SCAVENGAR HUNT

Abhinav Ganesh: aganesh@andrew.cmu.edu
Krishna Vudata: kvudata@andrew.cmu.edu

Group #7
18-551 Spring 2012

Table of Contents

1. Project Overview

1.1 Introduction

Augmented Reality is pushing the bounds of integrating what is computer generated with what we experience on an everyday basis. This concept has been introduced in video games and movies as having many futuristic applications but has yet to be relevant to real world problems. Until recently, augmented reality has generally been an impractical technology that can only be used with unwieldy headsets and visors backed by clunky hardware that is unable to handle the rigors of modern day computing. We intend to bring augmented reality to a more practical mindset and take advantage of its potential by utilizing the portability and computing power of our Android powered tablets to bring a more real time and useful user experience. We demonstrated the practical use of augmented reality in the form of a mobile game. The concept of our game is a scavenger hunt where one user marks locations and leave clues while other users must find these clues by viewing them on their mobile devices. These clues only exist virtually and are displayed as an overlay on real world artifacts.

1.2 Problem/Motivation

Augmented reality essentially involves finding some landmark or marker and then using the location of this marker to overlay video with realistic augmentations e.g. insert objects and pictures which seem realistic but do not actually exist. We feel that in order to make augmented reality a more practical technology we must move away from classical augmented reality technology that uses specialized preset markers like the one in Figure 1. Preset markers don't provide the flexibility or immersive user experience that natural landmarks can. Given the relatively new nature of markerless augmented reality, we would like to forge our own path in the domain by creating a fun and challenging game that showcases the potential everyday applicability of augmented reality.

Figure 1 – A typical AR marker used by traditional AR technologies

1.3 Novelty/Previous Work

Though many groups in the past have done projects relating to pattern recognition, we are looking to apply this in a very different way. Due to the increase in flexibility and computing power (inherent with having a tablet and the Android platform) we are able to execute more processes in real-time. Due to limitations with hardware in previous years, groups were not able to experiment with up and coming technologies like augmented reality. Our idea of exploring this technology in the form of an interactive game also provides a unique and fun twist to our project. In addition, our usage of natural landmarks is novel because many AR technologies are still limited to using preset standardized markers.

1.4 Methodology for Solution

The main problem involved in performing augmented reality is finding and recognizing the natural landmark i.e. object detection. There are several different algorithms which perform object detection, the following subsections provide a high level summary of how the object detection algorithms were tested and compared.

1.4.1 Data Set

In order to test our object detection algorithms, we compiled a set of images and videos of landmarks in various camera poses. These landmarks were chosen to be items that would be found in a natural setting (that a user might want to use in a scavenger hunt). Hence these landmarks need to be detailed enough that they can be recognized and distinct. The types of landmarks we chose were posters, labeled objects (i.e. ketchup bottle), and distinct patterns. Our choices were also motivated by having prior knowledge as to what types of landmarks worked well with the QCAR framework (this was explored in Lab 3).

Here a few samples of the different landmarks we tested on.

Figure 2

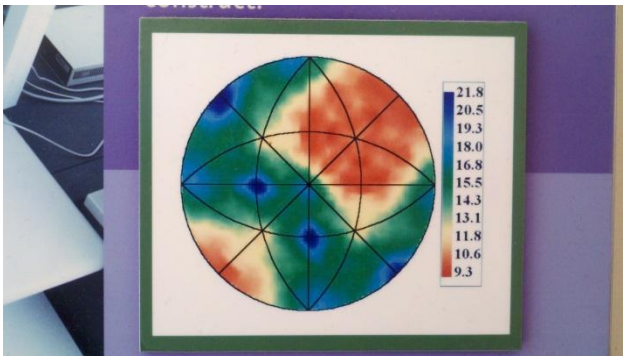

Figure 3

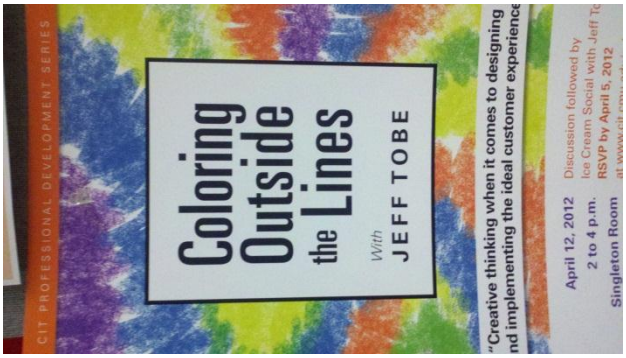

Figure 4

1.4.2 Breakdown of the Code Base

The object detection and tracking algorithms that we used were written in C++. We greatly leveraged the OpenCV C++ library because of our familiarity with the library (due to prior labs) and due to the numerous object detection algorithms it provides. OpenCV is also a very active open source library making it a very well tested and reliable source. The code we implemented used the various image processing and object detection algorithms as building blocks to construct an augmented reality application. We will go into more detail later on as to how these OpenCV functions were combined.

