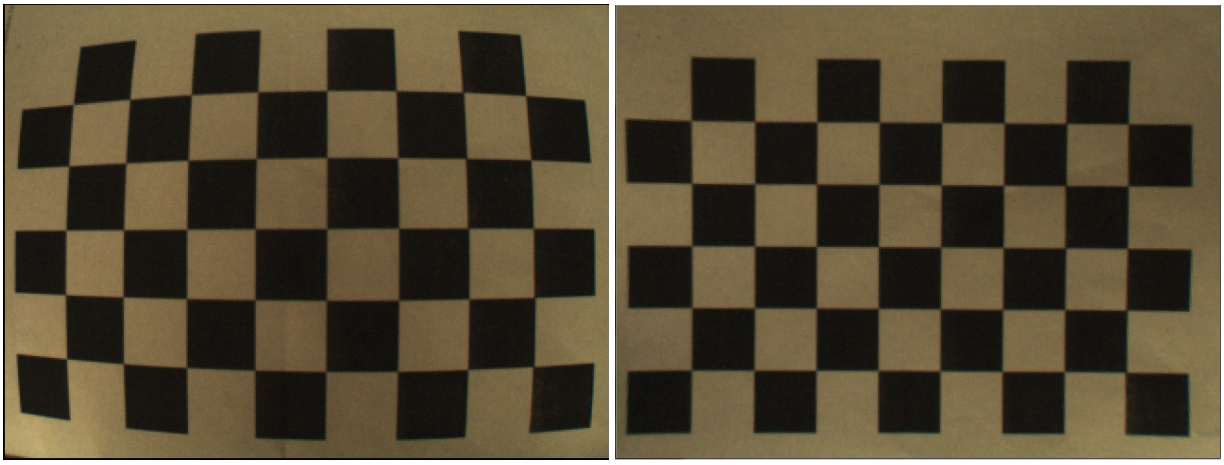
Figure 3.2: Augumented output after completion of AR loop.

1. Calculation of the two undistortion maps during AR system's startup. The two undistortion maps are lookup tables for x and y coordinates respectively, on how the pixels from source image should be mapped to the undistorted image.

2. The actual image undistortion during each frame in the AR loop. For each pixel in the destination image, the two undistortion maps calculated during AR system's startup are used for finding its corresponding pixel from the source image and takes its value.

OpenCV has a function called `undistort`[53] that performs both of the above at the same time, which actually calls the `initUndistortRectifyMap` function and the `remap` function [53]. The `undistort` function is very computationally inefficient for our purpose because the undistortion maps will be calculated in every frame wasting 10 milliseconds each, while resultant undistortion maps are exactly the same result for every frame.

In our AR system, to calculate the undistortion maps during program startup, we call an OpenCV function `initUndistortRectifyMap`, which requires the camera matrix and non-linear distortion parameters. The camera calibration procedure in Section 3.4 is used to obtain these parameters.



(a) (b)

Figure 3.3: Left: Uncorrected camera image of a checkerboard showing the effects of distortion in the lenses. Right: After applying undistortion.

In the main AR loop, the actual undistortion for every frame is performed on the GPU. Our implementation of the CUDA kernel is based on the OpenCV `remap`[53] function.

GPU-accelerated Image Undistortion

The two undistortion maps (lookup tables for undistortion) are copied into the GPU's memory before the AR loop begins, and remains there for the lifetime of the program. This avoids costly memory transfer from system memory to GPU's memory during each frame. The GPU memory for the distorted source image and undistorted destination image are also preallocated during startup, saving processing power from the need of constantly allocating and deallocating memory.

For each frame, the memory transfers between host and device associated with this section consist of:

- The transfer of the distorted source image from host memory to device memory.
- The transfer of the undistorted image from device memory back to host memory.

The CUDA kernel is launched in thread blocks of 256×1 threads each, with the grid dimension of $(\frac{\text{Image width}-1}{256} \times$ Image Height) blocks. Each thread is associated with a pixel in the source image, where each of them finds the undistorted coordinates of its coresponding pixel for the destination image, using the two undistortion maps calculated using `initUndistortRectifyMap`:

$$dst(x,y) = src(map_x(x,y), map_y(x,y))$$

Note that the pixel's destination coordinates are not integers, which means they actually lie somewhere in the middle of two pixels. The OpenCV implementation of `remap` uses bilinear interpolation [54] by default to resolve this. To match the results of OpenCV we needed to implement bilinear interpolation in this kernel as well.

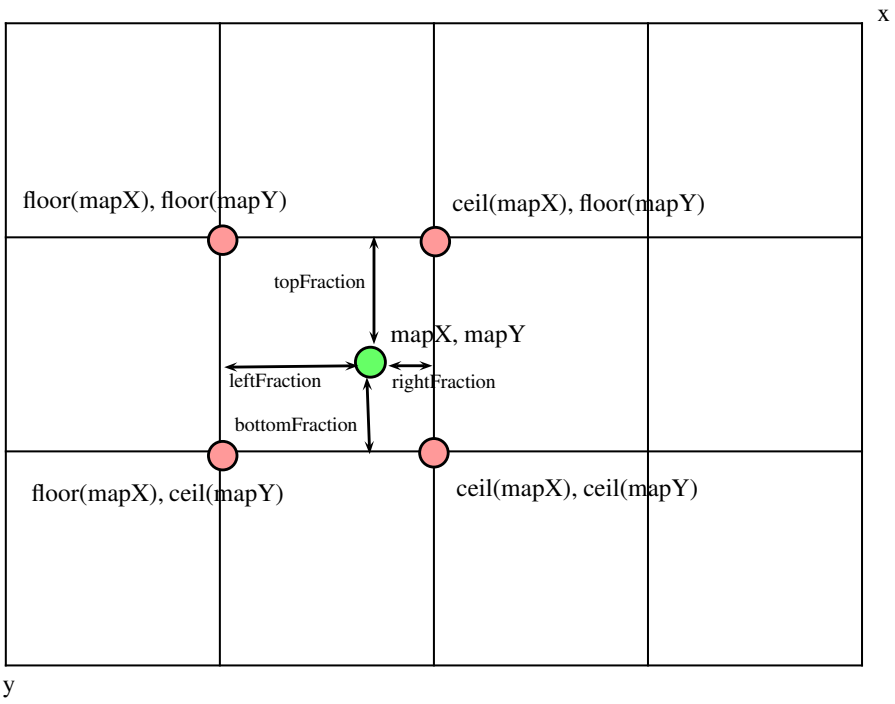


Figure 3.4: Bilinear Interpolation.

Each thread reads the blue, green, and red intensity values of four (4) discrete pixels within the rounding neighbor of the non-integer destination coordinates. These are scattered reads from the global memory, so the Fermi GPU's caches help in this situation.

- Top left pixel is located at `(floor(dstX), floor(dstY))`
- Top right pixel is located at: `(ceil(dstX), floor(dstY))`
- Bottom left pixel is located at: `(floor(dstX), ceil(dstY))`
- Bottom right pixel is located at: `(ceil(dstX), ceil(dstY))`

Now, the thread calculates the distance fraction that is above and below from nearest integer coordinate:

$$\text{Top fraction} = dst(Y) - floor(dstY)$$

$$\text{Bottom fraction} = ceil(dstY) - dst(Y)$$

And for the distance fraction on the left and right from the nearest integer coordinate:

$$\text{Left fraction} = ceil(dstX) - dst(X)$$

$$\text{Right fraction} = dst(X) - floor(dstX)$$

Now we use the weighted average to calculate the horizontal average color intensities:

$$\text{Bottom x-axis average} = \text{Left fraction} \times \text{Bottom left color value} + \text{Right fraction} \times \text{Bottom right color value}$$

$$\text{Top x-axis average} = \text{Left fraction} \times \text{Top left color value} + \text{Right fraction} \times \text{Top right color value}$$

And combine the two horizontal averages vertically to get the pixel's final color intensity:

$$\text{Final color} = \text{Bottom fraction} \times \text{Bottom x-axis average} + \text{Top fraction} \times \text{Top x-axis average}$$

The last three formulas are applied three times each, for each of the blue, green, and red intensity values.

After this kernel finishes, the corrected destination image is copied back to the host memory.

GPU-accelerated Color-Grayscale Conversion

The undistorted image still resides on the GPU's memory, we take this opportunity to perform color to grayscale image conversion on the GPU, without having to transfer memory data from the host to the device.

Our GPU implementation of this conversion is based on OpenCV's `cvtColor`[55] function, where each color is weighted differently in the following formula:

$$\text{Gray} = \text{Red} \times 0.299 + \text{Green} \times 0.587 + \text{Blue} \times 0.114$$

The CUDA kernel is launched in thread blocks of 256×1 threads each, with the grid dimension of $(\frac{\text{Image width}-1}{256} \times \text{Image Height})$ blocks. Each thread is responsible for converting one pixel from 24-bit color to grayscale using the above calculation, where they read their respective pixel's red, green, and blue intensity values from the source image in global memory, calculate the new grayscale using the formula above, and write their respective grayscale pixel into the destination image residing in global memory.

When the kernel finishes, the grayscale image is copied back to the host memory.

Overall, with GPU acceleration the combined undistortion and color-grayscale conversion takes about 2.2ms per frame on our hardware, including the time spent copying the original distorted image from the host to the device, the corrected color image from device to host, and the corrected grayscale image from device to host. This is roughly half the time of OpenCV's original CPU implementations. The low bandwidth and high latency of memory transfers between host and device limited the performance gain achieved using GPU acceleration.

3.2.2 Feature Detection

The first step of the AR system after receiving the camera's image, is to detect interest points. The purple circles in Figure 3.5 illustrate several examples of interest points detected from their images. The size of the circle indicates the scale of each interest point, and the line inside the circle indicates the computed orientation of the point. These circles are not displayed in the actual AR system, but are used for matching 3D points in the next step.

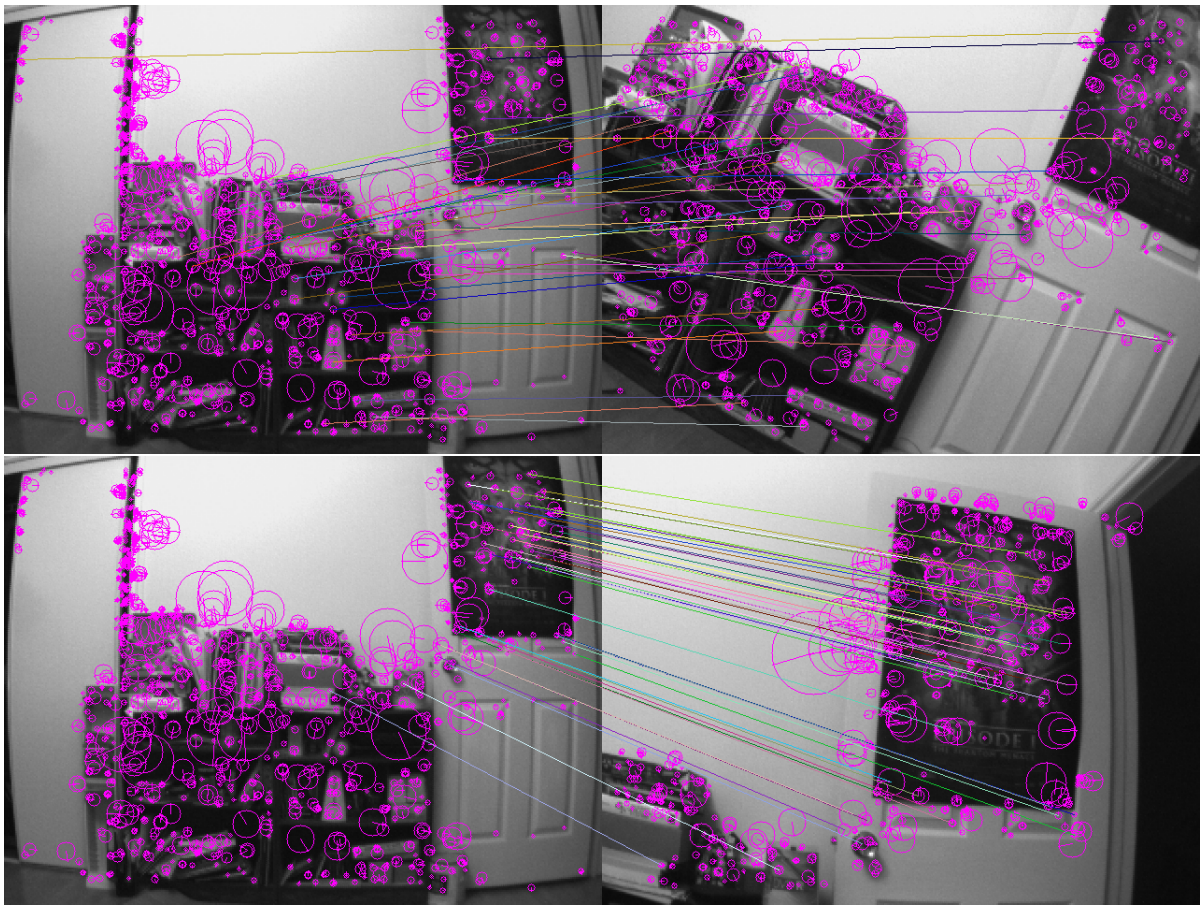


Figure 3.5: Interested points detected with GPUSURF, and using its descriptors to match.

Feature detection and descriptor generation is performed using Speeded Up SURF [23], which is an implementation of SURF to take advantage of the GPU's parallel processing power. It is implemented with NVIDIA's CUDA, and requires a CUDA-compatible NVIDIA GPU to run. From here onward, we will refer to this CUDA SURF implementation as GPUSURF.

In the laptop used for development of this AR system, for 640×480 sized images and default parameters, GPUSURF is capable of performing keypoint detection and descriptor generation in less than 20 milliseconds when plugged to AC power, and approximately 40 milliseconds when running on batteries.

3.2.3 Descriptor Matching

After the interest points from the camera image have been detected, they can now be matched to the 3D map's points using each of their descriptors. Figure 3.5 illustrates two examples of descriptor matching, where different coloured lines represent different matches of interest points in each of the image pairs. As the images show, a few of the matches are incorrect, but in the pose recovery step, these outliers are taken care of by RANSAC.

The GPUSURF implementation does not include any functionality to actually perform descriptor matching, so we had to write our function to perform GPU-accelerated matching.

In this AR system, exhaustive keypoint matching, where all vectors are compared (also known as brute force), is used with dot product as the comparison criteria. This is a highly data-parallel operation well suited for GPUs. Recursive tree-based matching algorithms such as Kd-trees [34] that are often mentioned in the literature are not exactly suitable to be coded for GPUs, and matching accuracy is sacrificed due to their approximative natures.

Step 1: Dot product calculation

By putting the two sets of descriptors into their respective matrices, the resulting matrix contains the dot product for all of the descriptor comparisons.

The descriptor sets are as follow:

- The first descriptor set represents keypoints from the image captured by the camera, arranged as a $N \times 64$ matrix where N is the number of keypoints in the camera's image. The descriptors in this matrix is different for every frame, and the contents are copied from host memory to device memory in every frame.

$$\text{Camera Descriptors Matrix} = \begin{bmatrix} d_{p0,0} & d_{p0,1} & d_{p0,2} & \dots & d_{p0,63} \\ d_{p1,0} & d_{p1,1} & d_{p1,2} & \dots & d_{p1,63} \\ d_{p2,0} & d_{p2,1} & d_{p2,2} & \dots & d_{p2,63} \\ d_{p3,0} & d_{p3,1} & d_{p3,2} & \dots & d_{p3,63} \\ d_{p4,0} & d_{p4,1} & d_{p4,2} & \dots & d_{p3,63} \\ d_{p5,0} & d_{p5,1} & d_{p5,2} & \dots & d_{p3,63} \\ \dots & \dots & \dots & \dots & \dots \\ d_{pN,0} & d_{pN,1} & d_{pN,2} & \dots & d_{pN,63} \end{bmatrix}$$

- The second descriptor set comes from the map file, arranged as a $64 \times M$ matrix where M is the number of 3D points in the database. The descriptors in this matrix is the same during the lifetime of the program, so the contents are copied from host memory to device memory during program startup only.

$$\text{Transposed 3D Map Descriptors Matrix} = \begin{bmatrix} d_{m0,0} & d_{m1,0} & d_{m2,0} & d_{m3,0} & d_{m4,0} & \dots & d_{mM,0} \\ d_{m0,1} & d_{m1,1} & d_{m2,1} & d_{m3,1} & d_{m4,1} & \dots & d_{mM,1} \\ d_{m0,2} & d_{m1,2} & d_{m2,2} & d_{m3,2} & d_{m4,2} & \dots & d_{mM,2} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ d_{m0,63} & d_{m1,63} & d_{m2,63} & d_{m3,63} & d_{m4,63} & \dots & d_{mM,63} \end{bmatrix}$$

The GPU memory space for both matrices is pre-allocated during program startup, avoiding the need to constantly freeing and reallocating memory for the descriptors of camera images in every frame.

