Structure From Motion in CUDA

Master thesis
60 credits

Svein Atle F. Arnesen
saarnese@ifi.uio.no
sveinatle@gmail.com
http://folk.uio.no/saarnese

1. November 2010
Structure From Motion in CUDA

Svein Atle F. Arnesen

1. November 2010
Abstract

State of the art image analysis techniques such as SURF feature detection and description are fused with multiple view geometry theory in order to implement a Structure from Motion pipeline. The Structure from Motion pipeline tracks features in the video stream and uses the tracking data to estimate the movement of the camera. The three-dimensional structure recorded by the video is then reconstructed by using the knowledge of the cameras and the projected points to triangulate the three-dimensional locations of the points. Special considerations and modifications to existing methods are made to leverage the advantages and minimize the disadvantages of reconstructing from video versus from sets of unrelated images. The implementation is designed to take advantage of the processing power of CUDA due to the sheer amount of data involved in processing a video.
Contents

1 Introduction ................................................. 6
   1.1 Problem statement .................................... 6
   1.2 Related work .......................................... 7
   1.3 Disposition ............................................. 9

2 Outline Of The Process .................................. 10
   2.1 The process from input to output .................... 10
   2.2 Ecosystem ............................................... 11

3 Interest Point Tracking ................................... 12
   3.1 The steps that make up interest point tracking .... 12
      3.1.1 Locating candidate interest points ............ 12
      3.1.2 Generating descriptors ......................... 14
      3.1.3 Matching ........................................... 18
   3.2 Resulting implementation of interest point tracking 20
      3.2.1 Input .............................................. 21
      3.2.2 Format conversion .................................. 21
      3.2.3 Integral image generation ...................... 23
      3.2.4 Interest point detection ......................... 25
      3.2.5 Interest point description ...................... 27
      3.2.6 Interest point matching ......................... 31
      3.2.7 End of the interest point tracking stage ....... 36

4 3D Reconstruction ....................................... 38
   4.1 Theory of perspective projections ................... 38
      4.1.1 Homogeneous coordinates and the projective space 38
      4.1.2 Translation ........................................ 40
      4.1.3 Perspective projection .......................... 41
      4.1.4 The camera matrix and intrinsic parameters .... 42
   4.2 Theory of the reconstruction problem ............... 43
1 Introduction

The idea of machines that see is not a recent one. Many a science fiction author has envisioned humans one day being able to create intelligent machines similar to humans, complete with vision and all. Though interest in the field may have been there for a while, actual research into computer vision harks back no further than to the late 1960s [RFM07]. Vision involves processing enormous amounts of information [Dav04], and so research has been at the mercy of the computational resources available. As computers have grown more powerful, so has the research into computer vision expanded immensely. Initially computer vision was thought to be a very simple problem to solve — the kind of problem a student could solve as a summer project [RFM07] — but as time progressed it dawned on researchers that the “problem of computer vision” was not in fact a single, easily solvable and well-defined problem. No, computer vision is not a “problem”; computer vision — as has become clear today — is a complex field composed of many wildly different problems and subfields, intersecting with a wide range of other fields such as image processing and image analysis, signal processing, geometry, pattern recognition, artificial intelligence and statistics as seen in Figure 1.

1.1 Problem statement

The purpose of this study is to experimentally investigate how state of the art techniques can be combined as an implementation of a complete video-to-structure pipeline in CUDA.

This is accomplished by applying techniques of Structure From Motion (SfM). SfM is an umbrella term for techniques which extract 3D structural information from a set of 2D images with different viewpoints. The techniques applied in this study involve detecting and tracking particular points in the
video, estimating camera motion based on the behavior of such points, and reconstructing the 3D scene from the tracked points and estimated cameras. Part of the results of this study is more insight into special considerations that can be made when dealing with video as input.

Leveraging the computational power of modern Graphics Processing Units (GPUs) is essential because of the large amounts of processing involved. Compute Unified Device Architecture (CUDA) is an architecture developed by NVIDIA Corporation [NVI10a]. CUDA enables developers to perform general-purpose parallel computations on the GPU. By using CUDA to implement the pipeline, this study will also highlight the primary concerns to keep in mind when implementing SfM in a parallel computing environment.

This study will only concern itself with video recordings of static scenes; that is, the structure is fixed and the camera is the only thing to move.

1.2 Related work

This study focuses on extracting structure from video by detecting and tracking particular 2D points and then using that information to estimate camera movement and 3D structure. The first stage of this process involves detecting and describing particular points — features or more precisely interest points — in the 2D images. These are points which can be found again in other images.

One algorithm for detection and description of interest points is Scale-Invariant Feature Transform (SIFT) [Low99]. SIFT finds interest points by locating maxima and minima in the Difference of Gaussians of a scale-space representation of the image. Scale-space is a representation where the original image is successively downsampled, creating a stack of successively smaller versions of the original image. Scale-space is an important concept in interest point tracking because it makes it possible to detect the same feature in different images with different scales. Difference in the scale of two images may be a result of the images having different resolutions, or simply that one of the cameras were zoomed in or located closer to the structure.
After interest points have been detected, SIFT then assigns orientations to the interest points based on the gradient in the area around the interest point, before it finally generates descriptors based on histograms of the gradient in smaller areas surrounding the interest point. Descriptors are also an important concept in interest point tracking; they are used for recognizing interest points from different images.

Speeded-Up Robust Features (SURF) [BTG06] is a newer method which is in many ways inspired by SIFT. Compared to SIFT it is more robust in the face of various transformations such as rotation, changing viewpoint and changes in illumination. It also introduces the use of integral images for speeding up calculations, and is in fact faster than SIFT.

SURF detects interest points as maxima and minima in a scale-space with a Haar-wavelet filter applied. Instead of downsampling the image to create the scale-space, SURF uses the integral image to apply the filter with increasing filter size. The integral image means that filters of any size can be applied with constant time complexity.

Features as detected by SIFT and SURF have well-defined positions within the image, which is why they are referred to as interest points. There are however other classes of features that may be used to extract information from an image. A common type of feature are edges which may be detected by various gradient transforms such as the Canny edge detector [Can86] or the Sobel operator [NA08]. Smallest Univalence Segment Assimilating Nucleus (SUSAN) [SB97] and the Harris operator [HS88] are two of several feature detectors designed to locate corners.

Single Image Dehazing [Fat08] is a different technique which employs statistical measures to determine relative depth in a single picture based on fog or haze. Other techniques, such as Shape From Texture [SF01], is able to estimate the gradient of textures by analyzing the local distortion of textures.

Hartley et al. show how the theory of multiple view geometry can be used to reconstruct a 3D scene by analyzing the 2D movement of the projected points...
between different images [HZ04]. The key idea here is to determine the movement of the camera by analyzing the parallax effect on the projected points, and then triangulate the 3D location of points.

Image analysis and computer vision have applications in a wide range of fields including robotics, security, motion tracking in the entertainment industry, medicine and astronomy [BK08]. This has motivated the creation of OpenCV, an open source software library of functions for performing many of the common tasks in computer vision. With modern, high performance GPUs becoming more common, implementations which take advantage of their computational power are also becoming common. CULA [Pho10] is a CUDA implementation of the LAPACK [LAP10] linear algebra library, providing functionality such as Singular Value Decomposition.

SfM techniques have also been put to commercial use by companies such as 2d3 [2d310] who supplies products for entertainment, industry and government, and Yotta DCL [Yot10] who specialize in highway surveying.

1.3 Disposition

Structure from motion is really the fusion of two unrelated knowledge bases: digital image processing and geometric algebra. This is reflected in the organization of this study by treating interest point tracking and 3D reconstruction separately. First of, Chapter 2 briefly describes the entire process in order to provide an overview and motivation for its various components. Chapter 2 also describes the expectations and requirements. The image processing aspect is tackled in Chapter 3. First the necessary theory is introduced, and then the resulting implementation is discussed and justified. The geometric aspect is handled in Chapter 4 which — much like Chapter 3 — first introduces the needed theory before discussing how it was applied in the implementation. Lastly, Chapter 5 concludes by summarizing the results.
2 Outline Of The Process

The ultimate goal of this study is to reconstruct the 3D structure recorded by a video camera. This is accomplished not by simply implementing a single algorithm in isolation, but by application of techniques and concepts ranging from image processing algorithms to algebraic geometry. Generating a 3D structure from the input image data involves several steps, transforming the data through several different forms. At this point we could jump right into the basic theory and gory details of each step; however, the purpose and relevance of each step — how it relates to the ultimate goal — may not be immediately obvious. This chapter is therefore intended to provide an outline of the entire process, to motivate the intermediate steps that lead up to the final reconstruction. It sets the scene before the following chapters delve into the details.

2.1 The process from input to output

There is no “the one way” to reconstruct a scene from an image or video source. The nature of image and video data allows for large variations in the characteristics of the input data, and extracting structure from such data can in a sense be considered an art. The approach applied in this study is fairly obvious given some background knowledge of computer vision.

The first step is — quite naturally — to load the video data. The video is loaded one frame at a time from the disk and then transferred to the GPU.

Once a frame has been uploaded to the GPU, it is analyzed to detect interest points. Descriptors are generated for each interest point so that it will be possible to compare interest points from different images. Additional frames are uploaded and analyzed similarly. Interest points that are extracted from each frame are then matched against the interest points in prior frames by comparing their descriptors. Interest points can thus be tracked across several frames. Information about the movement of all the interest points is stored in a large table of known interest points. This table, which is continuously updated
as new frames are processed, is the output of the interest point tracking stage.