The user interaction and game logic of the Scavenger Hunt was implemented on the Android platform. The Android platform provided us with an easy way to build a user interface on top of the native detection and tracking. More precisely, Android provides a Java UI framework that we used to construct the Scavenger Hunt application.

We also utilized Adobe Photoshop to design some of the custom UI elements in our application (such as backgrounds, widgets, and icons).

1.4.3 Method

The objective of our testing was to determine which object detection algorithms to use. Object detection is a process comprised of **keypoint detection**, **feature extraction**, **feature matching,** and **match grading**. There are several different ways to implement each of these steps. In order to test these various implementations we mixed and matched algorithms across these key steps and gathered performance statistics such as frame rate and time taken. In the end there is also a qualitative assessment (of the augmented reality experience) necessary to choose which implementations would work best and provide the best user experience.

2. Object Detection

Object detection is the task of detecting instances of semantic objects of a certain class (Object Detection, 2011). In essence this is the main logic in an augmented reality application. In relation to our scavenger hunt we need to be able to detect the landmarks specified by a user so we can augment them appropriately.

For the purposes of this discussion, the *trackable* refers to the image of the landmark that we are trying to detect and the *frame* is the image we are searching in. As mentioned previously, object detection can be broken down into multiple steps. .

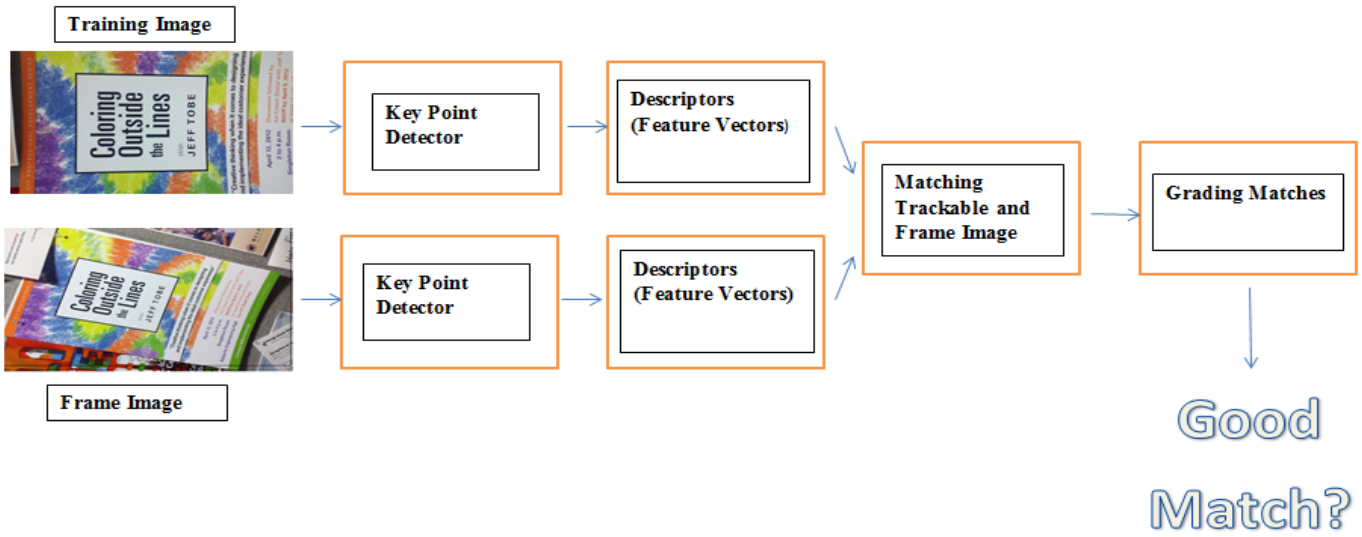

Figure 5 – Object Detection Pipeline

Figure 5 outlines these steps as a dataflow pipeline. The first step of object detection is to reduce the *trackable* and the *frame* to a set of meaningful points that describe the unique features of the image (**keypoint detection**). The next step is to describe these meaningful points as a feature vector (**feature extraction**). The next step is to search for the *trackable* in the *frame* by matching the *trackable* feature vectors with the *frame* feature vectors (**matching**). The final step is to rate

the matches and determine if we have successfully found the *trackable* (**match grading**). Now we will get into the details of each of these steps.

2.1 Keypoint Detection

Keypoint detection is the process of reducing the *trackable* and the *frame* to a set of interesting points that specify the location of unique features in the image.