These two matrices are multiplied together on the GPU using function `cublasSgemm` provided by the CUBLAS library (CUDA Basic Linear Algebra Subprograms) [56]. CUBLAS is a library of optimized GPU-

accelerated matrix operations that runs on CUDA, which is included with the CUDA Toolkit.

$$\text{Matching Scores Matrix} = (\text{Camera Descriptors Matrix})(\text{Transposed 3D Map Descriptors Matrix})$$

The result of this brute-force operating is an $N \times M$ matrix containing the dot products from each of the keypoint combinations in the pair. The rows are the keypoint indices for the camera image and columns are the index to a 3D point in the map file:

$$\text{Matching Scores Matrix} = \begin{bmatrix} \text{Score}_{p0,map0} & \text{Score}_{p0,map1} & \text{Score}_{p0,map2} & \dots & \text{Score}_{p0,mapM} \\ \text{Score}_{p1,map0} & \text{Score}_{p1,map1} & \text{Score}_{p1,map2} & \dots & \text{Score}_{p1,mapM} \\ \text{Score}_{p2,map0} & \text{Score}_{p2,map1} & \text{Score}_{p2,map2} & \dots & \text{Score}_{p2,mapM} \\ \dots & \dots & \dots & \dots & \dots \\ \text{Score}_{pN,map0} & \text{Score}_{pN,map1} & \text{Score}_{pN,map2} & \dots & \text{Score}_{pN,mapM} \end{bmatrix}$$

Step 2: Picking best matches

The second step of keypoint matching is to select the best match between each keypoint in the camera image and each 3D point in the map file, using the calculated dot product results in the previous step. The resultant matrix from the previous step remains in GPU's memory to be used for this step, avoiding time-consuming memory transfers between host and device.

A CUDA kernel function is launched with N threads on the GPU, where N is the number of keypoints in the camera's image. Each keypoint in the camera's image is assigned a thread, where the *i*th thread is responsible for the *i*th keypoint. The threads perform an exhaustive search through their keypoint's row in the dot product matrix, searching for the best and second best matches. Two thresholds are compared before the chosen matches are accepted or rejected:

- The relative threshold is the minimum ratio between the best and second best match.
- The absolute threshold is the minimum in order to discard weak matches.

Where a match is accepted, the index of the matching map file point is put into the matches array. Where no best match meets the thresholds, it will be deemed to have no match, and is assigned a -1 in the matches array instead.

Step 3: Removing multiple matches to the same 3D point

The last step of keypoint matching is to scan through the array of matching indices, to eliminate duplicate matches to the same 3D point, keeping the one with the best matching score. This ensures that only one keypoint from the camera image is matched to any given 3D point in the map.

This step is implemented as a CUDA kernel executed on the GPU, with the same number of threads as the previous step, which is the number of keypoints in the camera's image. For each thread, if its respective keypoint has a match, then it searches through the entire array concurrently for any other keypoint that has the same matching target as its, and compares their matching score. If the thread's own keypoint has a better matching score than the other keypoint, then that keypoint's matching index is set to -1, and the loop continues for this

thread. Otherwise, if the other keypoint has a better matching score than the thread's own keypoint, then the thread's own keypoint has its matching index set to -1, and the loop ends for this thread. All memory operations in this kernel are performed on global memory.

The final result, an array of indices of matches, is transferred from the GPU to the host, to be used by the remainder of the AR system.

3.2.4 Pose Recovery

After the descriptor matching step, a number of the interest points in the camera image now have correspondence to a 3D world point, allowing the estimation of the camera's pose in the real world.

Pose recovery is performed by OpenCV `solvePnPRansac` from the GPU module [43], using the iterative algorithm based on Levenberg-Marquardt optimization. This function takes at least four 2D-3D correspondences and the camera matrix, and utilizes RANSAC (Random Sample Consensus) [42] to reduce the effects of outliers that result from incorrect correspondences between image points and 3D points. In the OpenCV's implementation, in each RANSAC iteration, it picks four random 2D-3D correspondences and estimates the camera's position and orientation, and calls OpenCV's `solvePnP` function which is not GPU-accelerated.

The function then searches for the best result, which is the RANSAC iteration that contains the highest number of inliers. It then returns the best estimate of camera's pose and orientation in the 3D world. The computation time for pose recovery is proportional to the number of RANSAC iterations chosen.

We also experimented with a different pose recovery algorithm, ePnP [24] with this AR system, combined with our own GPU-accelerated RANSAC loop. Though it is clearly faster, it has a drawback. It is unable to recover the camera's direction correctly when all of the matched 3D points lie on the same plane, which is the case with most of our AR system's test scenarios.

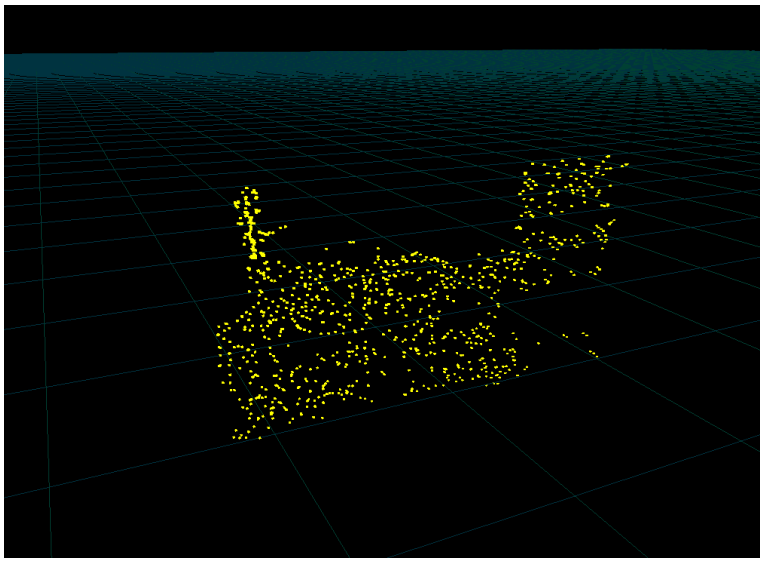
3.3 3D Maps

The 3D map file for this AR system is a collection of 3D points in the real-world space, and each 3D point has a set of 64-dimensional GPUSURF descriptors to allow the matching between 2D image points and 3D world points. Map files are created as an offline process using a separate program. This will be elaborated in Section 3.3.1.

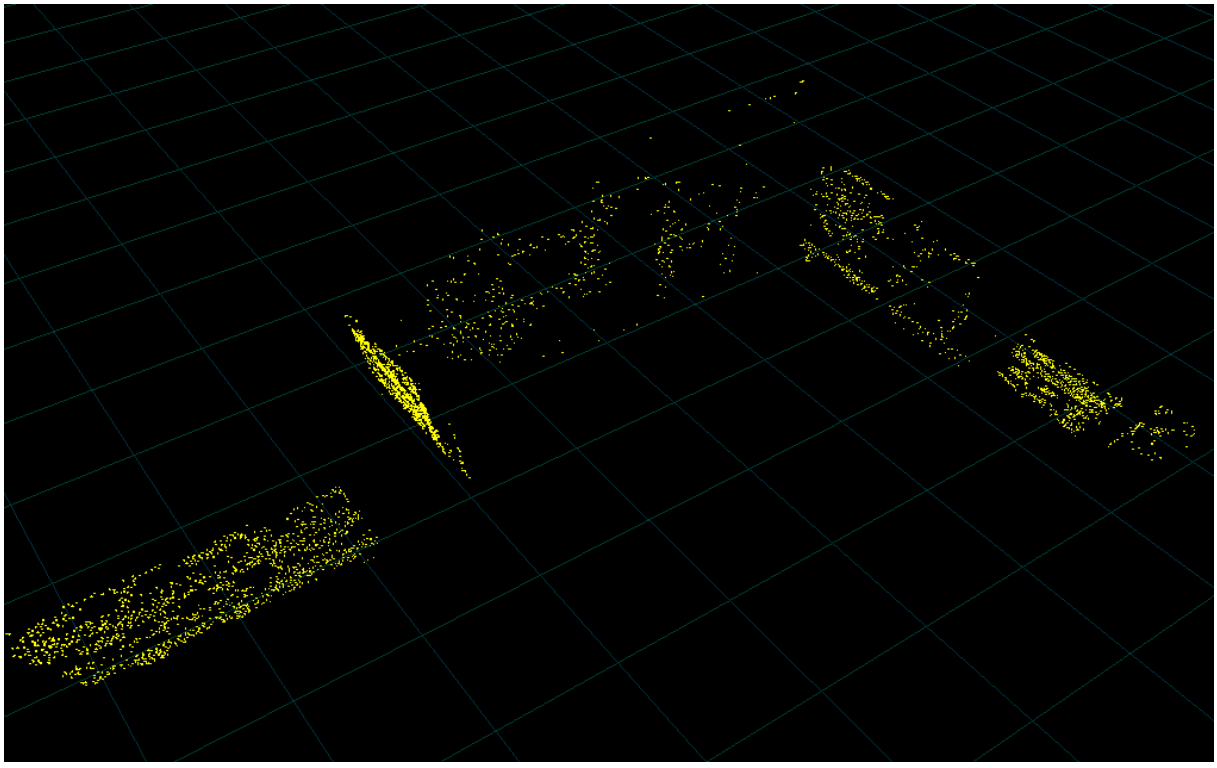
3.3.1 3D Map Generator

Our map creation program captures **Speeded Up SURF** interest points from locally planar vertical surfaces, such as the garage door or the brick arches in Figure 3.8. The user manually measures two 3D world points, and aligns the camera to the two world points. The 3D coordinates of each detected interest point are calculated using their image coordinates and the two manually-entered 3D world points,

The program assumes all interest points from each camera image lie upon a flat plane. Off-plane interest points, such as the snow on the ground in Figure 3.8a, the objects behind the arches in Figure 3.8b, can be manually removed using a graphical user interface.



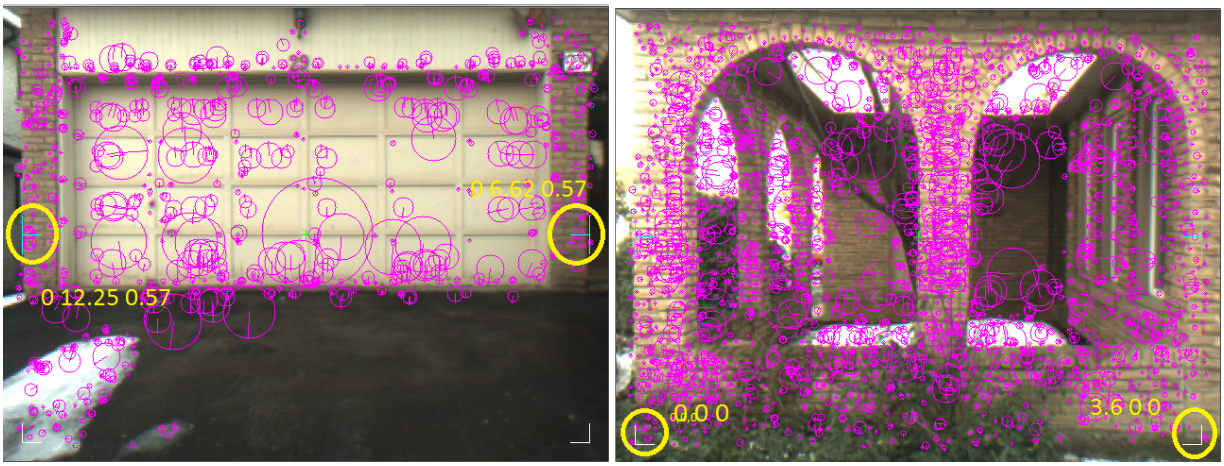
(a)



(b)

Figure 3.6: 3D visualizations of map files. Top: Single scene map. Bottom: Multi scene map

Figure 3.7: Camera loop of the map file generation program.



(a) (b)

Figure 3.8: The z-coordinates of the input reference can be either at the bottom white corners, or the cyan marks in the middle of the sides.

Step 1: Capturing a scene

The map file generation program begins the scene capturing phase with a camera input loop. Images captured from the camera are corrected for distortion in the same way done by our main AR system. The undistorted

camera images, along with GPUSURF features, are displayed the on the screen as shown in Figure 3.7.

The user points the camera to a reasonably planar wall that returns a good number of SURF features, say as a poster, and aligns the two reference markers on the sides into two chosen targets that are useful to be manually measured, such as a the corner or edge of a bookshelf. The reference markers can be either the cyan markers in the midpoint of the left/right camera edge, or the white markers in the bottom corners. The user then hits any key to capture the scene.

This program assumes the entire image from the camera is on a plane, and the vertical (z) axis is perpendicular to the horizon.

The circles are the GPUSURF detected features, where the size of the circle represents the size of the feature, and the lines inside the circle are the orientation of the feature. The white corner markings at the bottom left and bottom right of the image are the two points in the world coordinates to be physically measured. The center green cross is an aid to indicate the center of the image.

Step 2: Entering world coordinates

The program will ask the user whether to use the middle cyan markers or the bottom white markers as the reference world points. The user will then be asked for the x and y horizontal world coordinates of the two desired reference markers, which are manually measured by the user. Both reference world points are assumed to have the same z-position, so the program asks the user to input the z-coordinates once.

After the world coordinates of the plane pointed by the white corners are entered, the world coordinates of each GPUSURF interest points can be derived. First, find the vector components of the bottom effective edge of the image:

$$\text{vector}_x = x_{\text{corner 2}} - x_{\text{corner 1}}$$

$$\text{vector}_y = y_{\text{corner 2}} - y_{\text{corner 1}}$$

$$\text{vector}_{\text{combined}} = \sqrt{\text{vector}_x{}^2 + \text{vector}_y{}^2}$$

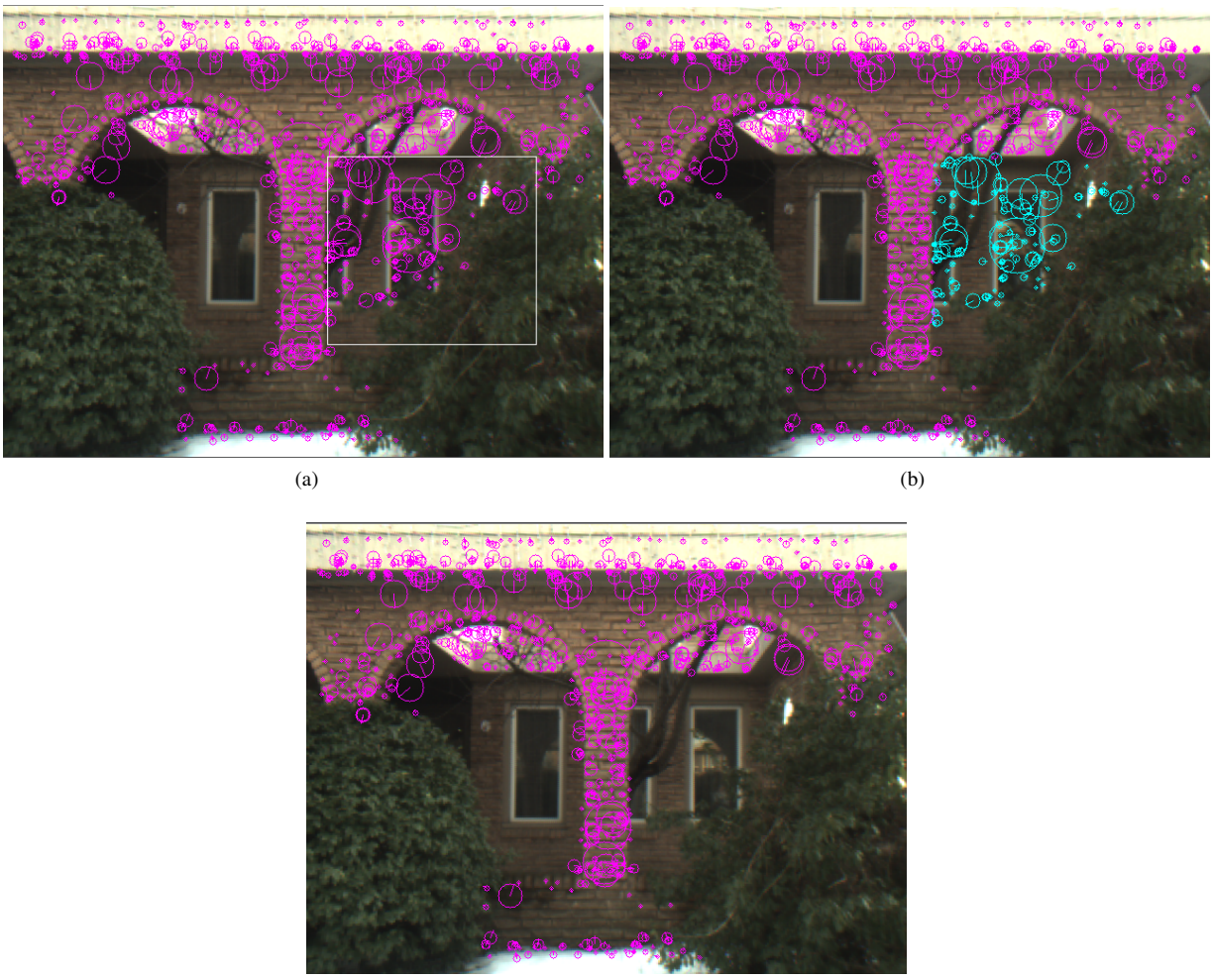
Now the world coordinates of each interest point can now be estimated as follows: (u and v are the screen coordinates of the interest point, after eliminating the outside buffer zone)

$$x_{\text{world}} = x_{\text{corner 1}} + \text{vector}_x \times \frac{u}{\text{Screen Width}}$$

$$y_{\text{world}} = y_{\text{corner 1}} + \text{vector}_y \times \frac{u}{\text{Screen Width}}$$

$$z_{\text{world}} = z + \text{vector}_{\text{combined}} + \frac{\text{Screen Height} - v - 1}{\text{Screen Height}}$$

For this AR system there is a 20-pixel buffer zone (outside the white brackets) around the edges of the image, so the outermost 20 pixels around the screen are ”not considered to be part of the image” in these calculations.