The second stage is the reconstruction stage. This stage uses information about the movements of all known interest points to estimate the motion of the camera from frame to frame. With this information in hand, the 3D positions of the points tracked in the table can be calculated by triangulation. A 2D point in the image plane actually represents a ray in 3D space from the focal point and through the 2D point in the image plane. The actual 3D point lies somewhere on this ray. Triangulating a 3D point comes down to considering the rays of all its projections in different images, and determining where in 3D space all these rays intersect. In real world applications there will be noise and other inaccuracies, so the rays will not perfectly intersect. In practice, therefore, the 3D location of the point is estimated by finding a least squares solution.

2.2 Ecosystem

This study limits itself to dealing with static scenes. Static scenes are scenes without any form of movement. The only variable is the viewpoint and rotation of the camera.

The interest point tracking, based on SURF, is a cornerstone of the implementation. Such interest point tracking relies on the structures being of a nature which results in a number of interest points which can be tracked. The source of such interest points is primarily variations in the textures of the structure, so the approach studied here will not work well on scenes which are largely composed of structures with very smooth surfaces; we need to assume that the structure has rich textures.

The purpose of this study is to investigate applying the techniques to video, so we assume that the input is a video and not a series of unrelated images. In accordance with this, we also assume that the video is somewhat stable and free from excessively jerky movements.
3 Interest Point Tracking

The previous chapter outlined the entire process of Structure From Motion from image data to 3D reconstruction. Hopefully that introduction made clear the purpose and motivation behind interest point tracking, which will now be investigated further.

3.1 The steps that make up interest point tracking

For purposes of discussion, we roughly split the process of interest point tracking into three steps. The first step is discovering the location of candidate interest points in a given image. The second step is generating a descriptor for each candidate interest point. Finally, the third step is the actual tracking, i.e. identifying and locating these same interest points in other images. This matching process of finding the correspondences between images is facilitated by the descriptors that were generated in the previous step. Each of these three steps will be detailed in the following sections, starting with locating candidate interest points.

3.1.1 Locating candidate interest points

The first step of tracking interest points across several images is finding the interest points. Images typically contain many features of different classes with varying properties that modulate their usefulness in different applications [SMB00]. Common classes of features are corners or points, edges and ridges, blobs and textures [NA08]. Finding these features can be accomplished by applying some operator, that is, a feature detector, to the image. Some well known feature detectors are the Canny edge detector, the Sobel edge detector and the Difference of Gaussians (DoG). These feature detectors compute a response map were local maxima or minima identify potential features. The Sobel edge detector is illustrated in Figure 2 and the Canny edge detector in Figure 3. Not all features, or feature detectors, are well suited for the task at hand though. This section
describes the most important characteristics [SMB98] of a detector as required by our application.

**Well-defined position** We will depend on the position of the features for calculating the motion of the camera and in turn the reconstruction of the 3D scene. The accuracy of these results will benefit from our features having well-defined, precise positions. A feature having a *well-defined* position means that the feature maps naturally to a single point in image space. This makes certain detectors such as edge detectors ill-suited for our uses because edges by definition extend over a larger area of image space. Figure 4 shows SIFT features detected in an image. A SIFT feature is located at the center of each square.

A property of images from a video source is that the baseline — the line connecting two focal points — between images is very small. This has the unfortunate effect that small inaccuracies in the registered positions of features may significantly affect the estimated 3D position of these features [Dav04]. This is detailed further in Section 3.2, but is noted here as well since it justifies the concern for striving for well-defined positions.

**Stability** The approach to 3D reconstruction employed in this study depends on being able to find the same point in several different images. This requires the response from the feature detector to be stable to a certain degree in the presence of noise and transformations. The input images in this particular application come from a video source. This application — compared to other applications where the input images come from a vast range of different cameras,
viewpoints, lighting conditions, etc. — is therefore afforded some flexibility with regards to the inherent stability of the feature detector, because each image will come from the same camera and have a viewpoint and lighting conditions very similar to the previous image in the video stream.

**Parallelizable and fast**  The feature detector needs to be parallelizable since it is to be implemented on the GPU, which is parallel in nature. This is not an issue for most detectors, but still warrants mentioning. A more important concern when working with video streams is that the detection process is efficient since there is a large amount of image data to process in a video stream. This naturally is of particular importance when the goal is real time processing.

Features that are stable, have well-defined positions and for which a descriptor may be generated are typically referred to as interest points [SMB98].

### 3.1.2 Generating descriptors

The second step in interest point tracking is identifying the interest points that have been found. This entails generating a descriptor for each detected interest point. A descriptor is a distinctive vector generated from one or more properties related to the interest point. The purpose of descriptors is to facilitate the matching process in the next step. An example of a very simple descriptor would be a single-element vector containing the luminosity at the precise location of the interest point. Another (this one highly distinctive) descriptor could simply be a 25-element vector constructed from the 25 pixels of a 5 by 5 grid centered on the interest point.

Different applications have different requirements for a descriptor. These are requirements for properties such as distinctiveness, reproducibility or invariance, and computational cost (both in terms of generating the descriptor and in terms of corresponding them afterwards). Unfortunately these properties are not innately orthogonal, e.g. a highly distinctive descriptor will tend to exhibit low invariance and/or high computational cost. Selecting a descriptor
therefore requires understanding the needs of the particular application, and choosing the descriptor whose balance of these properties best satisfy those needs. An application such as Bundler [SSS07], which expects image data from very different sources, may for instance allow higher computational cost in return for higher distinctiveness. The application studied here is an example of the contrary as will be seen in Section 3.2 when the descriptor employed in this study is discussed and justified.

**Distinctiveness** The distinctiveness of a descriptor refers to how well it differentiates between different interest points. The more distinctive a descriptor is, the less likely it is to confuse two different interest points for one another.

The first example descriptor, with only a single element (the luminosity at the precise interest point location), has a very low distinctiveness. Supposing image data with 256 possible degrees of luminosity per pixel means that this descriptor can at best only differentiate between 256 different interest points. In a typical image there are normally at least several hundred interest points, if not thousands [SSS07], so many interest points will necessarily get identical descriptors. Furthermore, we cannot expect the luminosity of a interest point to remain perfectly constant across images. There needs to be some leeway in the matching stage allowing some variation in luminosity, and hence some variation in the descriptor value. This, in a sense, "blurs" the descriptors, decreasing their distinctiveness and increasing the confusion.

The second example provided above, a 25-element vector comprised of the luminosity in a 5 by 5 grid of pixels centered on the interest point, has potentially high distinctiveness. The effective distinctiveness depends on the size of the 5 by 5 grid as compared to the size of the image and the level of detail in it. If the image has very high resolution then this descriptor would effectively degenerate into the single-element descriptor. If an appropriate relative size of the grid may be presumed, then this descriptor may be very distinct. The apparent relevance of relative size hints at the importance of another property of descriptors, namely
Reproducibility and invariance

The **reproducibility** of a descriptor is a measure of the constancy of its value when the data base for computing the descriptor is subjected to noise and other distortions external to the interest point itself.

Noise in this case may come from several sources:

- particles such as dust in the line of sight;
- the image capturing process (for instance due to a low quality CCD or insufficient lighting);
- compression artifacts; and
- aliasing caused by the discrete nature of digital images.

Noise of these kinds manifests itself as distinct, unpredictable alterations of the values of single pixels or small groups of pixels. Descriptors which are sensitive to the value of individual pixels, such as both of the two example descriptors introduced earlier, will therefore be sensitive to noise. In Section 3.2 it is shown how less naïve descriptors account for noise by considering larger groups of pixels, minimizing sensitivity to the effect that noise has on individual pixels.

**Invariance** — more specifically invariance to changes in illumination, viewpoint location, rotation, scale and perspective — refers to stability of a descriptor under certain transformations. Ideally the value of a descriptor is dependent only on the information related to the real world structure it describes, and independent of other states such as the position and orientation of the camera or illumination. In practice, however, this is not a trivial problem because an image, being just that, an *image* of the real world interest point, only contains a subset of information about the real world interest point. Furthermore, the transformation (illumination, viewpoint, etc.) will affect which subset of information is present in a given image. Different images will contain different
subsets of information, further exacerbating the creation of an invariant descriptor. Creating an invariant descriptor therefore involves deciding on a process and information base that yield the same value even when some of the expected information is missing.

The transformation will also transform the actual information. To illustrate, variations in illumination may effect a global increase or decrease in the values of the pixels, or a change in the focal point can enlarge an area of the image. Both cases may include the same information; it just happens to have been transformed. By accounting for the transformation it is possible to arrive at the same descriptor value regardless of the variation in illumination or focal point.

**Computational cost** As relevant as ever is the computational cost. There are actually two distinct computational costs associated with a descriptor. The first is the cost of *generating* the descriptor, and the second is the cost of the subsequent process of *comparing* pairs of descriptors. Both these costs can be further divided into the cost of loading and storing the necessary data from and to memory, and the cost of the actual computations performed on the data. These costs, however, are not fixed for any given descriptor, but depend on the environment — that is, the software and hardware — and the actual implementation. For one combination of descriptor and environment the cost of actual computation may be insignificant compared to the cost of memory accesses, while in another combination the memory may be fast enough or the computations extensive enough to render the cost of memory accesses insignificant.