The algorithm that we decided to use was the SURF (Speeded Up Robust Feature) detector. The specifics of how SURF works are outside the scope of this paper, but the general idea is that SURF determines points of interests or keypoints by looking for distinct locations in the image such as corners, blobs, and t-junctions (Bay, Tuytelaars, & Lucl). The idea is that these keypoints are resistant to scale, noise, differences in illumination, orientation, and partially invariant to affine distortion. SURF takes a variety of parameters which tweak the detection process, we found that the default values recommended by OpenCV to be quite effective. An example of keypoint detection in an image can be seen below in Figure 6.

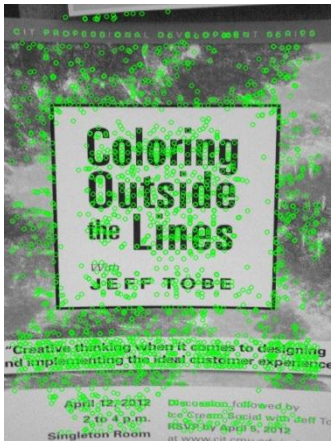

Figure 6 – Trackable image with detected keypoints marked

We also tried many other detectors apart from SURF. These other detectors have similar goals as SURF (which are to find points resistant to affine transformations) but differ in how these goals are attained. We found some detectors to be very fast but they lacked in accuracy or robustness. A comprehensive list of these results can be seen in table A. In the end we chose to use SURF because it provided the greatest combination of accuracy and speed.

Keypoint Detector/Feature Extractor	FPS (640x480 px)	Time taken to find all keypoints
SURF/SURF	0.476	672.3 ms
SIFT/SURF	0.447	1104 ms
STAR/SURF	5	53 ms
MSER/SURF	3.33	108.667 ms
GFFT/SURF	1.875	105 ms
FAST/SURF	1.875	9 ms

Table A

*SIFT (Scale-Invariant Feature Transform)
*MSER (Maximally-Stable Extremal Region Extractor)
*GFFT (Good Features To Track)
*FAST (Features from Accelerated Segment Test)

2.2 Feature Extraction

Feature extraction is the process of describing these aforementioned meaningful points as a feature vector.

The algorithm that we decided to use was again that of SURF. The SURF feature extractor works by computing a histogram of local gradients around keypoints. SURF speeds up the computation time by using integral images and only using a 64-dimensional feature vector (Bay, Tuytelaars, & Lucl). In essence the feature vector describes the region around the keypoint, capturing what is interesting (or its property of being resistant to affine transformations) about the keypoint. Again the parameters we decided to use were default values provided by OpenCV.

Similar to what we did with keypoint detection, we tried other feature extractors. Again we found some feature extractors to be fast but lacking in accuracy. A comprehensive list of results can be seen below in table B.

Keypoint Detector/Feature Extractor	FPS (640x480 px)	Time taken to find descriptors
SURF/SURF	0.476	1227.67 ms
SURF/SIFT	0.361	1774 ms
SURF/BRIEF	0.469	39.33 ms

Table B

*BRIEF (Binary Robust Independent Elementary Features)

2.3 Matching

Now that we have transformed both of the *trackable* and *frame* images to sets of feature vectors, we perform a matching operation to try to determine which *frame* feature vectors could correspond/be the same as *trackable* feature vectors. Performing the matching essentially equates to finding the "nearest" feature vector in the *frame* feature vector set for each of the *trackable*'s feature vectors. The nearness of a feature vector is determined using a standard distance metric in the vector space; in our case we used Euclidean distance. The intuition behind this step is that if the *trackable* landmark exists within the *frame* image, then the *frame* should contain feature vectors which closely resemble the *trackable* feature vectors. So in this step we generate a potential matching and submit it for grading at the next step.

The matching can be computed in a brute force manner, where a distance is calculated for each *frame* feature vector and the closest one is chosen. If there are n *trackable* feature vectors and m *frame* feature vectors, this operation takes O(nm) time. This is pretty expensive, especially when the *trackable* or *frame* contain a large number of keypoints. We instead perform an approximate nearest neighbor search for improved performance. The approximate search is done by storing the data into kd-trees. The advantage of this structure is that it serves as a sort of index of the vectors, allowing for efficient searching without necessarily viewing every element. At a high level, kd-trees partition the vector space in an incremental fashion such that entire partitions can be ruled out of the search at a time. This still offers worst case O(m) search time if we want to find the guaranteed nearest neighbor, but much better performance can be achieved by heuristically performing an approximate search. This is done by generating a fixed number of trees which are constructed by randomly choosing a fixed number of partitions and searching the trees in parallel for a closest neighbor. This does not necessarily return the optimal solution i.e. the guaranteed nearest neighbor, but in practice it performs reasonably well and many orders of magnitude faster. As seen in table C, the FLANN matcher performs 5 times faster than the brute force Matcher. This algorithm of performing approximate nearest neighbor searches in higher dimensions is implemented in the OpenCV FLANN (Fast Library for Approximate Nearest Neighbors) interface.