(a) (b)

(c)

Figure 3.9: Using the GUI to manually remove off-plane interest points.

Step 3: Removing off-plane interest points

After the world coordinates of each interest points are calculated, the user has the option to manually delete unwanted interest points from the image, particularly the off-plane points.

To select unwanted interest points that are to be deleted, the user clicks and holds the left mouse button and drags a rectangular selection box on them. The user hits the delete key or backspace to remove the selected interest points off the 3D map, or clicks the mouse anywhere to unselect the interest points. The user hits any other key to finish this step, and the remaining 3D points and their descriptors are saved into the 3D map file.

3.3.2 3D Map format

The 3D map has the following format, for each 3D point in the file: `3dpoint-num desc-num x y z detector-type num-desc d0 d1 d2 ... d63`

Where:

- `3dpoint-num` is the ID of this 3D point in the map. This ID starts at 0 for the first 3D point in the map.
- `desc-num` is always the same as `3dpoint-num` in this AR system. In future projects, there may be multiple descriptors for the same 3D point.
- `x, y, z` are the 3D world coordinates of this point.
- `detector-type` is an identifier for the descriptor generator. In this AR system, `UT_GPU_SURF`is used for this field.
- `num-desc` is the number of dimensions for the descriptors. For SURF, this is 64.
- `d0, d1, ..., d64` are the 64 descriptors.

3.4 Camera Calibration

This section covers the procedure to obtain the camera matrix (also called K-matrix) and distortion coefficients, which are needed in order to perform image undistortion and pose recovery. A camera calibration program using OpenCV was developed to find these matrices.

The camera matrix has 3 rows and 3 columns with the following form:

$$C = \begin{bmatrix} f_u & 0 & c_u \\ 0 & -f_v & c_v \\ 0 & 0 & 1 \end{bmatrix}$$

Where f_u and f_v are the focal length (zoom factor) in the horizontal and vertical axes, and c_u and c_v are the image center. The ratio of f_u and f_v is the aspect ratio of the pixels, which is 1:1 for the Firefly camera, and in this case these two values are equal.

The camera matrix is a component of the 'projection matrix' which maps 3D points to 2D points in the image. The camera matrix is also used to convert a pixel position in the image into angles in space.

As well as the camera matrix, non-linear distortion parameters are also necessary for most computer vision applications to correct the curvature of the camera image. A calibration procedure is necessary to calculate both the camera matrix and non-linear distortion parameters.

To find the coefficients of the camera matrix, a checkerboard-based calibration method is used. The OpenCV library [57] provides the necessary functions to find the camera matrix and the distortion coefficients.

The camera calibration program starts the camera loop, calls `findChessboardCorners` function to find a chessboard in the image. If a chessboard is detected, it returns the integer screen coordinates of the interior corners on the chessboard. Figure 3.10 illustrates the detected chessboard corners. Another function, `cornerSubPix`

is called to attain sub-pixel accuracy for each detected corners. The relative world coordinates of each chessboard corner are calculated from their sub-pixel screen coordinates.

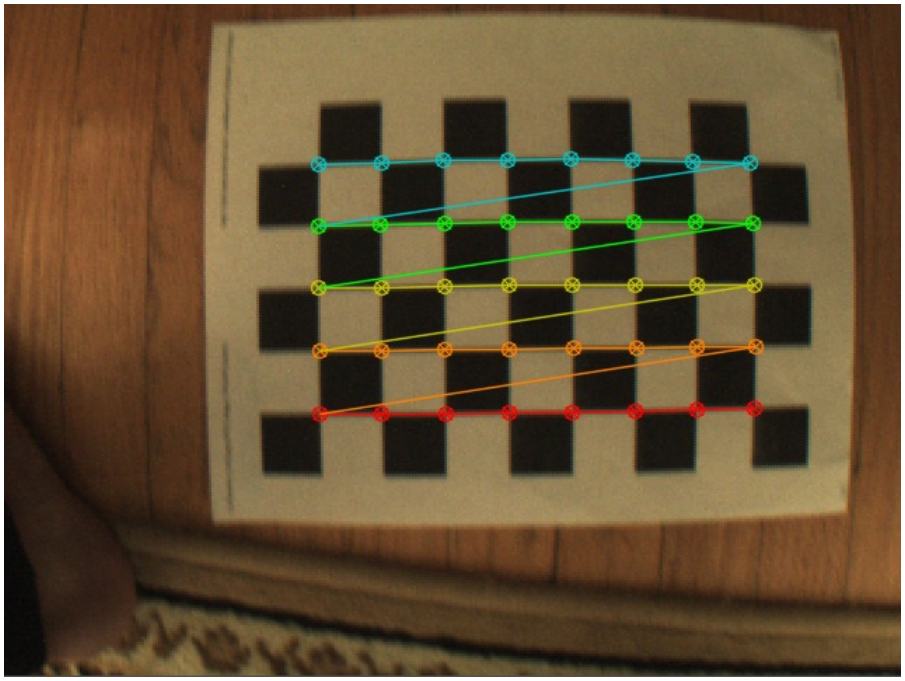


Figure 3.10: Using OpenCV's `findChessboardCorners` function to detect a chessboard with 5 rows and 8 columns.

This process is repeated until the chessboard is detected in 30 different images. With the correspondences between world coordinates and camera coordinates of each chessboard corner in all 30 images, the function `calibrateCamera` is finally called to perform the actual camera calibration. This function returns the calculated camera matrix and the distortion coefficients at the same time.

After the camera matrix is calculated, the user manually verifies the undistortion effects of the calibration. A camera loop runs again, undistorts the camera image using the `undistort` function, with the camera matrix and distortion coefficients just derived, and displays the undistorted results on the screen. If the user finds the undistortion effects satisfactory, the camera calibration parameters just calculated can be taken as final. Otherwise, if radial distortion is still noticeable, the camera distortion process should be repeated in order to obtain a new set of camera calibration parameters.

3.5 Graphic Rendering

After the camera's pose is retrieved, it is now possible to draw lines defined in world coordinates, and project them to the output to augment the camera image. This system as developed displays the output on the laptop's monitor, but in future work this will be displayed in an HMD. Figure 3.11 shows an example of the final output.

The two ends of the lines are projected using the following calculations:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = C \begin{bmatrix} R \mid T \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Where:

- x, y, z are the world coordinates of the input point.
- C is the camera matrix calculated using the checkerboard method.
- R is rotation matrix, and T is translation matrix, combined into a 3 by 4 projection matrix.

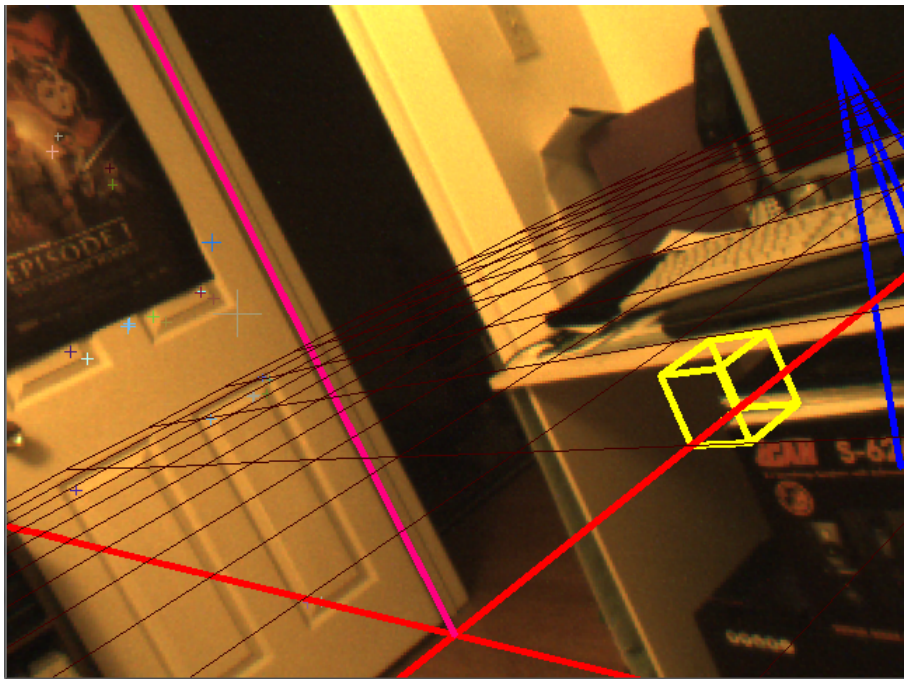


Figure 3.11: Drawing projected lines onto display.

3.6 Hardware

The Lenovo W520 laptop used during the development of this system contains the following hardware:

- CPU: Intel Sandy Bridge running at stock frequency of 2.2 GHz (Core i7 2720QM)
- System (host) memory: 4 GB DDR3

- GPU: NVIDIA Quadro 2000M [1](see below)

For this AR system, a CUDA-capable GeForce series GPU will suffice. A Quadro's GPU's improved double precision capabilities compared to a GeForce GPU is unnecessary, as none of the GPU-accelerated CUDA applications in this AR system use any double precision operations, but use single-precision floating point values [23]instead.

The Quadro 2000M mobile GPU operates at 1.1 Ghz clock. There are 4 Streaming Multiprocessors on the GPU die. It is built on the Fermi architecture, providing 48 CUDA cores per multiprocessor for a total of 192 CUDA cores, which is the number of warps that can be executed concurrently. As a warp is a group of 32 parallel threads executing the same instructions, this yields up to 6,144 parallel threads. There are 2 GB of DDR3 memory for the GPU, where this memory is called global memory in CUDA context.

This is not the fastest CUDA-capable mobile GPU available on the market as of the time this laptop was acquired. For instance, a GeForce GTX580M contains twice as many microprocessors (thus twice as many CUDA cores), and a slightly higher GPU clock speed of 1.24 GHz [2]. A laptop running with that GPU should provide faster performance than our laptop.

3.7 Modification of GPUSURF for Fermi Compatibility

Although its developers claimed that GPUSURF is under active development, no update have been made since April 2010. In particular, with the latest version released as of this writing (0.2.0), it does not work properly with the latest Fermi architecture GPUs without modification to the CUDA code. In fact, it did not even compile if the latest CUDA toolkits are used. The Lenovo W520 laptop's GPU, Quadro 2000M, has a Fermi architecture. It is designed and optimized for earlier, pre-Fermi GPU architectures.

To make GPUSURF to function properly in Fermi GPUs, its CUDA code was modified as per NVIDIA's Fermi Compatibility Guide [58]. The following is an excerpt of the GPUSURF code performing parallel reduction `descriptors.cu`, before modifying for Fermi compatibility [23]:

```
// compute thread IDs (row-major)
    int tid = __mul24(threadIdx.y,blockDim.x) + threadIdx.x;
...
   // allocate shared memory
    __shared__ float smem[2*5*5];
    // 2 floats (dx,dy) for each thread
    // (5x5 sample points in each sub-region)
...
  // sum (reduce) 5x5 area response
    __shared__ float rmem[5*5];
    // buffer for conducting reductions
...
```

[1] http://www.nvidia.com/content/PDF/product-comparison/Quadro_Mobile_Product_Comparison.pdf
[2] http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-580m/specifications

```
 // sum (reduce) from 16 to 1 (unrolled - aligned to a half-warp)
    if (tid < 16)
      {
        smem[tid] = smem[tid] + smem[tid + 8];
        smem[tid] = smem[tid] + smem[tid + 4];
        smem[tid] = smem[tid] + smem[tid + 2];
        smem[tid] = smem[tid] + smem[tid + 1];

        rmem[tid] = rmem[tid] + rmem[tid + 8];
        rmem[tid] = rmem[tid] + rmem[tid + 4];
        rmem[tid] = rmem[tid] + rmem[tid + 2];
        rmem[tid] = rmem[tid] + rmem[tid + 1];
      }
    __syncthreads();
...
```

In each of the reduction steps, it omits the `__syncthreads()` to improve performance. These take the advantage of an optimization technique that is mentioned in [58].

GPUSURF performs parallel reduction with this optimization technique in the computation of descriptors and orientations. To enable this code to work on Fermi or newer GPUs, it is necessary to add a pointer with the `volatile` identifier pointing to the shared memory where parallel reduction will take place, and have the memory operations go through the volatile pointers. The following is the above code with this modification:

```
// compute thread IDs (row-major)
    int tid = __mul24(threadIdx.y,blockDim.x) + threadIdx.x;
...
   // allocate shared memory
    __shared__ float smem[2*5*5];
    // 2 floats (dx,dy) for each thread
    // (5x5 sample points in each sub-region)
...
  // sum (reduce) 5x5 area response
    __shared__ float rmem[5*5];
    // buffer for conducting reductions
...
        volatile float *smem2 = smem;
        volatile float *rmem2 = rmem;

    // sum (reduce) from 16 to 1 (unrolled - aligned to a half-warp)
    if (tid < 16)
      {
```

```
        smem2[tid] = smem2[tid] + smem2[tid + 8];
        smem2[tid] = smem2[tid] + smem2[tid + 4];
        smem2[tid] = smem2[tid] + smem2[tid + 2];
        smem2[tid] = smem2[tid] + smem2[tid + 1];

        rmem2[tid] = rmem2[tid] + rmem2[tid + 8];
        rmem2[tid] = rmem2[tid] + rmem2[tid + 4];
        rmem2[tid] = rmem2[tid] + rmem2[tid + 2];
        rmem2[tid] = rmem2[tid] + rmem2[tid + 1];
      }
    __syncthreads();
...
```

In addition, GPUSURF utilizes CUDPP [59] to calculate the integral images, distributed together with it. The version supplied with GPUSURF predated the release of Fermi GPUs, and suffered this problems too, but an updated version of CUDPP is available. Replacing the included CUDPP with the newest version suffices for this part.

Chapter 4

Experiment Results

4.1 Setup

4.1.1 GPUSURF settings

For this AR application, it is necessary to strike a balance between robustness and processing speed. The GPUSURF parameters used for this AR system, in both map file creation and main AR loop, are as follows:

- 8 octaves, the maximum possible for GPUSURF.
- 9 intervals per octave, the maximum possible for GPUSURF. This is assumed to mean how many scale levels to be subdivided in each octave as mentioned in the original SURF paper [33].
- Interest operator threshold of 0.2.
- First octave scale of 2 - same as default. Any smaller will result in too many keypoints with scale too small to be useful for keypoint matching.
- Descriptor computation enabled, which is necessary for matching.
- Orientation computation enabled. The user moves the camera freely including rotation, so we need the rotational invariance. According to our tests, even without rotating the camera sideways, leaving orientation computation enabled returned better pose recovery results.

The parameters are chosen through experimentation in indoor and outdoor environments. The online documentation of GPUSURF only provides very brief descriptions on each of the runtime parameters, and there are no detailed explanation in their paper [23]. For the interest operator threshold, when this value is increased, fewer interest points are detected. Neither the GPUSURF author's paper [23] nor the original SURF paper [60] explained what the thresholds exactly mean. To improve matching performance with different distances between camera and 2D mapped surfaces, the number of octaves and intervals are set to the maximum supported.

4.1.2 RANSAC pose recovery settings

In this AR system, the number of RANSAC iterations was chosen at 64 in order to allow for a balance between stability of pose recovered and frame rate. Any higher number of RANSAC iterations will reduce frame rate without significant improvement of recovered pose's stability. The computational time for pose recovery portion is roughtly proportional to the number of RANSAC iterations used.

4.2 Indoor Experiments

4.2.1 Scenario 1: Bulletin Board in Staircase

Our first evaluation took place in the stairwell of a university campus with a highly occupied bulletin board. The 3D map (Figure 4.1) was taken on three surfaces: the bulletin board itself, the doors next to the bulletin board, and up one flight of stairs.

The number of GPUSURF features and 2D-3D correspondences over each frame of the test sequence are indicated in Figure 4.4.

The bulletin board is an ideal target to match 2D-3D correspondences, where the different scales of the interest points in both the 3D map and the camera image allowed correspondences to be found, whether the camera was close or far away from the bulletin board. Figure 4.2d shows successful pose recovery under motion blur which resulted from the camera motion.

Facing the doors without the bulletin board visible (Figures 4.3d), the recovered pose was unstable as very few 3D points from the door could be matched. However when a few 3D points on the bulletin boards were matched too (Figure 4.3e), the recovered pose became much more stable.

On the flight of stairs (Figure 4.2a, 4.2b), the metal railings look approximately planar when viewed from a distance, allowing that surface to be recorded to the 3D map and a pose to be recovered. However when the camera is extremely close to the railings while facing in an angle (Figure 4.2c), pose recovery failed as less than 4 2D-3D correspondences were matched.

