Markerless GPU accelerated augmented reality on android devices

Somers, A.J.M.

Award date:
2012

Eindhoven University of Technology
Department of Mathematics and Computer Science

Master's thesis

Markerless GPU Accelerated Augmented Reality on Android Devices

September 10, 2012

Author: ing. A.J.M. Somers
a.j.m.somers@student.tue.nl

Supervisor: dr. M.A. Westenberg
m.a.westenberg@tue.nl

Tutor: drs. A.A.G. Willems
ard.willems@alten.nl

Assessment committee: drs. A.A.G. Willems
dr. M.A. Westenberg
dr. A. Serebrenik

Abstract

This thesis describes an Augmented Reality (AR) framework for Android devices. Using the framework, virtual three dimensional objects are drawn in a view of the world captured with a camera, thus augmenting the reality. AR can for example be useful for visualisation or to assist with tasks. To enable a wide range of applications, the AR framework is developed for Android devices and designed to work in a wide variety of scenes.

To create AR, a virtual camera must be created which matches the real camera. To create a matching camera the camera extrinsic parameters, which are the position and orientation, and the camera intrinsic parameters such as focal length, need to match. This thesis describes how to automatically obtain both the camera intrinsic and extrinsic parameters. The extrinsic parameters are estimated by first detecting objects in the camera image and then using the position of these objects in the camera image to estimate the location and orientation of the camera.

The object recognition system is trained with reference images from objects after which it is able to robustly recognise objects under different viewing conditions such as different camera viewpoints and different lighting conditions. This is achieved by first finding matching keypoints between the camera image and the reference images. For the keypoint detection, description and matching several methods are evaluated in this thesis. Based on these matches an object recognition method is built which is able to reliably tell if and where an object is present in the camera image.

To create an immersive AR experience a high framerate is required. Therefore several optimizations are discussed, such as a graphics processing unit (GPU) implementation of the keypoint detection and description algorithm.

The resulting framework is a fully functional framework which incorporates all aspects required such as pose estimation and rendering. It shows that keypoint based object recognition is an effective way to create the required understanding of the environment needed to perform pose estimation. The framerate without GPU acceleration or tracking is just below one frame per second. Furthermore several methods are shown in which the framerate can be increased.

Acknowledgments

It is a pleasure to thank the people who made this thesis possible.

First I would like to thank my supervisor, dr. Michel Westenberg. During our regular meetings he showed great involvement in my project and supported me with sound advice.

I also wish to express my gratitude to my tutors drs. Ard. Willems and ir. David Hardy, which supported me throughout my project with advice, feedback and questions. Their support, ideas and guidance enabled me to complete this project.

I would also like to thank dr. A. Serebrenik for his membership in the assessment committee.

Finally I wish to show my gratitude to my parents, which encouraged and supported me during my studies. Without their unconditional love and support the completion of this project would not have been possible.

ing. A.J.M. Somers

Contents

1 Introduction

This thesis discusses the implementation of an augmented reality framework for Android devices. Augmented reality can be used in a wide variation of applications such as object visualization, information visualization or task assisting applications. To use augmented reality effectively there is need for a framework that requires minimal modifications of the environment in which it is used and at high speed such that the augmented reality feels like a reality. Previous research at Alten provided two frameworks. Those either required the placement of special markers or required a 3D camera and was limited in the environments in which it could be used. As such both frameworks were limited in their applications. This projects tries to overcome these limitations, making the framework more useable in practice.

1.1 Motivation

Computers can be used to provide information on numerous applications, but often it may be hard to relate the information to the physical world. For example a mechanic performing a complex task on a machine might have schematics to assist him. But he might have difficulty relating the information in the schematics to the actual machine. Therefore it could be beneficial to combine the view of the physical world and the information into a single view. AR is a technique which enables the user to see virtual objects such as the information in a real environment.

Android devices such as smartphones are ever increasing in popularity, computational capability and features, making them a good platform for AR applications. They fullfill the basic requirements for an AR application, namely they have a camera, a screen and the ability to run complex applications.

1.2 Perspective

This research has been performed at the Alten PTS company, which delivers consultancy in technical automation.
This work explores the possibilities of Augmented Reality (AR) on mobile devices, in this case an Android smartphone. The outcome of this project may be used in further research or as a prototype for possible clients.

Previous work at Alten PTS by Hardy [Har11] and Crijns [Cri12] has shown augmented reality on Android devices is possible but further work is needed to make it applicable for general use. Problems that arose are low frame rates and the requirement of special markers [Har11] or applicability in a limited number of environments [Cri12].

1.3 The Idea

An AR application draws virtual data in a view of a real environment, thus augmenting the reality. This can range from context-relevant information to virtual 3D objects which appear as real objects. This thesis focuses on virtual 3D objects. In order to draw 3D virtual objects in the right position, scale and orientation, such that they appear to be part of the real environment, the application should have some understanding of the environment. The most important step for an AR application is to derive the camera pose. The camera pose is the position and orientation of the camera in the world. Using this camera pose it is possible to correctly position virtual objects on a view of the real world.

Computer vision is used to gain an understanding of the world. While there are multiple other ways a camera pose could be derived, such as by GPS and compass or orientation sensors, the accuracy of these techniques is too low. Therefore computer vision techniques are used to analyse the camera image itself. To derive the camera pose from images captured by a camera, some knowledge about the real world is required. First the appearance of objects has to be known such that objects can be recognised. Next some location information about these objects is required such that when objects are recognised in camera images, the camera pose can be derived.

The object recognition step can be implemented in various ways, for example previous work by Hardy [Har11] used markers as special objects to recognize. In order to get a framework which does not require special objects, such as markers, to be placed in the scene, a general object recognition system based is implemented. This system is based on keypoint detection in images. The keypoint-based object recognition system is able to recognize many real objects from training images under different viewing conditions such as scale, illumination and viewing angle.

Unfortunately these keypoint-based methods are computationally expensive. Especially on an Android device with limited computational abilities this poses a problem. In order to improve performance, various optimizations are discussed such as a GPU implementation of the object recognition.

1.4 Document setup

This thesis starts with the project outline in chapter 3. Here the various components that make up the framework and how they are combined are outlined. Chapters 4 to 7 discuss key components in detail. In chapter 8 various ways in which the framework is optimised are discussed. This includes the use of the GPU to perform part of the computations and keypoint tracking. Appendix A has details on camera properties such as field of view and how they can be obtained.

2 Background

In this chapter the different types and methods that are used in order to create AR are discussed. In the following section typical applications are described and illustrated using examples. Then general AR techniques are discussed to show different strategies of implementing AR. Finally previous projects performed at Alten PTS and the techniques used are discussed. The techniques chosen in these previous projects have some limitations which this project overcomes.

2.1 Applications

AR has some interesting and useful applications. Applications can enable a user to visualize objects before they are created or purchased. This can help a user to see a certain design in a real environment without the cost of actually creating it. Other applications using AR often add information to a view to assist a user in a certain task or fulfill an informational need.

An example of AR for visualization can be found at online clothing store Tobi. They allow a user to see what clothes will look like using AR to project the clothing on the user as shown in figure 2.1

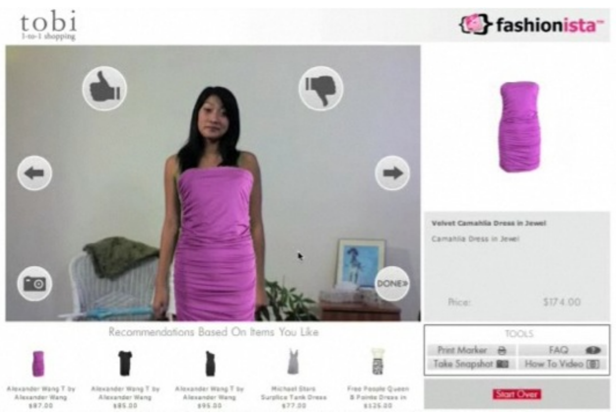
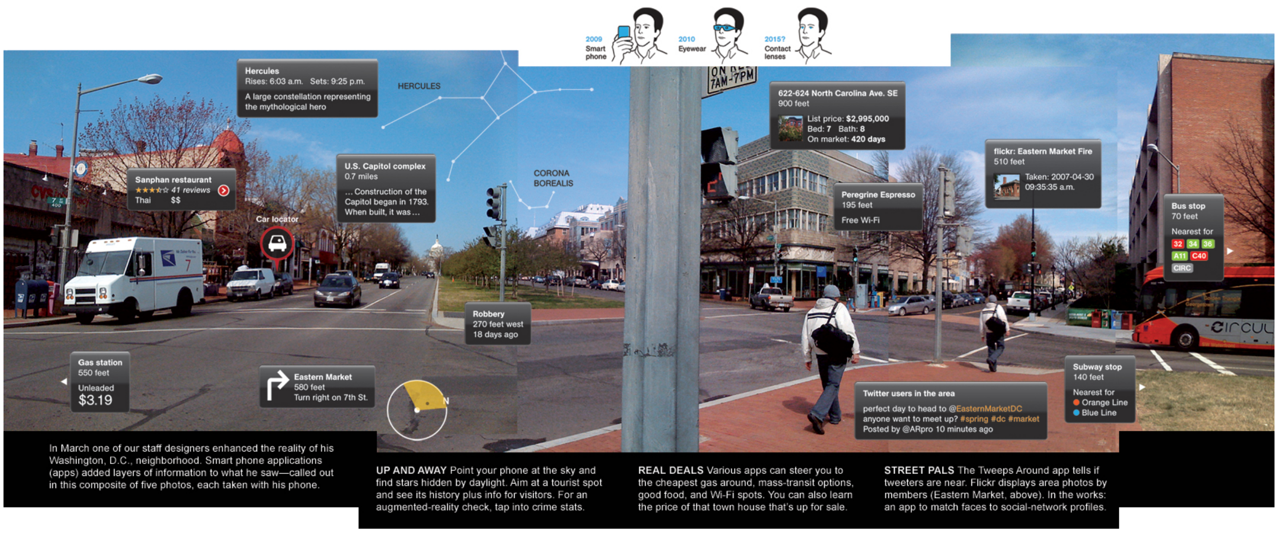

tobi
1-to-1 shopping
fashionista
DONE
Velvet Camahlia Dress in Jewel
Camahlia Dress in Jewel
Price:
$174.00
Recommendations Based On Items You Like
TOOLS
Print Marker
FAQ
Take Snapshot
How To Video
Start Over
Alexander Wang T by Alexander Wang $87.00
Alexander Wang T by Alexander Wang $85.00
Alexander Wang T by Alexander Wang $95.00
Michael Stars Surplice Tank Dress $77.00
Free People Queen B Pointe Dress in $125.00

Figure 2.1: *Tobi virtual dressing room*

AR can also be used to display information to the user, see figure 2.2 from [Fol10]. This is an image from *National Geographic Magazine* showing how AR could be used to fulfill many different informational needs. Some examples shown are commercial information of shops restaurants and gas stations, directions to get to a certain destination, public transport information and the location and names of constellations.



In March one of our staff designers enhanced the reality of his Washington, D.C., neighborhood. Smart phone applications (apps) added layers of information to what he saw—called out in this composite of five photos, each taken with his phone.

UP AND AWAY Point your phone at the sky and find stars hidden by daylight. Aim at a tourist spot and see its history plus info for visitors. For an augmented-reality check, tap into crime stats.

REAL DEALS Various apps can steer you to the cheapest gas around, mass-transit options, good food, and Wi-Fi spots. You can also learn the price of that town house that's up for sale.

STREET PALS The Tweeps Around app tells if tweeters are near. Flickr displays area photos by members (Eastern Market, above). In the works: an app to match faces to social-network profiles.

Figure 2.2: *Information AR by national geographic magazine*

The ARMAR project [HF11] uses AR to assist a user in a technical task, in this case maintenance and repair. In figure 2.3 a mechanic is wearing a head-mounted display showing directions to a maintenance task. The directions are shown as text and by projecting virtual objects which indicate the components involved and how the task should be performed.

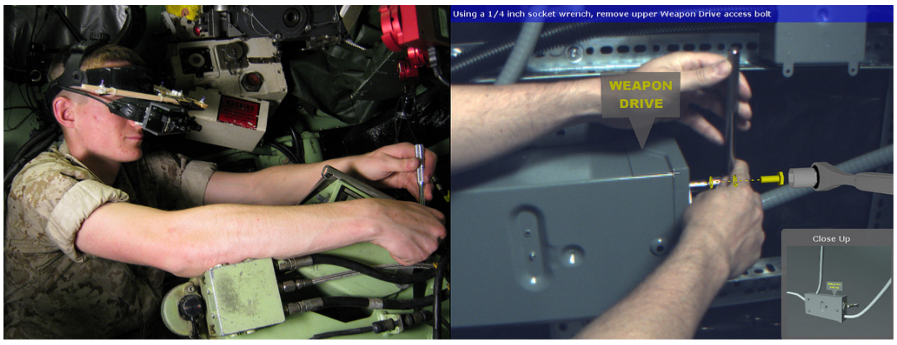

Using a 1/4 inch socket wrench, remove upper Weapon Drive access bolt
WEAPON DRIVE
Close Up

Figure 2.3: *Task assisting AR*

2.2 Previous work

This project was preceded by two other students working on related projects. David Hardy [Har11] has created an application focused on landscape development AR. His project has several useful subjects including camera calibration and pose estimation. His application used markers and ran at approximately 1.5 frames per second.

Thorstin Crijns [Cri12] has focused on using a stereo camera to create a point cloud which he then matched from other perspectives. Because of the hardware limitations the application had a limited range in which it could operate, making it suitable for indoor use only. Crijns did not use specific markers, but needed points with distinctive characteristics for the application to perform well. His application ran at approximately 1.9 fps.

2.3 Techniques

Every AR application has to have an understanding of the view of the real world in order to augment the view. While for some applications an approximate position (such as one acquired by GPS) and viewing direction (such as one acquired with a compass) are sufficient. For example most of the information in figure 2.2 could be shown using the approximate location and viewing angle. An AR application which places virtual objects in the scene has to have much better understanding of the view of the world to be able to correctly align the virtual and the real objects. If the virtual objects are not very precisely aligned, applications such as portrayed in figure 2.3 would not be possible. In figure 2.3 a screw is highlighted and an arrow is showing where the screw should be inserted. If the matching with the real world is not exact, this AR application would obviously not work. So the precision required depends on the application.

The precision that is achieved depends on the method of pose estimation. Pose means the camera position and viewing angle. The highest accuracy is achieved using computer vision techniques. Therefore the focus of this thesis is on pose estimation using computer vision techniques.

Computer vision is a broad subject and many approaches exist. Using markers has been a popular approach since markers are relatively easy to detect. The downside of markers is that they require modification of the scene that is to be augmented in order to work. A more general object recognition technique is therefore preferred. One such technique that has been popular is keypoint-based object recognition. By the detection of keypoints in an image, objects can be recognised even when they are viewed under different viewing conditions such as a different viewing angle. This makes the technique especially suitable for AR applications.

2.4 Summary

Different applications for AR exist. These can be categorised in visualisation using AR, AR for informational display and task-assisting AR. The techniques used to create these applications depend on the application. The main factor that determines the chosen technique is the accuracy in pose estimation required. For example task-assisting AR requires an accurate pose estimation. The most accurate pose estimation is achieved using computer vision techniques. Computer vision can be implemented in many ways. The implemented approach uses a general keypoint-based object recognition system.

3 Project Setup

The framework described in this thesis allows a user to use an Android device to see virtual objects placed within the camera images as if they were real three-dimensional objects in the environment. This means that when moving the camera the location, orientation and appearance of virtual objects changes relative to the physical world. To create a deep level of immersion the framework is optimized such that the frame rate is maximized and the latency is minimized. This is achieved by placing focus on performance during design of the framework and by the implementation of specific optimizations such as tracking and the implementation of algorithms on the GPU.

This chapter outlines the various components implemented in the AR framework and how they are combined to create a fully functional AR framework. In the following chapters several of these components will be discussed in more detail.

3.1 Platform

The framework described in this thesis runs on Android hardware such as smartphones or tablets. Modern hardware and software is used since AR is not feasible on older hardware [Har11] and the implementation uses new features introduced in recent Android versions.

3.1.1 Software

The application runs on Android devices. During the development of the framework the newest version available is used, which is Android 4.1 (Jelly Bean). While the framework is developed and tested on Android 4.1 the minimal version required is Android version 3 (Honeycomb) due to features used in the framework which are available since this version.

OpenCV (Open Source Computer Vision Library) [Bra00] is a library of programming functions mainly aimed at real time computer vision. It includes many graphics operations including keypoint extraction and matching. It is reasonably fast. The used version version, OpenCV 2.4.2, includes a Java API which is used for the implementation of the framework.

3.1.2 Hardware

OpenGL ES 2.0 capable hardware is required since older versions do not include programmable shaders. These shaders are required for both the GPU based optimisations and the rendering of the virtual objects. The device is required to have a camera, screen, fast CPU and GPU. The CPU and GPU need to be fast in order to achieve a high enough frame rate such that an AR application feels immersive. .The application is developed and tested on a Samsung Galaxy SII.

3.2 System overview

The main goal of an AR framework is finding the camera pose, meaning the location and viewing direction of the camera. If these are known the virtual world with the virtual objects can be aligned with the real world captured by the camera. This pose is calculated by detecting points in the camera image for which the 3D location in the world is known. This is achieved by creating a database of objects. Each object has one or more reference images which are pictures of the object and certain points in these images are annotated with their 3D location. When objects in the database are found in the camera image, their 3D location is used to estimate the pose of the camera.

3.2.1 System components

The framework consists of several key components. These components are displayed in the diagram below.

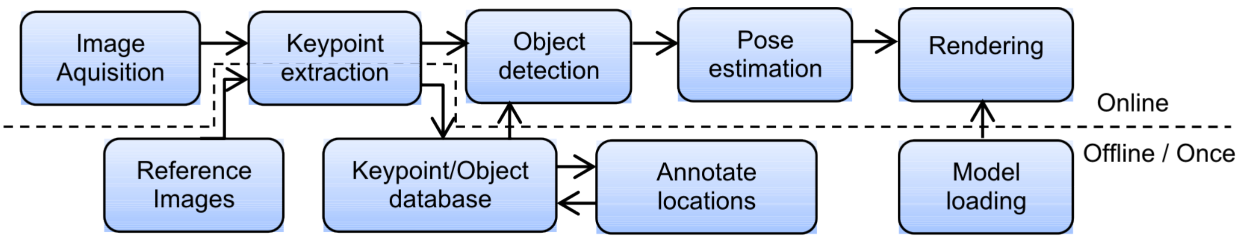


Figure 3.1: *Components in AR framework*

Several steps are performed for every frame, while others are done offline or during initialization. This is denoted by the dotted line.

For every frame a camera image is acquired. This camera image can then be corrected for camera production errors, resulting in what is called calibrated images. The calibration step is optional and can be omitted for performance reasons when only small errors are present. More details on camera calibration can be found in section 8.1.2 and A.

Then keypoints are extracted from the camera images. A keypoint is a point in an image which can be robustly found, independent of (limited) variations in scale, illumination and viewing angle. Keypoint extraction also includes the calculation of some characteristics of the keypoints and their neighborhoods, such that keypoints can be matched with previously recorded keypoints. Several algorithms to find these keypoints and compute their characteristics exist. These are discussed in more detail in chapter 4.

The keypoints found in the camera image are then matched with keypoints in a keypoint database. This keypoint database is generated during initialisation using a set of reference images and the same keypoint extraction algorithm as is applied to the online camera images.

Objects in the image are detected using matching keypoints. If a number of keypoints found in the camera image match with keypoints of an image of an object in the database, this can indicate that the object is present in the camera image. Chapter 5 discusses methods to test if an object is present in the camera view. If an object is present in the camera view, the object location, orientation and scale are calculated.

The pose, meaning the location and viewing angle of the camera, is estimated using the 3D location of points in the camera image. The pose estimation step, described in chapter 6, determines the 3D locations corresponding to points in the image using the location of detected objects in the camera image and their real 3D location.
When objects are defined using reference images, they are also annotated with location information. The annotations are created offline and describe the 3D location of points in the reference images.

Rendering uses virtual models and renders these models using a virtual camera that matches the real camera. This means the same camera properties such as field of view, but also the same location and viewing direction. The camera properties such as field of view are computed offline. This is discussed in appendix A. The viewing location and direction, or pose, is retrieved from the pose estimation step. The rendering process is discussed in chapter 7.

3.3 Implementation

While the key components are discussed in the following chapters, some less important implementation details are discusses in this section.

3.3.1 Camera aquisition

In previous projects (Hardy [Har11], Crijns [Cri12])one of the major bottlenecks was the image acquisition and calibration. This was partly because of the required conversions between image formats. In those approaches the camera images where captured in the camera's native YUV format. To draw a camera image using OpenGL it first had to be converted to RGBA and uploaded to a texture in the GPU memory. From Android API version 11 it is possible to use the camera images as textures in OpenGL ES using the SurfaceTexture class [API12]. This does not require uploading of texture data and does not require conversion. To get the image in main memory, such that OpenCV image processing can be performed the texture is downloaded from the GPU. So instead of converting and uploading the camera images, the image is only downloaded. When processing the image on the GPU as discussed in chapter 8 downloading the image data is not required making the current approach even more efficient.

3.3.2 Object database

The object database describes objects such that they can be used for detection and pose estimation. Objects are described by images, location annotations and possibly regions of interest.

For each object one or more images need to be supplied. Using multiple images taken under different viewing conditions enables better object recognition. If the object does not occupy the entire image a region of interest can be defined.

For each image one or more points have to be annotated with a 3D position.
Using at least four points for each image is recommended. Since at least four points are required for pose estimation, annotating at least four points enables pose estimation even when only a single object is detected.

The objects in images, called the reference images throughout this thesis, have to be (near) planar surfaces. This is a constraint used by multiple steps in the AR framework, such as object recognition and pose estimation. This is because only certain points in the camera image are matched with points in the reference images. The matching between all other points are found by interpolation. This is only a valid method for planar surfaces.

Once objects are defined using the reference images, regions of interest and annotations, the framework will create the keypoint database used during object recognition. Currently the OpenCV API does not allow storing and reloading of keypoint data using the Java API, therefore the keypoint database is generated when the application is initialised. This increases the load time and therefore the maximum database size. Support for keypoint storing and loading is on the OpenCV roadmap.

3.4 Summary

This thesis describes the implementation of an AR framework for Android devices and uses the OpenCV library where applicable to perform image processing. The framework consists of several steps. First a database describing objects is created. This database has reference images annotated with real location information. Using the reference images objects can be detected in camera images. When objects are detected the real location of these objects is used to determine the camera pose. Once the pose is determined virtual objects can be correctly rendered in the camera image.

Chapter 4 to 7 describe several steps of the AR framework in detail. Finally optimization techniques applied are discussed in chapter 8.