Matching Algorithm	FPS (640x480 px)	Time taken to complete matching
Brute Force Nearest Neighbor	0.297	1046.67 ms
Brute Force 2-Nearest Neighbor	0.33	1103.33 ms
FLANN Nearest Neighbor	0.395	202.33 ms
FLANN 2-Nearest Neighbor	0.476	212.33 ms

Table C

*Measurements were taken assuming SURF to be both detector and descriptor algorithm

Below is an example screenshot of SURF features which have been matched from the *trackable* on the left to the *frame* on the right.

Figure 7 – The result of the matching operation

There are various methods to perform matching beyond simply finding a single closest match. OpenCV provides 3 separate methods, `DescriptorMatcher.match()`, `DescriptorMatcher.knnMatch()`, and `DescriptorMatcher.radiusMatch()`. These methods respectively correspond to a closest neighbor search, k-nearest neighbor search, and a bounded radius search. Retrieving the closest match performs reasonably well, but after some experimentation, we found that k-nearest neighbor with k = 2 did not take too much extra time and provided additional information which was very useful for filtering and grading the matches in the next step. Radius match can be more efficient than k-nearest neighbor (especially when using the FLANN index as opposed to brute force), but we found it difficult to set a threshold which worked well across different landmarks. Thus we chose 2-nearest neighbor. Note that this results in a mapping of two *frame* feature vectors for each *trackable* feature vector.

2.4 Match Grading

Now that we have extracted features from the *trackable* and *frame* as well as computed a 2-nearest neighbor matching on the sets of feature vectors, we grade these matches to determine if it is a good set of matches and we have indeed found the *trackable* within *frame*. Grading is done in a two-step process. First, we filter the matches according to some preset metric to retrieve a set of "good" matches. Then we compute a homography matrix between the "good" matched keypoints to determine what affine transformation must be applied to the *trackable* image to warp it into the shape and orientation that it has within the *frame*. If the homography matrix "fits" the data well (as will be discussed in more detail, we have methods to rate how well a mathematical model such as a homography transformation fits a set of data) i.e. the "fit" is greater than some threshold T, then the match is considered an accurate match and the *trackable*

object is said to have been detected. Otherwise the *trackable* is declared to not have been present in the image.

Let us go into more detail as to what it means to filter the set of 2-nearest neighbor matches into a set of "good" matches. The idea here is to filter out matches which seem too large in distance or too ridiculous to possibly be an actual match. Just because the distance from vector A to vector B is less than the distance to vector C does not mean that vector B is almost equal or close to vector A. Initially, we implemented a sort of radius matching/filtering in which the threshold would depend on the *trackable* in some way. Initially our algorithm was to retrieve the minimum distance match, then only keep those matches with a distance less than k*min_dist. We tried setting k to values from 1-10, but we did not get very reliable performance this way. The problem was that there was too much variance in the data which the homography matrix was being computed on in the next step, which resulted in it being difficult to establish a reliable fit threshold T. So we implemented a different match filtering algorithm that reduced the variance greatly. We realized that what we really wanted to accomplish by filtering the matches was to get a set of matches which can all be said to be accurate matches or inaccurate matches (this accuracy is calculated in the next step when computing the homography). Now, take the case of knowing the 2 nearest matches for a given *trackable* feature vector. Let vector A be the *trackable* vector, B and C be the 2 *frame* feature vector matches for A, and finally let B be closer to A than C. Note that there are two possible configurations here. Either B is clearly closer to A than C is, or it is not clear that B is a much better match than C.
These two cases are pictured in figures 8 and 9.