4.2.2 Scenario 2: Inside a Lab

Our second evaluation, in a robotics lab, captured 3D points from multiple locally planar surfaces to create a more complex 3D map (Figure 4.5).

The number of GPUSURF features and 2D-3D correspondences over each frame of the test sequence are indicated in Figure 4.8.

Among the silver numbered boxes (Figure 4.6a, 4.6b), only the interest points that lie on the black numbers were useful for matching, and the interest points that lie between the boxes had to be manually deleted during 3D map creation. With only the silver numbered boxes on the 3D map, the recovered pose was very unstable, but adding the bricks on the back improved the stability of the pose considerably, but not perfectly.

The back wall with a movie poster, a T-shirt, and a Canadian flag (Figures 4.6d, 4.7a, 4.7b) allowed good matching performance from close-up and far away distances.

On the wall with the doors (Figures4.7c-4.7f), only a few interest points from a small spot on the doors were able to be matched, and none on the nearby walls. This resulted in very poor pose recovery and was very unstable, and increasing the number of RANSAC iterations did not help.

4.2.3 Scenario 3: Kitchen

In this kitchen, we captured the wall with a calendar, fortune poster, two sides of the refrigerator, the cabinet around the refrigerator, the front of the stove with surrounding cabinets, and the clock into the 3D map (Figure 4.9a).

The number of GPUSURF features and 2D-3D correspondences over each frame of the test sequence are indicated in Figure 4.12.

The calendar is not conducive to feature matching due to repetitive features, but the fortune poster on the right side does permit efficient matching (Figure 4.10a-4.10c). The light reflection has a negative effect on the ability to match descriptors in Figure 4.10a. The large amount of planar objects attached to the refrigerator allowed good descriptor matching.

We did capture 3D points from the brown cabinets behind the refrigerator and around the stove, but that were nearly impossible to match. For the cabinet behind the refrigerator (Figure 4.11d), matches are only possible when the camera was directly in front of it. For the cabinet around the stove, the interest points were unable to be matched and pose recovery failed (Figure 4.11c).

4.3 Outdoor Experiments

4.3.1 Scenario 4: Outdoor brick arches

For the first outdoor evaluation, we chose a house with brick arches with irregular brick patterns. To construct a 3D map for this location (Figure 4.13a), we used the brick arches as the reference planar surface, and all interest points that did not reside in the front of the arches (such as the snow on the ground) are manually deleted.

The number of GPUSURF features and 2D-3D correspondences over each frame of the test sequence are indicated in Figure 4.16.

These bricks did not allow large-scale interest points to be detected, limiting the maximum distance to conduct pose recovery, where Figure 4.14a was the furthest distance from the arches to extract camera pose.

When the camera tilted upwards as shown in Figure 4.14d, the camera reduced its exposure and increased shutter speed, causing the image to darken, and no interest points were available to be matched to the 3D map, and a pose could not be computed.

4.3.2 Scenario 5: Outside house

In the second outdoor scene we took the front of the house and the wooden fence as the target planar surfaces to be captured to the 3D map as in Figure 4.17a.

The number of GPUSURF features and 2D-3D correspondences over each frame of the test sequence are indicated in Figure 4.20.

The house could be tracked from as far as the sidewalk (Figure 4.18a, 4.18b), but there were abundant translation errors in the z-position while tracking the front of the house, with the virtual objects constantly bobbing up and down. However the x and y-positions recovered were accurate.

The recovered pose were much more stable when many 3D points from the wooden fence were matched (Figure 4.19e, 4.19f), but the fence did not allow much distance for tracking. We already attempted to increase the possible tracking distance of the fence during the 3D map creating step, by placing the camera at a distance away from the fence such that the width of the fence took only half the width of the camera image.

4.4 Computation Times

The following tables details the computation time for each processing step of the AR system within each frame. In each test sequence, 3D maps and raw video from the camera are recorded from the scene. Performance data for each sequence are generated by running the AR system using these videos as the ”camera input”. The overall frame rates are averaged over the entire video sequence.

	Staircase	Lab	Kitchen
Number of frames processed	484	229	327
Number of points in 3D map	1987	2328	2355
Number of 2D-3D matches	(Fig. 4.4)	(Fig. 4.8)	(Fig. 4.12)
Image undistortion and grayscale conversion (ms)	1.9	1.9	1.8
Interest point detection and descriptor generation (ms)	29.8	28.2	30.3
Descriptor matching (ms)	4.7	4.5	5.6
Pose recovery (ms)	32.3	36.7	41.3
Graphic rendering (ms)	3.2	3.9	3.2
Overall frame rate (fps)	**12.0**	**11.3**	**10.9**

Table 4.1: Breakdown of computation times of each step for indoor scenes

	Brick Arches	Driveway
Number of frames processed	299	306
Number of points in 3D map	3024	1925
Number of 2D-3D matches	(Fig. 4.16)	(Fig. 4.20)
Image undistortion and grayscale conversion (ms)	1.9	1.9
Interest point detection and descriptor generation (ms)	36.9	34.3
Descriptor matching (ms)	10.0	6.3
Pose recovery (ms)	30.5	33.4
Graphic rendering (ms)	3.8	3.6
Overall frame rate (fps)	**10.7**	**10.7**

Table 4.2: Breakdown of computation times of each step for outdoor scenes

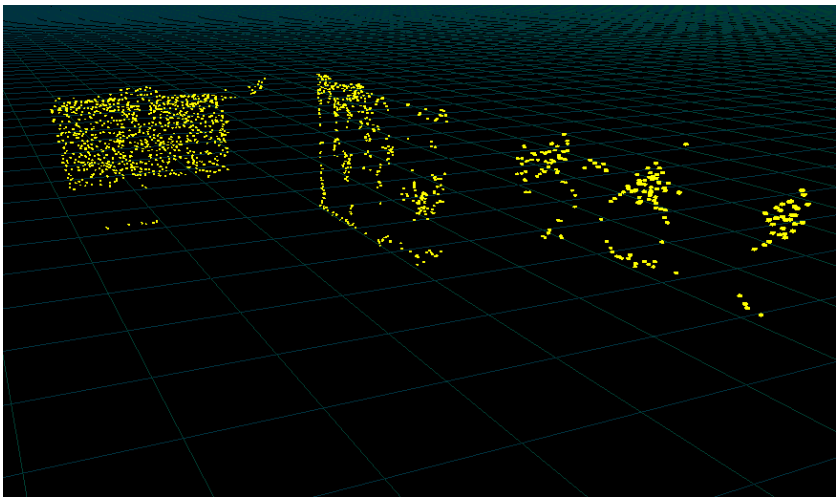
The three slowest steps within this AR system, in order from longest computation time to shortest computation times, are:

- SURF feature detection (28.2-36.9 ms)
- Pose recovery (30.5-41.3 ms)
- Descriptor matching (4.5-10 ms)

In the outdoor brick arches sequence, description matching took longer time to compute when compared to our other test sets. On average throughout the sequence, there were more SURF features detected in each video frame, and more 3D points in the map, when compared our other test sets.

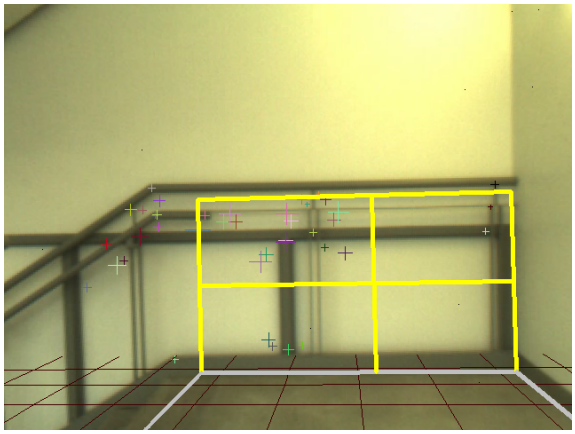
In both outdoor sets, GPUSURF feature detection were slightly slower than our indoor sets. One possible explanation was the higher average number of GPUSURF features detected in both outdoor sequences.

The virtual objects were projected onto the display by multiplying their 3D coordinates with the camera and projection matrices on the GPU, and drawn using OpenCV's HighGUI functions.

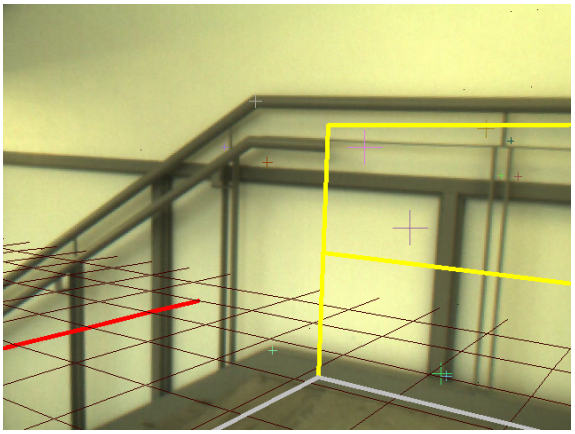


(a)

Figure 4.1: 3D map of the campus staircase.

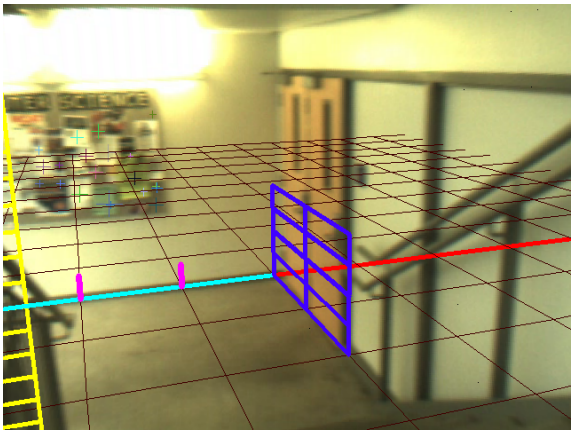


(a)



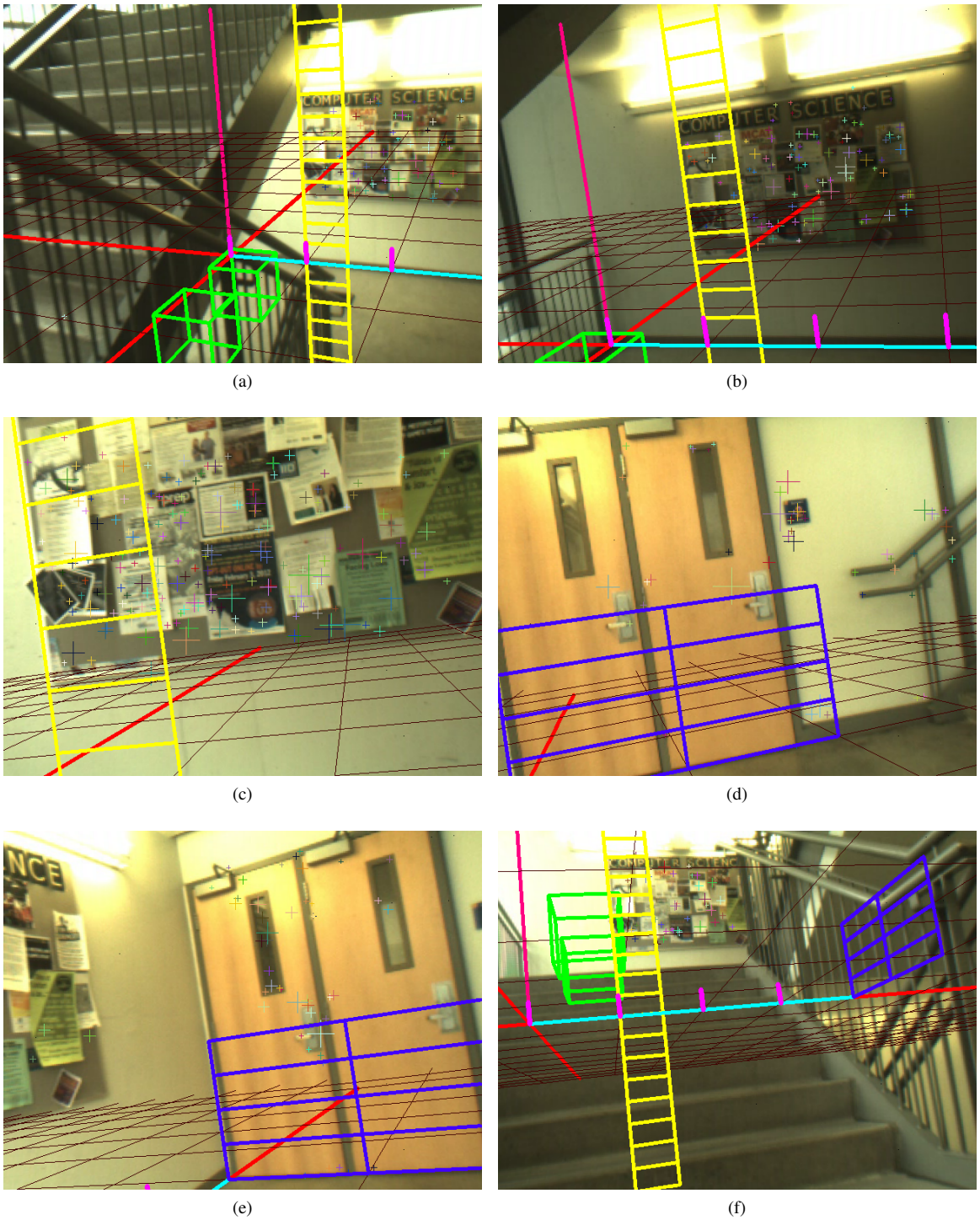
(b)

(c)



(d)

Figure 4.2: Augmented results in the campus staircase. 4.2a: Frame 14. / 4.2b: Frame 56. / 4.2c: Frame 65. / 4.2d: Frame 78.



(a) (b) (c) (d) (e) (f)

Figure 4.3: Augmented results in the campus staircase (continued). 4.3a: Frame 100. / 4.3b: Frame 147. / 4.3c: Frame 223. / 4.3d: Frame 299. / 4.3e: Frame 318. / 4.3f: Frame 475.