The first example descriptor, the single-element one, is an example of a descriptor with very low — minimal even — cost of both generation and comparison. Generating the descriptor requires loading one scalar from memory, and performing no computations. Comparing it requires, once again, loading one scalar for each descriptor. A common approach to comparing two descriptors is to simply calculate their Euclidean distance [Low99]. For a single-element descriptor, this amounts to performing a single subtraction operation on its value.
and the scalar value of the other descriptor.

Generating the descriptor in the second example requires loading 25 values from memory. Like the first example, generating the descriptor does not require any actual computations to be performed. The cost of generating this descriptor then is dependent only on the cost of loading and storing the relevant data from and to memory, the total cost of which will be 25 times that of the single-element descriptor. If the same approach to comparison is taken — based on Euclidean distance — then the cost of comparing two descriptors of this kind will be loading the 25 elements from each descriptor, performing 25 subtractions followed by 25 multiplications and 24 additions, and finally taking the square root. Hence the 25-element descriptor is significantly more costly than the single-element descriptor. Whether this extra cost can be justified depends on the particular application.

Other descriptors such as SIFT [Low99] and SURF [BTG06] are significantly more complicated and require extensive computations to generate the descriptor.

### 3.1.3 Matching

The goal of interest point tracking is to determine the relative displacement of the perspective projection of a set of 3D points in two images. Interest point tracking is but one method to solve what is known as the correspondence problem. The correspondence problem concerns mapping parts of one image to another image. Another method of solving the correspondence problem is by directly comparing the pixel data in parts of the two images to determine if those parts are similar [BM91]. This type of matching is useful in subsequent processing for generating a denser point cloud.

Matching is performed by comparing the two sets of interest points from the two images. Pairs of interest points which have similar descriptors are deemed a match. Ideally all interest points will be found in both images, but in practice a lot of interest points may not find their match because:

- they have moved out of the frame;
• the interest point detector failed to detect the interest point; or

• transformations such as noise or change in viewpoint or illumination cause
  the descriptor to be too different.

Sometimes a interest point may not be present in one frame, just to reappear
in the next.

**Time complexity**  Another aspect of computational cost is the time complexity
of the matching process of a set of descriptors. In the worst case scenario
all descriptors need to be compared to each other resulting in a polynomial
complexity \( O(n^2) \). The nature of some applications and some descriptors make
it possible to reduce this complexity. Keeping with the two original example
descriptors, an illustration of how the matching complexity for each of them
might be reduced will now be provided.

The time complexity of the matching process for the first example descriptor
could be reduced by binning the values. Upon writing the newly generated
descriptors back to memory, they could be stored in bins based on their values.
A fixed number of bins could be used, or a dynamic number of bins based
on the number of interest points. A fixed number of bins would still mean
a theoretical complexity of \( O(n^2) \). On the other hand, using \( \sqrt{n} \) bins would
reduce the complexity to \( O(n^{3/2}) \): there would be \( \sqrt{n} \) bins each containing on
average \( \sqrt{n} \) descriptors which would need to be compared to the remaining
\( \sqrt{n} - 1 \) descriptors in the bin.

A similar binning process could be applied to the second example. The
higher dimensionality of the second type of descriptor provides more options
for the binning parameter. Binning based on a single element — such as the
element representing the center pixel — out of the 25 elements is a possibility.
This is not a good approach though, because a single element being different in
two descriptors does not imply that the descriptors as a whole are significantly
different. Thus this approach may very well put nearly identical descriptors in
different bins. A more reliable approach would be binning based on the length
of the descriptor since the length is less sensitive to differences in individual elements.

As was mentioned, the nature of the application may also allow the complexity of the matching process to be reduced. The particular application in this study — retrieving structure from video data — is an example of such an application. Successive images from a video source will tend to be similar, exhibiting only small relative transformations. This implies that a interest point will not move far from one image to the next. This can be leveraged during the matching process by only comparing the descriptor of a given interest point to descriptors of interest points which are in close proximity. This is discussed further in the next section, where the resulting implementation is discussed.

3.2 Resulting implementation of interest point tracking

The resulting implementation for extracting three-dimensional data from video is divided into two stages. The first stage is interest point tracking based on the theory from this chapter. The second stage is reconstruction based on the tracked features from this chapter; Chapter 4 deals with that stage.

In this study, interest point tracking is implemented as a series of steps much like a pipeline. Information flows one-way from the raw input image data, through various intermediate forms, to a list of interest points that are tracked. In other words, there is no feedback from subsequent steps to earlier steps; each step depends only on information from previous steps. This particular approach was selected in an effort to keep the implementation minimal in terms of complexity.

The implementation utilizes the FFmpeg library [Bel10] for reading video files. The remainder of the pipeline is implemented in CUDA. Interest point detectors and interest point descriptors are based on SURF, but with some modifications which will become apparent. Once the raw input has been uploaded to the GPU (referred to as the device from now on), then all data and calculations are constrained to the device in order to avoid the unnecessary over-
head of data transfers. Minimizing uploading and downloading from the device is recommended in the Best Practices Guide, even to the extent of performing serial computations on the device despite serial computations being an inherently inefficient use of the device [NVI10b]. The output of this interest point tracking stage is a table of interest points with information about their movement across images. This table remains on the device and is made available to the second stage, reconstruction. Following is a discussion of each step in the pipeline.

3.2.1 Input

First the input has to be loaded and made available to the subsequent steps. The FFmpeg library is used for loading raw video data from disk into main memory. The FFmpeg library was selected because it can handle a large variety of input formats. After a frame has been loaded into the main memory by FFmpeg, it is then uploaded to the device to a layer. A layer is simply a uniform, rectangular block of memory. It is a generalization of a normal image intended to reduce coupling between the steps in the pipeline. It proved useful during development because it allowed flexibility and simplified experimentation — allowing any step to easily utilize any data, i.e. layer, that was available. As a layer stored in device memory, the frame is ready to be read (and potentially written to) by other steps.

3.2.2 Format conversion

Floating point values are used as much as possible throughout this implementation because they are easier and more intuitive to work with than integer values. The format of the data from the input step however is four 8-bit integer components. There are three possible approaches to deal with this:

Approach 1: Convert to floating point in global memory, before uploading to the device.
Approach 2: Upload to device memory and then convert to floating point.

Approach 3: Upload to device memory and have the next operator in the pipeline read it directly as integer values and convert to floating point internally.

The previous section already implied that Approach 1 was not used. The justification is that Approach 1 has two downsides: Firstly it would be using the processor on the host to sequentially perform a task that is more efficiently done in parallel on the device; and secondly, a conversion from four 8-bit integer values to four 32-bit floating point values quadruples the size of the data. Transferring data between the host and the device is considered slow [NV10b], and a good rule of thumb is therefore to minimize the number and size of such transfers. It is therefore preferable to do the conversion after transferring the raw data to device memory.

The second and third approach both handle the conversion on the device. The question is whether it should be done as a separate step (Approach 2) or if it should be baked into whatever the next step is (Approach 3). The primary disadvantage of Approach 2 compared to Approach 3 is that it requires additional storage for storing the converted layer. A secondary and less significant disadvantage is that it requires a separate kernel invocation. The big advantage of Approach 2 — still compared to Approach 3 — is that it results in separation of concerns. Where Approach 3 would mean mixing data conversion and whatever other logic is required by the next operator, Approach 2 keeps these concerns separate. This is very helpful in an experimental setting since new operators then can be created without worrying about whether they need to handle conversion as well, saving unnecessary thought, development, code duplication and computations. Maintaining the converted data in a separate layer also makes it easier to reuse it in other stages of the pipeline. The advantages of Approach 2 far outweighs the disadvantages, so Approach 2 was selected.
3.2.3 Integral image generation

One aspect of SURF that justifies the adjective Speeded-Up in its name is the use of integral images for speeding up computations. SURF makes extensive use of the integral image of a scene as part of interest point detection, and again as part of interest point description [BTG06]. The integral images are essential for completing these tasks with high efficiency at various scales. The integral images have found a second utility in this modified version of SURF for detecting and suppressing unstable interest points. Integral images will now quickly be defined before we look at parallelizing their generation.

Integral images The integral image $I_{\Sigma}$ of a source image $I$ is an image where the value $I_{\Sigma}(\vec{x})$ of a pixel at a location $\vec{x} = (x, y)^T$ is the sum of the values of all source pixels above and/or left of pixel $\vec{x}$. More formally:

$$I_{\Sigma}(\vec{x}) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i, j)$$

Integral images are useful for getting the sum of pixels in a rectangular area of the source image in constant time. Only four lookups and three additions or subtractions are required regardless of the size of the area.