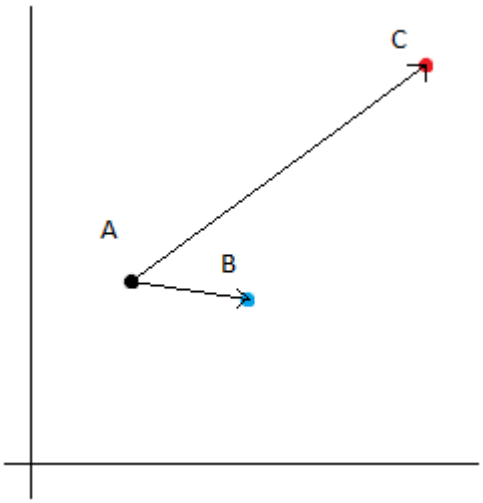

Figure 8 – B is much closer to A than C

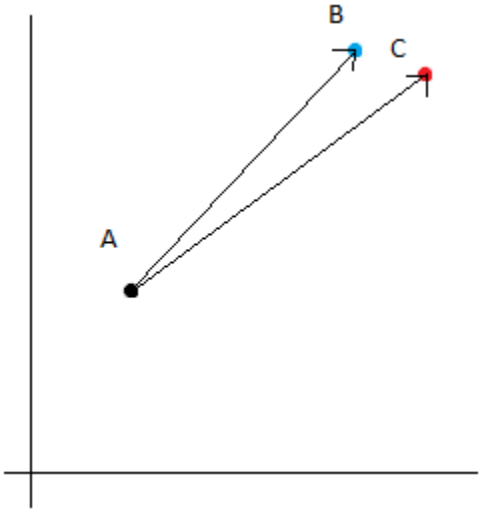

Figure 9 – It is not clear that *B* is a much better match than *C*

If we throw out matches where it is not clear that B is a much better match than C, then the resulting set of matches can be confidently said to be a consistent set of matches, we have no ambiguity in knowing whether some of these matches may be good but some of them aren't. With no ambiguity, the next step of calculating fit can be done much more precisely. In practice we found this method of filtering to result in a much larger discrepancy between the fit for when the object does exist in the frame as opposed to when it doesn't.

Finally, once we have our set of filtered potential matches, we compute the homography matrix, a matrix which details exactly how to transform pixels in the *trackable* image to the corresponding ones in the *frame*. This is essentially the problem of seeing how well a given mathematical model fits the given data. In this case, the mathematical model is a homography matrix and the data is the set of filtered matched keypoints. This can be done by using a method called RANSAC (RANdom Sample Consensus). Basically, RANSAC is method which robustly finds the parameters for a given mathematical model which result in the best fit of the given data. Exactly how RANSAC works is outside the scope of this paper, but the important thing to note here is that RANSAC will output a percentage indicating what percentage of the data is accurately represented by the best fit model. So finally, we can determine if the object was successfully detected by judging if this percentage of "inliers" was greater than some threshold. Now we can see that if the set of filtered matches is mostly consistent, then this makes it easier to determine if the data fits or doesn't fit the homography transformation model.

Once we know the homography matrix which relates points in the *trackable* coordinate space to the points in the *frame* coordinate space, we can use this matrix to transform any 2D image or structure and overlay it in *frame* to make it seem like it is actually overlaid on top of the *trackable* landmark in reality. In Figures 9 and 10 below, you can see the result of transforming a box the same size as the *trackable* image and overlaying it on the *frame* image.

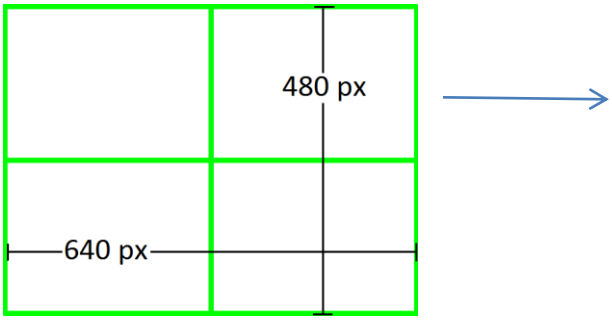

Figure 10 – Bounding box

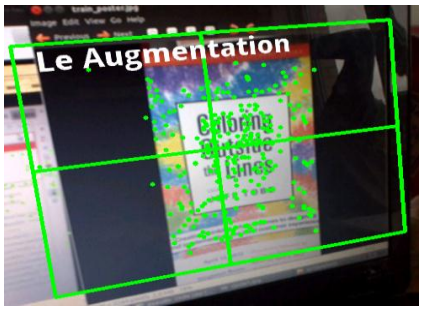

Figure 11 – Warped Bounding Box in 640 x 480px frame

The following is a pseudocode implementation of the object detection pipeline discussed thus far. In the context of detecting a landmark in a video or live video feed, we would process each frame of the video/feed at a time and input the image of the landmark as well as the video frame into the object detection pipeline.

```
int processFrame(Mat& frame) {

      // object detection pipeline:

      // extract descriptors from video frame

      vector<KeyPoint> frame_kpts = getKeyPoints(frame);

      Mat frame_kpt_descriptors   = getDescriptors(frame, frame_kpts);

      // match to test if object present in frame

      vector<DMatch> matches = getMatches(trackable_descriptors,
frame_kpt_descriptors);

      // Do initial thresholding to grade the match

      // Compute homography using RANSAC error metric

      // (RANdom SAmple Consensus)

      // to further grade the match

      if (good_match) {

                  // found Trackable!

            }

      }

}
```

3. Tracking

As noted from the statistics mentioned in the previous sections, object detection is a computationally expensive and slow process. Running on a Motorola Xoom tablet with a 1 GHz dual core processor and 1 GB of RAM resulted in a frame rate of only .275fps! There is, however, one big adjustment we can make to the object detection pipeline to greatly decrease the expensive computations that we do. This is based on the observation that once we have detected the *trackable* within a video frame, it is unnecessary to go through the entire process of object detection again. All that really matters is the end result of object detection, the keypoints in the video frame that correspond to keypoints in the landmark we are looking for. If we can figure out where the pixels corresponding to the keypoints in the previous frame moved in the next/new frame, then we will have computed the keypoints identifying the landmark without the going through the computationally expensive process of feature vector extraction, matching, and

grading. This concept of tracking how pixels in an image move to a new position in a subsequent image is called *optical flow*. By applying optical flow techniques to the detected keypoints of a landmark within a frame, we can track where these points move frame to frame, and thus keep track of where the landmark is from frame to frame, all without the cost of going through object detection again.