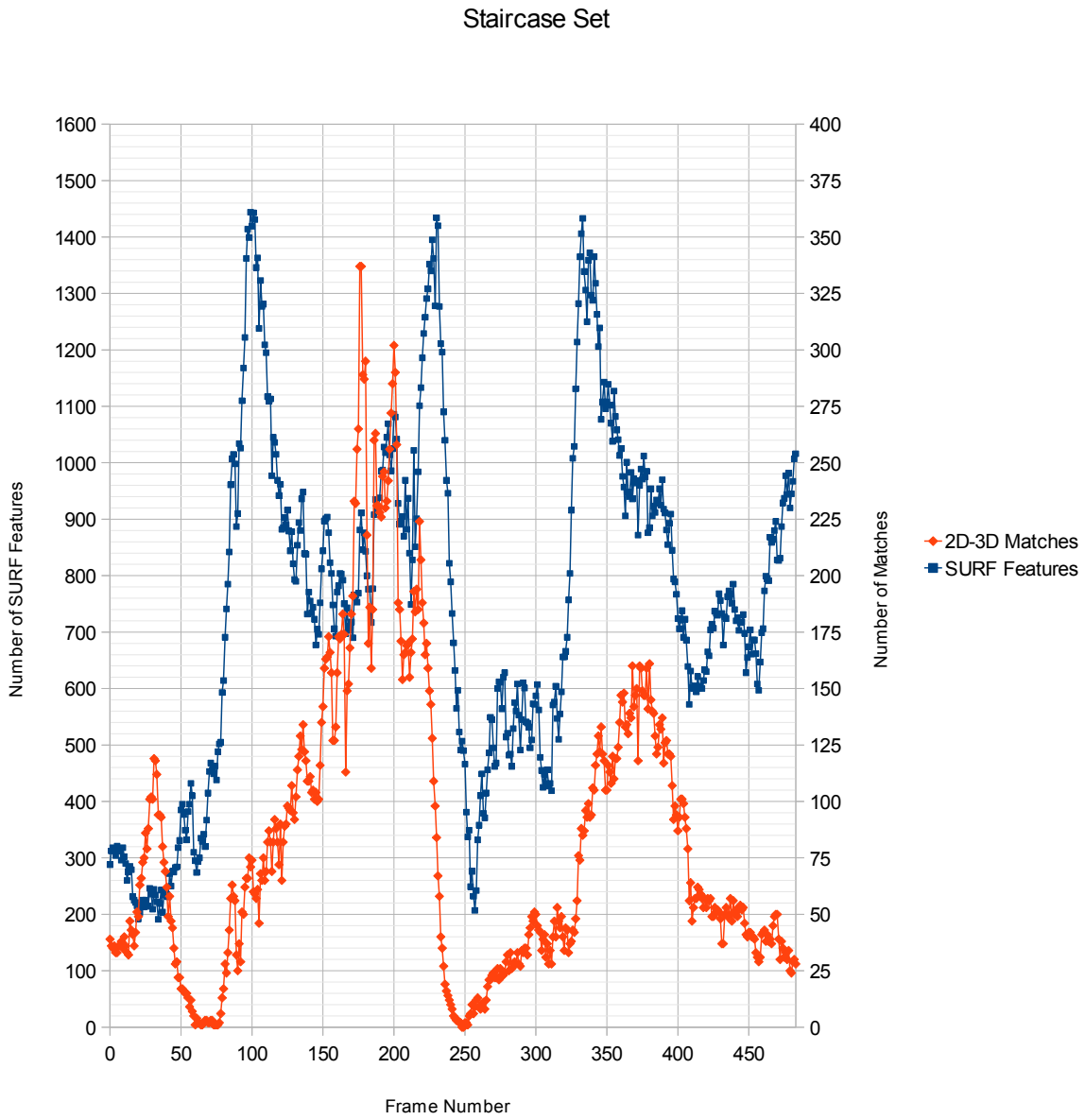
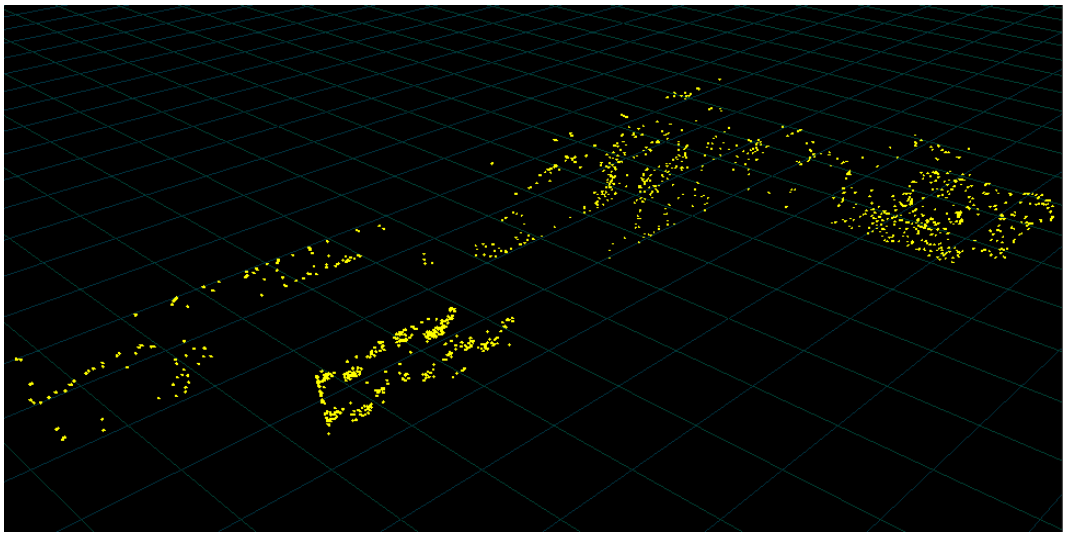
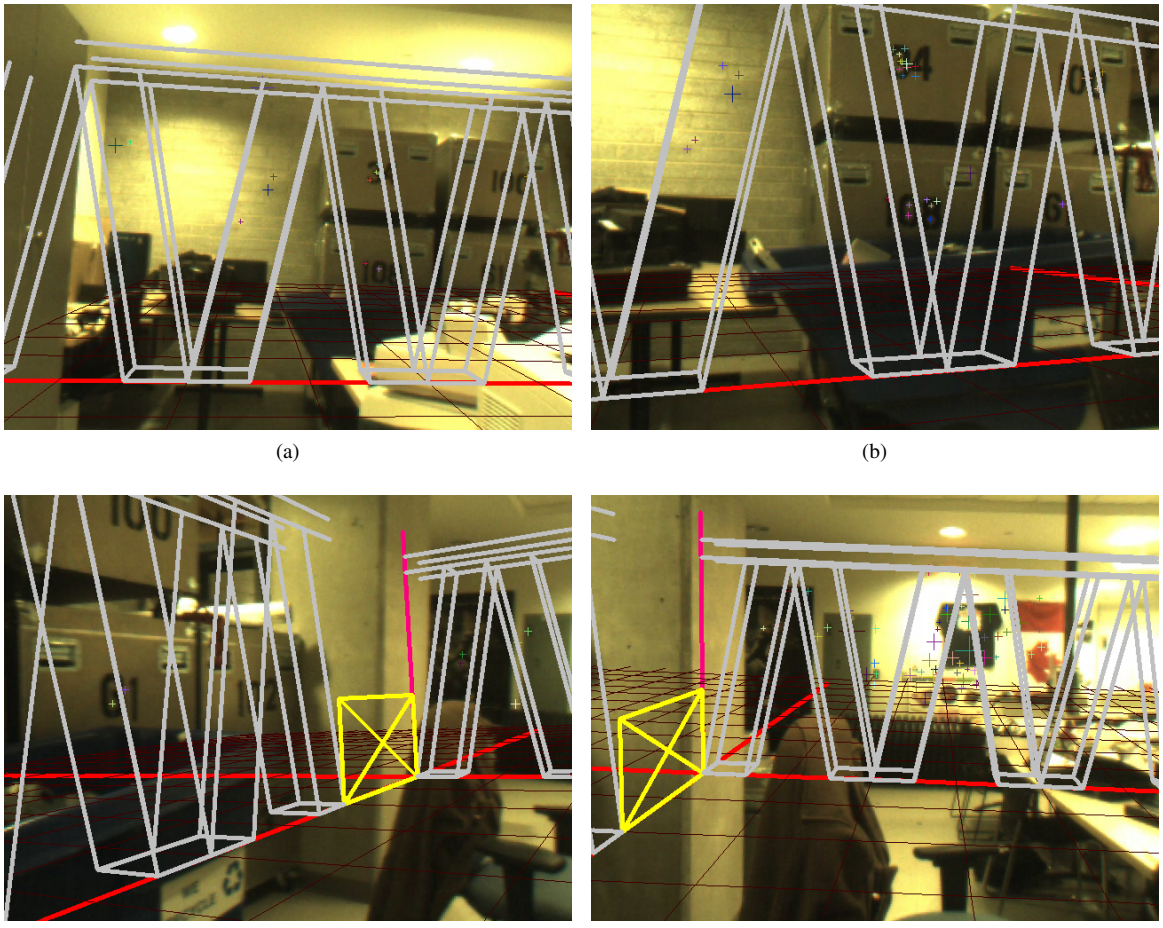


Figure 4.4: Staircase set: Number of SURF features detected (Blue) / Number of 2D-3D matches (Red)



(a)

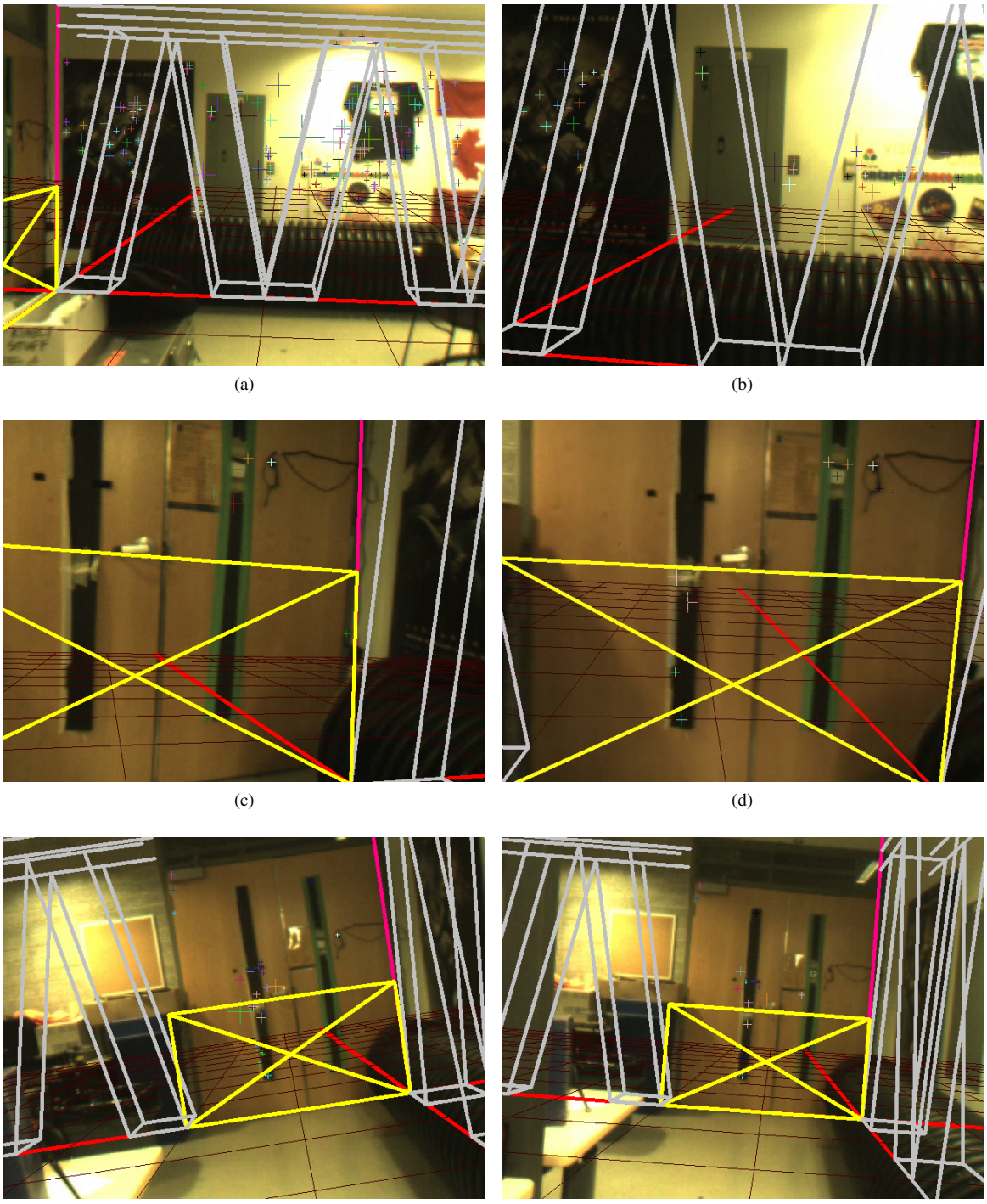
Figure 4.5: 3D map of a lab.



(a) (b)

(c) (d)

Figure 4.6: Augmented results in a lab. 4.6a: Frame 17. / 4.6b: Frame 36. / 4.6c: Frame 58. / 4.6d: Frame 70.



(a) (b) (c) (d) (e) (f)

Figure 4.7: Augmented results in a lab (continued). 4.7a: Frame 103. 4.7b: Frame 125. 4.7c: Frame 154. 4.7d: Frame 160. 4.7e: Frame 212. 4.7f: Frame 227.

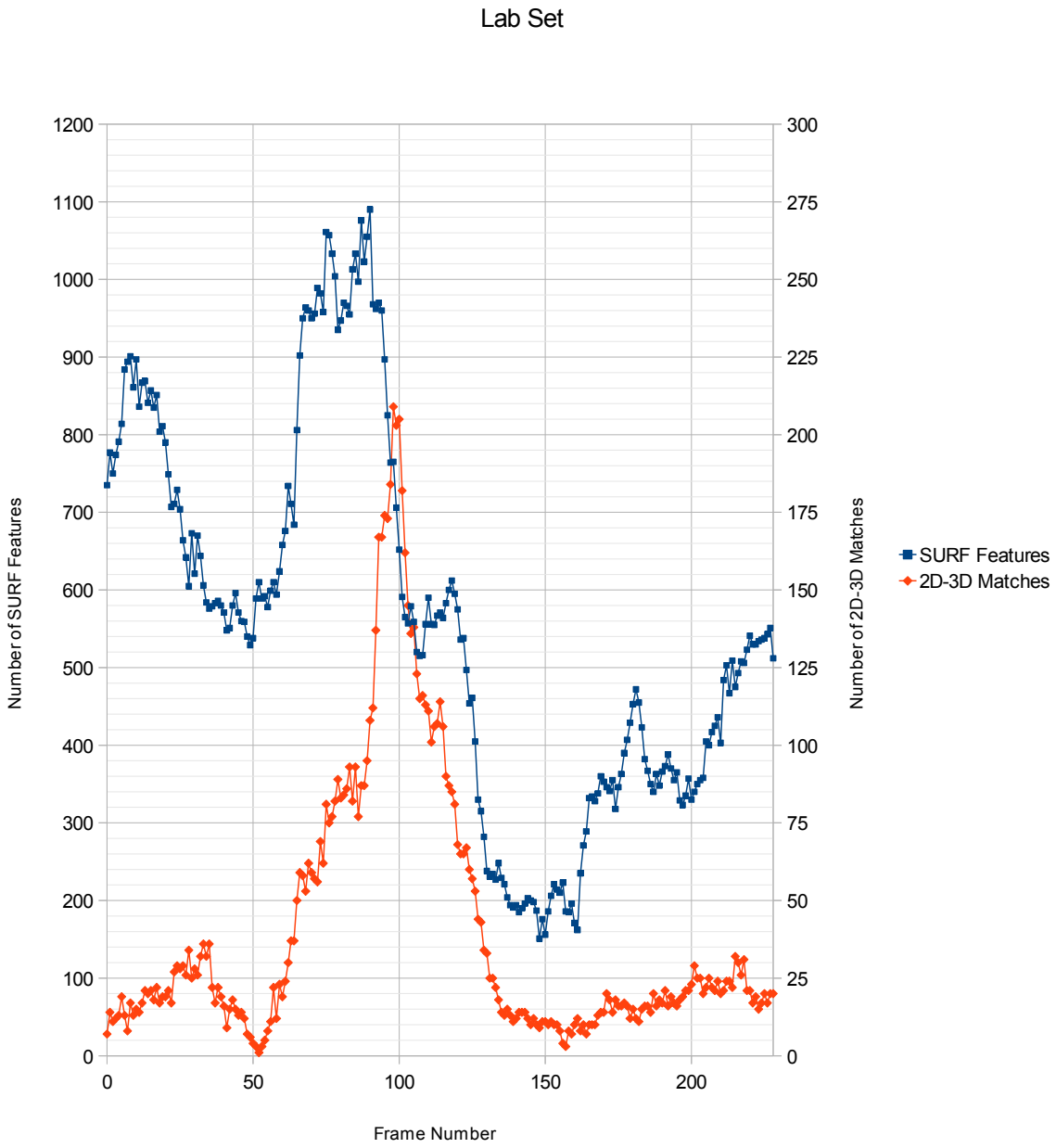
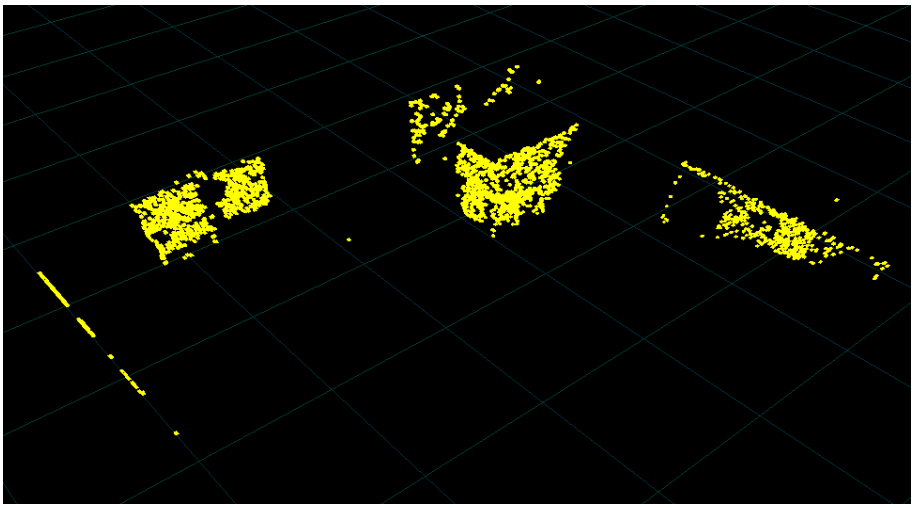
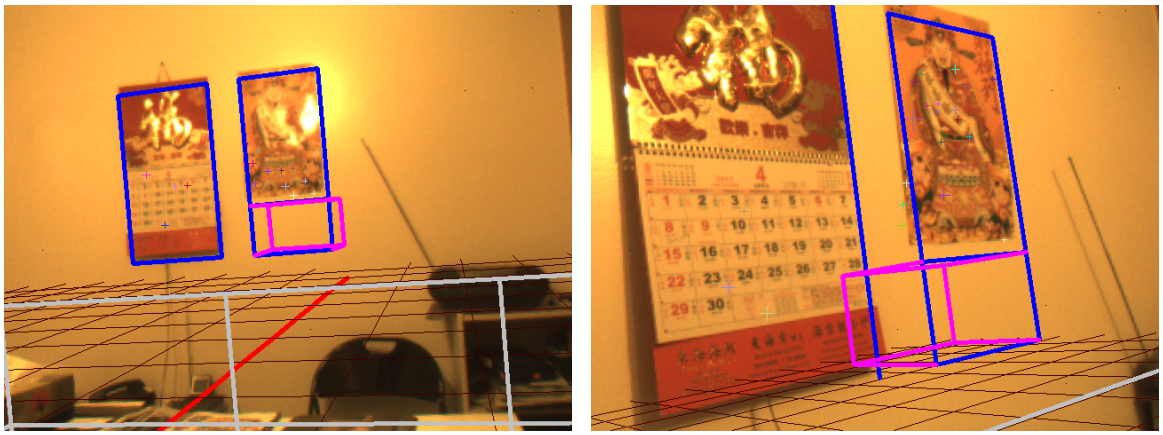


Figure 4.8: Lab set: Number of SURF features detected (Blue) / Number of 2D-3D matches (Red)

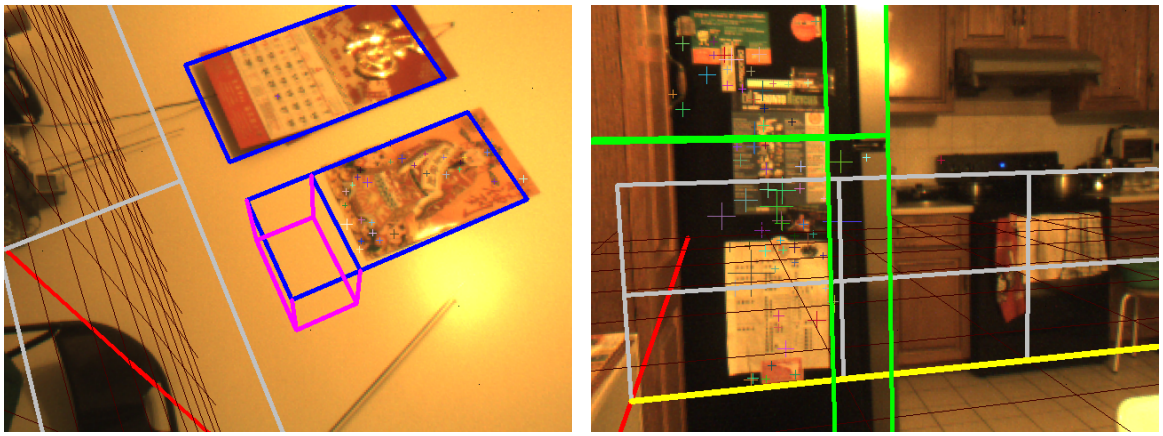


(a)

Figure 4.9: 3D map of a kitchen.