Parallelization The generation of integral images does not map naturally to a parallel process because the value of each output pixel is dependent on either the preceding output pixel, or on all preceding input pixels. It is none the less possible to parallelize the generation, allowing integral images to be calculated on the device. The first thing to recognize is that it can be divided into two separate steps. The first generating the vertically integrated image $I_{\Sigma}^{vert}(\vec{x})$ from the source image $I$, and the second generating the final integrated image $I_{\Sigma}$ by horizontally integrating the vertically integrated image $I_{\Sigma}^{vert}(\vec{x})$. Formally:
\[
I_{\Sigma}^{\text{vert}}(\bar{x}) = \sum_{j=0}^{j \leq y} I(x, j)
\]

\[
I_{\Sigma}(\bar{x}) = \sum_{i=0}^{i \leq x} I_{\Sigma}^{\text{vert}}(i, y)
\]

The process of horizontally integrating an image is identical to vertically integrating an image if the input and the output is transposed. The same algorithm can therefore be used for both steps. An iterative algorithm which gradually integrates larger parts of the image is used in order to parallelize the process. If \( v_i^p(x) \) is the value of the \( x \)-th pixel in part \( p \) after the \( i \)-th iteration, then it looks like this:

1. Start with a stride \( s \) of 1 pixel and a suitable part size \( w \), say 8 pixels.
2. For each part \( p \):
   (a) For each pixel \( x' \) in the part:
      1. Load the value \( v_{i-1}^p(x') \) from the previous iteration \( i - 1 \) into \( v_i^p(x') \). If this is the first iteration, then the value comes from the input image \( I \).
      2. For each pixel \( x = sk - 1, k \in 1, 2, 3, ... \) to the left of \( x' \), increment the value \( v_i^p(x') \) by the value \( v_{i-1}^p(x) \). Formally,
         \[
         v_i^p(x') = v_{i-1}^p(x') + \sum_{x = sk - 1, x < x'} v_{i-1}^p(x), \quad k \in 1, 2, 3, ...
         \]

The pixels \( x \) selected by Equation 1 are the last (right-most) pixels from each part in the previous iteration. The result is that \( v_i^p(x') \) becomes the integral of pixels within the current part \( p \).
3. Multiply stride and part size with the initial part size (8 pixels). This will always be $s = (w_0)^i$ and $w = (w_0)^{i+1}$ where $w_0$ is the initial part size.

4. Increment the iteration number $i$ and go to Step 2 unless the stride size is larger than the width of the input image.

The output of this step is a layer with the same dimensions as the input image. Each pixel $\vec{x}$ in this layer is a single floating point value, namely the sum $I_\Sigma(\vec{x})$ of all pixels above and to the left of $\vec{x}$.

### 3.2.4 Interest point detection

Interest point detection can begin once the integral image has been calculated. SURF first generates a response map at various scales by using the integral image to apply a filter of increasing size [BTG06].

**The filter** The filter is based on an approximation of the Hessian matrix. The actual Hessian matrix is defined as

$$
\mathcal{H}(\vec{x}, \sigma) = \begin{bmatrix}
L_{xx}(\vec{x}, \sigma) & L_{xy}(\vec{x}, \sigma) \\
L_{yx}(\vec{x}, \sigma) & L_{yy}(\vec{x}, \sigma)
\end{bmatrix}
$$

where $L_{xx}(\vec{x}, \sigma)$ is the convolution of the Gaussian second order derivative $\frac{\partial^2}{\partial x^2} g(\sigma)$ with the image $I$ in point $\vec{x}$, and similarly for $L_{xy}(\vec{x}, \sigma)$ and $L_{yy}(\vec{x}, \sigma)$ [BTG06]. This matrix is approximated as $\mathcal{H}_{\text{approx}}(\vec{x}, \sigma)$ by convolving with box filters instead of the actual Gaussian filters. The response of the filter is the determinant of the matrix, i.e. $\det(\mathcal{H}_{\text{approx}})$. Box filters are used instead because box filters of any size can be applied in constant time by using the pre-calculated integral image.

Having calculated the entries in $\mathcal{H}_{\text{approx}}$ as $D_{xx}$, $D_{yy}$, and $D_{xy}$, the determinant of $\mathcal{H}_{\text{approx}}$ can now be calculated as $\det(\mathcal{H}_{\text{approx}}) = D_{xx}D_{yy} - (wD_{xy})^2$. The weight $w = 0.9$ should in theory depend on the scale $\sigma$, but is kept constant in practice “as it did not have a significant impact on the results” [BTG06].
Applying different size filters results in a stack of layers containing filter response maps, with each map representing the response of a filter with a certain size.

SURF creates sets of response maps grouped in octaves of different resolutions. In this implementation things are kept simple by only creating a single set of response maps. This reduces the complexity of interest point localization in the next step.

**Interest point localization** Interest points need to have an exact location in the image in order to for us to use them in our calculations. Extracting these locations from the response maps is done by finding all local maxima.

SURF finds local maxima in scale space; that is, in three dimensions, using non-maximum suppression in a $3 \times 3 \times 3$ neighborhood around each pixel. In this modified version of SURF only a $3 \times 3$ non-maximum suppression is applied, so local maxima are found only within each map. The neighboring scales are disregarded. This is done for three reasons:

- it reduces the memory footprint since this way there is no need to keep several response maps around at the same time;
- it results in more interest points; and
- it allows points to be found in the bottom scale too.

If there are $n$ scales then SURF will only find interest points in scales $S^i$, $i \in [2, n - 1]$ because the $3 \times 3 \times 3$ non-maximum suppression requires a scale above ($S^{i+1}$) and below ($S^{i-1}$). Limiting the suppression to individual scales means that we can find interest points in the bottom scale $S^1$ as well. This is important because this is where the smallest and hence most precisely positioned interest points are located. Limiting suppression to individual scales also renders the relationship between scales unimportant, simplifying the algorithm.

The output of this step is, for each scale $S^i$, a map where each non-zero pixel represents a detected interest point.
Interest point discardation  Input images often contain objects with flat shading and no stable interest points. These areas still tend to generate a large number of interest points due to noise in the input and from compression. These interest points need to be discarded since they do not represent actual interest points in the scene, but random artifacts in the image. This is done by assigning each interest point a value $p_{xy}$ based on how much it stands out in the input image, and then thresholding based on that value. The value $p_{xy}$ for an interest point at a location $(x, y)$ is calculated as the difference between its corresponding input pixel $I(x, y)$ and the average of its surrounding pixels:

$$p_{xy} = I(x, y) - \frac{1}{(2(s+1))^2} \sum_{i\in[-s,s]} \sum_{j\in[-s,s]} I(x+i, y+j), \quad s \in \mathbb{N}$$

Interest points are then discarded if $p_{xy}$ falls below a certain threshold. Note that the integral image is reused here for calculating the average value of the surrounding pixels efficiently.

The output of this step is a set of layers, one for each scale $S^i$, with the same dimensions as the input image. Any pixels where an interest point was detected have a non-zero value; the rest, i.e. the majority, are zero.

3.2.5 Interest point description

Matching interest points between different images, or in this case, different frames, requires them to be comparable in some sense. SURF is intended for matching pairs of images which may have very different viewpoints. The downside of this is that all the interest points from the second image $I_2$ need to be considered when searching for an interest point from the first image $I_1$ [BTG06]. This means that SURF requires highly distinctive descriptors to reduce false matches. The process of generating these distinctive descriptors involves a large number of memory accesses [BTG06].
**Special considerations**  The needs of this application are different from the general case SURF descriptors were developed for. First, we are dealing with video data as opposed to distinct images with large baselines. Second, we are implementing in CUDA, where memory access is considered costly. Note that in this case we are talking about accessing global device memory from the device. This is faster than accessing it from the host, but it is still considered costly, and minimizing it is one of the high-priority recommendations \[NVI10b\]. The latter point implies that the large number of memory accesses should be reduced, and the former point enables us to do just that.

Since the input data is a video, the movement of the interest points will be somewhat predictable. Specifically, they can be expected to not move very far from one image to the next. This is leveraged in this implementation by localizing the search for matching interest points to a small area surrounding the expected location. This localization has several benefits.

- It reduces the number of interest points that need to be compared.
- The reduced search space relaxes the demands on descriptor distinctiveness.
- Localized search space lends itself better to implementation in the independent block centric CUDA architecture.

The relaxed demands on descriptor distinctiveness allows the descriptors for this application to be simplified compared to SURF descriptors, reducing computational cost both in terms of actual computations and of memory access. The implications of the other two benefits will be discussed in the next section, Section 3.2.6.

The descriptor implemented in this application is inspired by the SURF descriptor. SURF generates descriptors from a large number of sums of pixels distributed in a grid surrounding the interest point \[BTG06\]. The descriptor in this implementation is also based on sums from a grid centered on the interest
point, but the number of sums is reduced to 9 in a 3 by 3 grid. As with the SURF descriptor, the integral layer is utilized for calculating these sums.

Another aspect of the SURF descriptors which is simplified in this implementation is rotation invariance. SURF is intended to be highly invariant, including to rotation. The rotation from frame to frame in video is under normal conditions very small or non-existent, whereas rotation invariance as implemented in SURF is intended to deal with large rotations [BTG06]. The extra measures in SURF to achieve rotation invariance is therefore not employed in this implementation in order to avoid unnecessary computations.

**Generating the descriptors** Now to the actual construction of the descriptors. The output from the last step was a set of full size layers, but only a fraction of the pixels are interest points. It would be straightforward to just feed the entire layers into a CUDA kernel which then runs a thread per pixel, generating a descriptor if that pixel is an interest point. This would however waste a lot of computational resources since most pixels are not interest points, and hence most threads would not be doing any useful work.

Therefore — in order to maximize occupancy (another recommendation from NVIDIA [NV10b]) — a lightweight collector kernel is executed first. This collector kernel simply collects all interest points in a dense list. Using a counter $F_b$ per block $b$, the collector first determines the number $F_b$ of interest points in each block $b$. Then each block atomically increments a global counter $F_g$ by $F_b$ and retains the previous value of $F_g$ as $idx_b$. This, in effect, reserves $F_b$ slots in the dense list starting at $idx_b$. Each block then finally writes an vector for each interest point to the list, starting at their slot $idx_b$. The vectors that are written contain the position and scale that the interest points came from. The information about position and scale was implicit in the previous layers, but when constructing a separate, dense list of the interest points then this information needs to be stored explicitly.