4 Local Invariant Features

The AR application uses local invariant features as a first step in object recognition. The first section explains the general concepts involved, how these can be used and what the desired properties of a local invariant feature detector are. This theory is the basis for the explanation and comparison of different local invariant feature detectors in the following sections.

Finally matching algorithms are discussed. These matching algorithms compare features calculated by the different algorithms. Using matching, features from different images can be related.

4.1 Local Invariant Features

There are several approaches to computer vision and specifically object recognition. When the type of objects that have to be detected is known and can be characterised by certain properties, the object recognition system can be implemented by an algorithm that performs segmentation and detection steps to search for these characteristics.
For example, the work by Hardy [Har11] detects markers in a camera image. Since these markers consist of several straight lines with high contrast, the segmentation steps include the detection of high contrast areas and line segments within these areas.

While such a specialised detection algorithm works well when the type objects which have to be detected are known, this approach does not work as well when the object recognition system has to be able to recognise a wide variety of objects with different characteristics.

A major inspiration for general purpose computer vision systems has been the biological vision system. Research suggests that low level human vision works by finding edges, or gradient changes at various scales. Using these edges an internal representation of the visual scene is created. Then objects are classified matching gradients at various keypoints. This general model of the human vision system is also at the basis of the algorithms discussed in this chapter. [Low04]

The algorithms discussed in this chapter use local invariant features to describe images. The idea is that first keypoints are localised in an image. These keypoints are points that have "interesting" features. This means the keypoints are located in the image at points which are distinctive and descriptive for the image. After finding the set of keypoints these points are described by a descriptor. These descriptors capture what makes the keypoints interesting. These descriptors should also be comparable, such that different images can be compared to see if they contain the same sort of interesting points, and thus may contain the same objects. To make this matching possible when two images contain the same object, but under different viewing conditions, the descriptors should be invariant to some properties while remaining distinctive for other properties. For example to recognise objects that are moved or captured under different lighting conditions, the descriptors should be invariant to rotation and illumination changes, while they should be distinctive enough to recognise which features are similar.

Because the keypoints are local points in an image and are described in a way such that the descriptions are invariant to certain changes, these keypoints are called "local invariant features".

The local invariant features and their preferred properties are outlined below. These properties are an adaptation of the more general description in the extensive survey paper by Tuytelaars et al.[TM08]. This thesis discusses local invariant features in the context of the AR framework.

Local features A local feature in the context of image processing is a point or area in an image which differs from its immediate neighborhood. It is a point or area where one or more image properties such as intensity, color or texture change.

Invariant features An invariant feature is a feature which does not change under a certain family of transformations. For example the surface area of a shape does not change under rotation. In this example the feature called *surface area* is invariant of changes in *rotation*. The properties calculated around a feature can be (partially) invariant to a number of transformations such as rotation, scale, translation, affine transformations or illumination changes. When an algorithm is invariant to certain changes enables comparison of descriptors when the property changes. For example an object in an image should still be recognised when it is rotated, therefore the descriptor should be invariant to rotations.

Local invariant feature detector A local invariant feature detector comprises a part which can detect feature locations in an image and a part which creates a description of the feature. Usually some measurements are taken around a detected feature point and converted into a descriptor which describe the various properties around that point. These are measurements such as changes in image intensity or the appearance of certain colors.

Use of local invariant features Local invariant features are of special interest because they provide a limited set of well localized and individually identifiable and comparable anchor points which can in turn be used for object detection. Especially for AR applications it is important that features are invariant to small changes in for example scale, rotation and illumination such that objects in a camera image can be detected under varying viewing conditions.

Properties of an ideal local feature detector An ideal feature detector should have the following properties:

Repeatability Two images of the same scene, but under slightly varying viewing conditions, should result in a high number of corresponding features being detected in both images. Repeatability is achieved by having invariance to certain transformations and robustness to small variations.

Distinctiveness/Informative The descriptors of the features should be distinctive such that they can be matched with other features.

Locality While the feature descriptors are calculated using an area around a point, the feature should be well localized. This is especially important for AR since further pose calculations are based on the position of feature points in the camera image.

Quantity It is important to have enough features such that even small objects in the image can be detected, but not too many such that the computational cost remains low.

Accuracy The detected features should be accurately localized in image location, scale and shape.

Efficiency Feature detection should be inexpensive to compute and the descriptors should be easy to match.

Obviously some of these goals conflict with each other. For example *distinctiveness/informative* and *robustness* conflict since a feature detector which is insensitive to small changes might not be able to distinguish points which only differ a little bit. As such the importance of these properties depends on the application. For an AR application repeatability is the most important characteristic. Repeatability of a feature detector enables object detection under various viewing conditions. Furthermore distinctiveness should be such that detected points can be matched with reference points in a database. Finally efficiency is of importance since the AR application should run on an embedded device with limited computational power.

4.2 Algorithms

A local invariant feature algorithm comprises two parts. The first part localises features in the image. The second part creates descriptors of the features. While the descriptor part varies greatly in different algorithms, the keypoint localisation is based on the concept of corners in most approaches.

Keypoints should have the properties described in section 4.1, which includes repeatability, distinctiveness and locality. As described earlier, areas in an image with strong gradients, or edges, could provide points of interest. However because an edge consists of many points they do not provide good keypoints because points can not be well localised. Corners provide a better basis for keypoints. Corners are similar to edges, but instead of a strong gradient in one direction, a corner is defined as a point with a gradient in more than one direction[1]. Since corner-based keypoints are repeatable, localisable, accurate and informative, most algorithms use a corner detector as the base for keypoint localisation.

For the AR framework three keypoint detection and description methods are implemented. These are implemented mostly using OpenCV functionality. These methods are the Scale-Invariant Feature Transform (SIFT) by Lowe [Low04], Speeded Up Robust Feature (SURF) by Herbert Bay et al. [BETVG08] and Oriented FAST and Rotated BRIEF (ORB) by Rublee et al. [RRKB11]. These will be discussed in more detail below. All three methods discussed in detail work with pixel intensities, or grayscale images. Therefore the camera image is converted to grayscale as an initial step in all three algorithms. Extensions to these algorithms, such as the inclusion of color information, are discussed in section 4.2.4.

[1]Early corner detectors searched for edges, and then these where searched for corners, giving rise to the name corner detectors. Newer algorithms use the definition of multiple gradient directions. Using the newer definition a single white point in a black area is considered a corner. It turns out that these points are useful and should be included. However the term "corner" is not completely what one traditionally expects but is still commonly used.

4.2.1 SIFT

The Scale-Invariant Feature Transform (SIFT) algorithm by Lowe [Low04] is a very popular approach. It is one of the older techniques but is still widely used today.

The SIFT detector locates keypoints by searching for corners. This is done by looking at an approximation of the second spatial derivative of the image intensity at different scales. Using different scales is done to make the detector invariant to scale changes. The approximation of the second spatial derivative is used to find points which have a strong gradient change in the original image. Because edges also have strong gradient changes the points are tested to see if they are part of an edge or a corner. Then the keypoints locations are refined to sub-pixel accuracy and low contrast keypoints are removed. Finally the directions of the gradients around the keypoints are analysed to find the dominant direction of the keypoint. If multiple dominant directions exist, a keypoint is added for each dominant direction.

The descriptor is created by creating several histograms of the gradient directions in sections around the keypoint. The gradient directions are normalised using the keypoint's dominant orientation to make the descriptor rotation invariant. The histograms are interpolated to reduce boundary effects.

Detector

As noted in the previous section an approximation of the second spatial derivative is used to find points which have a strong gradient change in the original image. The second order derivative at a certain scale can be created using the Laplacian of Gaussian (LoG). The LoG is created by blurring the image using a Gaussian filter and then applying the Laplace operator. The Laplace operator is a second order differential operator. The amount of blurring, defined by a parameter σ which defines the scale of the LoG.

Because the LoG is expensive to compute, an approximation of the LoG, called the difference of Gaussians (DoG), is used instead. DoG images are created by blurring the input image using a Gaussian filter at different scales. Then these blurred images of different scales are subtracted. When the scale σ used to blur the images is chosen carefully, the resulting image is an approximation of the Laplacian of Gaussian (LoG). An example input image and a DoG are shown in figure 4.1 and 4.2. Note that the second derivative can be both positive and negative, therefore the image is normalised to the range $[0, 255]$.

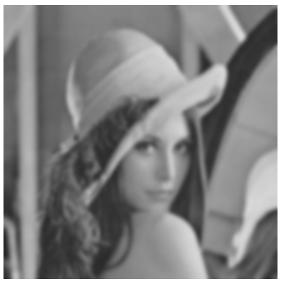


Figure 4.1: *Input image*

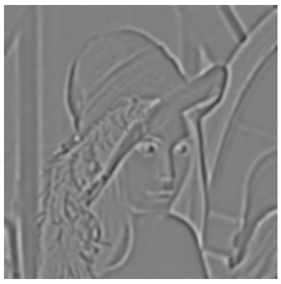


Figure 4.2: *a Difference of Gaussian*

Since feature detection should be invariant to scale, the DoG images at different scales are created. This is achieved using what is called scale space. Scale space first described by Witkin [Wit83] and further described by Lindenberg [Lin94] is a continuous function of scale. This means that points are defined in three dimensions, the spatial x and y dimension and a scale dimension σ. The scale space function $L(x, y, \sigma)$ is defined from the gaussian function $G(\sigma)$ and the input image $I(x, y)$ as:

$$L(x, y, \sigma) = G(\sigma) * I(x, y) \tag{4.1}$$

Where $*$ denotes a convolution. When this continuous function is sampled at different scales, this would result in the scale space sampling shown in figure 4.3. The images at higher scales, these are

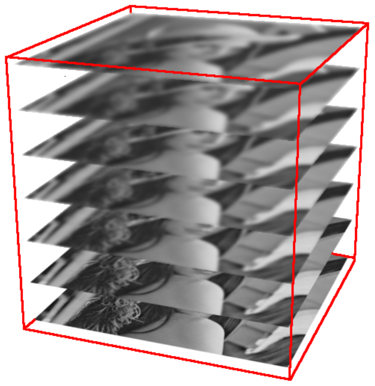


Figure 4.3: *Sampling of levels in scale space*

the images higher up in the stack shown in figure 4.3, are blurred a lot and thus have little detail. Therefore they can be stored and processed at a lower resolution. This down-sampling results in better performance without loss of quality. When images at higher levels of the scale space are smaller, the stack shown in figure gets a pyramid shape. Such a scale space is therefore called a scale space pyramid.

Two scale space pyramids are made. A Gaussian pyramid containing the blurred images, and the DoG pyramid, which is created by subtracting images from the Gaussian pyramid. Because images of different scales have to be compared, which is easier if images have the same resolution, the scale space pyramid is down-sampled in steps. Multiple images at the same resolution are made, but at different scales. These are called "levels". Then the image is down-sampled after which another set of levels is created. Each set of levels is called an "octave".

This process is outlined in figure 4.4. A grayscale image is blurred for multiple levels. These levels are subtracted to create the DoG levels. Then the image is down-sampled and used as input for the next octave.

Possible keypoints are located by finding extrema in the DoG scale space. This means the DoG images are searched for minima and maxima. A minimum or maximum needs to be lower or higher than the all the neighbours in an image at a certain scale, and the neighbours in the two neighbouring scales. In figure 4.4 the neighbours that are inspected to see if the green point is an extremum are displayed in red. This means the green point is considered an extremum if it is either the maximum or minimum of the 3×3 region around the point.

To locate minima and maxima in the DoG pyramid at n scales in each octave, the DoG pyramid needs to have $n + 2$ levels in each octave. This is because a lower and higher scale need to be inspected when locating the extrema. The Gaussian pyramid needs $n + 3$ levels in each octave because the DoG levels are created by subtracting a level $i - 1$ and from level i in the gaussian pyramid. The scales of the gaussian pyramid, determined by σ, are chosen such that the levels in which keypoints are searched have a constant increase in scale.

The extrema that are found in the DoG images are candidates for keypoints. The location of these candidates is refined to sub-pixel accuracy by fitting a 3D quadric function to the local sample. This

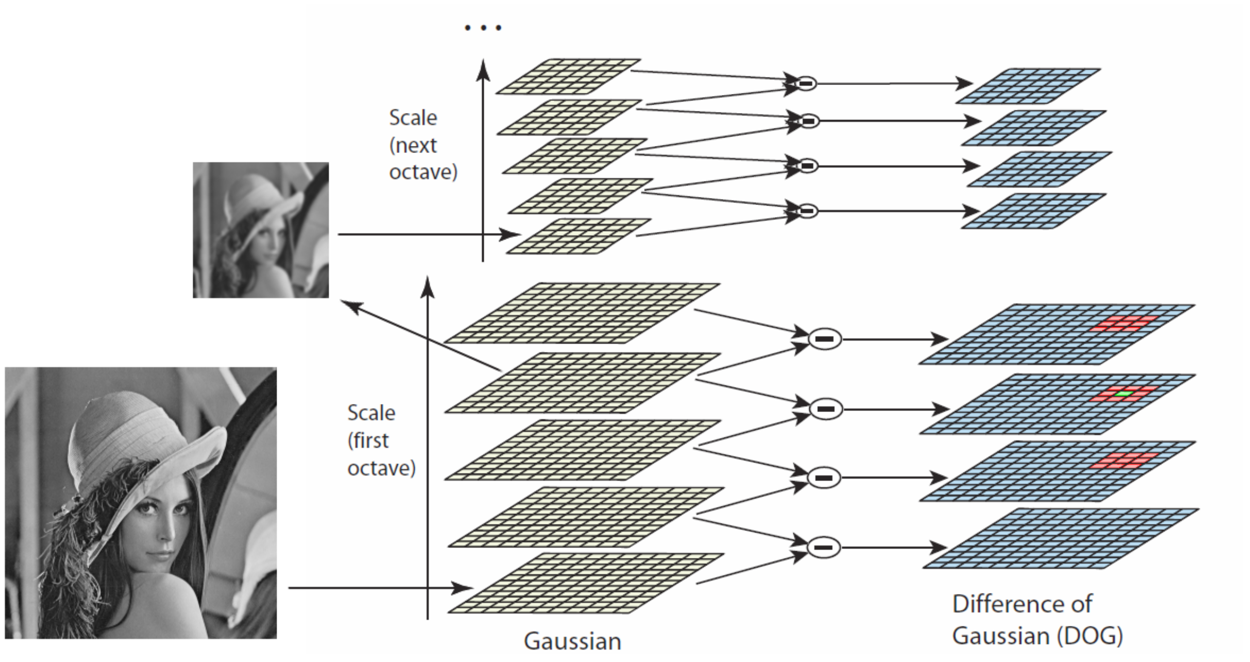


Figure 4.4: *From an input image multiple levels are created. Levels are subtracted to create a DoG. Then the image is down-scaled and the process is continued in the next octave. Extrema in the DoG levels are detected by looking at neighbours in spatial and scale dimensions. Image is partly from [Low04]*

method was developed by Brown and Lowe [BL02]. The idea is that the extremum might not lie exactly in a pixel center. Therefore the derivatives of the area around the extremum are used to estimate the true position of the extremum. The derivatives are estimated using pixel differences. In effect this is the reverse of the anti-aliasing, which is used when drawing shapes defined in real coordinates on a canvas with a limited number of pixels. In the case of sub-pixel refinement the image is anti-aliased, and the real-valued locations must be found.

After sub-pixel localisation keypoints with low contrast are discarded. The contrast of keypoints is determined using the gradients at the keypoint location. This is efficiently computed using by re-using the gradients computed for the sub-pixel localisation.

After low contrast suppression keypoints are tested to see if they are located at corners or edges. A corner is a point which has an edge in two perpendicular directions. A Hessian matrix H as shown in equation 4.2 is created. Function $L(p, \sigma)$ stands for the Laplacian of Gaussian at point p at scale σ. $L_{xx}(p, \sigma)$ and the variants stands for the second order derivative of this function in the corresponding x and y directions. In SIFT the DoG images are used as an approximation for $L(p, \sigma)$ and the derivatives are calculated using pixel differences in the DoG images at the location and scale of the keypoint.

$$H(p, \sigma) = \begin{bmatrix} L_{xx}(p, \sigma) & Lxy(p, \sigma) \\ L_{xy}(p, \sigma) & L_{yy}(p, \sigma) \end{bmatrix} \tag{4.2}$$

Calculating the eigenvalues of H will result in the principal curvatures of L at the keypoint. If there is one large and one small eigenvalue the keypoint is located at an edge while two large eigenvalues denote a corner. Unfortunately eigenvalues are expensive to calculate. But because only the ratio between the eigenvalues is needed, not their actual values, a more efficient method is possible. The technique, first proposed by Harris and Stephens [H88], uses the trace and determinant of the Hessian to determine the ratio of the principal curvatures. This is efficient to compute.

Finally the orientation of the keypoint is calculated. This is done by calculating the gradient angle and strength of points in a region around the keypoint. For each point the strength multiplied by a gaussian function such that points further away from the keypoints have a smaller influence. Then a histogram is created where each bin covers 10 degrees. Each weighted strength is added to the appropriate bin. The maximum of the bins in the histogram determines the keypoints orientation. If there are multiple maxima of similar magnitude, multiple keypoints are added at the keypoint location, one for each maximum.

Descriptor

The SIFT descriptor is based on the histograms of regions around the keypoint. The reason histograms are used instead of a list of points is that these histograms represent a region. As such the exact location of a point is less important, it will contribute to the same histogram if it is anywhere in the region which the histogram represents. This enables better keypoint matching when for example the viewpoint from which the keypoint is viewed changes. This idea is based on biological vision. As shown by Edelman et al. [EIP97], the biological vision system seems to work by detecting gradients at various scales, but the location of these gradients is allowed to shift over an area.

The calculation of the descriptor begins by computing the gradient angle and magnitude of points around the keypoint. The region depends on the scale of the keypoint. Next the angles are rotated relative to the keypoint orientation and the magnitudes are weighted by a using the gaussian function $G(x, y, \sigma)$ as shown in equation 4.3

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{4.3}$$

Where (x, y) is the location of the point relative to the keypoint and σ is the strength of the gaussian. A σ of half the keypoints are is used. Next 4×4 histograms are created which represents the weighted magnitude of angles in a subarea of the keypoint area. To prevent boundary effects the weighted magnitudes are added to the two closest bins in the four closest histograms. The contribution to the different bins in each histogram is weighted depending on how close the keypoint angle is to the center of those bins and the distance from the location of the point to the location of the center of the area the histogram represents. Figure 4.5 shows a representation of the histogram calculation step. The image shows a 2×2 descriptor array with 8×8 samples. The implementation uses a 4×4 descriptor array calculated from 16 samples. The 4×4 histograms each have 8 direction bins. Concatenating the values of all histograms results in a $4 \times 4 \times 8 = 128$ element feature vector. This vector is normalised to unit length to make the descriptor more invariant to changes in illumination. After normalisation all values are clamped between 0 and 0.2. This way large variations in lighting are dismissed, and thus more emphasis is on the gradient directions than the magnitudes. Finally the vector is normalised again.

4.2.2 SURF

Speeded Up Robust Features (SURF) is an algorithm that is inspired by SIFT conceptually but calculates the keypoints locations and descriptors completely different. SURF is designed to be more efficient to compute than SIFT while having similar matching performance.

The SURF detector searches for corner points by finding extrema in the determinant of the Hessian matrix H. This Hessian is the same as described an equation 4.2. Although the same Hessian is used in SIFT, the way it is calculated in SURF is different.

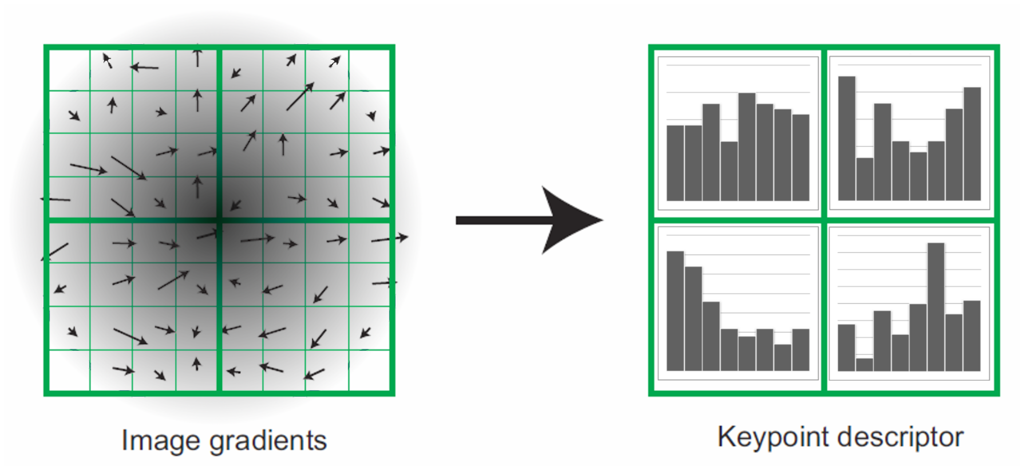


Figure 4.5: *On the left the area around a keypoint is displayed. The length and direction of the arrows indicate the normalised gradient direction and magnitude. The magnitudes are weighted by a gaussian window, indicated by the gradient. The right shows the histograms created for the different regions. The image is a modified version from [Low04]*

Similar to SIFT, the second derivatives of the image intensity at various scales need to be calculated. The derivative of an image convolved with a Gaussian filter is the same as the convolution of the image with the derivative of the Gaussian filter. The same holds for the second order derivatives. So if the derivatives of the Gaussians are calculated in advance, L_{xx}, L_{yy} and L_{xy} in equation 4.2 can be calculated in a single convolution. The reason SIFT did not use this, but rather performed the Gaussian filtering and derivative approximation separately, is that the Gaussian filters can be separated in a horizontal and vertical filter. If two equal-valued, one-dimensional Gaussian filters, where one is vertical and one is horizontal, are convolved, the result is the two dimensional Gaussian filter. Because convolution is associative (e.g. $f * (g * h) = (f * g) * h$), this means using the two one dimensional filters has the same effect as the two-dimensional filter. A 13x13 filter normally requires $13^2 = 169$ multiplications for each pixel. But the Gaussian, convolution can be performed by a convolution with a 13x1 filter followed by a convolution with a 1×13 filter. This results in $2 \times 13 = 26$ multiplications. Therefore the implementation used by SIFT is more efficient than using a single convolution. SURF reduces the computational complexity of computing the convolutions by using an approximation of the filters and an integral image. The integral image, which will be explained in more detail in section 4.2.2, is computed as a preprocessing step. Using this integral image the convolutions are computed using only a few lookups. The resulting images are searched for extrema in a 3×3 region similar to SIFT.

In SURF the gradients of points, needed for the keypoints orientation and the descriptor are calculated using Haar wavelets. These can be calculated efficiently using the integral image.