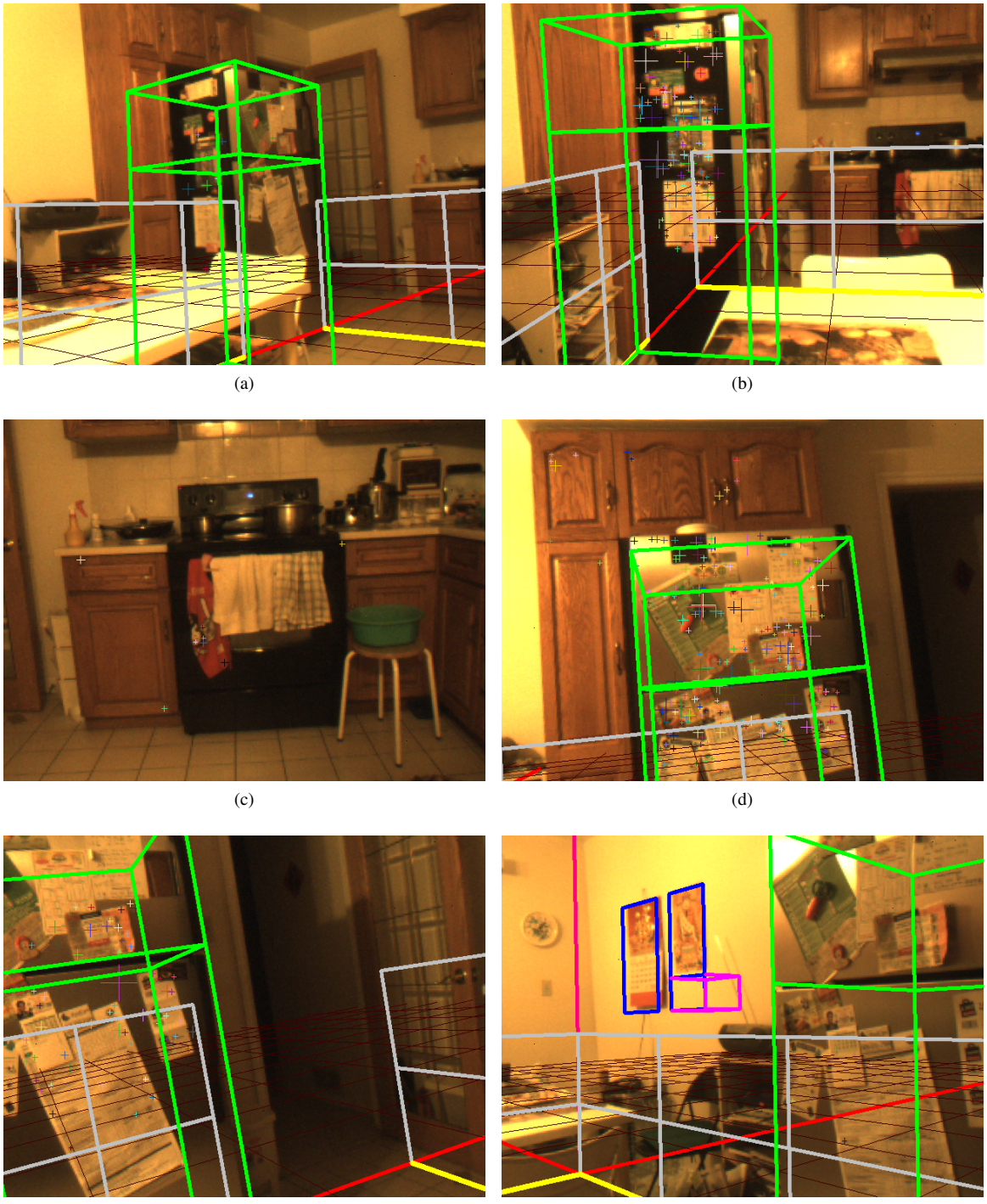


(a) (b)



(c) (d)

Figure 4.10: Augmented results in a kitchen.



(a) (b) (c) (d) (e) (f)

Figure 4.11: Augmented results in a kitchen (continued).

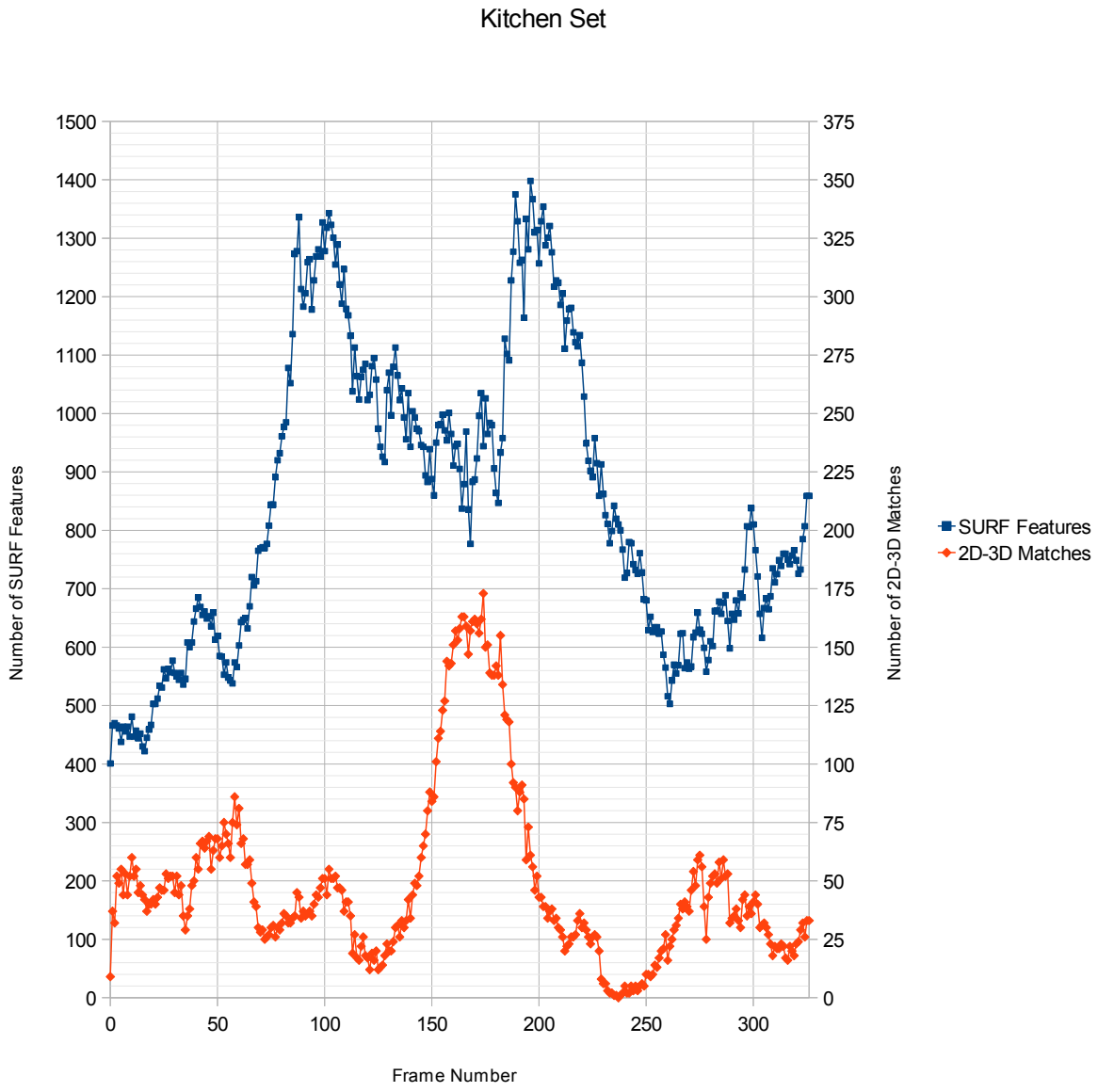
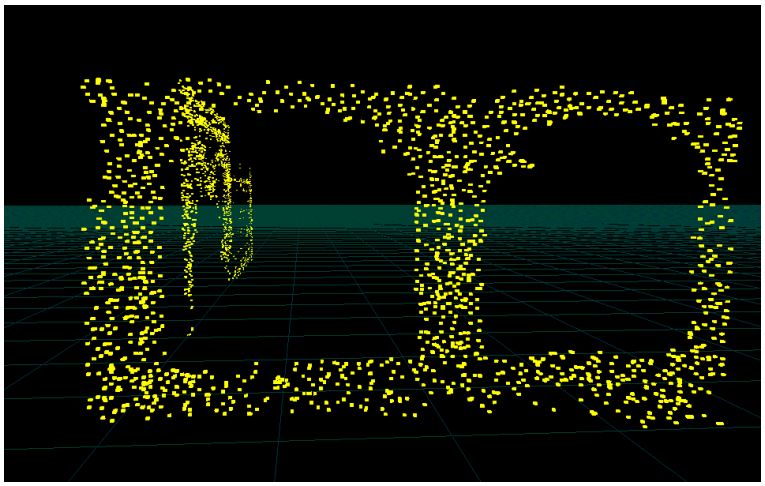
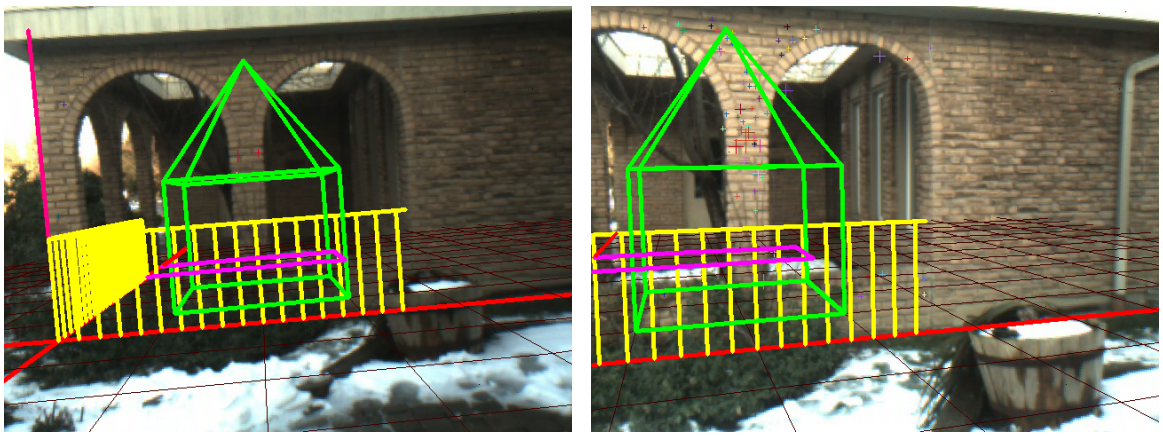


Figure 4.12: Kitchen set: Number of SURF features detected (Blue) / Number of 2D-3D matches (Red)

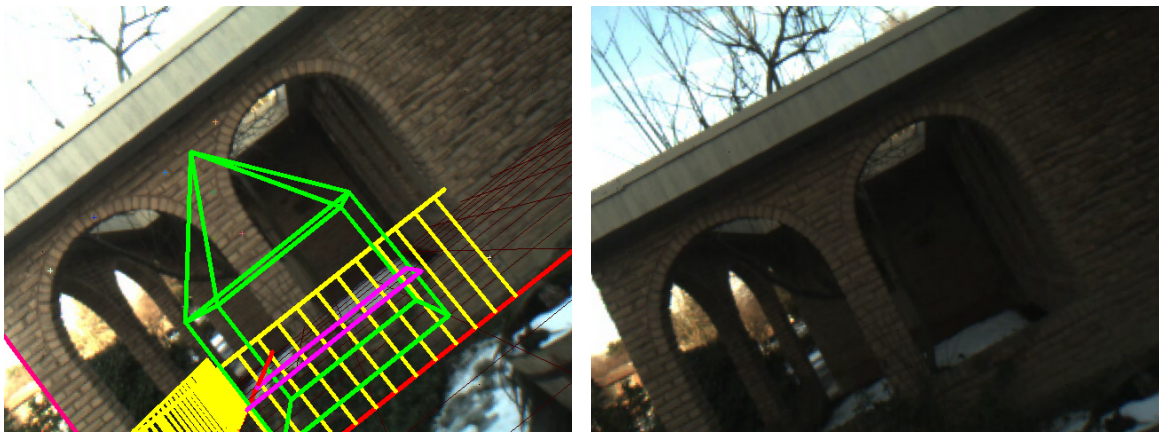


(a)

Figure 4.13: 3D map of a house.

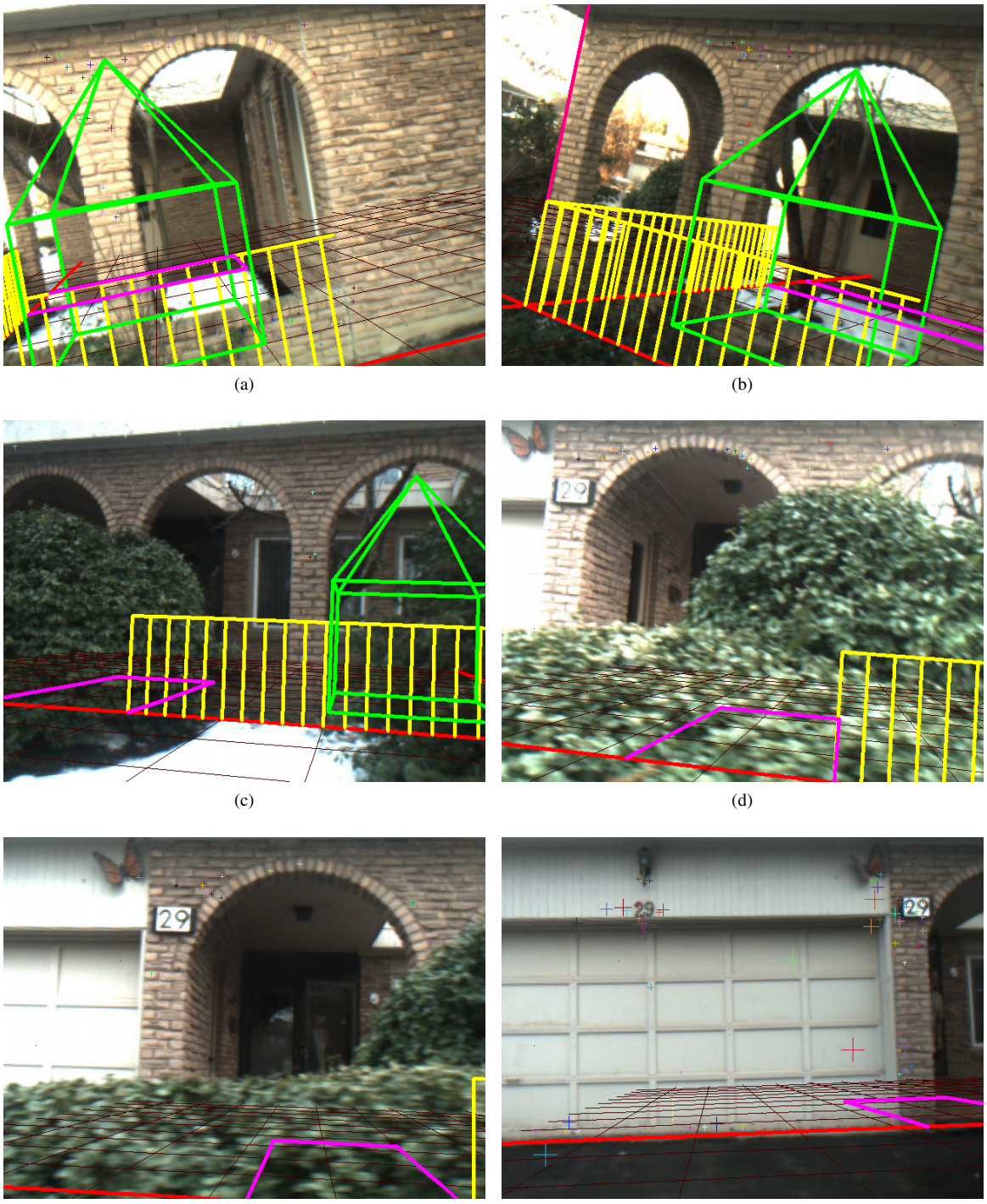


(a) (b)



(c) (d)

Figure 4.14: Augmented results in the outdoors. 4.14a: Frame 23. / 4.14b: Frame 74. / 4.14c: Frame 119. / 4.14d: Frame 135.