Now that we have a dense list of interest points we can apply the more
computationally expensive kernel without wasting threads. The descriptors in this implementation are vectors $\vec{d}_{xy}$ with 9 elements. Each element $d_{ij}^{xy}, i, j \in -1, 0, 1$ is the sum of pixels in a square. The size $s$ of each square is dependent on the scale $S^2$ at which the interest point was detected such that $s = 3z$. If the interest point is located at position $(x, y)$ and scale $S^2$ then

$$d_{ij}^{xy} = \sum_{k \in [1, s]} \sum_{l \in [1, s]} I \left( (x - \frac{s}{2}) + is + k, (y - \frac{s}{2}) + js + l \right)$$

$$= I_\Sigma \left( (x - \frac{s}{2}) + (i + 1)s, (y - \frac{s}{2}) + (j + 1)s \right)$$

$$+ I_\Sigma \left( (x - \frac{s}{2}) + is, (y - \frac{s}{2}) + js \right)$$

$$- I_\Sigma \left( (x - \frac{s}{2}) + (i + 1)s, (y - \frac{s}{2}) + js \right)$$

$$- I_\Sigma \left( (x - \frac{s}{2}) + is, (y - \frac{s}{2}) + (j + 1)s \right).$$

Each of the 9 sums can thus be calculated by accessing the integral image 4 times, for a total of $9 \times 4 = 36$ memory accesses and $36 \times \text{sizeof(float)} = 144$ bytes transferred. As it happens though, the squares are adjacent, which means that several of the values may be reused, decreasing the total memory access count to 16 and 64 bytes.

The output from this step is a layer with the same dimensions as the input images. All the interest points which were distributed over the different scales $S^i$ have now been collected in this one layer. The descriptors are stored in small lists $L_{bx,by}^{\text{new}}$, where each list is associated with a small area or block of the image such that

$$L_{bx,by}^{\text{new}} = \{ d_{xy} : bx \leq x/s < bx + 1 \land by \leq y/s < by + 1 \}, \ s \in \mathbb{N}$$

where $s$ is the desired size — width and height — in pixels of the blocks. This grouping in lists based on location is done with an eye to speeding up the next step, matching.
3.2.6 Interest point matching

Interest point tracking involves figuring out which interest points in a new image correspond to which existing interest points — interest points already known from previous images. The naive implementation of this matching process would simply compare all newly discovered interest points with all known interest points. Unfortunately this becomes impractical when dealing with tens of thousands of interest points in every frame, and, say, 30 frames every second. The central idea in this application of interest point tracking is therefore to leverage the special nature of video to speed up this process. The previous section already showed how video allows the descriptors to be more lightweight. In this section we will see how to speed up the matching process further by localizing the search for matching interest points in order to reduce the number of comparisons.

The idea  It would be straightforward to have two lists, one with all $k$ known interest points $f^{\text{known}}$ and one with all $n$ newly discovered interest points $f^{\text{new}}$, and simply comparing all of them. That would be $kn$ comparisons per frame, which would be a $O(n^2)$ time complexity with $n$ proportional to the dimensions of the video such that $n \propto \text{dim}^2 \propto \text{width} \times \text{height}$. This leads to a total time complexity of $O(\text{dim}^4)$. We can do better. The movement of features in a video is inherently fairly predictable. By predicting where a interest point will be in the next frame, we can significantly reduce the complexity of the search by only comparing interest points in close proximity to the expected location. There are several ways, or levels, of predictions that can be made regarding the next location of the interest point:

1. The interest point is likely to be found close to its previous location:

   There is only a small time interval between subsequent frames from a video so there will not be a large change of viewpoint from one frame to the next. This causes 3D points to be projected to nearly the same 2D location in subsequent images.
2. The movement, i.e. velocity and acceleration, of the interest point can be predicted based on its earlier movement:

Classical mechanics would suggest that the motion of the camera — or, relatively, the motion of the 3D points and their 2D projections — is predictable [Fit06]. Inertia caused by the mass of the camera and the object, e.g. hand and arm, and intention of the operator will normally causes the camera to follow a somewhat smooth and predictable path. This can be taken advantage of in two ways:

(a) The 2D projections of a point identified as a interest point can be tracked across several frames. From this a model — for instance based on velocity and acceleration — describing its movement in the 2D plane can constructed.

(b) The movements (extrinsic parameters) of the camera across several frames can be tracked, and from this a model describing the projection can be constructed. This could then be used to estimate the next projection. This estimation could be useful in the next level of predictions.

3. If the projection of the next frame is known or can be estimated, and the 3D position of the interest point has already been estimated, then the expected next position can be estimated.

The implementation of the first and second levels of prediction is discussed next. The third level of prediction is not employed in this implementation, but is interesting for future implementations as a way to speed the process up.

The trick In accordance with first level prediction as described previously, the goal is to limit comparisons such that we only compare known interest points and newly detected interest points which are in close proximity to each other. This is accomplished by partitioning the interest points into disjoint blocks based on their location in the 2D image. All interest points within a block are
then compared. If there are $k_b$ known interest points and $n_b$ newly detected interest points in a block $b$, then this equates to $k_b n_b$ comparisons per block. If we assume that interest points are distributed evenly amongst blocks, then there are $b = k/b = n/n_b$ blocks. The total number of comparisons per frame then becomes $k_b n_b b = (k/b)(n/b)b = kn/b$, a fraction of the $kn$ comparisons required by the naïve implementation.

A straightforward implementation of this would load each pair of known interest point and newly detected interest point and compare them. The total number of interest points loaded from memory would then be $2k_b n_b$, two for each comparison. Now, as it happens, threads belonging to the same CUDA block can share data [NV110c]. This means that a value can be loaded only once from memory, and then accessed by all threads in the block. It is important to structure the algorithm in a way that takes advantage of this (another recommendation[NVI10b]) because it significantly reduces the costly memory accesses. This is accomplished in this implementation by implicitly assigning each CUDA block to carry out all comparisons in a specific block $b$ of the image. Each thread in the block is responsible for one of the newly discovered interest points, so each thread loads one newly detected interest point for a total of $n_b$ loads. The threads then, in cooperation, compare their respective new interest points to one known interest point at a time. A known interest point is loaded once and all the threads in the block compare their respective interest point to it, before finally the next known interest point is loaded and the process repeated. Each known interest point is loaded only once since the threads are cooperating by sharing them. This makes for $k_b$ loads. The total number of interest points loaded from memory then becomes $n_b + k_b$ instead of $2k_b n_b$ — linear time complexity $O(n)$ instead of polynomial $O(n^2)$.

**The implementation** First of all, since we want to track known interest points over time, there needs to be some persistent information about them. This is implemented as a large persistent table with one entry per known interest
point:

• the last and second last known 2D position of the interest point;
• the estimated velocity; and
• the most recent descriptor.

There needs to be a way to read this table based on the location of each entry so that the comparisons can be performed in blocks of threads based on the location of the interest points. This is accomplished by maintaining a lookup layer which indexes into the known interest point table. The indices in this lookup layer are grouped small lists $L^{\text{known}}_{bx,by}$ in accordance with the location of the interest point they represent, exactly like the lists $L^{\text{new}}_{bx,by}$ of descriptors in the layer from the previous step. This way a list of all known interest points in an area of the image can be retrieved without iterating through the entire known interest point table. The scene is now set for efficiently comparing known and newly detected interest points. The matching process can now be carried out by invoking one block of threads per list tuple $(L^{\text{known}}_{bx,by}, L^{\text{new}}_{bx,by})$. As was described previously, each thread first loads one descriptor from $L^{\text{new}}_{bx,by}$. Known interest points are then, one at a time, loaded from $L^{\text{known}}_{bx,by}$ and compared in the individual threads.

Newly detected interest points are compared to known interest points by calculating the Euclidean distance of their descriptors. If the distance is below a certain threshold, then it is tentatively considered a match; the final, definite, match is the descriptors with the shortest distance. Once each thread has determined which known interest point matches, if any, they update the known interest point table with the new positions, velocities and descriptors. Newly detected descriptors for which there where no matching known interest point are added to the known interest point table as brand new interest points.

These were the basics of the implementation. Next a few improvements will be discussed.
Improvements

There is a downside to the approach as explained above; there is no flow of information between blocks. If an interest point moves to a different block then it will not be recognized as a known interest point. It will be interpreted as a brand new interest point. This means that interest points can only be tracked for a small period of time — a small set of images — even though they might in reality exist for a long time. This has two implications:

- Each interest point will in effect be "split" into many separate interest points since the image space is partitioned into a large number of blocks. The result is each interest points becomes many separate points in the reconstruction.

- Only a small set of images will be available for reconstructing the 3D location of the interest point. This impacts the precision of the reconstruction in two ways:
  
  - It becomes more sensitive to noise, for instance due to the discrete nature of digital images.
  - The interest point will only appear to be observed for a short period of time, during which the camera will not move very far. The result is that the total baseline of the images in which the interest point was observed will be small. The negative impact of small baselines will be explained in Section 3.2.