The SURF descriptor, like SIFT, creates a region divided in 4×4 sections around the keypoint. This region is rotated using the keypoint's main orientation to achieve rotation invariance. Again the orientation and magnitude of the points in the sections are used to create a description of the section. While SIFT created a histogram for each region, SURF computes the sum of the x and y components of all gradients and the absolute values of the x and y components of the gradients. This results in shorter descriptors making them faster to match.

Detector

The maxima of the determinant of the Hessian in scale space is used to locate corner points. The Hessian is repeated in equation 4.4.

$$H(p,\sigma) = \begin{bmatrix} L_{xx}(p,\sigma) & Lxy(p,\sigma) \\ L_{xy}(p,\sigma) & L_{yy}(p,\sigma) \end{bmatrix} \tag{4.4}$$

To find these points a scale space pyramid containing an approximation of the determinant of the Hessian is used. The scale space pyramid is similar to the scale space pyramids constructed in SIFT, containing several octaves, each containing several levels. But the images in the scale space pyramids in SIFT are the Gaussian and DoG images, while the SURF scale space pyramid contains the approximation of the determinant of the Hessian.

The determinant of the Hessian is:

$$det(H) = L_{xx}L_{yy} - L_{xy}^2 \tag{4.5}$$

L_{xx}, L_{yy} and L_{xy} can be computed by convolving the image with the derivative of a gaussian. A Gaussian filter is displayed in figure 4.6. The first-order partial derivative with respect to x is given in figure 4.7, and the second-order mixed derivative with respect x and y is shown in figure 4.8. A convolution of an image with a filter like the one in image 4.8 corresponds with L_{xy} in equations 4.4 and 4.5. As visible in the image the kernel size for the derivatives should be larger than the filter for the corresponding Gaussian filter. Near the edges the values should be near zero, which is not the case in figures 4.7 and 4.8.

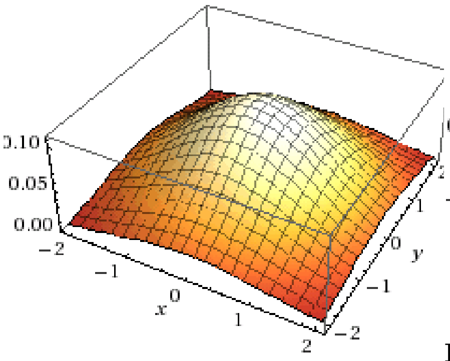


Figure 4.6: *Gaussian filter with* $\sigma = 1.2$*,* $G(1.2)$

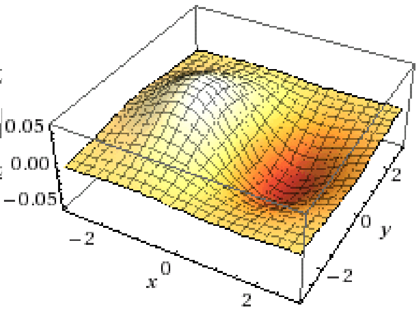


Figure 4.7: *First order derivative,* $\frac{\partial}{\partial x}G(1.2)$

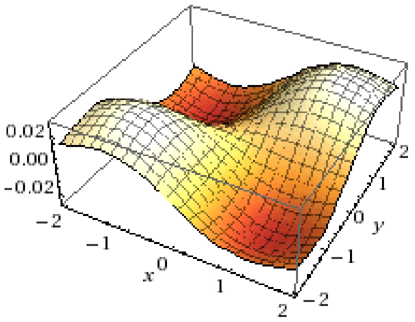


Figure 4.8: *Second order derivative,* $\frac{\partial^2}{\partial x \partial y}G(1.2)$

As noted in the introduction, computing a 2D dimensional kernel which is not separable is expensive. Because derivatives of Gaussian filters are not separable and larger than the original filters the convolution of these filter would require too many computations. Therefore an approximation of the filters is used as shown in figure 4.9 and 4.10. Figure 4.9 shows a discretized version of the second order partial derivative with respect to y. Because the filter contains positive and negative values the image is scaled such that gray represents zero. The approximation shown in figure 4.10 is created by making regions which have a constant value. Similar approximation filters are made for the second order partial derivatives in the x direction and the y direction.

Using these approximations, calling them D_{xx}, D_{yy} and D_{xy}, an approximation of the determinant of the Hessian is calculated as shown in equation 4.6:

$$det(H_{approx}) = D_{xx}D_{yy} - (wD_{xy})^2 \tag{4.6}$$

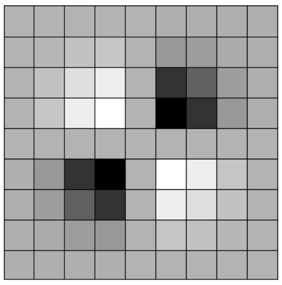


Figure 4.9: *Second order derivative,* $\frac{\partial^2}{\partial x \partial y} G(1.2)$*, image from [BETVG08]*

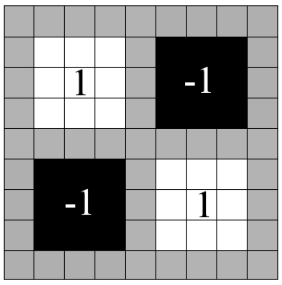


Figure 4.10: *Approximation, image from [BETVG08]*

Here a weight w is added which results corrects for approximations errors in these new filters, such that energy conservation is achieved.

The main reason these filters are changed such that they comprise of several constant regions, is that there is an efficient way to compute the sum of all pixel intensities in a region using integral images, as explained in the next paragraph. These area sums can be computed using four lookups in an integral image and some additions. So to compute the filter response, for each region the sum of all pixels that correspond to a pixel region are computed using the integral image and then multiplied by the region intensity (either 1 or -1 in figure 4.10). So instead of multiplying all points in the filter, a fixed number of lookups and additions is used. This requires four lookups per region. So to compute the filter shown in figure 4.10, which has 4 regions, $4 \times 4 = 16$ lookups are required. This is the same for any filter size, where the cost of a convolution with the original filter would grow exponentially with filter size.

In integral images, also known as *summed area tables*, each pixel $p = (x, y)$ contains the sum of all pixel values above and to the left of the point in the source image. The integral image I is defined using a source image i as:

$$I(x, y) = \sum_{\substack{x' \leq x \\ y' \leq y}} i(x', y') \tag{4.7}$$

This image is computed efficiently in a single pass. The sum of all pixel values in any axis aligned rectangle can be computed using the integral image values of the corner points of that rectangular region. For example the sum of all pixel values in the region shown in figure 4.11 can be computed using the values of the integral image at points A, B, C and D as shown in equation 4.8.

$$\sum_{\substack{A(x) < x' \leq C(x) \\ A(y) < y' \leq C(y)}} i(x', y') = I(C) + I(A) - I(B) - I(D) \tag{4.8}$$

The scale space pyramid of determinants of the Hessian is thus constructed by first creating a integral image. Then the different approximation filters are used to to determine the determinant of the Hessian at a point as shown in equation 4.6. Because the filter can be evaluated at any size the same integral image is used for all octaves and levels. The only thing to note is that the filters are scaled such that they contain a center point, meaning their width and height must be an odd number.

The actual keypoint locations are determined by looking for extrema in a 3×3 region in the scale space pyramid, similar to the approach of SIFT. Also similar to SIFT the method of Brown [BL02] is used to interpolate in image and scale space.

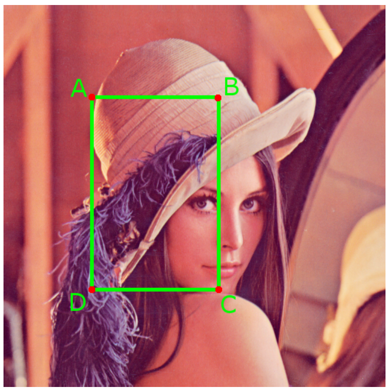


Figure 4.11: *Rectangle in image defined by corners A, B, C and D*

The keypoints orientation is calculated similar to SIFT, meaning the orientation and magnitude of the gradients of points around keypoint are determined and the magnitudes are weighted using a gaussian window. The main difference is the way the orientation and magnitude of points are calculated and how the main orientation is extracted from these values. The gradients are calculated using a Haar filter as shown in figure 4.12 and 4.13, where black means -1 and white $+1$. These filters are also evaluated using the integral image. The keypoint orientation is calculated from these responses using a sliding window of size $\frac{\pi}{3}$, and finding the orientation such that the sum of the responses in the window is maximised. This is shown in figure 4.14, where responses are plotted as blue points, the orientation is shown by the red arrow and the window is shown by the grey area.

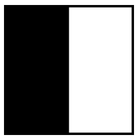


Figure 4.12: *Haar wavelet filter to compute response in x-direction, image from [BETVG08]*

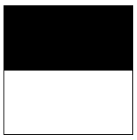


Figure 4.13: *Haar wavelet filter to compute response in y-direction, image from [BETVG08]*

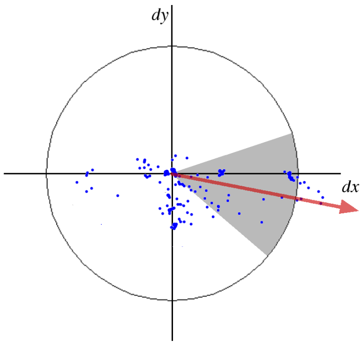


Figure 4.14: *Keypoint orientation detection using sliding window*

Descriptor

The SURF descriptor, again similar to SIFT, creates a region around the keypoint with a size depending on the scale in which the keypoint was found. This area is divided in 4×4 sections. The area is rotated depending on the keypoint orientation to create rotation invariance. For all point in the region the magnitude and orientation are computed, where the magnitude is weighted using a Gaussian window.

The difference with SIFT is how the orientations and magnitudes are computed and how a descriptor is created from these values. The orientation and magnitude of the gradients of the points in the region around the keypoint are computed using the Haar wavelet filter also used during keypoint orientation calculation. And where SIFT creates interpolated histograms from the orientations and weighted magnitudes, SURF uses different sums of these values.

The descriptor is calculated using the sum of the weighted response of the Haar filter in the x and y direction, called d_x and d_y. The descriptor d for a single region is shown in equation 4.9

$$\vec{d} = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|) \tag{4.9}$$

The descriptor is generated by concatenating the descriptor vectors from the 16 sub regions. Because a single vector consists of four values, the total descriptor length is $16 \times 4 = 64$, half the size of the SIFT descriptor. This results in faster matching compared to SIFT descriptors.

4.2.3 ORB

The ORB is based on the FAST keypoints detector and BRIEF descriptor. The ORB method adds a orientation component to FAST and rotates BRIEF to make these methods rotation invariant. The method is developed to be a much faster alternative for SIFT or SURF.

Detector

The FAST keypoint detector by Rosten and Drummond [RD06], works by inspecting the points in a circle centered around a point. If all the pixels in a continuous section of the circle are higher than the center point plus a threshold or lower than the center point minus the threshold, the center point is a keypoint candidate. Because all pixels in a continuous section need to be either high or low, a circle can often be rejected after several comparisons. This makes the FAST method fast.

The FAST method will also return keypoint candidates located on edges, therefore ORB uses a Harris corner measure, also used by SIFT and SURF, to reject edge points. The FAST detector only detects features of a certain scale. Therefore the input image is down scaled several times, and keypoints are detected in each scale. The orientation of a keypoint is determined using the intensity centroid [Ros99]. The angle θ of a point is determined as shown in equation 4.10

$$\theta = atan2(m_{01}, m_{10}) \tag{4.10}$$

where

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y) \tag{4.11}$$

where $I(x, y)$ is the image intensity at point (x, y).

Descriptor

For the descriptor an oriented version if the BRIEF descriptor by Calonder and Lepetit et al. [CLO+12] is used. The BRIEF descriptor is a binary string based on the relation of pairs of points in an image patch around a keypoint. Each pair is tested to see which point has the largest image intensity. So for each pair p of points $p = (x, y)$ where x and y are points in the image patch I, the binary value b_p is defined by:

$$b_p = \begin{cases} 1 & \text{if} I(a) < I(b), \\ 0 & \text{if} I(a) \geq I(b) \end{cases} \tag{4.12}$$

The BRIEF descriptor d is the vector of the binary values of 256 pairs:

$$d = \sum_{i=1}^{256} 2^{i-1} b_i \tag{4.13}$$

Each pair consists of two points. All points are randomly generated where the probability of each points is determined by a Gaussian distribution around the image patch center. This means that points near the center are more likely to be chosen. So to create the 256 pairs, 512 points are randomly chosen. Such a set of pairs is shown in figure 4.15. To create a oriented version of BRIEF, the pairs

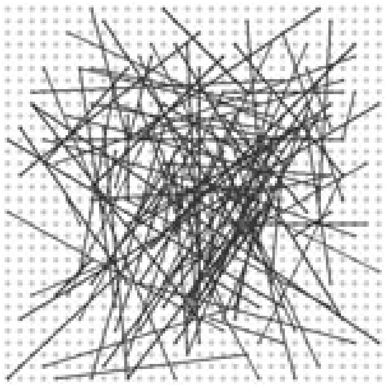


Figure 4.15: *Pairs used in BRIEF descriptor, created using a random gaussian distribution around the center. Image from [CLO+12]*

are rotated depending on the keypoint orientation. Furthermore a different distribution is used of pairs. When a distribution such as the one shown in figure 4.15 is used for rotated keypoints is not the optimal distribution.

The orientation of a keypoint depends on the dominant gradient in the image patch around the keypoint. If an image patch is rotated depending on the keypoint orientation, the dominant gradient will always have the same orientation, for example from the left to the right there will be a gradient from high to low intensities. Pair of points around the horizontal axis will most likely have the same result for all keypoints, meaning the point to the left will almost always have a higher intensity. So pair distributions with more pairs closer to the vertical axis will create more informative descriptors. Such a distribution is therefore used in ORB. The actual distribution is generated automatically by selecting the best pairs from a large set of random pairs. The best pairs are those of which the binary value from equation 4.12 has the largest variance.

4.2.4 Variations

A possible extension to the above algorithms, SIFT, SURF and ORB, is the use of color images instead of grayscale images. A problem when using color information is how to remain invariant to illumination changes. When the rgb values are used the algorithms becomes sensitive to illumination changes. An approach is to convert the rgb color to another color space. An example is to calculate the hue and use that together with the intensity from the original SIFT.

For SIFT alone several approaches are suggested. Some of these are HSV-SIFT [BZMn08], HueSIFT [vdWGB06], C-SIFT [AHF06] and W-colour-SIFT, C-colour-SIFT, h-Colour-SIFT, hue-Color-SIFT and hsv-Colour-SIFT [BG09].

As shown in the comparison of many of the color based SIFT variations by van de Sande [vdSGS10], the inclusion of color information has a small effect on the matching performance. His work shows a mean average precision gain of 7% in keypoint matching. The computational cost of these techniques is much higher because these techniques require color conversion, processing multiple values for each pixel, for example three when using RGB, and results in much larger feature vectors, which are more expensive to compare. Because speed is important for the framework, the precision gain these techniques offer is not worth the extra computational burden. A possible exception might be when the framework is used for outdoor applications. Burghouts [BG09] shows that changes in illumination direction has a negative influence on SIFT performance. In these cases the color based SIFT variations have a larger performance gain than the 7% mentioned by [vdSGS10]. In outdoor applications the main light source is the sun, which moves during the day, and as such color based SIFT variations can be considered.

Another variation to algorithms is to compute more distinctive descriptors. Such a technique is PCA-SIFT as described by Ke and Sukthankar [KS04]. In this work SIFT is modified to use Principal Components Analysis (PCA) to analyse the gradients around a keypoint. This creates a more compact descriptor holding more information. The research suggest that this result in faster and more accurate matching. However the computational cost of PCA is much higher than the histogram based method used in the original SIFT method. Therefore the overall cost of PCA-SIFT is higher. Furthermore, a comparison paper by Juan [JG09] suggest that PCA-SIFT only performs better in terms of repeatability when illumination changes, while it performs much worse under affine transformations.

4.2.5 Comparison and Conclusions

SIFT, SURF and ORB are implemented and tested. Research such as [TL12], [Bul12] suggests that SIFT has the best matching performance overall. This means it scores consistently high for several test cases where variations in scale, rotation and viewpoint are considered. Other research suggest the opposite, such as the papers of SURF and ORB [BETVG08], [RRKB11]. This suggests that there are different cases in which the different algorithms excel. Because different papers test the same properties like repeatability and have different results it is important to test the algorithms in a setting in which the performance is tested with respect to the applicability for AR application.

An application is used to test the performance of the different algorithms in an AR setting. This application has a database with ten distinct objects. Keypoints matching is performed and from the keypoints a homography is calculated as described in chapter 5. Using the homography the bounding boxes of detected objects are drawn in the camera image as shown in figure 5.1. By manual inspection of the appearance of the bounding boxes the performance of the different algorithms is determined. Several test are performed. First the correctness of the bounding box is tested. If it appears, does it appear at the correct location? The next question is when does the bounding box appear? This is tested in which range the bounding boxes are shown, which is tested by both moving the camera towards and away from the object, by looking at the objects at increasing angles as well as rotating the camera. Combinations of these movements are tested as well. Next the consistency of the errors is tested. This means is the object stationary when the camera is stationary, or does the bounding box move/jitter. This is also tested using constant movement. Finally the speed of the algorithms is determined by looking at the framerate during the above mentioned tests. By looking at the framerate all aspects are considered. This means keypoints detection, description and matching.

While different settings have been tested, the default settings in OpenCV, which correspond to the values used in the original papers of each method, resulted in the best performance. Therefore the experiments used the default settings.

The experiments showed that SIFT performed the best, followed by SURF, while ORB performed quite badly. The difference between SIFT and SURF was mainly that the range, in both the scale, viewing angle and rotation dimension, in which good results were achieved where a bit smaller for SURF. Also within the range in which good results where achieved, sometimes using the SURF method the bounding box moved a bit from frame to frame, while the SIFT method gave more consistent results. The ORB methods had a very small range in which a bounding box created, especially in the scale dimension. And there was a large variation in the bounding box location and dimensions from frame to frame. The conclusion of the quality tests is that both SIFT and SURF seem viable options while ORB is not suited for the AR framework.

When looking at the performance ORB is much faster than either SURF or SIFT, and SURF is a bit faster than SIFT. SURF achieved a little bit over a frame per second while SIFT achieved a little bit under a second.

Due to the quality test, either the SIFT or SURF method seem viable options, where SIFT the best choice. The difference in framerate between SIFT and SURF is relatively low, but SURF performs better in terms of speed. While SIFT is a bit slower it is expected that a GPU implementation of this method will have a bigger relative speed gain when compared to a CPU implementation of the SIFT algorithm, because the filtering steps used by SIFT map well to the parallel nature of the GPU.

Because of the superior quality SIFT and because it lends itself better for a GPU implementation, SIFT will be used as reference throughout the implementation and in the results in chapter 9. Although the results will be created using the SIFT method, the SURF and ORB methods are also available in the AR framework.

4.3 Matching

Each of the above methods result in a feature vector describing a keypoint. To find similar keypoints these feature vectors from different keypoints are compared. These vectors consist of a number of values. The similarity between feature vectors is defined by the euclidean distance. So to find a match for a certain keypoint in our camera image the nearest neighbour, using the euclidean distance between vectors, has to be found.

4.3.1 Second Nearest neighbor filtering

Keypoint matching is based on similarity to allow for small difference between keypoints. This is required to make the matching robust under variations such as viewing angle or changes in illumination. The similarity matching results in a distance score of the match between two keypoints. While a simple approach to increase the reliability of matches would be to define a threshold on the maximum distance between keypoints this is not the best way to filter keypoint matches.
Imagine two images of an object with a pattern and some unique points. If keypoint detection is performed between the two images of this object it is likely that a lot of keypoints are found on the pattern and some of the unique points. Now if the keypoints from these images are matched, a lot of the points in the pattern may have a strong match with keypoints in another area of the pattern because in a pattern a lot of points are similar. So both the wrong matches and the corrects matches

will have a low distance score. When the threshold is increased to remove the incorrect matches on the pattern, the correct matches will also be removed.

This example goes to show that strong matches (small distance score) might indicate similar patches in images, but might not always include the best keypoints for object recognition. And while a pattern is a specific case, one could imagine that a large database of reference images will include a large number of keypoints which are similar, and therefore less useful for object recognition.

A solution to this problem is proposed by Lowe [Low04]. The idea is to find the best and second best match for each keypoint in the camera image and then threshold based on the ratio between the distance score of the best match compared to the second best match. This results in a stricter threshold for matchings between keypoints if similar keypoints are also present in the database. The research by Lowe shows that only accepting keypoints for which the distance score is at most 80% of the second best match is optimal.

4.3.2 FLANN

Because of the high dimensionality of the feature vectors and the high number of keypoints, finding nearest neighbours is a computationally expensive operation and brute force matching all keypoints would provide a major bottleneck. Unfortunately no better perfect solutions than exhaustive search exist. There are however approximation algorithms which give approximate solutions which are close to the correct solution but run an order of magnitude faster than brute force searching. The approximation algorithm called FLANN by Muja and Lowe [ML09] is used in the framework.

The research [ML09] showed that for different datasets, different algorithms are most efficient. There was not a single algorithm that performed best in all cases. FLANN contains several algorithms to perform approximate nearest neighbour searching, such as a priority search or randomized kd-trees searching. Depending on the dataset the most efficient algorithm is automatically selected.

As noted it is an *approximate* nearest neighbour algorithm, meaning it might not give the correct match. This does not pose a problem as noted by Lowe in the SIFT paper [Low04]. This is because the solution given might not be the exact nearest neighbour, but it will be very close to the nearest neighbour. When keypoints are distinct the approximate nearest neighbour search will most likely return the correct result. However when multiple keypoints are very similar the probability that the incorrect keypoint is returned increases. As noted in the previous section these similar keypoints will be filtered anyway and therefore using the approximate nearest neighbour search does not significantly reduce the quality of the final set of matches.

4.4 Summary

An image can be characterised by a set of keypoints. Several algorithms exist which locate keypoints at "interesting" points in an image in a predictable and repeatable manner. For these keypoints feature vectors are calculated, such that different keypoints can be compared. This is used to find similar points in different images. When strong matches are found between several points in two images this may indicate the same object is shown in both images, which is used as described in the next chapter.

Several algorithms are implemented and tested. SIFT performed the best, meaning the matches found using SIFT are often correct and well localised. SIFT is the slowest algorithm but it is expected that it is more suitable for an GPU implementation than the other algorithms. This is further discussed in chapter 8.

5 Object Recognition

This chapter discusses the detection of objects in the camera view using keypoint matches between the camera image and reference images. It details the steps taken to turn matching keypoints as discussed in chapter 4 into object locations, orientations and scales.