(a) (b) (c) (d) (e) (f)

Figure 4.15: Augmented results in the outdoors (continued). 4.15a: Frame 207. / 4.15b: Frame 281. / 4.15c: Frame 298. / 4.15d: Frame 345. / 4.15e: Frame 358. / 4.15f: Frame 432.

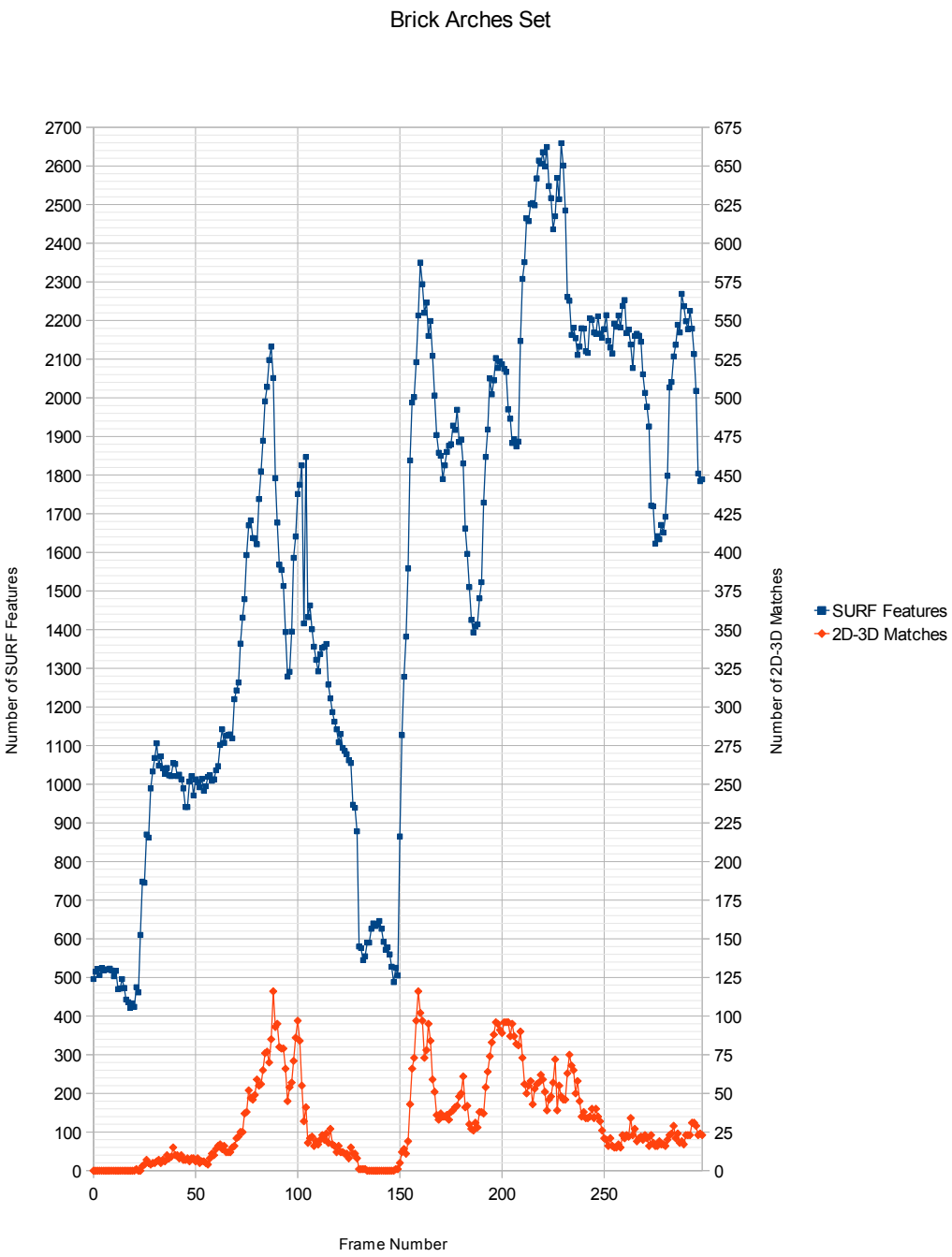
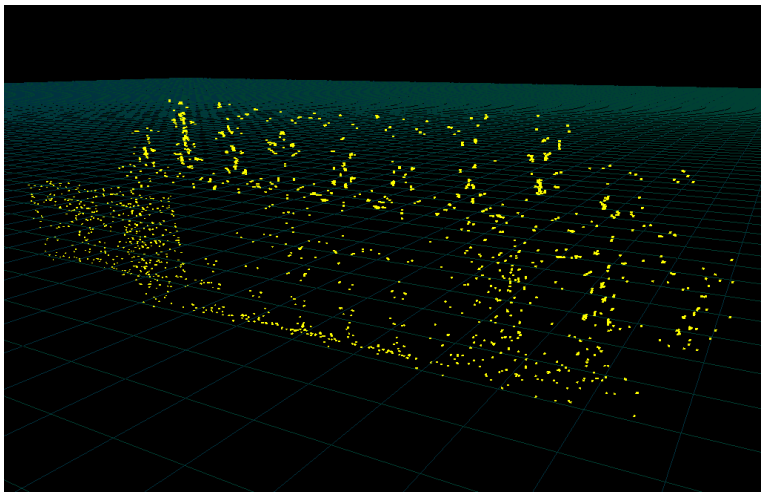
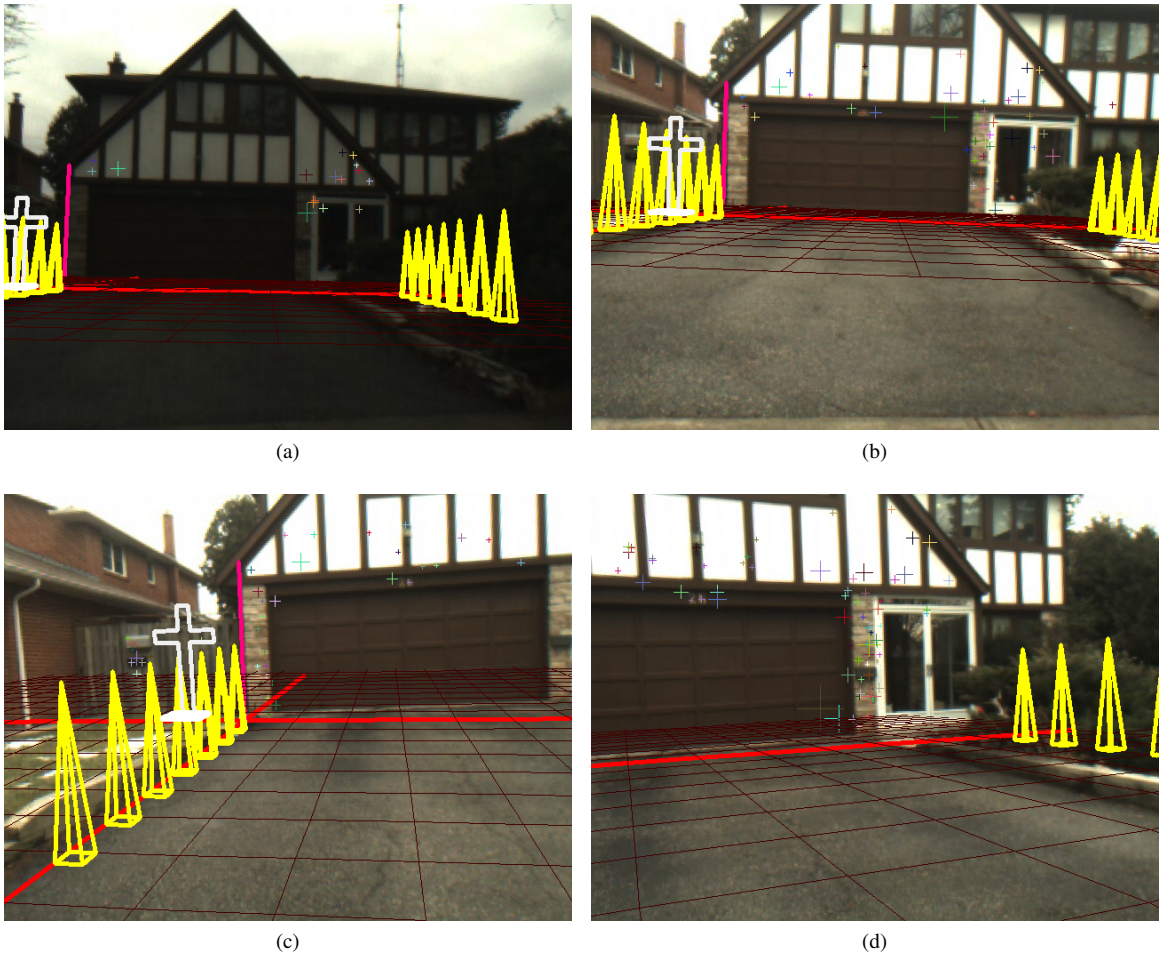


Figure 4.16: Brick arches set: Number of SURF features detected (Blue) / Number of 2D-3D matches (Red)



(a)

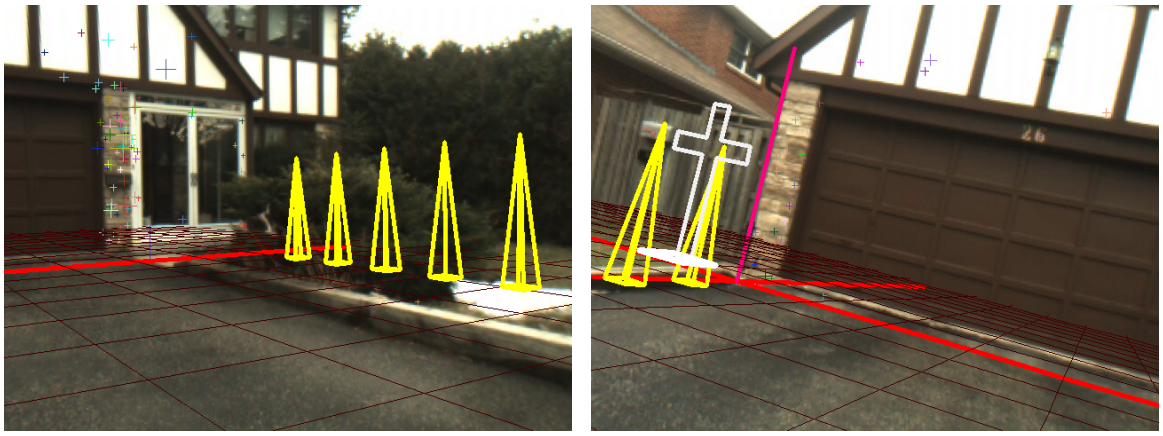
Figure 4.17: 3D map of a house.



(a) (b)

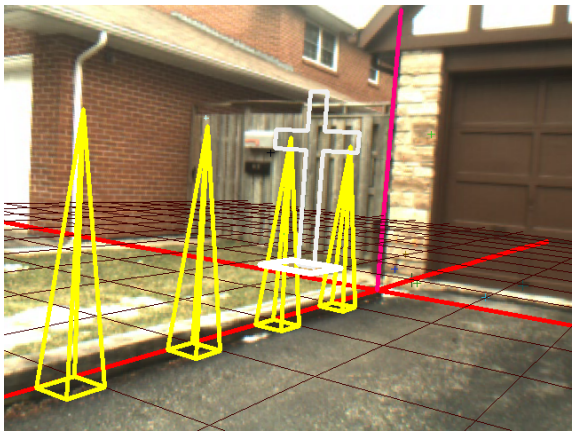
(c) (d)

Figure 4.18: Augmented results in the outdoors. 4.18a: Frame 6. / 4.18b: Frame 49. / 4.18c: Frame 93. / 4.18d: Frame 124.

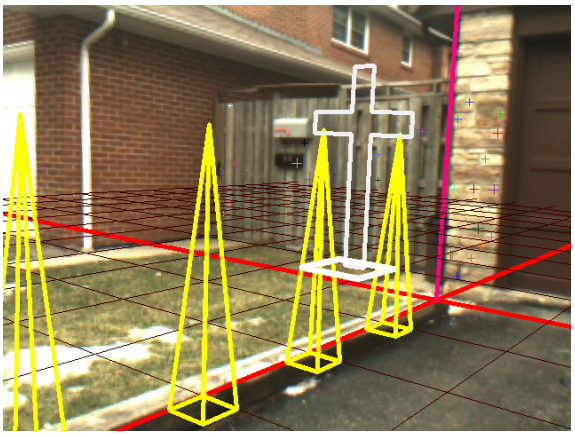


(a)

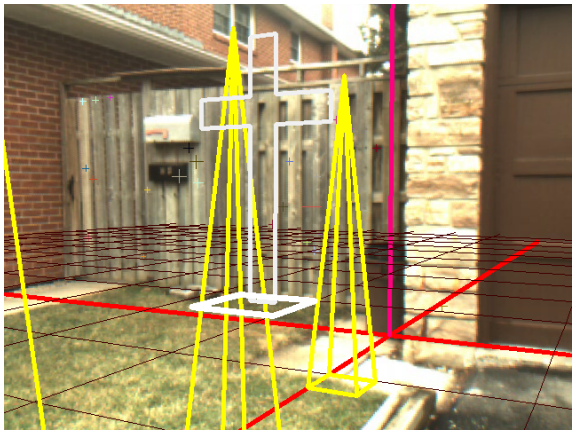
(b)



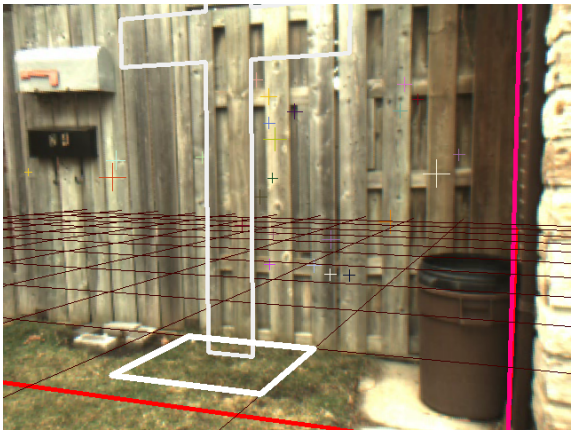
(c)



(d)



(e)



(f)

Figure 4.19: Augmented results in the outdoors (continued). 4.19a: Frame 140. / 4.19b: Frame 192. / 4.19c: Frame 214. / 4.19d: Frame 227. / 4.19e: Frame 250. / 4.19f: Frame 276.