This implementation deals with this problem by allowing interest points to move to other blocks under certain circumstances. An estimate of the velocity of each interest point is tracked continuously. This velocity is used to more accurately predict where a interest point will move to in the next frame. The index of a certain interest point is then copied into appropriate list \( L_{bx',by}' \) from \( L_{bx,by} \) whenever the prediction suggests that the interest point will move from block \((bx, by)\) to block \((bx', by')\). This allows the second block, \((bx', by')\), to also check for the interest point. This way the interest point can be tracked across blocks indefinitely, reducing the frequency of interest point "splitting".
A second downside is that the velocity can only be estimated after the interest point has been observed twice within the same block. This is a problem because the size of the search space is fixed, independent of the resolution of the video, so interest points that move too fast will never be observed twice in the same block. Hence they will not be tracked. To be precise, this applies to interest points with high velocity $\vec{v}$ (relative to resolution) compared to the block size $\vec{s}_b$ (also relative to the resolution). If $\vec{v} = \text{displacement}_{\text{pixels}}/\text{resolution}_{\text{pixels}}$ and $\vec{s}_b = \text{blocksize}_{\text{pixels}}/\text{resolution}_{\text{pixels}}$, then interest points where $\vec{v} > \vec{s}_b$ will not be tracked. In an ideal implementation the interest points would be tracked without concern to blocksize. This is not an ideal implementation.

Another improvement which is made is to factor in the expected position when comparing interest points. The distance between the descriptors of a known interest point and a new interest point is modulated by the distance to the expected position of the known interest point. The effect of this is that a new interest point which is located closer to the expected position is more likely to be selected as the correct match, even if there are other new interest points whose descriptors are slightly closer to the known interest point descriptor. The motivation behind this is that occasionally there will be several interest points with similar descriptors within a block; accounting for the expected position increases the likelihood that the new interest points are matched to their appropriate known features.

3.2.7 End of the interest point tracking stage

A table of known interest points and their movements from frame to frame has now been constructed. The information in this table will continuously update as each frame is processed.

Only a subset of the known interest points will be observed in a given frame. This is due to interest points going out of sight and due to instabilities in the detection, description and matching of interest points. For this reason, a supplementary list is created with indices to all interest points that were
observed in the last frame. This makes it possible to access all the interest points which were observed in the last frame without iterating through the entire list of known interest points.

By tracking the changes in the known interest point table it will now be possible to deduce the movement of the camera, and in turn the 3D location of each known interest point. This process — known as reconstruction — is the second and final stage of this implementation, and is the topic of the next chapter.
4 3D Reconstruction

Reconstructing 3D structure from 2D image data requires understanding how the 2D image data is created from the 3D structure in the first place. Therefore the necessary and most important basic theory of 3D projections, perspective projection in particular, is introduced before developing the theory of reconstruction. Then the application of the theory in a basic implementation is detailed.

4.1 Theory of perspective projections

Any point in the real world, or more generally any point in any 3D space $\mathbb{R}^3$, can be represented by a 3-element vector $\vec{X}$. This vector is the coordinate of the point relative to some coordinate system. This point, or a set of points, can be transformed by a $3 \times 3$ matrix $A$ as $\vec{X} \mapsto A\vec{X}$. This transformation can be interpreted as rotation, scaling and skewing of the point or set of points relative to origin of the coordinate system. Alternatively it can be interpreted as a change of coordinate system. The difference in interpretation lies in what is changing: the actual position of the points or just their reference frame [Str88].

Notice that $A$ as defined above does not and cannot include translation, as in $\vec{X} \mapsto A\vec{X} + \vec{b}$, because translation is not a linear transformation. As it happens though, a shear in a vector space $\mathbb{R}^n$ looks like a translation in the subspace $\mathbb{R}^{n-1}$. Thus it is possible to represent 3D rotation, scaling and translation all as a single 4 by 4 matrix. This is accomplished by upgrading from Cartesian coordinates in Euclidean space $\mathbb{R}^3$ to homogeneous coordinates in projective space $\mathbb{P}^3$, specifically the real projective space $\mathbb{RP}^3$ [Ma,+03].

4.1.1 Homogeneous coordinates and the projective space

The projective space $\mathbb{P}^1$ can be defined as the set of all lines in $\mathbb{R}^4$ that pass through the origin [HZ04]. In a similar vein one can also define the projective space $\mathbb{P}^2$ as the set of all lines in $\mathbb{R}^3$ that pass through the origin. To aid un-
nderstanding and for consistency, these are also the spaces that will be used here to introduce homogeneous coordinates and projective spaces, but rest assured that the theory extends naturally to $\mathbb{P}^3$ and $\mathbb{R}^4$ [HZ04].

The projective space $\mathbb{P}^2$ has already been defined as the set of all lines in $\mathbb{R}^3$ that pass through the origin. Each such line actually represents a single point in the projective space. This means that any point on such a line in $\mathbb{R}^3$ is equivalent to any other point on the same line. This property is reflected by the coordinate system used for projective spaces: *homogeneous coordinates*. Define the homogeneous coordinates in $\mathbb{R}^3$ of a point in $\mathbb{P}^2$ to be the triple

$$X^h = (x^h, y^h, w^h)$$

where $(x^h, y^h, w^h) \neq \vec{0}$. The triple $X^h$ then identifies a point

$$\vec{X}^e = (x, y) = (x^h/w^h, y^h/w^h)$$

in Euclidean space $\mathbb{R}^2$. This scaling by the third component $w^h$ suggests that a point $\vec{X}^e$ may be identified by infinitely many triples $X^h$. This is indeed the case; since $\mathbb{P}^2$ has been defined as all lines in $\mathbb{R}^3$ that pass through the origin, it follows that multiplying a homogeneous coordinate $X^h$ by any scalar $\lambda \neq 0$ does not change which point in $\mathbb{P}^2$ that it represents because the new coordinate lies on the same line. That is,

$$(x^h, y^h, w^h) \equiv (\lambda x^h, \lambda y^h, \lambda w^h)$$

and specifically

$$(x^h, y^h, w^h) \equiv (x^h/w^h, y^h/w^h, 1) = (x, 1)$$

because $(\lambda x^h, \lambda y^h, \lambda w^h)$ necessarily lies on the line passing through the origin and $(x^h, y^h, w^h)$. 

39
Any point in the Euclidean space, be it \( \mathbb{R}^2 \) or \( \mathbb{R}^3 \), has a single corresponding point in projective space. The converse however is not true. The projective space includes a set of points, points at infinity, not included in Euclidean geometry [HZ04]. The existence of these points are not of concern for the application of projective geometry employed here, but they are mentioned for completeness. These points are as their name suggests located at infinity, where they form a single line, the line at infinity. Any direction through \((x^h, y^h, w^h)\) has an associated point at infinity \((x^h, y^h, 0)\). The set of all points \((x^h, y^h, 0)\) form the line at infinity, which can be thought of as encircling all other points \((x^h, y^h, w^h), w^h \neq 0\), i.e. \( \mathbb{R}^2 \).

### 4.1.2 Translation

The motivation for introducing homogeneous coordinates and projective spaces was to make it possible to represent non-linear translation transformations in euclidean space \( \mathbb{R}^3 \) as linear skew transformations in projective space \( \mathbb{R}^4 \) so that translations could be implemented as matrix operations. Homogeneous coordinates now allow the transformation \( \vec{X} \mapsto A\vec{X} + \vec{b} \) to be expressed as multiplication with a single matrix

\[
B = \begin{bmatrix}
A & \vec{b} \\
\vec{0} & 1
\end{bmatrix}
\]

as shown for any Euclidean space \( \mathbb{R}^n \) next. The homogeneous coordinates of a point \( \vec{X}_e = (x, y) \) in Euclidean space \( \mathbb{R}^2 \) is \( \vec{X}^h = (xw^h, yw^h, w^h) = (x^h, y^h, w^h) \) for any \( w^h \). Select \( w^h = 1 \) such that \( \vec{X}^h = (x, y, 1) \) for simplicity. In general,
for any $\mathbb{R}^n$, $X^h = (X^e, 1)$. The transformation then becomes

$$B X^h = \begin{bmatrix} A & \vec{b} \\ \vec{0} & 1 \end{bmatrix} \begin{bmatrix} X^e \\ 1 \end{bmatrix} = \begin{bmatrix} A X^e + \vec{b} \\ 1 \end{bmatrix}$$

which are the homogeneous coordinates of $AX^e + \vec{b}$. Notice that the bottom row of $B$ being $\begin{bmatrix} 0 & 1 \end{bmatrix}$ and selecting $w^h = 1$ ensures that $w^h$ always comes out equal to 1, thus keeping things simple with $X^h = (X^e, 1)$. The vector $\vec{b}$ in $B$ is actually a skewing transformation in $\mathbb{R}^3$, which manifests itself as a translation in the Euclidean plane at $w^h = 1$.

### 4.1.3 Perspective projection

The behavior of real world cameras may be approximated by the pinhole camera model [HZ04]. The pinhole camera model describes the projection of a 3D point onto the 2D image plane of an ideal pinhole camera. An ideal pinhole camera is a camera with an infinitely small pin hole or aperture through which the light travels before it strikes the image plane at a distance $f$ from the pinhole. The value $f$ is the focal length of the camera. The pinhole camera model does not include a lens; meaning that the rays travel straight through the pinhole.