A matching between keypoints in a scene and one of the reference images is shown in figure 5.1. This is the result of keypoint detection and matching as discussed in chapter 4. The keypoints are shown

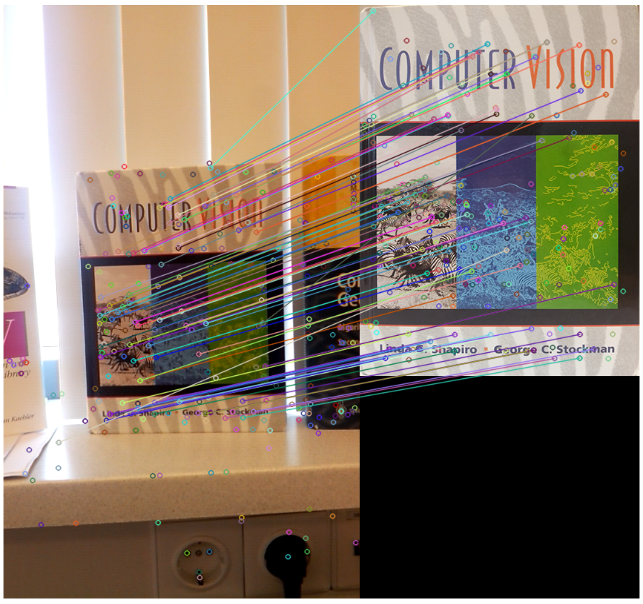


Figure 5.1: *Matching keypoints between live view and reference image*

as circles and the matches between keypoints are shown as lines between keypoints. While a number of lines are correct, there are also wrong matches or keypoints that are not matched. These errors are present because the matching process described in chapter 4 includes errors. This can not be solved by changing the matching criteria. Because making the matching more strict to reduce the number of false matches will also reduce the number of correct matches. And making the matching less strict to increase the number of matches will also introduce more false matches. When looking at single keypoints the matching process as described in the previous chapter provides an optimal balance between correct matches and false positives.

Further filtering is possible when considering multiple keypoints should match to the same object. As described in chapter 4, keypoints have a location, scale and orientation. If an object is in the camera view, multiple keypoints should match the reference image. But there is also a relation between those matches. Because objects are planar surfaces, all keypoints in the camera image belonging to a certain object should have the same spatial relation as in the reference image, aside from affine transformations introduced by movement and rotation of the object. Furthermore all points should have a similar scaling and rotation compared to the keypoints in the reference image. If many keypoints appear to be part of the same object, this is a strong indication that an object is present, but also a strong indication that these matches are correct. This is used to filter bad matches. A Hough transform, described in section 5.1, is used to find the most probable locations of objects. Then all keypoints matches are tested to see if they are part of a set of matches that agree on an object location, scale and orientation. If there are no other matches that indicate an object is in the camera view at a certain location, scale and orientation, the keypoint match is most likely a false positive, and is discarded.

After the Hough transform based filtering, the remaining set of matches is used to compute a transformation matrix which describes the relation between points in the reference image and the camera image. This transformation matrix, called a homography, is calculated in section 5.2. Once the Homography is calculated, is can be used to find the matching location of all points in both images, not just the keypoints. This is required to find the location of parts of the object in the camera image that do not have matching keypoints. For pose estimation, as explained in the next chapter, the location of points in the view that correspond to predefined points in the training image have to be known. These may not be part of the set of matched points, but can be located in the camera image using the homography.

The homography found with the object recognition steps described above can be used to find the corresponding location of all points of a reference image in the camera image. To create the image shown in figure 5.2, the location of the corner points of the reference image are located in the camera image. This is used to draw a bounding box. The green rectangle denotes the bounding box of the object in the reference image.

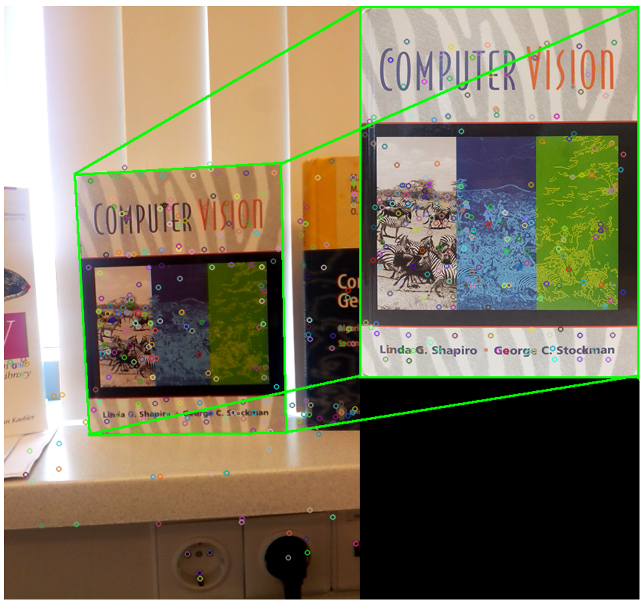

COMPUTER VISION
COMPUTER VISION
Linda G. Shapiro • George C. Stockman
Linda G. Shapiro • George C. Stockman

Figure 5.2: *Matching object in live view and reference image*

Because keypoint matching and the filtering steps described above only allow for limited affine transformations between the reference image and the camera image, object recognition fails if the angle at which the reference image is taken differs too much from the angle at which the object is viewed in the camera image. The solution is to have multiple reference images for a single object. How this is implemented is discussed in section 5.3

Finally support for multiple reference images belonging to the same object, but captured at different angles is discussed. This enables object recognition with greater differences in viewing angle.

5.1 Hough Transform clustering

As shown in figure 5.1 the matching process described in chapter 4 includes errors which can not be identified by looking at individual keypoint matches. But because all keypoints belonging to the same object lie on the same plane, all keypoints should be rotated, scaled and translated by the same amount. If many keypoints in the camera image match with a certain object and all have a similar rotation scaling and translation, they are most likely correct matches.

When looking at figure 5.1, it is easy to see that some keypoints are matched incorrectly while there is also a set of matches which seem to agree on the location of the object. These matches that agree can be identified because they have the same spatial relation between them, besides affine transformations. The idea of finding good matches by finding those that agree with each other is implemented using Hough transform clustering [Low04]. Using the Hough transform the most frequently occurring object locations, orientations and scales are determined, after which all keypoints that are not similar to those properties are discarded.

5.1.1 Hough transform background

The Hough transform, originally proposed and patented by Hough [Hou62], and refined and made popular by Duda and Hart [DH72], is a method to find the location of shapes such as lines or circles in an image. The idea of the Hough transform is define a set of parameters which describe the shape. Then points in the image are considered. Each point may indicate the presence of the shape with certain parameter values. If a point indicates the presence of such a shape, votes for these values are added in an "accumulator array", an array in which all the votes for each combination of parameters are counted. After processing all points the maxima in the accumulator array indicate the most likely parameter values.
Consider an example of using Hough transform to search for circles in figure 5.3. In the example the

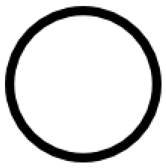


Figure 5.3: *A circle*

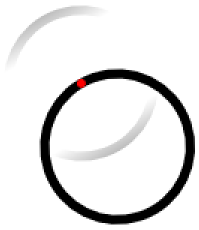


Figure 5.4: *Considering single point on edge (red dot)*

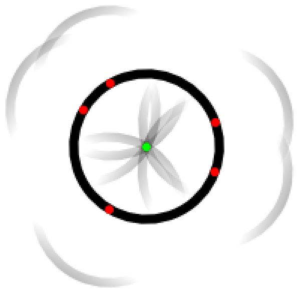


Figure 5.5: *Considering multiple points, maximum denoted by green point*

circle radius r is known, but the center point (x, y) is unknown. Since there are two unknowns, the accumulator array is two dimensional. In this example the accumulator array is drawn on top of the circle image. When a point on the circle is considered, shown by a red dot in figure 5.4, the center point location can be estimated. It must be somewhere at distance r, therefore points at distance r get a vote. If the gradient at the point is considered as well, the number of possible locations for the center point can be reduced. This is shown by the shading on the vote circle in figure 5.4. The darker region indicates likely locations for the circle center.

If multiple points on the circle are considered, as shown in figure 5.5, the point which has accumulated the most votes is the actual center point. This maximum is shown by the green dot.

The accumulator array used to store the votes consist of discrete values. Effectively the accumulator array is a multidimensional histogram. The resolution of the accumulator array can be adjusted to match the certainty by which parameters can be estimated. If in the example the circle in the input image was not a perfect circle, or if only an approximate radius was known, the resolution of the accumulator array can be lowered to increase robustness. When a lower resolution, or larger bin size, is used the robustness is increased at the cost of precision of the estimation.

Finally the Hough transform can be used to estimate any number of parameters. If for example the radius of the circle was not known, only a range of possible radii, the radius could be added as a parameter, thus requiring a three dimensional accumulator array. Then for each point votes for each for different locations at each of the possible radii are stored in the accumulator array. When this is done the maxima in the three dimensional accumulator array denotes both the circle location and circle radius.

5.1.2 Hough transform for keypoints

For the AR framework the Hough transform is used to find the probable location (x, y), scale s and orientation o of objects in the camera image. To estimate x, y, s and o a four-dimensional accumulator array is required. To support multiple objects a fifth dimension is added to the accumulator array. The object identifier is stored in this dimension. This way the Hough transform can be performed for many objects at once.

Each keypoint in the camera image is analysed too see which values it supports for these 5 parameters. As described in chapter 4 each keypoint has a position as well as an orientation and scale. The orientation and the scale of a keypoint are based on the gradients near the keypoint so they have to be normalised to get an absolute orientation and scale. The keypoints are normalised using the values of the matching keypoints in the reference image. This is shown in figure 5.6 to 5.8. Figure

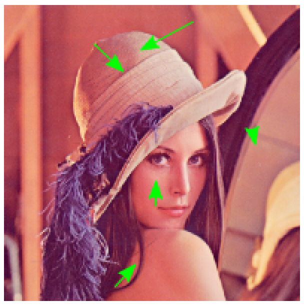


Figure 5.6: *Reference image with keypoints*

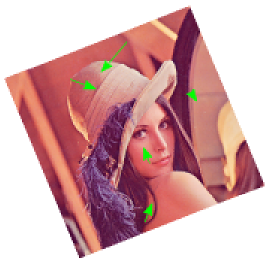


Figure 5.7: *Rotated and scaled image with keypoints*

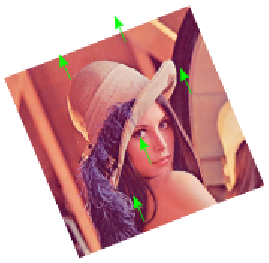


Figure 5.8: *Normalised keypoints*

5.6 shows an image with several keypoints, denoted by arrows. Each keypoint has an angle as shown by the direction of the arrows, and a scale, shown by the length of the arrows. Figure 5.7 shows the same image rotated and scaled. This is similar to how and object might appear in a camera image, although other transformations such as shearing might also be present. After keypoints are matched, the normalised values are calculated. The scale of the normalised keypoint is calculated by dividing the scale of the keypoint in the transformed image by the scale in the reference image. The angle of each normalised keypoint is the difference between the angle of the keypoint in the reference image and the transformed image. The resulting values are shown in figure 5.8.

A similar normalisation process is used to calculate the center of the object, although this is not shown in the figures. First the vector from the keypoint location to the reference image center is determined. This vector is scaled and rotated using the normalised scale and orientation. Then, using the keypoint location in the transformed image and the vector, the center of the object in the transformed image is determined.

The vote for a keypoint match v_i, given a match between a keypoint in the camera image k_i and a keypoint in the reference image k'_i, where k_i has a location, scale and orientation: $k_i = (x_i, y_i, s_i, \theta_i)$ and k'_i has a location, scale, orientation, object identifier and center point: $k'_i = (x'_i, y'_i, s'_i, \theta'_i, id'_i, c'_i)$, is defined by equation 5.1:

$$\begin{aligned} v_i &= (id, s, \theta, x, y) \\ \text{where} \\ id &= id'_i \\ s &= \frac{s_i}{s'_i} \\ \theta &= \theta_i - \theta'_i \\ \vec{v} &= c'_i - (x'_i, y'_i) \\ x &= x_i - \vec{v}_x \\ y &= y_i - \vec{v}_y \end{aligned} \tag{5.1}$$

In the example the image has only been rotated and scaled. When the viewpoint of the camera changes, the 3D rotation of the object as seen by the camera results in projective transformations. This has several side effects. For example, when an object at viewed from an angle, points closer by appear larger. These projective transformations are not captured using the (x, y), s and o parameters estimated in the Hough transform. As noted by Lowe[Low04], these are only a approximation of the full six degrees-of-freedom in which an object can be moved. This approximation results in errors when objects are viewed from an angle. As noted in the introduction, the accumulator can be seen as a histogram, where the resolution in the different dimensions of the accumulator array is equal to a histogram's bin size. To allow clustering to work while objects are rotated in all possible degrees-of-freedom, the bin size is increased. Larger bin sizes allow the clustering to work for larger viewpoint changes, but will also reduce the precision of the estimated parameters, therefore an optimal between precision and robustness is determined. In the angle dimensions bins span 30 degrees, in the scale dimension bins span 0.5 units and in the position dimension bins span 0.25 units.

Each keypoint votes for the closest two bins in each dimension, except in the position dimensions where the number of votes depends on the length of the vector to the object center. A larger vector results in more votes. Because the vector is scaled and rotated using the angle and scale, errors in the calculation of these values, combined with the error introduced by perspective transformations, will result in a bigger error in the estimated location for longer vectors. This is compensated by placing more votes for longer vectors.

Lowe suggests a fixed bin size in the position dimensions which depends on the maximum scale found among keypoints belonging to a certain object. This was not chosen since there are a number of problems with this approach. This requires a preprocessing step to find the maximum. This maximum might be due to a false match resulting in a wrong bin size. Secondly this would require a different bin size for each object, and thus a different accumulator array for each object. In the chosen approach the object id is added as a fifth dimension and all keypoints are stored in the same accumulator array. This eliminated the need to sort all keypoints and process them per object. The downside with the chosen approach is that the number of votes is higher. While Lowe's approach always places votes in the closest two bins, resulting in a fixed number of votes, the chosen approach uses a variable number of votes depending on vector size, where the total number of votes is equal or larger than the approach by Lowe.

Because in each of the five dimensions the possible number of bins is quite high, even with large bins, using a multidimensional array as an accumulator array is not feasible on embedded devices. But since the data in the accumulator array is very sparse the accumulator array is implemented using a HashMap, where the key is determined by the parameters and the value depends on the number of votes for these parameters. In this way only bins which actually contain data are stored.

After processing each keypoint, the maxima in the accumulator array denote the likely position, scale and orientation of objects in the camera image.

5.1.3 Filtering with the detected locations

As described above the Hough transform is used to find the probable locations, scale and orientations of objects in a camera image. These values are found by finding the maxima in the accumulator array for each object. For each keypoint the normalised values are compared to the found location, orientation and scale of the object the keypoint belongs to. If the keypoint did not vote for the found values the keypoint is discarded.

5.2 Homography

The set of good matches, that is the result of filtering as discussed above, describes a relation between the location of certain points in reference images and the corresponding location of these points in the camera image. Since points lie on a plane in both images, a homography (or collineation) can be computed which describes the relation between any point in the reference image and the corresponding location in the camera image. A homography describes a perspective transformation between points in two planes. It is defined by a 3x3 homography matrix. Given a point $p = (x, y)$ in the reference image we can find the corresponding point $p' = (x', y')$ in the camera image using this matrix $\mathbf{H}$. Given:

$$p = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, p' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}, \mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{5.2}$$

The point p' is then:

$$p' = \mathbf{H}p \tag{5.3}$$

The homography can be used to find the location for all points in the same plane. For example transforming the corner points of the reference image using the homography results in the corner points of the object in the camera image. This is used to draw the outline of the object as shown in figure 5.2. More importantly the homography is used to determine the location of the points for which we know the world location. This is used for camera pose estimation as described in chapter 6.

5.2.1 Homography calculation

OpenCV includes functionality to compute a homography from point correspondences. Details on the calculations of a homography can be found in the book *Multiple View Geometry in Computer Vision* [HZ04] but are outside the scope of this thesis.

The idea of the algorithm use a set of points p_i in the original image and corresponding points q_i in another image. Then an initial guess for homography $\mathbf{H}$ is used to project the points p_i from one plane to the other as shown in equation 5.3. Then the distance of the transformed points p_i' to the location they should be, the points q_i is calculated. The sum of the squared distances is called the reprojection error e_{reprj} as shown in equation 5.4.

$$e_{reprj} = \sum_i d(p', q)^2 \tag{5.4}$$

where function $d(x, y)$ returns the distance between point x and y.

Next the homography is iteratively changed such that the reprojection error is minimized. This is done using the Levenberg-Marquardt Algorithm (LMA) [Lev44], [Mar63].

Even after filtering with the Hough transform, some of the keypoint matches may be incorrect. When calculating the homography using all matches, where the homography is determined by a least squares scheme as described above, these incorrect matches will reduce the quality of the homography. This means that when incorrect matches are present, the minimisation of the reprojection error will result in a homography which projects all points such that no single point has a large error, but all points have a small error. A better solution would be if a subset of points could be found such that the reprojection error for that subset is very low while the size of the subset is as large as possible. RANSAC [FB81] is used to find this subset.

5.2.2 RANSAC

RANSAC (which is short for *RANdom SAmple Consensus*) is an iterative method which can be applied to fit a model to samples when outliers are present in the sample set. First a subset of the samples is used to calculate the parameters of the model. Then all samples are compared to the model to see which samples fit the model. If the model fits a large number of samples the model is accepted, otherwise the model is recalculated from all inliers. This is repeated until a good model is found, or until the maximum number of iterations is reached.

When calculating the homography matrix, the homography is the model which has to be calculated and the in-/outliers are determined by projecting the points from the reference image using the homography and checking if they are close to the position of the matching keypoint in the camera image.

While RANSAC can be easily used when using OpenCV's functionality to find the homography, OpenCV always returns a homography, even if the model is not good and the execution only stopped because the maximum number of iterations is reached without finding a good model. Therefore the model returned by OpenCV is checked by projecting all the points from the reference image and checking which points are within the reprojection threshold used to calculate the homography. The reprojection threshold is the maximum distance between the real location of keypoints in the camera image and the location of points generated by transforming the keypoints from the reference image using the homography. So all points used to actually calculate the homography, or inliers in RANSAC terminology, have the relation shown in equation 5.2.

$$|p' - \mathbf{H}p| \leq T_{reproject} \tag{5.5}$$

where p' is the actual location location of a point in the camera image and $\mathbf{H}p$ is the calculated position using the matching point p from the reference image and the homography $\mathbf{H}$ and $Treproject$ is the reprojection threshold.

When the number of outliers is large the model is less likely to be accurate. Therefore at least 50% of the matches need to be inliers or the model is discarded and the object is considered not to be in the camera image.

5.2.3 Transformed surface check

While object recognition allows for certain affine transformations between the camera image and the reference image, the shape of detected objects is similar to the shape in the reference image. This means that any shape transformed by the homography should only include small affine transformations and should still be similar to the initial shape. This is used to check if the homography is valid by transforming a square. The square is transformed using the homography and the angle of all corners of the transformed square are calculated. If corners are not within 30 degrees of a right angle, the homography is likely to be incorrect and the object match is discarded. The choice for 30 degrees is based on experiments which showed that failed homography calculation attempts often result in angles near 0 or 180 degrees and that the viewing range in which keypoint detection is possible is limited. Because the angle between the reference image and the camera is limited, the possible angle of the corner of the transformed square is also limited. Experiments showed this is within 30 degrees.

5.3 Multiple angles

SIFT is designed to be invariant to affine changes, but it fails when the viewing angle of the reference image is very different from the angle at which the object is viewed in the camera image.

The feature vector is calculated using an area around the keypoint in image space, so the actual surface area of the object of which the feature vector is calculated changes when looked at from an angle. Since the area changes the feature vector changes as well. Also when looking at an object at a large angle, the features that are closer by occupy more pixels compared to the features that are further away. So when two features have the same scale in the reference image, they may have a different scale when looked at from an angle. This also changes the calculated image center when performing the Hough transform.

So firstly the feature vectors change such that matching between keypoints fails, and if the keypoints match, the object recognition fails because they might no longer agree on the object scale or location.

The problem of decreasing keypoint matching performance is also visible in figure 5.9. This graph shows the number of matching keypoints when the angle between the reference image and the camera image increases. In figure 5.9 the number of matching keypoints assigned to a training image, in a small database of reference images, is shown when the angle between the training image and the viewing angle increases. It shows a trend in which the number of matches decreases when the viewing angle increases. Similar results are found for all tested images.

To enable correct object recognition over a wide range of viewing angles multiple reference images for a single object are used. The object to be detected is photographed from multiple angles and during

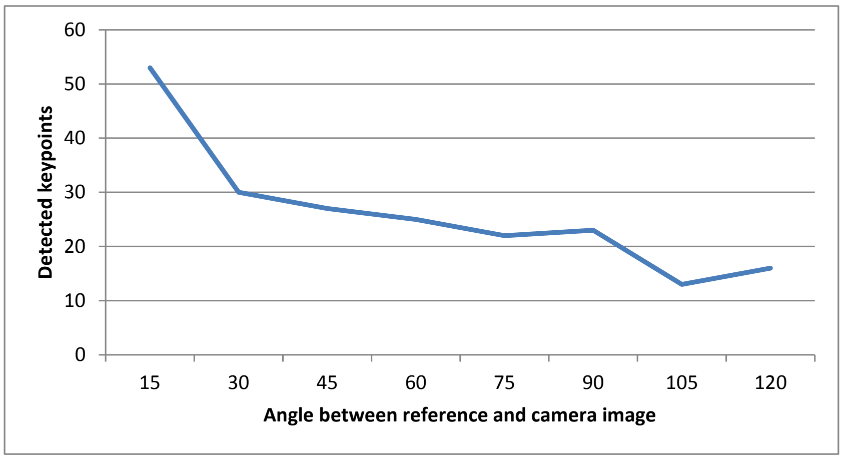


Figure 5.9: *Matching keypoints at increasing angles*

recognition the best angle for an object is selected in a preprocessing step such that for each object an image is available which is similar to the angle the object is currently viewed at.

5.3.1 Implementation

For the initial implementation photographs of an object taken from several angles where taken. Then these images where all added to the database. The problems that arise when adding them without further processing and the modifications required to correctly handle multiple images belonging to a single object are discussed below.