The actual algorithm we use to track keypoints is the sparse iterative version of the Lucas-Kanade optical flow estimation method in pyramids. We chose this algorithm because it is designed to effectively track a sparse set of pixels, as is the case with keypoint tracking. This algorithm is implemented in the OpenCV Motion Analysis and Object Tracking module (OpenCV documentation, 2012). At a high level, a basic explanation of the Lucas-Kanade algorithm is that instead of considering single pixels at a time, it looks at small regions of pixels. Assuming that each region has not drifted much from one frame to the next, it performs a search within a bounded area in the new frame, centered at the last known location. The point to which the region has moved to is determined using a least squares error criterion.

To add augmented reality to the frame, we can use the tracked point information to infer a general location of the *trackable* within the frame and place the augmentation. But this does not change the orientation of the augmentation or tilt it as if it is truly overlaid on top of the landmark. There is an additional piece of information required to perform realistic augmentation on top of the *trackable* landmark, the homography matrix. We need a homography matrix relating the points in the previous frame to the next frame. Given the warped overlay image which would be overlaid on the previous frame and a homography matrix relating points in the previous frame to those in the next frame, we can apply the homography matrix to the warped overlay to get a new overlay image. To get this homography matrix, we compute the homography matrix between tracked points from the previous frame to the next frame.

One problem that we noticed with repeatedly applying a homography transformation to an image was that it resulted in a blurring effect on the image. To handle this, we premultiplied the newly computed homography matrix with an accumulator homography matrix to compute an overall homography matrix.

Essentially, the following relation is true for the homography matrices from frame to frame.

$$\begin{bmatrix} x' \\ y' \\ \sim \end{bmatrix} = H_n H_{n-1} \cdots H_2 \, H_1 \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Where H_n is the homography matrix from frame *n-1* to *n* (the tilde means that value is ignored). Thus we can clearly see that an overall homography matrix is given by the equation

$$H_{overall} = H_n H_{n-1} \cdots H_2 \, H_1$$

This overall homography matrix can be applied to the original 2D augmentation image to get the overlay image.

The overall speedup achieved by implementing tracking versus just running object detection all the time is fairly significant. After setting the key point detector and descriptor with SURF and using FLANN 2-Nearest Neighbor as the matcher, we compared the frame rates for just running object detection versus object detection and tracking. The results are shown below.

Method Specified	FPS (640x480 px)
Pure Object Detection	.476
Object Detection with Tracking	8.765

Table D

*Frame Rates calculated on a laptop with a dual core 2.1 GHz processor and 4 GB RAM

Clearly, by combining tracking with object detection, we are able to achieve much faster frame rates.

4. Demo

For our demo we asked volunteers to play our Scavenger Hunt game which was present in the form of an Android application. The various screens and game rules will be detailed below.

Figure 12

In the screenshot above we have the introductory splash screen. Here we give the user the option to play the game or get access to some other basic functionality that is typically found in Android applications.

In the next screen we give the user the option of either creating a scavenger hunt or joining one. In the screen below, we provide the user with functionality to create a hunt.

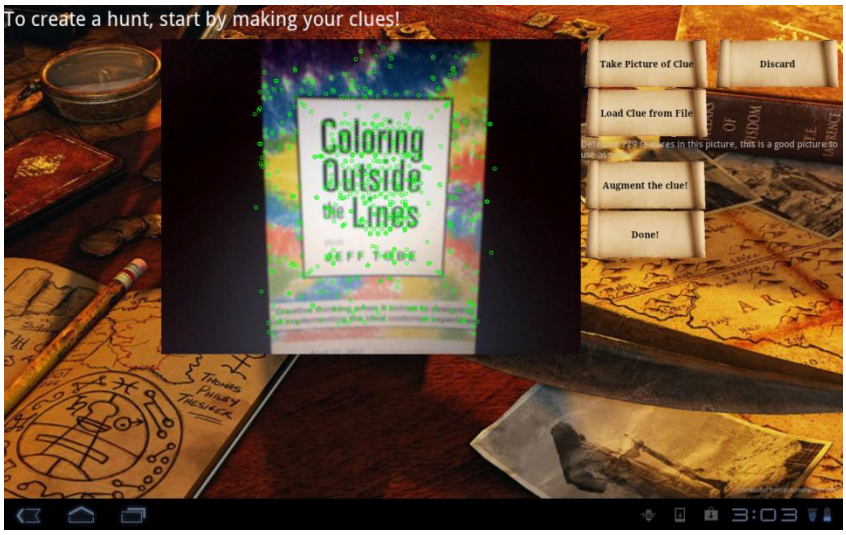

Figure 13

We can see above how the user has the ability to instantiate landmarks. This is done by taking a picture of the landmark or loading an image file.