Assume for the remainder of this section that everything is scaled relative to the focal length such that $f = 1$. The resulting image point, represented with homogeneous coordinates $\vec{x}^h_p$, located on the image plane is related to the 3D point $\vec{X}^h$ by a $3 \times 4$ projection matrix $P$:

$$\vec{x}^h_p = P \vec{X}^h$$

This matrix transforms the point $\vec{X}^h$ to a point $\vec{x}^h_p$ in projective space $\mathbb{P}^2$ with origin located at the focal point of the camera. This projective space can
be visualized in $\mathbb{R}^3$ as the rays through the origin. The resulting image is the intersection of these rays and the image plane located at $w^h_P = f = 1$. Performing a perspective division on the coordinates $\vec{x}^h_P$ — dividing them by $w^h_P$ — brings them into this plane. This one-to-one mapping of 3D lines to 2D points in the image plane aligns with the intuitive understanding that there is a line, i.e. an infinite number of points, in $\mathbb{R}^3$ which maps to any given point in the image plane. The perspective division produces the Cartesian coordinates $\vec{x}^c_P$ of the Euclidean image point much like how the homogeneous coordinates $\vec{X}^h$ of $\mathbb{P}^3$ divided by their fourth component $w^h$ coincide with the Cartesian coordinates $\vec{X}^c$ of the 3D Euclidean point.

The projection matrix $P$ only encodes the translation and rotation of the camera; these are the only parameters of an ideal pinhole camera (because we assumed $f = 1$). Translation and rotation are external to the camera and are therefore referred to as extrinsic parameters. Add a lens or unfix the image plane, and things get more interesting. Adding a lens or unfixing the image plane gives rise to intrinsic parameters, the existence of which complicate things, in particular the reconstruction process.

### 4.1.4 The camera matrix and intrinsic parameters

As was seen in Section 4.1.2, a general matrix $4 \times 4$ matrix $A$ may encode rotation, scale, skew, reflection and translation in $\mathbb{R}^3$. The definition of $P$ will now be expanded to account for the intrinsic parameters. The new projection matrix is $P = K[R|\vec{t}]$. The extrinsic parameters (think external movement) of a camera are encoded in the matrix $[R|\vec{t}]$; the intrinsic parameters (think image plane) — scale, skew and reflection — are encoded in the camera matrix $K$.

The intrinsic parameters are closely related to the location of the image plane relative to the focal point. The distance from the plane to the focal point defines the scale. Whether the plane is located between the focal point and scene, or behind the focal point, defines the reflection. Finally, the combination of location and rotation or orientation of the plane defines the skew.
Most real world cameras keep the focal point centered on the plane and produce images as though the plane is located between the focal point and scene. The only unknown intrinsic parameter is the focal length. Yet, since the theory allows for skew and reflections, we need to account for those parameters as well when attempting to reconstruct the scene. This is the topic of Section 4.2.2.

4.2 Theory of the reconstruction problem

Reconstruction is the process of recovering 3D structural data from 2D image data. It is necessary to have at least two images to generate a structure with any kind of certainty. The reason is that each image point in an image could come from any one of an infinite number of points along a line in 3D space. Extensive a posteriori information about the scene may help to narrow down the likely relative depth of some points in a single image but, triangulation using two or more images on the other hand does not require extensive a posteriori information, and is therefore the core focus of this study.

The complexity of reconstructing a scene depends on the available input data. The simplest case is when image point correspondences between two or more images are known along with the projection and camera matrices for each image. In this case a metric reconstruction can be calculated in a fairly straightforward manner. A metric reconstruction is a reconstruction which differ from the original structure only in scale. If the projection and camera matrices are not known, then these will need to be calculated first. Section 4.2.1 provides the needed theory for estimating initial projection matrices. Then Section 4.2.2 provides the needed theory for calibrating the cameras. As was seen in Section 4.1.4 the image plane of a theoretical camera is not necessarily centered on the focal point; calibrating the cameras is the process of determining their intrinsic parameters which define the image plane. A reconstruction based on uncalibrated cameras is a projective reconstruction. A projective reconstruction preserves straightness and intersections, but not angles or distances.
4.2.1 Determining relative projection matrices

A projection matrix transforms a set of known 3D points to the 2D image plane. The 3D points being known means that their positions are known relative to some frame of reference, some coordinate system. When these positions are not known, as is the case for the reconstruction problem, then there is no single, unique projection matrix. Any projection matrix will have an infinite number of valid sets of 3D points even though the 2D image of the projected points is constant.

However, for an arbitrary projection matrix $P_1$ for the image $\vec{x}_1$ of a general point set $\vec{X}$, there exists a unique corresponding projection matrix $P_m$ for each other image $\vec{x}_m$ such that

$$\vec{x}_1 = P_1 \vec{X}, \text{ and}$$
$$\vec{x}_m = P_m \vec{X}. \tag{2}$$

Conversely, the pair of images $\vec{x}_1$ and $\vec{x}_2$ of a general point set and their corresponding projection matrices $P_1$ and $P_2$ have precisely one solution point set $\vec{X}$. A general point set is a set of points which do not all lie within the same plane [HZ04]. More formally the set of n points $\{\vec{X}_n\}$, $\vec{X}_n \in \mathbb{R}^3$ is a general point set if and only if the matrix $\begin{bmatrix} \vec{X}_1 & \vec{X}_2 & \cdots & \vec{X}_n \end{bmatrix}$ has rank 3.

If we select an arbitrary projection matrix $P_1$, say

$$P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

for our first image, then it is possible to find a matrix $P_2$ which satisfies $\vec{x}_2 = P_2 \vec{X}$. The points $\vec{X}$ are currently unknown so, there is an infinite number of possible solutions for $P_2$. Finding a solution for $P_2$ amounts to calculating the fundamental matrix and then using it to construct a valid projection matrix $P_2$. 
The fundamental matrix  The fundamental matrix [HZ04] $F_{12}$ of two images $\vec{x}_1$ and $\vec{x}_2$ relates the two images through the equation

$$\vec{x}_2^T F_{12} \vec{x}_1 = 0. \quad (3)$$

If the two images are images of a general 3D point set then there is a unique fundamental matrix that relates them.

Expanding Equation 3 we get an equation

$$\vec{x}_2^T F_{12} \vec{x}_1 =
\begin{bmatrix}
x_2^j & y_2^j & w_2^j \\
x_2^j & y_2^j & w_2^j \\
f_{11} & f_{12} & f_{13} \\
f_{21} & f_{22} & f_{23} \\
f_{31} & f_{32} & f_{33}
\end{bmatrix}
\begin{bmatrix}
x_1^j \\
y_1^j \\
w_1^j
\end{bmatrix}
= x_1^j x_2^j f_{11} + y_1^j x_2^j f_{12} + w_1^j x_2^j f_{13} \\
+ x_1^j y_2^j f_{21} + y_1^j y_2^j f_{22} + w_1^j y_2^j f_{23} \\
+ x_1^j w_2^j f_{31} + y_1^j w_2^j f_{32} + w_1^j w_2^j f_{33} = 0 \quad (4)$$

in $F$ per point $\vec{X}^j$. At first glance this equation has 9 degrees of freedom, meaning we require at least 9 pairs of image points of the point set $\vec{X}$. The scale of the fundamental matrix is unimportant [HZ04], reducing it to 8 degrees of freedom. Additionally, $\det F = 0$, providing an extra constraint. This means that it is possible to generate $F$ from only 7 pairs of points. Given enough pairs of image points $\vec{x}_1^j$ and $\vec{x}_2^j$ we can construct a set of equations like Equation 4 which then can be solved for $F$. The set of equations may be written as homogeneous linear equation

$$A \vec{f} = 0$$
where

\[ A = \begin{bmatrix}
    x_1^1 & x_1^2 & x_1^1 & w_1^1 & y_1^1 & y_1^1 & w_1^1 & w_1^1 & w_1^1 \\
    x_2^1 & x_2^2 & x_1^2 & w_2^1 & y_1^2 & y_1^2 & w_2^1 & w_2^1 & w_2^1 \\
    x_1^1 & x_1^2 & x_2^2 & y_1^1 & y_1^2 & y_1^2 & w_2^1 & w_2^1 & w_2^1 \\
    x_2^1 & x_2^2 & x_2^1 & y_2^1 & y_2^2 & y_2^2 & w_1^1 & w_1^1 & w_1^1 \\
    \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
    x_j^1 & x_j^2 & x_j^1 & w_j^1 & y_j^1 & y_j^1 & w_j^1 & w_j^1 & w_j^1 \\
    x_j^1 & x_j^2 & x_j^1 & w_j^1 & y_j^2 & y_j^2 & w_j^1 & w_j^1 & w_j^1 \\
    x_j^1 & x_j^2 & x_j^1 & y_j^1 & y_j^2 & y_j^2 & w_j^1 & w_j^1 & w_j^1 \\
    x_j^1 & x_j^2 & x_j^1 & y_j^1 & y_j^2 & y_j^2 & w_j^1 & w_j^1 & w_j^1
\end{bmatrix} \]

and

\[ \vec{f} = \begin{bmatrix}
    f_{11} \\
    f_{21} \\
    f_{31} \\
    f_{12} \\
    f_{22} \\
    f_{32} \\
    f_{13} \\
    f_{23} \\
    f_{33}
\end{bmatrix} \]

which can be solved for \( \vec{f} \) and hence \( F \).

The fundamental matrix \( F_{12} \) encodes information about the relationship between the two projections \( P_1 \) and \( P_2 \). Having selected an arbitrary projection \( P_1 = [I|0] \) for the first image we can now use the fundamental matrix to construct a \( P_2 \), which will satisfy Equation 2. Hartley et al. demonstrated that given \( P_1 = [I|0] \), then \( P_2 \) can be calculated as

\[ P_2 = [SF_{12}|\vec{e}_2] \]

where \( S \) is any skew-symmetric matrix and \( \vec{e}_2 \) is the \textit{epipole} in the second image. Selecting \( S \) to be the matrix representation of the cross product with the epipole \( \vec{e}_2 \) we get an expression for \( P_2 \) in terms of the fundamental matrix and the epipole:

\[ P_2 = [[\vec{e}_2] \times F_{12}|\vec{e}_2] \]
**Epipoles** The epipole is the point in the image plane where the focal point of another projection would be projected. Given $F_{12}$ then the epipole $\vec{e}_1$ of the second projection, in the first image, is defined as $F_{12}\vec{e}_1 = \vec{0}$ [HZ04]. Oppositely, the epipole $\vec{e}_2$ in the second image is defined as $F_{12}^\top\vec{e}_2 = \vec{0}$.