Background removal

Creating a reference image of a rectangular object viewed from an angle will include some background area as shown in figure 5.10. Since keypoints found in this area need to be excluded, a region of interest

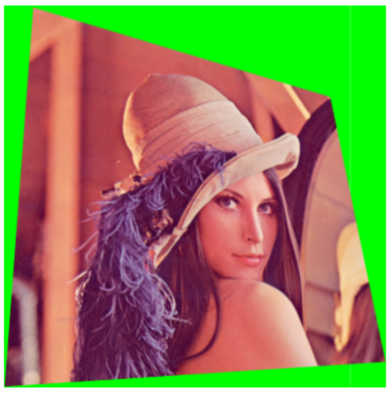


Figure 5.10: *Included background in green*

filter is implemented. This filter removes all keypoints that are not within the region of interest.

Regions of interest are defined using four corner points. The regions of interest can be defined by just these keypoints because objects have to be planar surfaces, which in practice are often rectangular.

The filter works by first detecting keypoint locations as normal. Then each keypoint that is not inside the region of interest is filtered before keypoint feature vectors are computed.

This method enables the use of regions of interest without changing the SIFT implementation. Only reference images have regions of interest, and these are processed in advance, so earlier bounds checking will not increase, but in fact lower, the performance at run-time. This makes this an efficient method to implement regions of interest.

Multiple reference image problems

When adding the images of an object as seen from multiple angles as individual objects, meaning there is no other modification to the framework and the images are added as if they were distinct objects, the recognition performance decreases. There are two reasons this is happening. Firstly the nearest neighbour filtering as discussed in section 4.3.1 results in higher thresholds for keypoints that are not distinct. This means that if there are many similar keypoints in the database they are less likely to be matched. When multiple images of an object are added to the database, the keypoints are likely similar, resulting in less keypoint matches. The second problem with this approach is that matches are divided over the different reference images of the object. This is also shown in figure 5.11. This graph shows how matches are divided over two reference images. The reference images are taken 30 degrees apart. The graph shows how many detected keypoints where matched with each reference image when the camera viewed the object from several angles at 15 degrees intervals. Figure 5.11

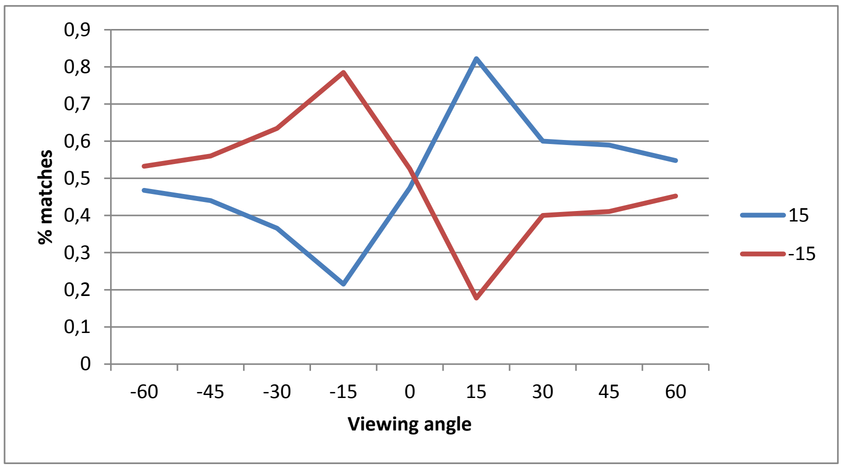


Figure 5.11: *2 training images taken 30° apart; Matching keypoints at various angles.*

shows that when the viewing angle is close to the angle of a reference image the most keypoints are matched to that image. When the viewing angle is between the two training images the keypoints are evenly divided. And for the other cases most keypoints are matched to the closest matching reference image, with decreasing performance when the angle increases further away from both training images.

This makes sense since, as shown in figure 5.9, keypoints matching performance decreases at increasing angles between training images and viewing angles. One more important thing to note when looking at the graph is that even if the angle is very close to a training image, some keypoints are still matched to the other training image. This makes the absolute number of keypoints matched to a reference image lower if multiple images are supplied, even if one exactly matches the current viewing angle.

Selection of best image

The problems of similar keypoints in the database which interfere with the nearest neighbour filtering and the spread of matching keypoints over the several reference images are solved by the introduction of a preprocessing step which selects the best reference image for each object. In this preprocessing step keypoints are matched to all images in the database. During this preprocessing step keypoints are matched as before but the nearest neighbour filtering is disabled. The matches in each image are summed for each object. Then for each object which has matches in the view, the training image that has the highest number of matches is selected. As also shown in figure 5.11, this is probably the image which was captured at an angle closest to the current viewing angle. After the best training image has been selected for each object in the view the keypoints are matched as before. Now nearest neighbour filtering can be enabled again since at most one training image for each object is considered.

Implementation using Masking

As described in the previous section a subset of the original set of reference images is created. In this subset only the best image for each object is selected. This subset is used in a second matching step. Performing matching on a subset of all keypoints in the database is supported in OpenCV using masks to denote which keypoints should be considered during matching and which keypoints should be ignored.

The main drawback of this technique is that the efficient approximate searching implemented in OpenCV as discussed in section 4.3, called FLANN, does not support masking. So after the initial selection brute force matching has to be used. Since the selection process also indicates which objects may be present in the view (objects with at least a couple of matches), the inefficient brute force matching is only performed on the small set of keypoints of objects that may be in the current view. The number of keypoints that are matched during the second matching step depends on the number of possible objects in the view and not the size of the database of reference images. As such the performance does not depend on the database size and the brute force matching has a minimal impact on performance even when using a large database.

The second best matches, required for nearest neighbour filtering, are also selected from this small set of images. To have the same result as when there was only one image for each object, the nearest neighbour should be selected from the entire database excluding the images of the object the keypoint belongs to, except for the one that was selected as the best. This would require masking and thus brute force searching. In this case almost the entire database has to be considered, which makes this approach infeasible for large databases.

How this affects the effectiveness of the nearest neighbour filtering has not been tested. Also the performance of large databases could not be thoroughly verified for this report due to the limited size of the training database during testing.

5.3.2 Results and Conclusions

Multiple images are required to correctly match objects from different viewing angles. The presented approach selects the image that is closest to the current image during a preprocessing step. If this preprocessing step is not used there are problems with the nearest neighbour filtering and keypoints matches being divided over the different images belonging to an object. Using the selection preprocessing step these drawbacks are avoided.

The nearest neighbours filtering can be used, although the nearest neighbours are selected from the limited set of keypoints.

The number of keypoints assigned to the best match is maximised as shown in figure 5.12. It shows the number of keypoints that are matched with two training images when looking at them from several angles at 15 degrees intervals. There are two training images, taken 30 degrees apart. The number of keypoints matched in the first two bars for each angle show the number of matches without preprocessing while the third and fourth bars show the number of keypoints when the best is selected (Note that either the third or fourth bar is always zero). Figure 5.12 shows that using the preprocessing

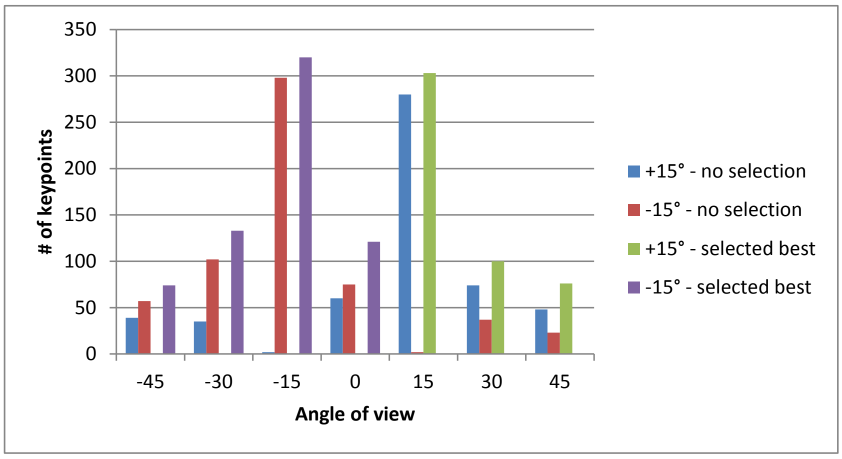


Figure 5.12: *# of Matching keypoints with and without selecting best image first.*

step to select the best image results in a larger number of keypoints matched with the image closest to the viewing angle (and zero matches to the other image). A higher number of keypoints for the best match enables more accurate object recognition.

Finally the solution is efficient despite having to use the brute-force matcher, since this brute-force matching is only required for the second time the keypoints are matched. During this second matching objects are considered of which at least multiple keypoints were found during the first matching. Therefore having to resort to brute force matching should not lead to decreased performance for larger training databases since the number of objects detected in the view depends on the number of objects in the view and not on the training database size.

5.4 Summary

Object recognition is performed for two purposes. First the set of matches generated by the keypoint detection and matching algorithms described in the previous chapter might include false matches. Because individual matches belong to objects and several matches should therefore be related, matches that are not related to other matches are removed. Secondly, using the calculated homography the relation between all points in the reference image and corresponding points in the camera image is described. This is used to perform pose estimation as discussed in chapter 6. Finally support for multiple training images for a single object is added such that objects can be recognised from varying viewing angles.

6 Pose Estimation

When rendering a virtual object, a virtual camera has to be created which defines from which position and orientation you want to view the virtual object, which are called the camera's extrinsic parameters, and some properties such as the focal length of the virtual camera, which are called the camera's intrinsic parameters. To draw virtual objects in a real view, this virtual camera needs to match the both the intrinsic and extrinsic properties of the camera which captured the view of the real world.

The camera intrinsic properties are described by the focal length, principal point, skew, radial distortion and tangential distortion. These intrinsic parameters depend on the properties of the hardware and settings of a camera. These intrinsic properties and how they can be determined is described in appendix A. The location and orientation are called the camera extrinsic parameters or pose. They change when the camera moves relative to the objects it is capturing.

Since the camera extrinsic parameters need to be known for correct rendering of virtual objects, these parameters are estimated each frame. These extrinsic parameters, or pose, are estimated using the 3D world location of a set of 2D points in the camera image. As shown in the previous chapter the location of objects in the camera image is calculated. For each object some 3D positions are given for certain points in the reference image. Using the homography, as calculated in the section 5.2, the positions of the 2D points that have a corresponding 3D position can be found in the camera image. This chapter discusses how to use this information to calculate the pose.

To intuitively understand how images can be used to estimate the camera pose and why camera intrinsic parameters are required, consider the following:
Imagine you get a photograph of a scene in which you recognise some objects. Then using the photograph and your knowledge of the real location of some points in the image you will be able to deduce the location and orientation of the camera when the photograph was taken. To make an accurate guess you will have to know something about the camera as well, such as the field of view and zoom level. This is required to make a distinction between the camera being close to an object or that the camera zoomed in on an object in order, which is required to accurately estimate the camera position. You could either try to guess those properties based on what you see in the photograph or you could get this information from another source such as the hardware specifications of the camera.

The above example describes the problem of pose estimation. It can be defined as:

Definition Pose estimation is the estimation of the location and orientation of the camera relative to one or more objects based on the 2D location of these objects in an image made by the camera.

Note that some research may state pose estimation in terms of translations an object has made in

front of an camera. In other words the pose of objects compared to a fixed camera is estimated. In effect these two ways to look at pose estimation are the same and solutions to camera pose estimation can be easily changed to object pose estimation and vice versa.
This is more obvious when looking at the relation between the components involved:

$$sm' = A\left[R|t\right]M' \tag{6.1}$$

Where s is the scale, m' the point in the image, A the camera intrinsics parameters matrix, $\left[R|t\right]$ the rotation/translation matrix and M', the point in 3D.
Pose estimation algorithms estimate $\left[R|t\right]$, which describes how the camera, or the objects in the reverse direction, are moved compared to a world coordinate system. Together with A it describes the relation between 3D points in the world, and 2D points in the image. Some pose estimation algorithms can also estimate A while other algorithms require the camera intrinsics to be given. The algorithms discussed in this chapter do not need to calculate A because this is done as a prepocessing step. This is discussed in appendix A.

6.1 Preliminaries

The problem of pose estimation from a 3D-2D point correspondence set, also called *Perspective-n-Point problem* (or PnP-problem), is solved in various ways. All methods need at least 4 points because this is the minimum number of points needed to have an unique solution. If there are less points there are multiple points from which a camera can view these points such that they appear at the same 2D location in the 2D plane.

The methods described below are either able to work with coplanar points, non coplanar points or both. When points are coplanar this means that there is a plane such that all points are part of that plane. When points are non coplanar this means there is not a single plane such that all points lie on that plane.

Hardy [Har11] describes how to estimate the pose using direct linear transformation which uses singular value decomposition to find a solution. After a initial solution is found he uses the Levenberg-Marquardt Algorithm (LMA) [Lev44], [Mar63] to optimize the solution. The main drawback of the method as described is that all 3D points have to be coplanar but he suggests the method can be extended to eliminate this requirement.

OpenCV includes several methods which were investigated on their working and applicability.

Firstly OpenCV includes a method which is based on the Levenberg-Marquardt Algorithm which does not require the 3D points the be coplanar.

A more recent method is POSIT [DD95], which is an iterative version of the POS (Pose from Orthography and Scaling) algorithm. The POS algorithm approximates the perspective projection with a scaled orthographic projection and finds the rotation matrix and the translation vector of the object by solving a linear system. It it faster to compute than previous methods and does not require an initial guess. It works when points are not coplanar.

The method by Gao et al. [GHTC03] solves the problem for exactly four object and image points.

The Efficient Perspective-n-Point Camera Pose Estimation or EPnP method by Lepetit et al. [LMNF09] is a non-iterative solution to the PnP problem which can be calculated in $O(n)$ time for both coplanar and non-coplanar points.

6.2 Implementation

The 2D-3D set is constructed from known points in detected objects in the image. The objects are planar surfaces themselves, so the keypoints from a single object are coplanar. But since multiple objects can be found in an image not all 3D points have to be coplanar.

In the framework the zooming level and camera resolution are fixed, and therefore all camera intrinsics are fixed for a certain device. Therefore the camera intrinsics can be precomputed, as described in appendix A, and considered a given during pose estimation. Since the parameters are fixed the pose estimation algorithm used in the framework does not have to estimate these.

As noted the 2D-3D set may include non coplanar 3D points and as such the approach by Hardy, without extending it to handle non-coplanar keypoints, is not applicable. The efficient POSIT approach is also not applicable since keypoints are not always coplanar.

The P3P solution by Goa et al. is less suited for the 2D-3D set calculated since the number of points in the set is not fixed and may be much larger than 4. A subset of 4 points could be used, but this would reduce the quality of the solution. Using multiple subsets and combining the results of several P3P estimates is possible but would require a new method to combine these results.

The iterative Levenberg-Marquardt Algorithm (LMA) as implemented by OpenCV and the EPnP method (also implemented in OpenCV) are both applicable to the problem and are both tested. The LMA method performed more robustly than the EPnP method. While the research indicates the results should be similar, with the EPnP method having the advantage that it is faster to compute, the results of the OpenCV implementation of EPnP pose estimation are less consistent. There is larger variation in the estimated pose when the camera is almost stationary. Furthermore the estimated orientation is sometimes 90 degrees off when using EPnP.

The estimated pose is used for rendering virtual objects. So small errors in the estimated pose will result in the virtual objects being rotated or moved slightly. If the pose estimation algorithm has a consistent error, the virtual object do not precisely align with the real world, but will appear at a fixed location relative to the real world. If on the other hand the error varies from frame to frame, the virtual objects move relative to the real world. This breaks the illusion that the objects are real. Therefore a consistent error is preferred.

Because the LMA method has a more consistent error and the EPnP method sometimes returns the wrong orientation, the LMA method is used for pose estimation in the framework.

6.2.1 Rotation vector and matrices

The OpenCV pose estimation returns the translation vector t as described in equation 6.1 and a rotation vector r. A rotation vector is a convenient and most compact representation of a rotation matrix since any rotation matrix has just 3 degrees of freedom. To turn this rotation vector to a rotation matrix R as in equation 6.1 Rodrigues' rotation formula is used. This conversion is implemented in OpenCV. To calculate the rotation vector the following is used. The vector t encodes both the angle and the direction. The rotation vector length denotes the angle θ. After θ is determined the vector t is normalised to $u =< u_x, u_y, u_z >$. Then letting

$$W = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix} \tag{6.2}$$

The rotation matrix R is defined by

$$R = I + \sin\theta W + 2\sin^2\frac{\theta}{2}W^2 \tag{6.3}$$

Where I is the identity matrix.

6.3 Summary

The camera pose can be determined using a set of points in the camera image for which the 3D location is known. Pose estimation from 3D-2D point correspondence set, also called the *Perspective-n-Point problem*, is solved using a iterative Levenberg-Marquardt Algorithm which is able to handle both coplanar and non-coplanar points, with a minimum of 4 points.

7 Rendering

Rendering is the process of generating an image using a computer program. For AR this means drawing both the camera image and virtual models such that the virtual models appear as if they are real objects in the physical world. For this the movement and rotation of the virtual objects should match the movement and rotation of the real world view captured by the camera. This is achieved by using a virtual camera that matches the properties of the real camera.

Virtual three dimensional objects can be drawn on Android devices using the OpenGL ES graphics library. OpenGL ES is a version of the OpenGL graphics library for Embedded Systems such as Android devices. Rendering 3D geometry using OpenGL consists of several steps including the transformation of 3D points, or vertices, to 2D locations on a screen, and the coloring of pixels that appear on the screen. While OpenGL performs many more steps these two are of particular interest for the AR framework since they control where the 3D models appear in the view and their appearance. Where they appear, this includes viewing angle, perspective, position and scale, is important because these properties have to match the physical world such that the virtual objects seem real, tangible objects. Next the appearance is important to further create the illusion that the virtual objects are real. This includes creating lighting which matches the real lighting conditions.

Creating a virtual camera in OpenGL which matches the real camera is discussed in the next section. Methods to further make the virtual objects appear part of the real environment are discussed in section 7.2.

7.1 Virtual camera in OpenGL

A virtual camera in OpenGL is defined by certain matrices. How these matrices relate to a virtual camera is explained in the next section. The following sections discuss how to calculate the different matrices involved.

7.1.1 Rendering with OpenGL

Virtual 3D models are defined by polygons, such as triangles. These polygons are defined by vertices. For example a triangle is defined by 3 vertices. Transforming a polygon from 3D coordinates to the 2D screen is mostly handled by the OpenGL library, only a matrix which transforms these vertices from 3D coordinates to clip space[1] has to be supplied. Using the supplied matrix the OpenGL libary calculates which pixels on the screen are occupied by a polygon, and if more than one polygon occupies a certain pixel, OpenGL determines which polygon is closest to a viewer and should be used to fill a pixel on the screen.

This matrix that an OpenGL user needs[2] to supply, called the model-view-projection-matrix, directly transforms vertices to clip space. As the name suggest it is composed of three parts: a model matrix, a view matrix and a projection matrix.

Each matrix is a 4x4 transformation matrix. A transformation matrix allows arbitrary linear transformations to be represented in a consistent format. The transformation matrix can represent both affine transformation, such as translation and rotation, as well as projective transformations. The exact workings of transformation matrices are outside the scope of this thesis. More on transformation matrices and especially how they are used in computer graphics and OpenGL can be found in the OpenGL Red Book [WNDS99].

Model-View-Projection-Matrix

The model-view-projection-matrix is created by multiplying the model, view and projection matrices as shown in equation 7.1.

$$M_{mvp} = M_p \cdot M_v \cdot M_m \tag{7.1}$$

where M_{mvp} is the combined model-view-projection matrix, M_p the projection matrix, M_v the view matrix and M_m the model matrix. Transforming a point by multiplying it first with M_m, then M_v and finally M_p yields the same result as multiplying the point with M_{mvp} making it faster when a lot of vertices have to be transformed.

The model matrix M_m transforms vertices from object space coordinates to world space, the view matrix M_v transforms world space coordinates to camera space and finally the projection matrix M_p transforms camera space coordinates to clip space. Each space denotes that the coordinates are relative to an object. For example object space means the coordinates are defined relative to the object coordinate system. If the object is to be placed at a certain position, rotation and scale in the world, these transformations can be encoded in the model matrix. When the vertices of the model are multiplied by the model matrix the resulting coordinates are relative to the origin and axis of the world coordinate system. More on different spaces can be found in the OpenGL Red Book [WNDS99].

7.1.2 Model-View-Projection-Matrix and the virtual camera

As described in chapter 6, the virtual camera used to render virtual objects in an AR application should match the real camera. To create a virtual OpenGL camera that matches the real camera, the model-view-projection matrix needs to be calculated from the camera intrinsic and extrinsic parameters. The extrinsic parameters, or pose, describes the coordinate system transformations from 3D world coordinates to 3D camera coordinates. In OpenGL this transformation is performed using the view matrix. The extrinsic parameters define the projection of the world on the image plane. In OpenGL this projection is defined using the projection matrix (and setting the viewport size).

[1]Clip space is almost like screen space. The transformation from clip space to screen space uses homogeneous coordinates, screen resolution and near/far planes. Since this is handled by OpenGL and is not relevant to the AR framework this is not discussed further.

[2]When using OpenGL ES 2.0 there is no specific requirement on how to implement transformations, yet this standard way of performing transformations has been implemented.

Model matrix

The model matrix is used to transform vertices from a local object coordinate system to world coordinates. Using this matrix enables models to be defined without specifying their location in the world but instead use a local coordinate system. Then using the model matrix, the model can be placed, scaled and rotated. If the exact location of the model in the world is known, the coordinates can also be defined in world coordinates directly in which case the model matrix is not needed.

View matrix

The view matrix is used to transform vertices from world coordinated to camera coordinates. The view matrix describes how the camera moved relative to the world or, if considering the camera to be stationary, how the world moved in front of the camera.

This matrix is calculated using the result from the pose estimate from chapter 6. The pose estimation resulted in a rotation vector r and a translation vector t. OpenCV uses another coordinate system than OpenGL, namely in OpenGL the positive y direction is up, and the positive z is towards the viewer while OpenCV uses the reverse directions. These axis need to be inverted on the rotation vector t and the rotation matrix R. This can be done using a matrix multiplication for t:

$$t' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} t = \begin{bmatrix} t_1 \\ -t_2 \\ -t_3 \end{bmatrix} \tag{7.2}$$

And similarly for R, including a transpose to change from OpenCV's row-major ordering to OpenGL which uses column-major order matrices:

$$R' = R^\intercal \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} = \begin{bmatrix} R_{11} & -R_{21} & -R_{31} \\ R_{12} & -R_{22} & -R_{32} \\ R_{13} & -R_{23} & -R_{33} \end{bmatrix} \tag{7.3}$$

OpenGL requires 4x4 matrices using a homogenous coordinate w. Using $w = 1$ this results in the appending of a row with 3 zero's and a one. When defining the matrix to invert the angles as M_i as

$$M_i = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{7.4}$$

The calculation of M_v from t and R is

$$M_v = \left(\begin{array}{c|c} R' & t' \\ \hline \mathbf{0} & 1 \end{array} \right) = \left(\begin{array}{c|c} \mathbf{R}^\intercal M_i & M_i \mathbf{t} \\ \hline \mathbf{0} & 1 \end{array} \right) = \begin{bmatrix} R_{11} & -R_{21} & -R_{31} & t_1 \\ R_{12} & -R_{22} & -R_{32} & -t_2 \\ R_{13} & -R_{23} & -R_{33} & -t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.5}$$

Projection matrix

The projection matrix is used to transform vertices from camera coordinates to clip coordinates. Clip coordinates are transformed to screen coordinates by OpenGL. The projection matrix describes how the 3D world is projected on a 2D surface.