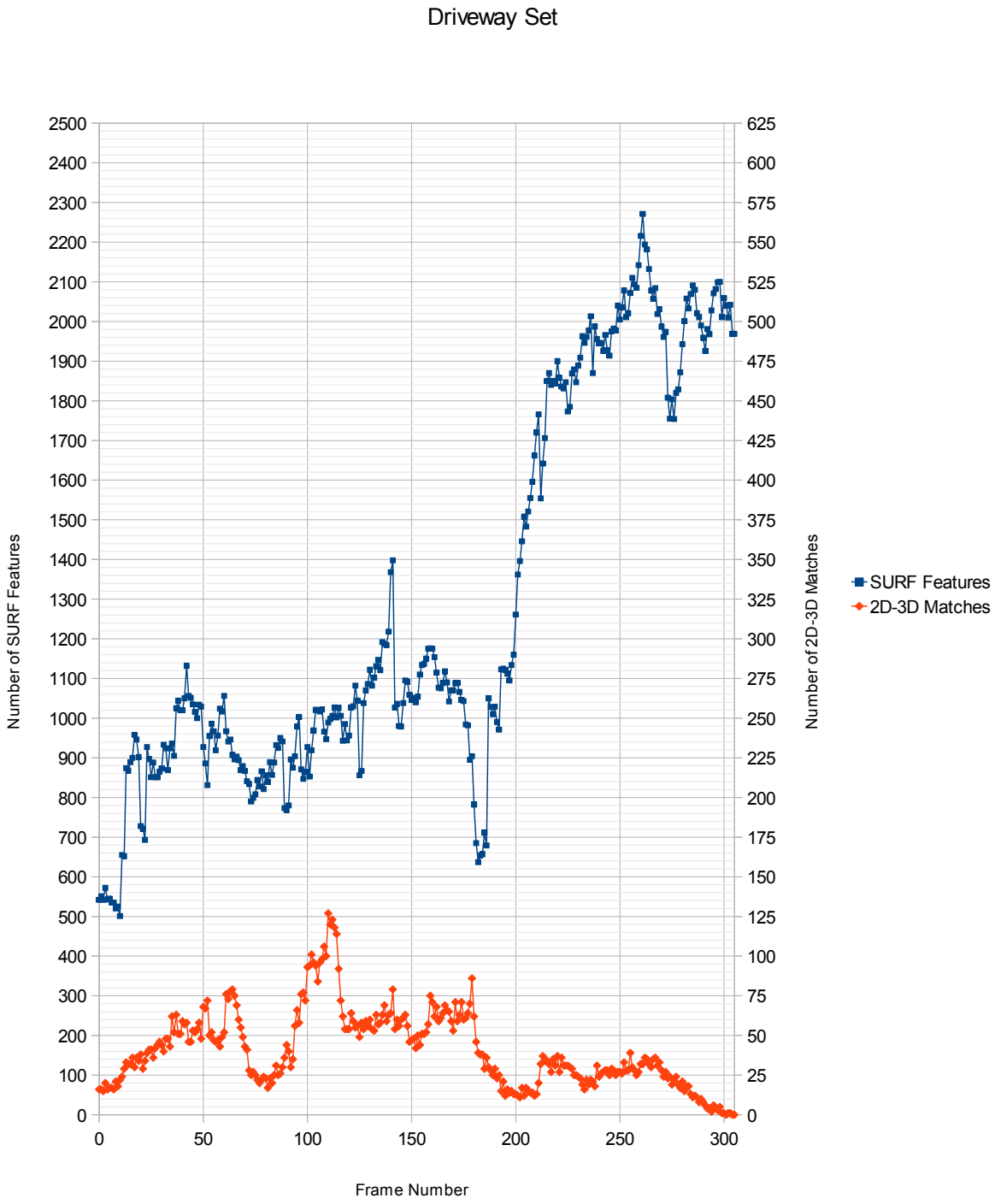


Figure 4.20: Driveway set: Number of SURF features detected (Blue) / Number of 2D-3D matches (Red)

Chapter 5

Conclusion, Discussion and Future Work

We implemented a markerless augmented reality system, running on a single CPU, single GPU laptop equipped with a single monocular RGB camera:

- Modified the source code of Speeded Up SURF (GPUSURF) [23], a GPU-accelerated CUDA implementation of SURF (Speeded Up Robust Features), to gain compatibility with Fermi and later GPUs (Section 3.7).
- Implemented GPU-accelerated brute force interest point matching, where CUBLAS library used for calculating dot products through matrix multiplication, and written new CUDA code to implement maximum dot product selector.
- Added GPU acceleration to front end image undistortion.
- Achieved consistent frame rates of 10.5-12 frames per second with 640×480 pixels resolution, using only robust algorithms without resorting to problematic optical-flow tracking as most other systems do.

We also developed the following additional support programs for our AR system:

- A camera calibration program, using OpenCV's checkerboard detection functions, to find necessary camera parameters.
- A 3D map program generator, created from one or more planar surfaces from the real world. The 3D points are captured using GPUSURF interest points from a camera image, assuming all points to lie on a plane, and their 3D world coordinates are calculated using pixel coordinates and two manually-measured reference world points at either the camera's bottom corners or midpoint of vertical edges.

5.1 Performance

The three slowest steps within this AR system, in order from longest computation time to shortest computation times, are:

- SURF feature detection (28.2-36.9 ms)
- Pose recovery (30.5-41.3 ms)
- Descriptor matching (4.5-10 ms)

We had to sacrifice some computation speed in GPUSURF to maximize its ability to match its descriptors by maximizing its scale invariance abilities. Pose recovery is a major performance bottleneck of our AR system, as the actual pose recovery computations are not GPU-accelerated. The computation time for pose recovery is approximately proportional to the number of RANSAC iterations used.

With a more powerful mobile GPU such as a GTX 580M [1], given that it has twice as many microprocessors and a slightly higher clock speed, we expect the computation time of GPU SURF can be reduced to approximately 15-19 ms, and descriptor matching times reduced to approximately 2.8-5.5, after accounting for memory transfers between host memory and GPU memory.

The projection of virtual objects were performed on the CPU using OpenCV's HighGUI functions. We expect the virtual object rendering time to be greatly cut down if this was implemented through OpenGL.

The frame rate performances in our evaluations were performed with the laptop plugged into AC power. While on battery power, the processing speed of this AR system is reduced to approximately 6 frames per seconds, due to the Quadro 2000M GPU throttling down its CUDA computing power, presumably a measure to limit power draw from the laptop's lithium-ion batteries.We were unable to find any method that can defeat this GPU performance limiter.

5.2 Limitations

As this AR system depends heavily on the ability to detect and match SURF features, scenarios with only textureless walls will not work as they do not yield matchable SURF interest points. Repetitive features, such as a block tiles or wall calendars, did not work well either, as each repetition of the feature will yield very similar descriptors that did not distinguish from each other well. Additionally, SURF descriptors are not possible to be matched when subjected to viewpoint differences in excess of 45 degrees.

Though we have demonstrated our AR system in two outdoor scenarios, neither of them were subjected to direct sunshine. Extreme difference in illumination, such as shadows cast by direct sun exposure on the reference planar surfaces, will not work, as such situations are way beyond the limits of SURF's illumination invariance capabilities, as the shadows result in completely different SURF feature detection results.

5.3 Future Work

As all GPU accelerations used in this AR system are implemented through CUDA, only NVIDIA CUDA-supported GPU can run it. However, there is a possibility to port this AR system to OpenCL in order to allow ATI GPUs to run, as an OpenCL implementation of SURF [22] is available as of this writing.

[1] `http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-580m`

Making a GPU-accelerated RANSAC pose recovery based on the OpenCV's iterative algorithm is another possibility to speed up this step even further, by having each of the RANSAC pose recovery loops assigned to a thread to be executed concurrently. These algorithms are complex and their implementation will present major challenges.

Finally, to enable the creation of 3D maps on non-planar objects, we are considering to incorporate Bundler [61] into the 3D map making process. Bundler allows 3D point clouds to be built using multiple camera images taken from different viewpoints.

Appendix A

Glossary

- AR: Augmented Reality.
- CPU: Central Processing Unit.
- GPU: Graphic Processing Unit.
- GPUSURF: Alternate name of Speeded Up SURF, a GPU-accelerated implementation of SURF by University of Toronto's UTIAS group.
- Octave: In scale-invariant feature detectors such as SIFT or SURF, 2^{octaves} is the scale of the large image compared to the smallest image within the image pyramid. For example, an image pyramid with 3 octaves perform feature detection at 1x, 1/2x and 1/4x-scaled image copies.
- OpenCV: Open Source Computer Vision Library.
- Pose: The position and orientation of the camera in a 3D space.
- Pose recovery: To calculate a camera's pose in a 3D world, for example this can be achieved using multiple pairs of 2D image point / 3D world point correspondences.
- RANSAC: RAndom Sample Consensus.
- SIFT: Scale-Invariant Feature Transform, by David Lowe.
- SURF: Speeded Up Robust Featues, by H. Bay et al.

References

[1] M. Fiala, "Artag, a fiducial marker system using digital techniques," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2, pp. 590–596, 2005.

[2] W. Piekarski and B. Thomas, "Arquake: The outdoor augmented reality gaming system," *Communications of the ACM*, vol. 45, no. 1, pp. 36–38, 2002.

[3] F. K. Wacker, S. Vogt, J. J. A. Khamene, Ali, S. G. Nour, D. R. Elgort, F. Sauer, J. L. Duerk, and J. S. Lewin, "An augmented reality system for mr image-guided needle biopsy: Initial results in a swine model," *Radiology*, vol. 238, no. 2, pp. 497–504, February 2006.

[4] G. Derry, "Avatar's augmented reality and blackmagic." [Online]. Available: http://blackmagic-design.com/community/communitydetails/?UserStoryId=9124

[5] J. Shi and C. Tomasi, "Good features to track," in *CVPR*, Seattle, Washington, USA, June 1994, pp. 593–600.

[6] H. Eltoukhy and K. Salama, *Electrical Engineering*, vol. 89, no. 10, 1999. [Online]. Available: http://scien.stanford.edu/2002projects/ee392j/eltoukhy_salama_report.pdf

[7] P. Barnum, B. Hu, and C. M. Brown, "Exploring the practical limits of optical flow." [Online]. Available: http://hdl.handle.net/1802/286

[8] K. Streib and J. W. Davis, "Extracting pathlets from weak tracking data ."

[9] G. Lee, C. Nelles, M. Billinghurst, and G. Kim, "Immersive authoring of tangible augmented reality applications," in *IEEE ISMAR 2004: Arlington, VA, USA*, Nov. 2004.

[10] O. Oda and S. Feiner, "Goblin xna: A platform for 3d ar and vr research and education." [Online]. Available: http://graphics.cs.columbia.edu/projects/goblin/

[11] "Nintendo 3ds - hardware features and overview at nintendo." June 2011. [Online]. Available: http://www.nintendo.com/3ds/hardware#/3

[12] "Ar play: Augmented reality gaming on ps vita system." [Online]. Available: http://us.playstation.com/psvita/apps/psvita-app-ar.html

[13] "Opencv: Open source computer vision library." [Online]. Available: http://opencv.org/

[14] "Next generation fermi architecture." [Online]. Available: http://www.nvidia.com/object/fermi_architecture.html

[15] Cuda c programming guide. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[16] "Nvidia kepler compute architecture." [Online]. Available: http://www.nvidia.com/object/nvidia-kepler.html

[17] www.nvidia.com/object/cuda_home.html.

[18] "Ati stream: Finally, cuda has competition," June 2009.

[19] "Simply hired: Opencl vs. cuda vs. ati stream," June 2009. [Online]. Available: http://www.simplyhired.com/a/jobtrends/trend/q-opencl%2Ccuda%2C%22ati+stream%22

[20] "Opencl - the open standard for parallel programming of heterogeneous systems." [Online]. Available: http://www.khronos.org/opencl/

[21] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *CoRR*, vol. abs/1005.2581, 2010.

[22] P. Mistry, C. Gregg, N. Rubin, D. Kaeli, and K. Hazelwood, "Analyzing program flow within a many-kernel opencl application," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 10:1–10:8. [Online]. Available: http://doi.acm.org/10.1145/1964179.1964193

[23] P. Furgale and C. H. Tong. (2010) Speeded up surf. [Online]. Available: http://asrl.utias.utoronto.ca/code/gpusurf/

[24] F.Moreno-Noguer, V.Lepetit, and P.Fua, "Accurate non-iterative o(n) solution to the pnp problem," in *IEEE International Conference on Computer Vision*, Rio de Janeiro, Brazil, October 2007.

[25] C. H. C and M. Stephens, "A combined corner and edge detector," in *Proceedings of the 4th Alvey Vision Conference*, 1988, pp. 147–151.

[26] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking." in *IEEE International Conference on Computer Vision*, vol. 2, October 2005, pp. 1508–1511. [Online]. Available: http://mi.eng.cam.ac.uk/~er258/work/rosten_2005_tracking.pdf

[27] ——, "Rapid rendering of apparent contours of implicit surfaces for realtime tracking," in *British Machine Vision Conference*, June 2003, pp. 719–728. [Online]. Available: http://mi.eng.cam.ac.uk/~er258/work/rosten_2003_rapid.pdf

[28] P. Mainali, Q. Yang, G. Lafruit, L. Van Gool, and R. Lauwereins, "Robust low complexity corner detector," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 4, pp. 435–445, 2011.

[29] K. Mikolajczyk and C. Schmid, "Scale & affine invariant interest point detectors," *International Journal of Computer Vision*, vol. 60, no. 1, pp. 63–86, 2004.

[30] M. Gevrekci and B. Gunturk, "Illumination robust interest point detection," *Computer Vision and Image Understanding*, vol. 113, no. 4, pp. 565–571, 2009.

[31] J. Banks and P. Corke, "Quantitative evaluation of matching methods and validity measures for stereo vision," in *The International Journal of Robotics Research*, July 2001, pp. 512–532.

[32] D. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the IEEE International Conference on Computer Vision*, 1999, pp. 1150–1157.

[33] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (surf)," *Computer Vision and Image Understanding (CVIU)*, vol. 110, no. 3, pp. 346–359, 2008.

[34] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, Sept. 1975.

[35] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *J. ACM*, vol. 45, pp. 891–923, November 1998.

[36] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *International Conference on Computer Vision Theory and Application VISSAPP'09)*. INSTICC Press, 2009, pp. 331–340.

[37] P. Azad, T. Asfour, and R. Dillmann, "Combining harris interest points and the sift descriptor for fast scale-invariant object recognition," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4275–4280, 2009.

[38] C. Wu, "Siftgpu: A gpu implementation of scale invariant feature transform (sift)." [Online]. Available: http://cs.unc.edu/~ccwu/siftgpu/

[39] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg, "Pose tracking from natural features on mobile phones," *Proceedings - 7th IEEE International Symposium on Mixed and Augmented Reality*, pp. 125–134, 2008.

[40] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *Computer Vision (ICCV), 2011 IEEE International Conference on*, nov. 2011, pp. 2564 –2571.

[41] D. Rodrguez and N. Aouf, "Robust harris-surf features for robotic vision based navigation," *IEEE Conference on Intelligent Transportation Systems*, pp. 1160–1165, 2010.

[42] M. Fischler and R. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography." in *Communications of the ACM*, 1981, pp. 381–395.

[43] Opencv 2.3.1 documentation: Camera calibration and 3d reconstruction. [Online]. Available: http://opencv.itseez.com/modules/gpu/doc/camera_calibration_and_3d_reconstruction.html#gpu-solvepnpransac

[44] G. Simon, A. Fitzgibbon, and A. Zisserman, "Markerless tracking using planar structures in the scene," in *Proc. International Symposium on Augmented Reality*, Oct. 2000, pp. 120–128. [Online]. Available: http://www.robots.ox.ac.uk/~vgg

[45] I. Skrypnyk and D. Lowe, "Scene modelling, recognition and tracking with invariant image features," *ISMAR 2004: Proceedings of the Third IEEE and ACM International Symposium on Mixed and Augmented Reality*, pp. 110–119, 2004.

[46] A. Lee, J.-Y. Lee, S.-H. Lee, and J.-S. Choi, "Markerless augmented reality system based on planar object tracking," in *Frontiers of Computer Vision (FCV), 2011 17th Korea-Japan Joint Workshop on*, feb. 2011, pp. 1 –4.

[47] B. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision. in imaging understanding workshop." in *Proc. 7th International Joint Conference on Artificial Intelligence (IJCAI'81)*, 1981, pp. 674–679.

[48] G. Klein and D. Murray, "Parallel tracking and mapping for small ar workspaces." in *ISMAR*, 2007, pp. 225–234.

[49] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking." in *IEEE International Conference on Computer Vision*, vol. 2, Oct 2005, pp. 1508–1511.

[50] T. Lee and T. Hollerer, "Hybrid feature tracking and user interaction for markerless augmented reality," in *Virtual Reality Conference, 2008. VR '08. IEEE*, march 2008, pp. 145 –152.

[51] C. Arth, M. Klopschitz, G. Reitmayr, and D. Schmalstieg, "Real-time self-localization from panoramic images on mobile devices," *Mixed and Augmented Reality, IEEE / ACM International Symposium on*, vol. 0, pp. 37–46, 2011.

[52] S. Hollister, "Ps vita 'magnet' tech does augmented reality without the cards," 3 2012. [Online]. Available: http://www.theverge.com/2012/3/8/2853770/ps-vita-ar-magnet-gdc-2012

[53] Opencv v2.4.1 documentation: Image processing. [Online]. Available: http://opencv.itseez.com/modules/imgproc/doc/geometric_transformations.html

[54] "Coding bilinear interpolation." [Online]. Available: http://supercomputingblog.com/graphics/coding-bilinear-interpolation/

[55] Opencv v2.4.1 documentation: Miscellaneous image transformations. [Online]. Available: http://opencv.itseez.com/modules/imgproc/doc/miscellaneous_transformations.html#cvtcolor

[56] "Cublas library." [Online]. Available: http://developer.nvidia.com/cublas

[57] "Open source comptuer vision library." [Online]. Available: http://www.intel.com/research/mrl/research/opencv

[58] "Cuda fermi compatibility guide." [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Fermi_Compatibility_Guide.pdf

[59] "cudpp - cuda data parallel primitives library." [Online]. Available: http://code.google.com/p/cudpp/

[60] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Surf: Speeded up robust features." in *Computer Vision and Image Understanding (CVIU), Vol. 110, No. 3*, 2008, pp. 346–359.

[61] R. S. Noah Snavely, Steven M. Seitz, "Photo tourism: Exploring image collections in 3d," in *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, 2006.