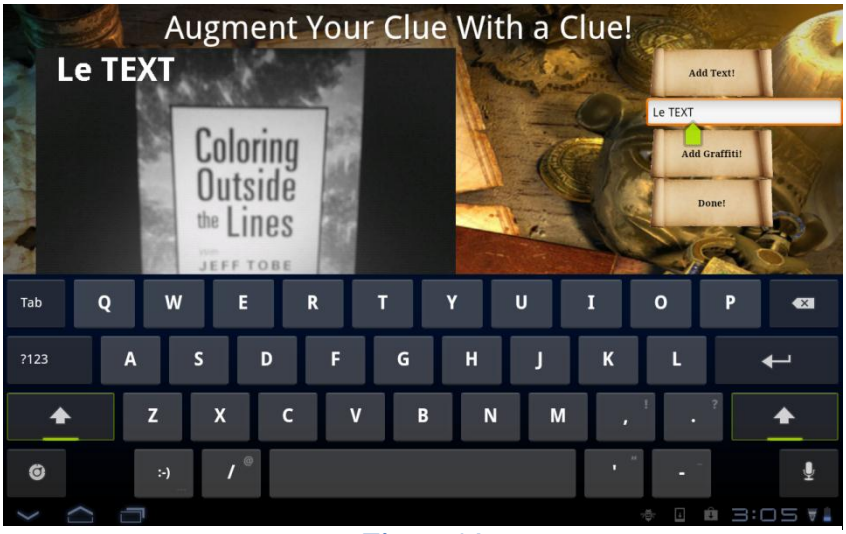

Figure 14

In the above screenshot we can see how the user has options to augment the landmarks that they have marked as clues with text. Once this process is complete the user has the option of adding more clues to the hunt or finishing the process and giving it the next player to find the clue.

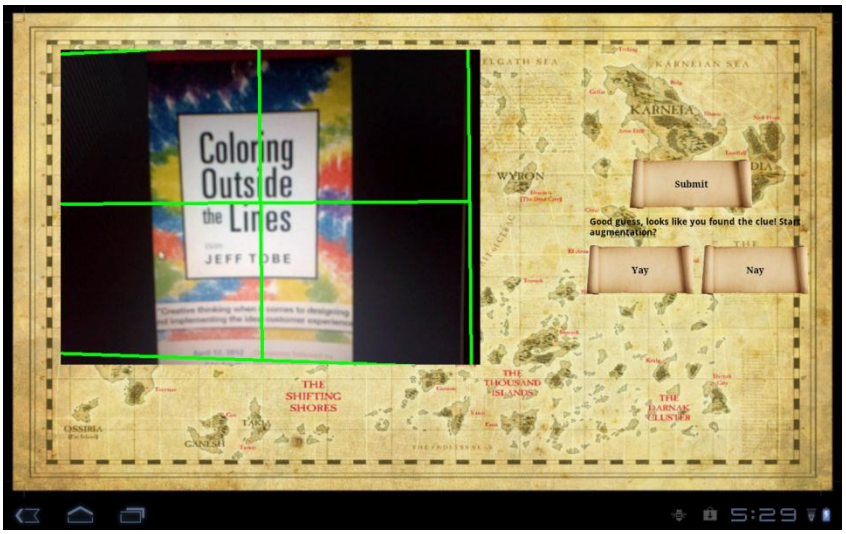

Figure 15

In the screenshot above we can see what a person who is joining the hunt and looking to find clues would see. The user has the opportunity to guess at landmarks by taking pictures of them and submitting their guesses. The user then receives feedback notifying the user of being correct or incorrect. If they are correct they have the opportunity to see the first players augmented text. The game ends when all of the hidden clues are found.

Due to the limited space in the demo room we decided to print out pictures of various images in order to simulate the action of finding unique natural landmarks and marking them as clues.

5. Discussion of Results

In this section we would like to discuss some of the errors we found and some of the steps we took to mitigate these errors. One of the first errors we found was that during tracking, some of the keypoints would drift erratically, hence causing the homography matrix to be calculated incorrectly. Another problem we faced was that we were not properly filtering out matches during the object detection. In order to address this problem we implemented the k nearest neighbor algorithm to filter more effectively. Another problem we faced was with the implementation of various detectors (in our object detection pipeline). We were not able to get the robust results that SIFT and SURF were providing and thus chose SURF as the faster but still robust keypoint detector.