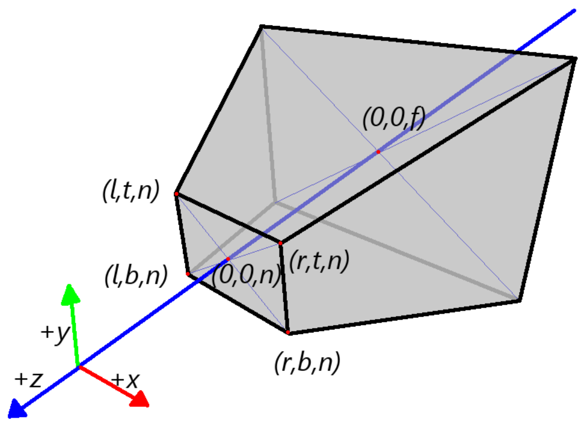


Figure 7.1: *Clipping planes*

The projection matrix can be defined using clip planes as shown in figure 7.1. The clip planes are calculated using the camera intrinsics described in appendix A. These clip planes themselves are defined by six variables, l, r, b, t, n and f. The values stand for left, right, top, bottom, near and far where near and far are the distance to the nearest and furthest planes as seen from the camera. These are the planes facing the camera. The other planes are defined using the distance from the center at the near plane. So (l, t, n) stands for the top left position at the near plane. Using these six values all planes are defined.

The near and far plane can be chosen freely, while the other four variables depend on the chosen value for the near plane and the field of view.
The field of view is defined by the vertical angle of view and the horizontal angle of view, which can be calculated from the focal lengths of the camera intrinsics as shown in equation 7.6 and 7.7, McCollough [McC93].

$$\alpha_x = 2\tan^{-1}(\frac{w}{2f_x}) \tag{7.6}$$

$$\alpha_y = 2\tan^{-1}(\frac{h}{2f_y}) \tag{7.7}$$

The α is the angle of view in either the horizontal or vertical direction, w is the width and h the height, depending in which dimension the angle needs to be calculated, and f is the focal length in the chosen dimension.
Using the field of view the values l, r, b and t can be calculated as shown in equations 7.8 to 7.11.

$$r = n * \tan(\alpha_y \frac{\pi}{360}) \tag{7.8}$$

$$l = -r \tag{7.9}$$

$$t = n * \tan(\alpha_x \frac{\pi}{360}) \tag{7.10}$$

$$b = -t \tag{7.11}$$

If the focal lengths are equal, so $f_y = f_x$, t and b can also calculated using the aspect ratio, which is the ratio between the width w and the height h, as shown in equations 7.12 and 7.13.

$$t = l * (w/h) \tag{7.12}$$

$$b = r * (w/h) \tag{7.13}$$

Android includes a Matrix library which is able to create a projection matrix using the six parameters defining the clip planes or it can be calculated manually as defined in equation 7.14.

$$M_p = \begin{bmatrix} \frac{2n}{l-r} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{7.14}$$

More on this and these equations can be found in the OpenGL Red Book [WNDS99].

Principal Point

The camera intrinsics also includes an actual image center or principal point. This principal point is not equal to the camera image center when the sensor in the camera is not precisely aligned with the lens.

In an early version of the framework these parameters where used when estimating the pose of the camera. Then an a-symmetrical frustum was created which matched the frustum of the camera. This resulted objects being rendered close to the required location, but not exactly. The result was a few pixels off the required location. As discussed in the paper by Bougnoux [Bou98] the principal point is hard to determine accurately. Camera calibration based on images, such as described in appendix A, results in large errors in the estimated principal point. The errors in the initial implementation are probably due to errors in this determination of the principal point.

As further discussed in the paper by Bougnoux [Bou98] the influence of the principal point on rendering is minimal. The error of interpreting the offset of the principal point as a translation of the camera is negligible. This means that using the image center as the principal point will result in a translation of the estimated camera position when performing pose estimation. Using this incorrect pose for rendering where the the image center is again used as the principal point results in very small errors in the resulting image. As shown in an example in [Bou98], when the principal point is up to 50 pixels from the image center and the focal length is 700, the error in the final image is less than 0.0144 pixels. These values are comparable to the values determined for the test device.

Using the image center as principal point is implemented and this method outperformed the initial approach in terms of accuracy. The new approach did not show a noticeable error during testing. As such the image center is used as the principal point during pose estimation, and the viewing frustum is created as described in section 7.1.2.

7.2 Appearance

Using the above method, the virtual world is aligned with the real world. This means that the virtual objects are drawn with the correct scaling, position rotation and perspective. And when de camera moves the virtual objects move accordingly. For some applications this is enough. For example in the ARMAR example shown in figure 2.3 in chapter 2. In this example AR is used to assist in a technical task. Here the virtual objects are always visible and virtual objects are clearly distinguishable from real objects. For other applications however it might be preferred if the virtual objects appear to be real objects. If for example AR is used to visualise virtual objects in an real environment, such as the Tobi store example in chapter 2, a virtual object should appear as real as possible.

Using the methods described so far, the virtual models are drawn on top of the view of the real world as an overlay. To achieve more realism two problems with this approach have to be addressed.

First occluders have to be taken into account. When occluders are not taken into account, the virtual objects are always drawn in front of all objects in the camera view. To make virtual objects appear part of the environment, parts of the environment must sometimes be drawn in front of the virtual objects as shown in figure 7.2.

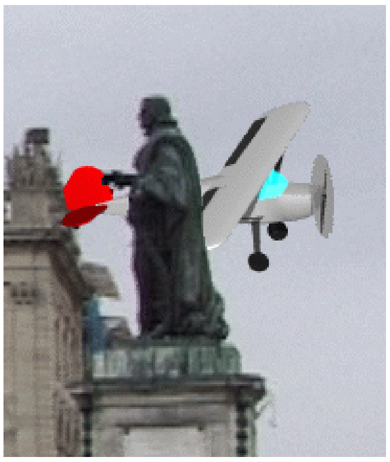


Figure 7.2: *Virtual plane occluded by a statue. Image from [LoB00]*

Another problem with rendering is creating realistic lighting. In order to make virtual objects appear real the lighting of virtual objects should match the lighting of the surrounding objects in the camera view. Ideas to solve these issues are addressed in the following sections, however these are not implemented in the framework. These are not implemented because, as explained in the following sections, there are multiple ways to further merge virtual objects with a scene, and the optimal solution is highly application specific. Furthermore the implementation of such techniques is outside the scope of this thesis.

7.2.1 Occluders

When occluders are to be taken into account there are two main approaches that can be taken. First the environment can be modeled. This means the real environment is modeled in the virtual world. If there is a complete model of the real world this model can be easily used to find parts of the virtual object that should not be drawn in the view. The approach could be to render the complete environment using OpenGL writing only to the depth buffer. Then when the virtual objects are drawn the depth buffer is used to find which pixels should be drawn and which pixels are occluded by a real object. Obviously this approach requires a precise model of the real environment which is labour intensive to create.

A second approach is to acquire the information automatically. One approach could be to use a camera which captures depth information such as the Microsoft Kinect camera. Such an approach is shown by Clark et al. [CP11]. Since no Android devices with such a camera exist this is not applicable to the framework described in this thesis. Another approach is to perform object recognition, and to assign some objects a position. This works well when there are only few possible occluders. This approach is shown by Lepetit and Berger [LoB00]. In this approach, objects which should be in front of virtual objects are manually selected in images. Then when creating the AR, the occluding objects are detected in the camera image. The contour of the occluding objects are determined and using this contour, billboard images are created. Then the final image is created by drawing the virtual objects in front of the camera image, and then the billboards on top of both the camera and the virtual objects.

The advantage of this approach is that it performs the occlusion semi-automatically. The drawbacks are that some manual work is required for each scene and that each occluder is a billboard without depth, which is either completely in front or behind a virtual object.

7.2.2 Lighting and shadows

In computer graphics the level of realism largely depends on the realism of the lighting of virtual objects. Lighting includes the illumination of objects and the lack of it. While shadows are just the lack of lighting it is often considered as a special case when rendering using OpenGL. This is because OpenGL usually works with local illumination models because these are fast to compute. But these local illumination models do not include indirect lighting. Therefore indirect lighting is often estimated using constants and shadows are generated using a different process. Therefore these are handled separately here as well.

Lighting

Lighting a virtual scene is usually determined using light source information, such as color, intensity, location and direction, and material properties such as color and reflectiveness. The problem when showing virtual objects in a real world is estimating the light source information. There are two approaches to create more realism. The first approach is to model the information while the second approach is to acquire the required information from sensors.

The modeling approach can be used when the environment in which the virtual model is shown is known and static. For example if a virtual model is to be shown indoors in a room with fixed lighting conditions, these lighting conditions can be modeled in the application.

The second approach is to acquire the lighting condition information. Computer vision could be used to estimate the lighting condition. For example the approach by Aittala [Ait10] estimates the lighting conditions automatically. The approach taken uses a ping-pong ball. The ping-pong ball is used to find the lighting from all directions in a hemisphere by analysing how the half of the ping-pong ball facing the camera is illuminated. A result of this work is shown in figures 7.3 and 7.4.

Shadows

Two aspects have to be considered when using shadows in an AR application. First the real environment might cast shadows on the virtual object, and secondly the virtual object might cast shadows on the real environment.

To find objects in the real environment that cast shadows requires knowledge about both the geometry of the environment and the light sources. This could be modeled for a static environment.

To make the virtual object cast shadows on a real scene also requires knowledge of the geometry and light sources of the real scene. The approach by [JNA+05] uses approximate knowledge of the geometry and lighting of a real scene to detect shadows in a scene to create matching shadow casting, in direction and color, for virtual objects.

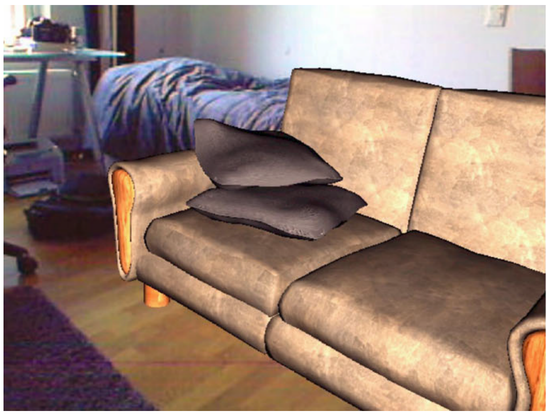


Figure 7.3: *Using basic OpenGL rendering, from [Ait10]*

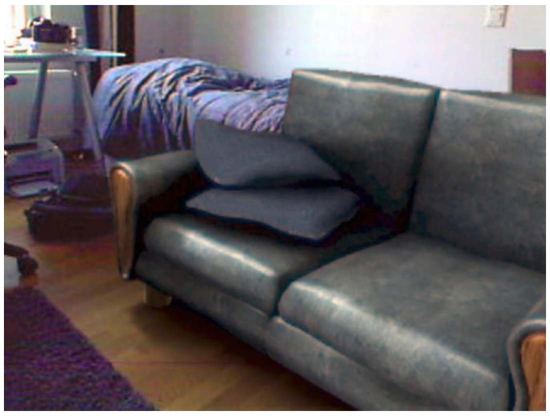


Figure 7.4: *Using estimated lighting conditions, from [Ait10]*

7.2.3 Conclusions

For both problems, occlusions and lighting, several approaches exist. These can be categorised by methods that acquire the information needed to create the effect and methods that model the information. The most appropriate approach depends on the level of realism required, the context of the application and on the amount of effort that one is willing to invest. Since this thesis describes a general AR framework, which is not developed with a particular application in mind, and because implementing these features is outside the scope of this thesis none of the appearance methods have been implemented. These are ideas that could be uses when the framework is used for a particular application.

7.3 Implementation

The implementation of the rendering part of the framework uses OpenGL ES 2.0 to draw the virtual models. The model-view-projection matrix as described is used to transform all vertices. To simplify the tasks of 3D file loading and resource management the open source Rajawali framework is used [Ipp12].

7.4 Summary

OpenGL is used to draw 3D models. Using a virtual camera in OpenGL which matches the real camera the virtual objects are correctly rotated, scaled, translated and projected. For advanced matching with the real world, such as detecting occluding objects, matching lighting and shadow casting, application specific solutions are presented.

8 Optimization

The implementation of the framework using the techniques described in the previous chapters resulted in a functional framework, but the framerate was low. For an immersive AR experience the framerate should be maximised. Profiling of the framework showed that most of the time was spent on feature detection and feature vector computation. To increase the framerate, two strategies are implemented. First, SIFT is implemented on the GPU. The second strategy is to track keypoints between frames. This way keypoint detection and feature vector computation does not have to be performed every frame.

Finally GPU accelerated camera correction is also discussed. Camera correction "revises" an image for imperfections in camera hardware. These imperfections are described in appendix A.

This chapter first discusses the use of the GPU for general purpose calculations and the implementation of SIFT and camera correction using the GPU. Section 8.2 discusses the theory and implementation of feature tracking.

8.1 GPU Acceleration

While graphics hardware has been primarily developed for graphics rendering, programmable GPUs can also perform general calculations, also called *General-Purpose Graphics Processing Unit (GPGPU) programming* [OLG+07], [OHL+08], [FH08]. As GPUs have highly parallel single-instruction, multiple-data (SIMD) architectures, these GPGPU programs are specifically well suited to perform parallel instructions on both tasks and data. This is because the GPU consists of many cores, much more than are available on current CPUs. While these cores are slower and simpler than the cores of a CPU, the large number of cores results in a much higher combined calculation performance. This means when looking at, for example, the number of floating points multiplications, a GPU can perform many more multiplications per second than a CPU, provided that all cores are used. This means a GPU is particularly suited to perform calculations on large sets of data where the result of one calculation is not required for another calculation. An example at which a GPU excels is performing a transformation of a large list of points. This is because they can be processed in parallel (so the task is parallelizable) and since the same transformation is used for all points, the SIMD architecture can be used to perform multiple transformations in a single pass. A problem which is not very well suited to be solved using GPGPU programming is when a single point has to be transformed by a long list of transformations. This is because the output of a transformation is used as input for the next step, therefore the operations can not be run in parallel. In this case only a single core is utilised, and because the GPU cores are slower than CPU cores, the GPU implementation would be slower than a CPU implementation.

The reason a GPU could be used to perform calculations is that when the power of a GPU is utilized, a GPU implementation can be much faster than a CPU implementation. Furthermore a system is able to perform both GPU and CPU programs simultaneously. This means that if part of an algorithm is implemented on a GPU, this will reduce the workload for the CPU, which can increase the overall performance.

Originally GPUs where designed to perform specific and fixed rendering tasks. When parts of this fixed function pipeline became programmable trough shaders, the GPU started to be used for other tasks than just rendering. There are two types of shaders, called vertex shaders and fragment shaders. When rendering calls are sent to the GPU to render geometry to an image buffer, the vertex program is used to process each vertex of the geometry. Then for all geometry which appears in the buffer, the fragment program is called for each pixel. The purpose of the fragment program is to determine the color of a pixel.

When the GPU is used for traditional rendering of three-dimensional geometry, the vertex shader will transform geometry from object coordinates to clip coordinates using a model-view-projection matrix as discussed in chapter 7. Then the pixel shader will compute the color of the pixels in the output buffer. This can be determined using for example a lookup in a texture map and some lighting calculations.

To use a GPU to perform general purpose calculations these shaders can be exploited. A problem which can be solved by a GPU program is determining the maximum of a list of pairs of numbers between 1 and 100. The program should thus create a list of numbers containing the maximum of each pair. This could be implemented on the GPU by creating a texture containing the pairs, where in each pixel the red channel contains the first number of the pair and the green channel the second number. Then a single quad is rendered. The vertex shader should transform the quad such that the quad is a single pixel high and has a width equal to the number of pairs. Then the pixel shader, which is automatically called for each pixel, should lookup the pixel in the texture corresponding to the pair it should compare, and output the maximum of the red and green channel. After rendering the buffer can be read back to main memory, which results in a list containing the maximum of all pairs.

While this is a very basic example it shows the basic flow of GPGPU programs which is:

1. Create textures containing the data that needs to be processed, and pass optional variables
2. Render geometry such that vertex/fragment shaders which perform the calculations are called
3. Either read back the results, or use the output as input for successive rendering steps

In the example the data was read back to main memory. In more complicated algorithm, the output of a rendering step can also be used as input for successive rendering steps before the results are read back to main memory. This is achieved by using the framebuffer which has been rendered to, as a texture in the next step. When using multiple steps it is important to use this method instead of transferring the data back and forth between the main memory and the GPU memory because this transfer of data is slow.

As described above the GPU can be used for general purpose calculations through the use of shaders, however trying to use shaders to perform general purpose calculations is beyond the purpose for which the GPUs where originally developed. This is also the reason that performing calculations using this method is not very intuitive. To make GPGPU programming on GPUs easier frameworks such as CUDA [NVI07] and OpenCL [Khr08] have been developed. Unfortunately there are no implementations of such frameworks available for Android. However, shaders are available on all Android devices supporting OpenGL ES 2.0.

While Android devices supporting OpenGL ES 2.0 support shaders, the functionality provided in these shaders is a bit limited compared to the support of current desktop GPUs. The capabilities of mobile GPUs are similar to the capabilities of desktop GPUs of several years ago. Most notably the current mobile GPUs only have 8-bit per pixel RGBA buffers. Also only single render targets are available. (Multiple Render Targets (MRT) enables fragment shaders to output multiple colors). This means a fragment shader on Android can only output 32 bits. Furthermore the fragment shaders use at most 16-bit floating point numbers, also called half floats, internally. Finally the Samsung Galaxy SII used for development supports non-power-of-two (NPOT) textures , but experiments showed that the GPU probably scaled these textures to power of two textures internally. This resulted in some interpolation between pixel values when using NPOT textures. While this might not be a problem for games, this is a problem when performing GPGPU programming.

8.1.1 SIFT GPU

SIFT has been implemented on a desktop GPU [Wu07] as well as mobile devices [Kay10], [Bre11]. The desktop GPU implementation is very complete, but relies heavily on functionality not available on mobile devices, such as multiple render targets and floating point buffers. Therefore this method is not usable for android devices. The implementation described in [Kay10] is very incomplete and does not include keypoint descriptor calculation. Also no source is provided. The implementation in [Bre11] is complete in the sense that it contains both keypoint detection and descriptor calculations, but it is very different from the OpenCV implementation. Some steps are completely missing and other steps are implemented differently, reducing the complexity of the algorithm, but probably also reducing the quality of the implementation. The implementation is for IOS devices which is quite different from Android devices.

Because none of the methods can be used for the AR framework described in this thesis, a new implementation of SIFT on a GPU is developed. This implementation is equal to the OpenCV implementation of SIFT where possible, but some changes had to be made, for example due to the limited precision and the limited number of bits a shader can output.

System Overview

As described in section 4.2.1, the SIFT algorithm works with grayscale images, so the input is first converted to grayscale.

To maximise the use of the SIMD architecture and to maximise the use of the texture memory, four levels of keypoints are created in each octave. Because the image is in grayscale, a single RGBA texture can store four levels, each channel containing a level. This requires six levels in each octave in the DoG pyramid and seven levels in the Gaussian pyramid. The seven levels of the Gaussian pyramid are stored in two textures. The first texture contains the levels 1 to 4 and the second level contains levels 4 to 7. This means level 4 is duplicated. This is because this enables the calculation of each DOG level using only one texture as input. This duplication of this level, when looking at both the calculation time and storage required, is "free". Calculating three levels is just a fast as calculating four levels because of the SIMD instructions. Also there is no extra storage required because textures have four channels.

The DoG pyramid is created by subtracting the GBA channels of each of the two textures from the RGB channels for each octave. This results in two textures for each octave, each texture containing three levels.

The OpenCV implementation uses 16-bit fixed point buffers for both the Gaussian pyramid and the DoG pyramid. This was initially implemented for the GPU approach as well. This was done by converting the 16-bit floating point numbers used in the shaders to fixed point notation, and then storing these values each using 2 channels, so the values where stored using 16 bits of precision. So a single fixed point number would use either the RG or BA channels. There where two problems with this approach. First this only allows the GPU to process two levels in a rendering pass instead of four. This doubles the time required to create the scale space pyramids. This, together with the additional cost created by the conversion of floating points to fixed point notation using two channels, made this approach too slow. So the images are stored using 8 bits per value instead. However storing the scale space pyramids in 8-bit precision results in noise, which creates many (bad) extrema in the DoG. This has been solved similar to [Bre11], using a blurring step with a Gaussian filter on the DoG levels. This removes the high frequency noise. The extrema found after performing the blurring step on the DoG are very similar to those found by the OpenCV implementation, which means the set of extrema has a large overlap.

After creating the smoothed DoG pyramid the extrema are detected by inspecting all neighbours in a 3×3 region around each point. This is done for four levels in each octave. Because the results are binary, a point is either an extremum or not, multiple octaves can be written to a single buffer. Using different values for each octave, the octave in which the keypoint was found can later be determined. Storing all octaves in a single texture is important because the buffer containing the extrema is read back to the main memory which is slow. Reading back this data is required because compacting a list of points is difficult on the GPU. Therefore the buffer is transferred to main memory and the locations of the extrema are extracted from the image.

Next sub-pixel location calculation, low contrast suppression and edge suppression are performed. These require some processing for each keypoint. The sub-pixel location calculation, low contrast suppression and edge suppression are performed in a fragment shader using an implementation similar to the OpenCV implementation. To run the fragment shader once for each keypoint, a single pixel for each keypoint is drawn. To prevent errors which occur when using NPOT buffers, these pixels are drawn in a buffer with dimensions which are powers of two, using the smallest power-of-two such that the buffer is large enough to contain all n points. So the width w and height h of the buffer is calculated as shown in equation 8.1:

$$w = h = cpot(\sqrt{n}) \tag{8.1}$$

where $cpot(x)$ is a function which returns the smallest power of two greater or equal to x. After rendering the buffer is read back to main memory. When calculating the sub-pixel location, it can be that a point is actually closer to another pixel. These points are processed in iterations of the above described method similar to the iterations in the OpenCV implementation. The result of the sub-pixel location calculation, low contrast suppression and edge suppression step, is a subset of the original keypoints. Furthermore the keypoint positions are improved to sub-pixel accuracy.

For successive steps the gradients in the x and y directions are required. These are calculated for each pixel in the four levels in the Gaussian pyramid which correspond with the levels in the DoG pyramid in which keypoints are detected. For each octave two gradient images are created, one containing four gradient images in the x direction and one containing the four gradient images of the y direction.

Next a buffer is created which contains tiles of 16×16 pixels. Again a buffer with power of two dimensions is created, only 16 times as large in each dimension. Then for each keypoint a 16×16 pixels large quad is drawn. The fragment shader calculates for each of the 16×16 pixels the magnitude and orientation of the gradient of points around the keypoint, using the gradient images calculated in the previous step. The orientation is converted to a bin, and the magnitude is multiplied by a Gaussian function. This weighted magnitude is converted to 16-bit precision (similar to the half float specification [cit08], but excluding special cases like "NaN" and "Infinity") and stored in two channels.

The keypoint main orientations are calculated by drawing a single pixel for each keypoint. In the fragment shader a histogram is created using the bins and weighted magnitudes of each pixel in the 16×16 tile corresponding to the keypoint which was created in the previous step. The keypoint orientations are determined in the same way as the OpenCV implementation. Each orientation is stored in a channel of the texture. This allows for up to four major orientations for each keypoint.

Then a step creating 16×16 tiles similar to the keypoint orientation step is performed. Again the orientation and weighted magnitudes of points around the keypoint are calculated, but now the distance to the bin center is stored as well. Furthermore the vertex shader is used to rotate the keypoint's image patch using the keypoint orientation. This way the orientations are normalised. In the OpenCV implementation the keypoint orientation is subtracted from the calculated orientations for each point, but in the GPU implementation it is more efficient to rotate the image patch, and the end result is (almost) the same.

For the descriptor an 8×8 quad is drawn for each keypoint. This quad will contain the 4×4 histograms of a keypoint. So each histogram is represented by 2×2 pixels. In each pixel two floating point values, converted such that each number is stored in two 8-bit channels as before, are stored. This results in $2 \times 2 \times 2 = 8$ values. Each value represents a bin, and in each of the eight bins, the sum of the weighted magnitudes is stored. In the fragment shader it is determined which two bins should be calculated by the fragment shader, and then all pixels belonging to that bin are processed to calculate the bin's value.

In OpenCV each point in the image patch contributes to the two closest bins in the four closest histograms. The contribution of a single value to to each of the 8 bins is calculated using trilinear interpolation. This interpolation is based on the difference of the keypoint angle and the bin center angle and the distance of the point to the histogram centers. On the GPU calculating a value in a fragment shader and distributing the result over multiple pixels is not possible. Therefore the reverse is done to calculate the histograms. Instead of processing each point and storing the results in multiple bins, the values are calculated per bin, and the corresponding points are gathered. This is an example of a scatter verses a gather approach, which is often mentioned in GPGPU programming texts because a GPU is not capable of scatter algorithms. Because a keypoint contributes to the four closest histograms in the OpenCV implementation, a histogram contains contributions from points in an area which is larger than a 4×4 pixel region. The keypoint image patch is divided in 4×4 regions of 4×4 pixels each. While there are also 4×4 histograms, due to the interpolation of the histograms, the points that contribute to a histogram are from the 4×4 pixels in a region and half the area of the neighbouring regions. This is shown in figure 8.1. Here the center 4×4 area represents a region. The four neighbouring regions are also drawn. The pixels that contribute to the center histogram are shown in grey. When a histogram is at the border there are obviously fewer neighbours. So the histograms are calculated by looking at all grey pixels. Because only two bins can be stored by the fragment shader, this means that all the pixels which influence a histogram will be inspected four times. And because the area of a histogram is up to twice the size of the histogram (the total number of gray pixels is $2 \times 4 \times 4 = 32$), this means each pixel is inspected 16 times. This is a disadvantage of having a parallel architecture and a maximum of 32-bit output.

Next the length of the vector is calculated. This is done by drawing a single pixel for each keypoint. For each keypoint the corresponding $16 \times 4 = 64$ pixels representing the 16 bins are inspected and used to calculate the vector length.

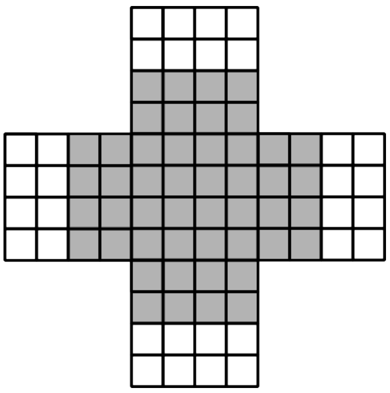


Figure 8.1: *Area in which points contribute to histogram*

Then a buffer with an 8×8 tile for each keypoint is created containing normalised and clamped histograms. These are created by using the histograms and the length calculated in the previous step. The histogram values are divided by the length and then clamped in the range $[0, 0.2]$ as described in [Low04]. Then the length of the new vector is calculated in the same way as before.

Finally a buffer containing 4×4 tiles for each keypoint are created. Here the values of the clamped histogram are normalised using the length calculated in the previous step. The 8 histogram bins are stored in a single pixel using four bits per bin.

The final buffer is read back to main memory. The contents are reordered such that the values of the eight bins of each of the 16 histograms are in succession. This results in the 128 value long SIFT descriptor.

Details on the some steps, such as how values of steps are encoded in the pixel color, will be discussed below.

A tiled Image

During several steps tiled images are created. This section shows what these images look like. An image which has been used during debugging is shown in figure 8.2. This image shows a tiled image containing 16×16 patches of several keypoints. Each tile contains the intensity of the patch of a keypoint from the Gaussian pyramid. The keypoints are rotated, equally to how the patches are rotated during descriptor calculation, which is visible by the fact that the points are mostly bright on the left side and dark on the right side. The Gaussian function which is used to weigh the magnitudes during keypoint orientation calculation and descriptor calculation is used to weigh the intensities, resulting in circular shapes. It is visible that the selected keypoints are located at corner points. The features shown in the image patches have a similar scale, this is because the 16×16 tiles contain samples from a region which depends on the keypoint scale as well as location and orientation. The bottom part of the image is empty. This is because the buffer has power of two dimensions, which results in some excess space if the square root of the number of keypoints is not a power of two.

Grayscale conversion

Conversion to grayscale is achieved by taking a weighted average of the different channels. The image intensity I of an RGB colored pixel P is

$$I = 0.299P_r + 0.587P_g + 0.114P_b \tag{8.2}$$

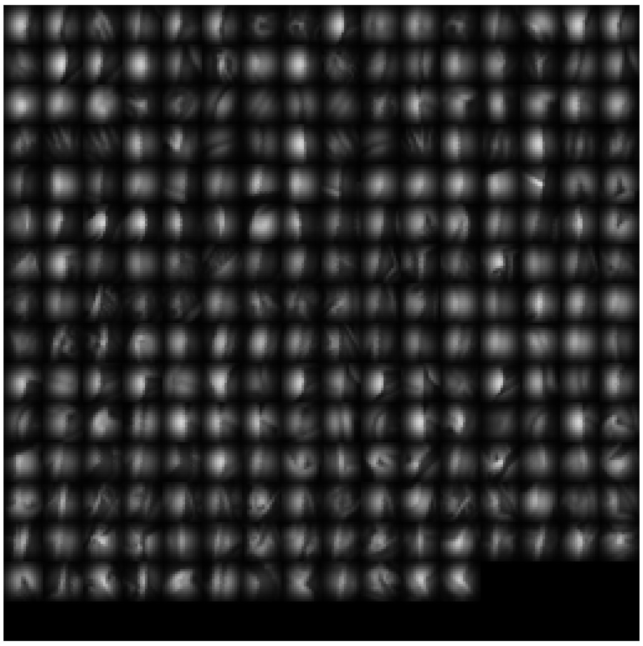


Figure 8.2: *Gaussian-weighted, rotated and tiled image patches*

In a pixel shader this can be achieved in a single operation by performing a dot product using the pixel and the vector containing the weights:

$$I = P_{rgb} \cdot \begin{pmatrix} 0.299 \\ 0.587 \\ 0.114 \end{pmatrix} \tag{8.3}$$

Gaussian pyramid

The Gaussian blurring is performed in two steps, using a separate horizonal and vertical filter. The Gaussian blurring is performed on four levels, which are stored in the four channels of the image, simultaneously. In the horizonal step a single input image is used for all four levels. In the vertical step the levels from the horizontal step are blurred. Because all four levels should have a different amount of blur, four different kernels are supplied to the shader. The approach used to calculate the kernels is different from the OpenCV implementation. In the OpenCV implementation each level is computed from the level below. This means the sigma parameter which is used describes the amount of blur needed to get from one level to the next. When four levels are calculated at once, the sigma should describe the amount of blur needed to get from the input image level to the desired level. So in the GPU implementation the relative sigmas are calculated in a similar manner as in OpenCV, but they are relative to the input image, which might be more than a single level lower.

The kernel size depends on the largest sigma parameter, which is determined by the highest level. As noted in the overview two textures are created. The first holds levels 1 to 4. In this step the initial image, or an image from the previous octave is used as the input image. Then the second set is created using the fourth level, thus the alpha channel of the first texture, as input. Using the last channel from the first set, instead of using the same input image as was used to create the first set is more efficient, because the additional blur required is smaller. The kernel size depends on the amount of blur, so a smaller kernel is required when using level four as input for the second set. The second step creates the second set of blurred images, which holds the levels 4 to 7. This means level four should be copied from the first channel without additional blurring. This is achieved by using a kernel in which all values are zero, except for the center value which is one.

Local extrema detection

As mentioned in the overview the local extrema are detected by inspecting the 3×3 region around each point. Because this buffer needs to be read to main memory it is efficient to combine the detected extrema in different octaves into a single image. This is achieved by rendering the output of all octaves to the same buffer with additive blending. To distinguish the keypoints a different intensity is used in each octave. For the intensity I the following is used depending on the octave o:

$$I = 0.5^o \tag{8.4}$$

This will result in 0.5 for octave 0, 0.25 for octave 1, 0.125 for octave 2, etcetera. Intensities are in the range [0,1] in shaders, but are automatically converted to [0,255] when stored to a buffer. Therefore the values read back will be 128, 64, 32, etcetera. These numbers correspond to in a single bit in the 8-bit number. 128 sets the most significant bit, 64 the second most significant bit, etcetera. Thus bit masking can be used to determine if a certain pixel represents an extremum in a certain octave. This method thus allows for 8 octaves to be stored in a single image.

Low contrast suppression, eliminating edge responses and sub-pixel accuracy

The low contrast suppression, eliminating edge responses and sub-pixel accuracy steps are performed in a single shader because they share calculations. The implementation is similar to the OpenCV implementation. A system of linear equations needs to be solved during these steps. Because such functionality is not available as a library function, this has been implemented. Gaussian elimination is used to solve the linear system.

Because the steps performed in this shader can have multiple outcomes, the first channel is used to encode the result. This result is either "low contrast", "edge", "relocate" or "ok". When the result is either "low contrast" or "edge", the point has either low contrast or is located at an edge, so all keypoints for which these values are returned are discarded. When the result is "relocate" this means the sub-pixel accuracy location is closer to another pixel or scale. All points for which this happens are processed in successive iterations. To calculate the new position used in the next iteration, the x, y and scale offset are stored in channel two to four by the shader. When the result is "ok", this means the pixel has high contrast and is not located on an edge. These points will be used for further processing. The x, y and scale offset are stored in channel two to four by the shader. These are used to adjust the keypoint location to sub-pixel and sub-level accuracy.

When the result is "relocate" the offsets are either $+1$, -1 or 0. For example when the scale result is "1", the keypoint location should lie in the next scale. These are encoded in the result by scaling the $[-1, 1]$ range to [0.1]. When the result is "ok", the offsets are in the range [-.5, .5], because if they where larger the points would be relocated. Therefore the offsets can be encoded in the pixels by adding .5.

During the calculations a determinant of the hessian is calculated. The determinant of a 2×2 matrix is defined as:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc \tag{8.5}$$

When the numbers a, b, c and d are relatively large but the difference between ad and bc is small this can lead to precision problems. These are solved by scaling the values before calculating the determinant. Using the equality shown in equation 8.8 the determinant is scaled as shown in equation 8.7.

$$ad - bc = n^2(\frac{a}{n} \cdot \frac{d}{n} - \frac{b}{n} \cdot \frac{c}{n}) \tag{8.6}$$

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = n^2 \begin{vmatrix} \frac{a}{n} & \frac{b}{n} \\ \frac{c}{n} & \frac{d}{n} \end{vmatrix} \tag{8.7}$$

Keypoint Orientation Assignment

The calculation of the orientation is a two step process. First the gradients in the x and y dimensions are calculated using pixel differences, using these gradient images the orientation and magnitude of points in an image patch around the keypoint are determined. The number of points that are sampled is always the same, but the area which is sampled depends on the keypoint scale. For each keypoint 16×16 points are sampled. The results of all keypoints are drawn as 16×16 images tiled in the output buffer. In each image the bin belonging to the orientation and the Gaussian weighted magnitude are stored.

The last step is creating a histogram of the 16×16 samples and detecting maxima in the histogram. The histogram generation and maxima detection is equal to the OpenCV implementation. Using the maxima keypoint orientations are calculated. To create an orientation estimate which is of higher precision than the histogram bin size, the actual orientation is computed using the values of the bin which contains a maximum and it's neighbours.

Because there can be multiple angles the four channels are used to encode the angles. To denote a channel is not used, (when there are less than four orientations) 255 is stored in the buffer. This leaves the [0.254] range to encode the angle. The angle in degrees is therefore multiplied by $\frac{254}{360}$. This results in a resolution of $\frac{360}{254} \approx 1,417$ degrees. Because the angle was estimated using bins spanning 10 degrees this seems enough. Finally storing at most four orientations seems enough. During testing at most three orientations where returned for a single keypoint.

Half Float Conversion

When performing GPGPU programming intermediate results are stored in a texture. But while the shaders work with floating point numbers with 16-bit precision, the textures only contain four 8-bit numbers for each pixel. When the intermediate results need to have a higher precision this can be achieved by encoding the 16-bit number in two 8-bit numbers.

To use this a function to encode a 16-bit floating point number in two 8-bit numbers and a function which can decode these numbers are required. This has been implemented similar to the IEEE 16-bit floating point number specification [cit08], but the special cases representing "Not a number" and "Infinity" and subnormal numbers are ignored.

A 16-bit half float uses one bit to store the sign, five bits to store the exponent and ten to store the mantissa. The mantissa is assumed to have an implicit lead bit with value one, unless the exponent field contains only zeros. So while only 10 bits of are stored total precision of the mantissa is 11 bits. The exponent bias is 15.

Because bit shifting is not available in GLSL, the $mod()$ and $exp2()$ functions and multiplications/divisions are used. For example to remove the most two significant bits of f, $f = mod(f, 64)$ is used. To shift bytes three positions to the left $f = f * exp2(3)$. Other operations are performed in a similar manner.

Two things are important to note. First the values are converted automatically by OpenGL between the $[0, 1]$ range and $[0, 255]$ when data is transferred between shaders and the buffers. This means that to write 128 in the buffer, the number 0.5 needs to be output by the shader, and vice versa. Secondly when performing these bit operations the expected range is $[0, 255]$, therefore the encoding and decoding functions multiply and divide the values by 255 at the beginning/end of the function. While operations are also possible without this re-scaling this makes it easier to see what is happening, because the operations are similar to bit-shifting operations. During operations the values are still floating point numbers, so the $floor$ function is required after some divisions to remove the remainders.

The source code for these functions is available in Appendix B.

Results and Future work

While most of the above described method has been implemented some issues with the descriptor calculation where not solved. The length calculation results are incorrect. Unfortunately due to time constraint further testing is not possible. It seems however that precision problems might be the cause, similar to those encountered when the determinant of the Hessian was calculated. When calculating the length of the vector the squared values are summed. This results in a very large number after processing a couple of numbers. When small numbers are added to the sum, the problem with combining small and large numbers occurs. For the GPU implementation to be fully functional this problem has to be overcome.

Because the final step has to be modified the performance of the algorithm can change, but most likely the change to the running time will be minimal. The GPU implementation has been tested by performing all steps on a 512×512 image. The input image is the famous "Lena" image used throughout this thesis. The average processing time, calculated using 50 iterations on the Galaxy SII smartphone, is $0.914s$. This is only marginally faster than the OpenCV implementation. The performance in terms of quality could not be tested because of the error in the feature calculation, but the quality of the individual steps has been tested. For example the images in the Gaussian pyramids and DoG pyramids created by the OpenCV implementation and GPU implementation were compared. Few pixels in the resulting images are different and these pixels have differences of at most 2. The keypoint location scales and orientations were compared. While not the exact same solution is created, which is not possible due to for example only having 16-bit precision, the solutions are very similar, returning similar points in each set. The keypoint locations, orientations and scales in the Lena image, calculated using the GPU implementation, are shown in figure 8.3

While the speed gain compared to the OpenCV SIFT implementation is minimal the GPU implementation might still offer a significant speed gain because using the GPU implementation will reduce the workload of the CPU. This way other AR tasks such as object recognition and pose estimation can be performed simultaneously.

While some performance increase is to be expected, there are some improvements possible to the method described above. For example the buffers are transferred back and forth between GPU and main memory in several steps. Some of these transfers could be eliminated. For example locating the keypoints after extrema detection could be done on the GPU using the HystoPyramid method [DZTS08]. Furthermore the number of rendering calls can be decreased by not making individual calls for each keypoint or tile, but make single calls to render an entire buffer. Another possible optimisation is reducing the size of the Gaussian filter kernels. While making them smaller eventually

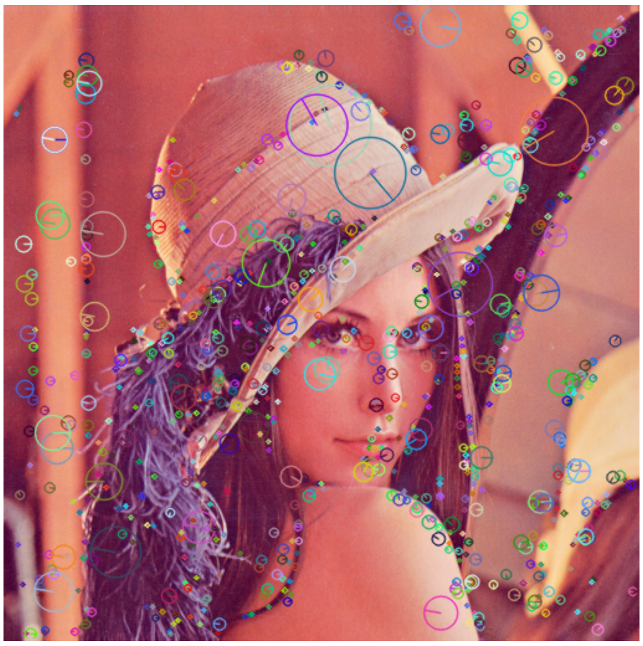


Figure 8.3: *Keypoints detected using GPU SIFT. Circle size denoted the size, the line denotes the orientation.*

leads to reduced quality, some reductions might be possible without decreasing the quality of the keypoint detection step.

But most importantly next generation mobile GPUs will most likely start to catch up with their desktop versions. This is a trend which has been observed in mobile computing and is most likely to continue. This means that next to increased performance, increased functionality like floating point buffers and multiple render targets will most likely become available. This will make a more efficient implementation of the SIFT GPU algorithm possible.

8.1.2 Camera correction

As described in Appendix A the camera might have certain defects. These include the non-linear deformations such as radial distortion and tangential distortion. Because these are not linear, these can not be implemented using an efficient matrix multiplication. While OpenCV contains functionality to correct images containing these deformations, it is too slow to include in the AR framework. A more efficient method is implemented using the GPU.

The method contains two steps. First a preprocessing step is performed to create a lookup table texture, and a second step which uses the lookup table. First a two-channel image is created of the same size as the camera images. This image is filled such that each pixel p_i is calculated depending on the coordinates of the point (x_i, y_i) and the width w and height h of the image:

$$p_i = (x_i/w, y_i/h) \tag{8.8}$$

The result is an image in which each point corresponds to the texture coordinates. A 16-bit precision image is used in order to have a high precision. Next this image is transformed using the OpenCV camera correction method. This image is converted into a four-channel image with eight bits per channel. The values of the corrected image are encoded to use two channels for each value as described in section 8.1.1. This image is uploaded to a texture on the GPU, this texture will be called the lookup texture. Now when the camera image is drawn, instead of using the texture coordinates normally used to fetch the pixel data, the texture coordinates are used to fetch the two encoded values in the created lookup texture. These values are used as texture coordinates when fetching the pixel from the camera texture.

When the image containing the texture coordinates was transformed, the coordinates where changed to a new location. This means that each point in the transformed image contains the texture coordinates from the original image. Because the values in the lookup texture thus contain the source pixel location for each pixel in the corrected image, the above method will create a corrected image.

This is fast because for each pixel only an extra texture lookup and floating point value decoding has to be performed. The actual deformations in the camera where found to be minimal. The quality of the camera on the Samsung Galaxy S2 is good, such that minimal errors are present. The new texture coordinates are not located at pixel centers, and thus interpolation is used to draw the camera image. This leads to slightly reduced image quality. Because of the slight image quality reduction due to interpolation and minimal image quality gain due to corrections, the method is not required for devices with a good camera. On devices with a camera which have larger deformations this method might be more useful.

8.2 Feature Tracking

Because the pose of a camera most likely changes by a small amount in successive frames, the keypoints in the camera view in some frame are most likely relatively close to the location in the previous frame. If the keypoints from the previous frame are tracked between frames the keypoint detection, feature calculation and matching does not have to be performed. Because these operations are computationally expensive, tracking the movement of keypoints is faster.

Tracking is implemented using OpenCV functionality. OpenCV implements the Lucas-Kanade optical flow in pyramids [yB00]. This method is able to track a sparse set of points in succesive frames. It works by inspection a small region around a point to find the new location.

When tracking is enabled the framework starts using SIFT normally to find keypoints, detect objects and estimate the camera pose. Once a pose has been determined the set of keypoints that resulted in the estimated pose are stored. This set only includes keypoints that were used in the pose estimation, so for example points that where filtered during the object recognition are excluded. For the successive frames SIFT is not performed, instead the keypoints are tracked using the Lucas-Kanade method. This will update the locations of points. The Lucas-Kanade method as implemented by OpenCV returns a status for each point which denotes which keypoints could be located in the new image. Using these statuses, points which could not be located are removed. While this removes failures, there are often still errors is the new set of locations. Often the location of one or more keypoints are not tracked correctly. These badly tracked points are removed using a homography. This homography is the same as the homography described in section 5.2. It is used to reject points which where not tracked correctly. During the homography calculation, badly tracked points will automatically be ignored because of the RANSAC method described in section 5.2. After the homography is constructed all the points that are outliers are removed from the set of tracked points. The homography is also checked for correctness by transforming a square as described in section 5.2.3.