6. Addressing Feedback

One of the suggestions provided to us was to try some non OpenCV implementations for our object detection and tracking algorithms. As we looked into these implementations we realized that integrating non OpenCV code with OpenCV code was not going to be a trivial task. This realization also coincided with the fact that we had to devote a large portion of our resources towards Android programming. We had an especially heavy emphasis on Android programming because we needed to program game logic into our application that would seamlessly hook into the more signal processing oriented OpenCV functionality. Thus we relegated this task to future work.

Initially we had decided that when looking for clues as part of our scavenger hunt we were going to run our object detection code in real time. Once we realized that this process was going to be very computationally expensive and slow, we received suggestions to use a more efficient approach that takes advantage of the GPS capabilities of the Motorola tablet (by turning on the heavy object detection algorithms only when you are close to the clue). We considered this suggestion but were concerned about getting enough resolution with the GPS. Thus we decided to instead change the paradigm of the game to avoid having to run the object detection code all the time. By making the second player (who is looking for clues) take pictures of what he thinks may be the clue and submitting the guess, we were able to keep the quality of the user experience high.

7. Future Work

Currently our object detection with regards to the descriptors and extractors we are using can definitely be improved. One of the ways to improve their effectiveness is by tweaking the parameters of various descriptors and extractors (includes SURF as well as other descriptors and extractors mentioned earlier). As mentioned earlier, we believe that we could try non OpenCV implementations of the aforementioned algorithms. These implementations might have been able to give us better performance or results. Continuing with this trend of trying different algorithms and methods we would also like to explore object detection in the context of machine learning as opposed to the matching pipeline. By having the opportunity to train on some of the images that we gathered we may have been able to increase the accuracy and robustness of our results.

Another potential optimization that we didn't have a chance to look into would be to make use of GPU's. Because GPU's can do calculations in a more efficient manner than CPU's we could offload some of the calculations necessary for object detection on to the GPU thus making our object detection process much faster.

Now that we have addressed some of the algorithmic changes with regards to our application let's talk about the potential for changes in terms of overall game logic and in game features. We had already alluded to this in the feedback section but the use of GPS would be very useful to the game structure. By having a GPS component, the player who is looking for clues can have a better idea as to where to look for hidden clues. This could be implemented as some sort of hot and cold meter that would assist the player. One of the concerns with having this GPS component would be getting enough resolution if the area of the scavenger hunt is very small like a room for example. One of the possible solutions that we didn't get a chance to explore was the use of differential GPS. By using a separate device in a fixed location we may be able to gauge distances in a small room by having the main device get its direction with reference to the separate fixed device. Apart from the GPS component we would also like to have the game structured so that both players in the game have separate tablets (since we would like to simulate a real life scavenger hunt). This would require communication between tablets and a reprogramming of the application such that both tablets could play the same game despite being on two different devices.

Building upon the future work that was mentioned previously we also felt that there could be more done with regards to how the clues in the hunt were being made and how we could add a dimension of competitiveness in the game. With regards to the clues currently we can only augment text but eventually we would like to be able to augment 3-D clues and potentially have more complex clues (such as slider puzzles) that would add a level of difficulty to the hunt. Because we are creating a game, we would like to eventually come up with a scoring system that would allow multiple users to join hunts and record their scores online and compare their performances. We see this as something that could be done by implementing a server that could communicate the necessary information.

8. Work Breakdown

Week	Tasks	Both	Primary	Secondary
2/27	Research existing AR libraries/Collect images for testing	X		
3/5	Research existing AR libraries/Collect images for testing	X		
3/19	Test chosen libraries on data and finalize algorithms to be used		K	A
3/26	Finalize implementation of algorithms for object detection and tracking		K	A
4/2	Prepare for Mid-Project Report	X		
4/9	Port algorithms to Android/Develop UI		A	K
4/16	Port algorithms to Android/Develop UI		A	K
4/23	Test on Android Platform	X		
4/30	Clean up code and prepare for demo/final report	X		

9. References

Object Detection. (2011, July 23). Retrieved May 2012, from Wikipedia: http://en.wikipedia.org/wiki/Object_detection

OpenCV documentation. (2012, April 28). Retrieved May 2, 2012, from OpenCV: http://opencv.itseez.com/

- Main resource for API documentation as well as tutorials/examples for various OpenCV functions

Bay, H., Tuytelaars, T., & Lucl, G. V. (n.d.). SURF: Speeded Up Robust Features. Zurich.

- Used as a reference to better understand how SURF feature detection works

Combitech. (2010). ARmsk. Retrieved from ARmsk: http://armsk.org/

- Used as a resource for understanding how to implement markerless AR

Rappoport, N., Puschinsky, A., & Richardson, E. (2011). realgraffiti. Retrieved from Google Code: http://code.google.com/p/realgraffiti/

- Provided guidance with regards to how to put together different elements of markerless AR since realgraffiti application accomplishes some of the things we want to do with our application