Once the homography is calculated the pose estimation can be performed using the calculated homography after which the rendering steps are performed. This continues until a homography can not be constructed or when the number of keypoints is lower than a threshold. During testing a minimum of eight appeared a good minimal threshold.

So the tracking starts when a pose has been estimated using SIFT, tracking continues as long as points are tracked correctly. This results in a homography which is for pose estimation as before. When tracking fails the AR framework switches back to the SIFT method.

Tracking is much faster than using SIFT. Using the tracking method about 3 frames per second are achieved. The resulting homography is of similar quality as the homographies created using SIFT. Especially when a strict re-projection threshold is used and when the minimal number of keypoints is at least eight. If the re-projection threshold or minimal number of keypoints is lowered the quality of the homography is lower, but this will reduce the number of times the AR framework switches back to SIFT. An additional advantage of tracking is that keypoints are often correctly tracked even if the view angle of the camera changes a lot. The range is about double the range at which SIFT can correctly identify keypoints.

Using tracking increases the performance a lot. The downside is that the speed of the system changes abruptly when switching between tracking and SIFT. Also no new objects are detected once tracking is started. This means if a user pans from left to right such that one-by-one several objects appear in the view, the new objects are not recognised because no new keypoints are added once tracking has started. However, when using SIFT in this setting more and more objects will detected and subsequently be used for pose estimation, which results in a more accurate pose estimate. A possible extension to the tracking algorithm is the use of asynchronous updates. This means that SIFT runs in the background at a lower framerate, while tracking is performed in another thread. The tracking thread provides fast updates to the pose estimation. This way new objects could be added while tracking and there would be no speed change because of switching between tracking and SIFT mode because they would run simultaneously.

9 Results and Conclusions

This thesis describes a complete implementation of an AR framework for Android devices. For AR the first requirement is calculating the camera intrinsics and extrinsics, because these are needed to construct a virtual camera which can be used to draw virtual objects.

The camera intrinsics are computed once using a checkerboard. The extrinsics are computed using object recognition. After detecting objects, their 3D locations can be used to estimate the camera extrinsics, or camera pose. The object recognition is based on detecting and matching keypoints. The keypoint detection and description is implemented using SIFT. This method is able to match keypoints under varying conditions such as changing lighting conditions or camera viewpoints. SIFT outperforms similar methods on quality. After keypoint matching is performed, the location of objects is deduced from the set of keypoint matches. This results in a homography. This homography is used to create a 2D-3D correspondence set from which the camera intrinsics or pose is deduced.

This work shows that AR based on image recognition is possible on Android devices. Compared to previous work this has the advantage that markers or special hardware are not required. Several optimisations are investigated to increase performance. A GPU implementation shows improvement over the OpenCV implementation and could possibly be improved upon further, especially if GPU hardware develops. Tracking gives and improvement of a factor of three in framerate. The downside of tracking as implemented is the switching between tracking and SIFT detection. This switching does not allow adding points to the set of tracked points and results in large changes in framerate when switching occurs.

9.1 Visuals

Figure 9.1 to 9.3 show results of the AR framework in which a virtual box is drawn relative to a book. As the camera moves, the box moves accordingly.

9.2 Future work

Future work could focus on improving the framerate. While the current framework provides a working system the low framerate makes is unsuitable for most applications. Improving the framerate could be achieved by optimizing the GPU implementation. Some suggestions to improve the current implementation are already given, but most importantly the next generation of phones will most likely

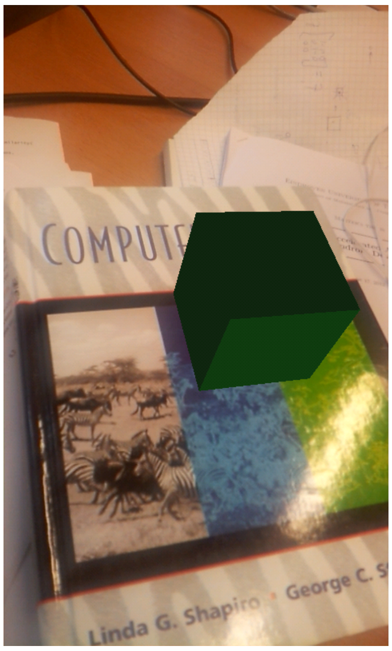


Figure 9.1: *Result1*

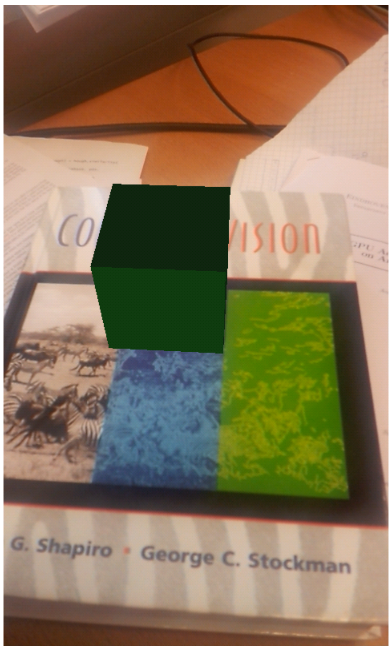


Figure 9.2: *Result 2*

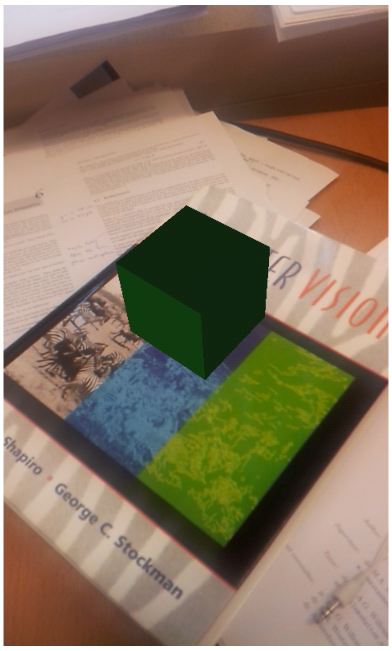


Figure 9.3: *And another angle..*

make a much more efficient implementation possible. Also the optimal combination of using both the GPU and the CPU could be investigated. The application should be designed such that the GPU and CPU part are not executed sequentially, but are executed in parallel. Next the tracking method shows great potential. Future work could investigate asynchronous updates.

Other future work could implement the suggestions made in section 7.2 which describe how the appearance of the virtual objects can be improved by adding for example shadows and lighting matching the environment or by taking occluders into account.

A Appendix

A camera uses a lens to project the world on a light sensitive sensor to create an image. There are several parameters that describe how the world is projected onto the sensor, and thus how the world appears in a two dimensional image. These parameters are:

focal length Distance to focal point of the lens, the further away the focal point, the smaller the field of view. The focal length might be different in the x and y dimensions.

principal point Point which corresponds with the center of the lens, should be near the image center.

sensor size Size of the sensor.

skew Coefficient between the x and the y axis, often zero.

radial distortion Barrel distortion, pincushion distortion or both. See figure A.1 and A.2

tangential distortion Distortion which occurs when the lens is tilted compared to the lens.

A principal point which is not equal to the image center, a non zero skew coefficient, radial distortion and tangential distortion are caused by production errors. When a camera has such production errors an image from this can be corrected in software when the parameters are known. This process is called camera calibration. When the image is corrected the images are called calibrated images.

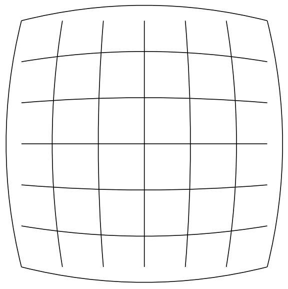


Figure A.1: *Barrel distortion*

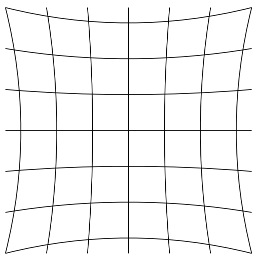


Figure A.2: *Pincushion distortion*

The focal length, principal point and skew parameters are linear parameters. This means there is a transformation matrix which represents these parameters. This is shown in equation A.1.

$$sm = A\left[R|t\right]M \tag{A.1}$$

Where s is the scale, m the point in the image, A the camera intrinsics parameters matrix, $[R|t]$ the rotation/translation matrix and M, the point in 3D. The Matrix a contains the intrinsic parameters and is defined in equation A.2.

$$A = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{A.2}$$

Here f_x and f_y are the focal length in the horizontal and vertical direction expressed in pixel units, γ is the skew coefficient and c_x and c_y together denote the principal point.

The radial and tangential distortions are non-linear parameters and can not be represented using a linear transformation. They are represented by coefficients $(k_1, k_2, k_3, k_4, k_5, k_6)$ and (p_1, p_2)

The transformation in from M to m can also be written as:

$$\begin{aligned} \begin{pmatrix} x \\ y \\ z \end{pmatrix} &= R \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + t \\ x' &= x/z \\ y' &= y/z \\ u &= f_x * x' + c_x \\ v &= f_y * y' + c_y \end{aligned} \tag{A.3}$$

where $M = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$ and $m = \begin{pmatrix} u \\ v \end{pmatrix}$.

To incorporate radial distortion and tangential distortion, equation A.3 is extended to A.4, which uses the coefficients $(k_1, k_2, k_3, k_4, k_5, k_6)$ and (p_1, p_2):

$$\begin{aligned} l \begin{pmatrix} x \\ y \\ z \end{pmatrix} &= R \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + t \\ x' &= x/z \\ y' &= y/z \\ x'' &= x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2 p_1 x' y' + p_2 (r^2 + 2 x'^2) \\ y'' &= y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1 (r^2 + 2 y'^2) + 2 p_2 x' y' \\ \text{where} \quad r^2 &= x'^2 + y'^2 \\ u &= f_x * x'' + c_x \\ v &= f_y * y'' + c_y \end{aligned} \tag{A.4}$$

A.1 Camera intrinsic parameters calculation

To determine the camera intrinsic parameters multiple methods exist. Methods which use special equipment such as orthogonal planes and methods which analyse images taken with the camera, often with a checkerboard pattern. The method which uses images of a checkerboard pattern is used for the framework because it is the easiest to perform.

The idea of the method is that the corner points of the checkerboard are detected in the camera images. Using multiple of these checkerboard images and initial guesses of camera poses and camera parameters, an iterative two-step algorithm is performed. In the first step the camera pose estimate is improved using estimated camera properties and the known checkerboard dimensions similar to the pose estimation described in chapter 6, then in the second step the camera parameters are estimated using the camera poses and the location of the corners in the checkerboards in the camera images. In each iteration both the estimate of the camera intrinsics and the poses of the camera are improved. There are several methods which employ this general idea. The method used in the framework is the method by Zhang [Zha00], which is available in the OpenCV library.

A.2 Implemented tooling

The AR framework includes functionality to determine the camera intrinsics. A user has to use a checkerboard pattern. Any checkerboard can be used, but because the dimensions need to be known these have to be supplied. Then multiple images of the checkerboard have to be taken. When the checkerboard is detected in the camera view, the corner points are highlighted as shown in figure A.3 and a image can be captured. After an image is successfully captured, the area of the captured checkerboard is used to fill the camera view with a partially transparent color as shown in figure A.4. This way the user can see which area of the screen is used by checkerboards. This is useful because the most accurate results are acquired when the checkerboards are distributed over the entire screen. After several images are taken from different angles, the camera intrinsics can be automatically computed. The values calculated using this method will then automatically be used when performing AR.

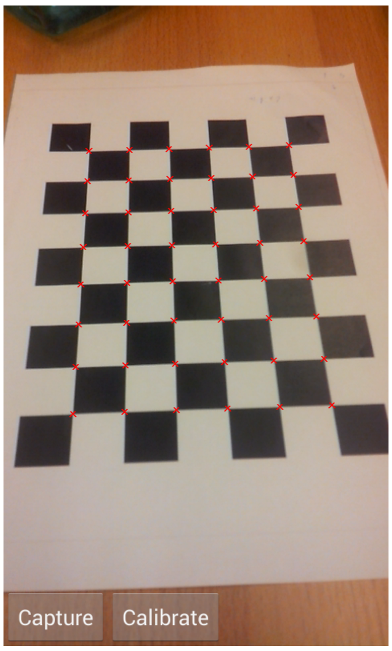


Figure A.3: *Once the checkerboard is detected the corners are highlighted.*

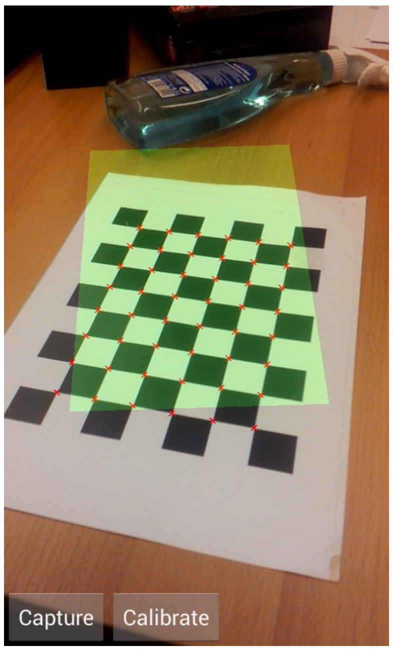


Figure A.4: *Once an image is captured the area of the checkerboard is shown using a translucent area.*

B Appendix

This appendix contains code snippets.

B.1 Encoding/Decoding

Encoding and decoding a 16 bit floating point number in two 8 bit numbers in GLSL such that they can be stored in 8 bits per pixel buffers.

```
vec2 encode16(float f) {
        float f2 = abs(f);
        if(f2==0.0){ //special case
                return vec2(0);
        }
        float sign = step(0.0,-f);
        float exponent = floor(log2(f2));
        float mantissa = f2/exp2(exponent);
        Exponent -= (1.0 - step(1.0, mantissa));
        Exponent +=  15.0; //exponent bias
        vec2 result = vec2(128.0 * sign  + floor(mod(exponent, 32.0)*4.0) +
                mod(floor(mantissa*4.0), 4.0), floor(1024.0*mod(mantissa,0.25)));
        return vec2(1.0/255.0) * result; //scale values to [0,1] will be scaled back
                                          //to [0,255] by OpenGL when written to buffer
}

float decode16(vec2 v)
{
        v *= 255.; //scale to [0,255]
        float sign = 1. - step(128.0,v.x)*2.; //set MSB
    //get bits 2 to 6 and subtract bias
        float exponent = floor(mod(v.x,128.) / 4.) - 15.;
        if(exponent==-15.)
                return 0.;
    //create number from bits 5 and 6 from 1st number and second number
    //and add implicit bit
```

```
        float mantissa = mod(v.x,4.0)*256. + v.y + float(0x400);
        return sign * exp2(exponent-10.0) * mantissa;
}
```

Bibliography

[AHF06] Alaa E. Abdel-Hakim and Aly A. Farag. CSIFT: A SIFT descriptor with color invariant characteristics. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '06, pages 1978–1983, Washington, DC, USA, 2006. IEEE Computer Society.

[Ait10] Miika Aittala. Inverse lighting and photorealistic rendering for augmented reality. *The Visual Computer*, 26:669–678, 2010. 10.1007/s00371-010-0501-7.

[API12] Android API. Android surfacetexture API, August 2012.

[BETVG08] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (SURF). *Comput. Vis. Image Underst.*, 110(3):346–359, June 2008.

[BG09] Gertjan J. Burghouts and Jan-Mark Geusebroek. Performance evaluation of local colour invariants. *Comput. Vis. Image Underst.*, 113(1):48–62, January 2009.

[BL02] Matthew Brown and David Lowe. Invariant features from interest point groups. In *In British Machine Vision Conference*, pages 656–665, 2002.

[Bou98] S. Bougnoux. From projective to euclidean space under any practical situation, a criticism of self-calibration. In *Computer Vision, 1998. Sixth International Conference on*, pages 790 –796, jan 1998.

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[Bre11] Maxime Brenon. An iOS app featuring SIFT descriptors extraction with OpenGL ES 2.0. https://github.com/Moodstocks/sift-gpu-iphone, 2011. [Online].

[Bul12] Christopher Bulla. Local Features for Object Recognition. In *POSTER 2012*, 2012.

[BZMn08] Anna Bosch, Andrew Zisserman, and Xavier Muñoz. Scene classification using a hybrid generative/discriminative approach. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(4):712–727, April 2008.

[cit08] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008.

[CLO$^+$12] Michael Calonder, Vincent Lepetit, Mustafa Ozuysal, Tomasz Trzcinski, Christoph Strecha, and Pascal Fua. Brief: Computing a local binary descriptor very fast. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34:1281–1298, 2012.

[CP11] Adrian Clark and Thammathip Piumsomboon. A realistic augmented reality racing game using a depth-sensing camera. In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry (VRCAI '11)*, pages 499–502, Hong Kong, China, 2011. ACM.

[Cri12] Thorstin Crijns. Augmented reality for indoor applications on mobile devices. Master's thesis, Technische Universiteit Eindhoven, the Netherlands, 2012.

[DD95] Daniel F. Dementhon and Larry S. Davis. Model-based object pose in 25 lines of code. *Int. J. Comput. Vision*, 15(1-2):123–141, June 1995.

[DH72] Richard O. Duda and Peter E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, January 1972.

[DZTS08] Christopher Dyken, Gernot Ziegler, Christian Theobalt, and Hans-Peter Seidel. High-speed marching cubes using histopyramids. *Comput. Graph. Forum*, pages 2028–2039, 2008.

[EIP97] Shimon Edelman, Nathan Intrator, and Tomaso Poggio. Complex cells and object recognition, 1997.

[FB81] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[FH08] Kayvon Fatahalian and Mike Houston. A closer look at GPUs. *Commun. ACM*, 51(10):50–57, October 2008.

[Fol10] Tim Folger. *The National geographic magazine.* Washington :National Geographic Society,, 2010. http://ngm.nationalgeographic.com/big-idea/14/augmented-reality.

[GHTC03] Xiao-Shan Gao, Xiao-Rong Hou, Jianliang Tang, and Hang-Fei Cheng. Complete solution classification for the perspective-three-point problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(8):930–943, August 2003.

[H88] C. Harris and M. . A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988.

[Har11] David Hardy. Augmented Reality for Landscape Development on Mobile Devices. Master's thesis, Technische Universiteit Eindhoven, the Netherlands, 2011.

[HF11] Steven Henderson and Steven Feiner. Exploring the benefits of augmented reality documentation for maintenance and repair. *IEEE Transactions on Visualization and Computer Graphics*, 17(10):1355–1368, October 2011.

[Hou62] Paul Hough. Method and Means for Recognizing Complex Patterns. U.S. Patent 3.069.654, December 1962.

[HZ04] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[Ipp12] Dennis Ippel. Rajawali: An OpenGL ES 2.0 Based 3D Framework For Android. `http://www.rozengain.com/`, 2012.

[JG09] Luo Juan and Oubong Gwon. A Comparison of SIFT, PCA-SIFT and SURF. *International Journal of Image Processing (IJIP)*, 3(4):143–152, 2009.

[JNA$^+$05] Katrien Jacobs, Jean-Daniel Nahmias, Cameron Angus, Alex Reche, Celine Loscos, and Anthony Steed. Automatic generation of consistent shadows for augmented reality. In *Proceedings of Graphics Interface 2005*, GI '05, pages 113–120, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[Kay10] Guy-Richard Kayombya. SIFT feature extraction on a Smartphone GPU using OpenGL ES2.0. Master's thesis, Massachusetts Institute of Technology, the Netherlands, 2010.

[Khr08] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[KS04] Yan Ke and Rahul Sukthankar. PCA-SIFT: a more distinctive representation for local image descriptors. In *Proceedings of the 2004 IEEE computer society conference on Computer vision and pattern recognition*, CVPR'04, pages 506–513, Washington, DC, USA, 2004. IEEE Computer Society.

[Lev44] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly Journal of Applied Mathmatics*, II(2):164–168, 1944.

[Lin94] Tony Lindeberg. Scale-space theory: A basic tool for analysing structures at different scales. *Journal of Applied Statistics*, pages 224–270, 1994.

[LMNF09] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. Epnp: An accurate o(n) solution to the pnp problem. *Int. J. Comput. Vision*, 81(2):155–166, February 2009.

[LoB00] Vincent Lepetit and Marie odile Berger. Handling occlusions in augmented reality systems: A semi-automatic method. In *Proc. Int. Symp. Augmented Reality 2000 (ISAR 00), IEEE CS Press, Los Alamitos, Calif*, pages 137–146, 2000.

[Low04] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[Mar63] Donald W. Marquardt. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *SIAM Journal on Applied Mathematics*, 11(2):431–441, 1963.

[McC93] Ernest McCollough. Photographic topography. *Industry: A Monthly Magazine Devoted to Science, Engineering and Mechanic Arts*, 54(1):399–406, January 1893.

[ML09] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[NVI07] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[OHL$^+$08] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879 –899, may 2008.

[OLG$^+$07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[RD06] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European Conference on Computer Vision*, volume 1, pages 430–443, May 2006.

[Ros99] Paul L. Rosin. Measuring corner properties. *Computer Vision and Image Understanding*, pages 291–307, 1999.

[RRKB11] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. ORB: An Efficient Alternative to SIFT or SURF. In *International Conference on Computer Vision*, Barcelona, 11/2011 2011.

[TL12] T. Trzcinski and V. Lepetit. Efficient Discriminative Projections for Compact Binary Descriptors. In *European Conference on Computer Vision*, 2012.

[TM08] Tinne Tuytelaars and Krystian Mikolajczyk. Local invariant feature detectors: a survey. *Found. Trends. Comput. Graph. Vis.*, 3(3):177–280, July 2008.

[vdSGS10] K. E. A. van de Sande, T. Gevers, and C. G. M. Snoek. Evaluating color descriptors for object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1582–1596, 2010.

[vdWGB06] J. van de Weijer, T. Gevers, and A.D. Bagdanov. Boosting color saliency in image feature detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(1):150 –156, jan. 2006.

[Wit83] Andrew P. Witkin. Scale-space filtering. In *Proceedings of the Eighth international joint conference on Artificial intelligence - Volume 2*, IJCAI'83, pages 1019–1022, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[Wu07] Changchang Wu. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT). `http://cs.unc.edu/~ccwu/siftgpu`, 2007.

[yB00] Jean yves Bouguet. Pyramidal implementation of the Lucas Kanade feature tracker. *Intel Corporation, Microprocessor Research Labs*, 2000.

[Zha00] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:1330–1334, 2000.