A concept which is related to epipoles is epipolar lines. In short, the epipolar line $\vec{l}_2$ in the second image, corresponding to a point $\vec{x}_1^j$, from the first image, is the projection in the second image of the ray from the focal point of the first image through the point $\vec{x}_1^j$. Formally, the epipolar line corresponding to the point $\vec{x}_1^j$ is $\vec{l}_2^j = F_{12}\vec{x}_1^j$, and the epipolar line corresponding to the point $\vec{x}_2^j$ is $\vec{l}_1^j = F_{12}^\top\vec{x}_2^j$. Epipolar lines have their uses, they may for instance be used to speed up the search for matching interest points if the projections of both images are known, but they are not utilized directly in the approach in this study and so will not be detailed further.

**Characteristics of projection matrices generated from the fundamental matrix** Since any skew-symmetric matrix $S$ may be used to construct $P_2$, we see that there are an infinite number of possible solutions for $P_2$. Each valid solution for $P_2$ has a corresponding set of points $\vec{X}$ for which Equation 2 is valid; that is, each solution for $P_2$ implies a certain source structure. This ambiguity means that we are only able to estimate a structure up to a projective similarity of the original structure. This ambiguity is a result of not knowing the intrinsic parameters, $K_m$, of the two cameras. When we chose $P_1 = [I|0]$ we implicitly chose $K_1$ and $[R_1|\vec{t}_1]$ such that $K_1[R_1|\vec{t}_1] = I$. In other words, the actual intrinsic parameters $K_1$ and extrinsic parameters $[R_1|\vec{t}_1]$ are not unambiguously specified, only their relationship. When we construct $P_2$ from $F_{12}$ we implicitly settle on an arbitrary intrinsic parameters $K_2$ and, as a result, arbitrary extrinsic parameters $[R_2|\vec{t}_2]$. The randomness of the intrinsic parameters cause the associated reconstruction to only be within a projective similarity of the correct reconstruction because the reconstruction has to fit the intrinsic and extrinsic parameters of both cameras.
If we were to generate the projection $P_3$ of a third image in the same manner, then we would get yet another random set of intrinsic parameters $K_3$ and an associated reconstruction. This reconstruction will generally not be the same as the first one, though they would be within a projective similarity [Ma,+03]. This means that in order to combine the information from several images into one reconstruction, we need to somehow deal with the unpredictable nature of the intrinsic parameters. This is done by adjusting the projection matrices through a process called camera calibration.

4.2.2 Calibrating the cameras

The reconstruction created in the previous section is only within a projective similarity to the actual scene. Upgrading it to a metric construction involves calibrating the cameras [HZ04]. Calibrating the cameras adjusts the intrinsic parameters of the cameras, represented by the calibration matrix $K$, to match that of the real world cameras which recorded the images. Adjusting $K$ also affects the reconstructed 3D points $\vec{X}$ since there is a one to one relationship between them. The calibrated reconstruction $\vec{X}$ becomes a metric reconstruction, meaning that all angles in it are equal to their real counterparts.

Methods for calibrating cameras may be classified as either manual or automatic [HZ04]. Manual methods require some additional information about the scene to be known. A common approach is to include a calibration object such as a surface with a checkered pattern in the images. The cameras can then be calibrated by using the knowledge that the angles of the squares in the pattern should be right angles. The downside of manual methods is of course the need for additional knowledge of the scene; if such knowledge is unavailable, then manual methods cannot be used. This is were automatic calibration methods become interesting.

Automatic calibration, or auto-calibration, is based on assumptions about one or more of the intrinsic parameters of the cameras. There are different mathematical methods for auto-calibration, but they are all based on the dual
image $\omega^*_i = K_i K_i^\top$ of the absolute conic [HZ04].

**Auto-calibration using the dual image of the absolute conic and the absolute dual quadric** The absolute conic is a conic on the plane at infinity. It is useful for automatic calibration because it is dependent on the intrinsic parameters of the camera, but independent of the extrinsic parameters. For a given camera, the absolute conic and its image remain fixed regardless of the motion of the camera. By considering the absolute conic, and placing constrains on its dual image $\omega^*_i$, we in effect place constrains on the intrinsic parameters $K_i$ since $\omega^*_i = K_i K_i^\top$.

The absolute dual quadric $Q_{\infty}^*$ is a 4 by 4 symmetric, homogeneous matrix of rank 3 that is related to the dual image by $\omega^* = P Q_{\infty}^* P^\top$ [HZ04]. In metric 3D space, the space we want our reconstruction to be in, $Q_{\infty}^* = \tilde{I} = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}$. The projection $P$ is not necessarily metric, but in writing the dual image as

$$\omega^* = P Q_{\infty}^* P^\top = P \tilde{H} \tilde{H}^\top P^\top$$

then $PH$ will be metric. Thus by finding $H$ we can find a calibrated projection $PH$.

Determining $H$ is done by applying constraints on some of the elements of $\omega^*$ and using those constraints to construct a set of equations for $Q_{\infty}^*$. The absolute dual quadric $Q_{\infty}^*$ can then be decomposed as $Q_{\infty}^* = \tilde{H} \tilde{H}^\top$. The dual image $\omega^*$ of the absolute conic is can be written in terms of the intrinsic parameters of the camera as

$$\omega^* = KK^\top$$

$$= \begin{bmatrix} \alpha_x^2 + x_0^2 & x_0 y_0 & x_0 \\ x_0 y_0 & \alpha_y^2 + y_0^2 & y_0 \\ x_0 & y_0 & 1 \end{bmatrix}$$

where $\alpha_x$ and $\alpha_y$ define the aspect ratio, $x_0$ and $y_0$ define the principal point.
(the point in the image plane closest to the focal point), and \( s \) defines the skew \[HZ04\]. Applying constraints involves, for example, deciding that the principal point should be located at the origin of the image plane, or that the aspect ratio \( \alpha_y/\alpha_x \) is equal to that of the input image. If we know that the principal point of the camera is located at origin then \( x_0 = y_0 = 0 \). This means that the elements \( \omega_{13}^* \) and \( \omega_{23}^* \) such that

\[
(P_i Q_\infty^{*\top} P_i^\top)_{13} = (P_i Q_\infty^{*\top} P_i^\top)_{23} = 0.
\]

(5)

If we have several projections \( P_i \) with the same intrinsic parameters then all their respective absolute quadrics \( Q_\infty^{*\top} \) are identical, so \( Q_\infty^{*\top} = Q_\infty^{\ast\ast} \). Since \( Q_\infty^{\ast\ast} \) is symmetric it has 10 degrees of freedom. Equations 5 provide 2 constraints per projection, so 5 projections will be enough to solve for \( Q_\infty^{\ast\ast} \). Expanding \( (P_i Q_\infty^{*\top} P_i^\top)_{13} \) and \( (P_i Q_\infty^{*\top} P_i^\top)_{23} \) for each of the 5 projections gives a set of 10 linear equations in the 10 unknown elements of \( Q_\infty^{\ast\ast} \). That set of equations may then be written as \( Aq_\infty^* = 0 \) where \( q_\infty^* \) is a vector of the 10 unknown elements. This can then be solved by singular value decomposition \([GK65]\).

Having calculated \( Q_\infty^{\ast\ast} \) for a set of projections \( P_i \), we can now find metric projections by decomposing \( Q_\infty^{\ast\ast} = HH^\top \) and multiplying the projections by \( H \) such that \( P_i^{\text{metric}} = P_iH \).

By observing that

\[
\vec{x}_i = P_i \vec{X} = P_iHH^{-1} \vec{X}
\]

we see that the pointset \( \vec{X} \) also can be upgraded from projective to metric by

\[
\vec{X}^{\text{metric}} = H^{-1} \vec{X}
\]

such that

\[
\vec{x}_i = P_i \vec{X} = P_iHH^{-1} \vec{X} = P_i^{\text{metric}} \vec{X}^{\text{metric}}.
\]
4.3 Resulting implementation of reconstruction

The starting point for performing the reconstructing is the table of known interest points from the first stage, supplemented by a list of indices to the interest points which where observed in the current frame. This table is updated once every frame with new information about the movement of all observed interest points. Using the theory introduced above, the information about the movement of each interest point since the previous frame can be used to determine the camera movement from the previous frame to the current frame. Once the motion of the camera has been determined, then we can triangulate the known interest points which were observed in the current frame.

Since the frames come from a video, there are a couple special considerations:

- Most interest points will have been observed in several frames.

- The baseline between successive frames is very small.

The first consideration is an advantage; the more samples there are of the position, the more accurately its real 3D position can be estimated. We will take advantage of this when estimating the 3D position of points by involving as many samples as possible in the calculations. The second consideration is a disadvantage [Dav04]; small baselines negatively impact the accuracy of the estimated position. For this reason we will avoid using samples with small baselines where possible.

When calculating the movements of the camera and the location of the reconstruction, we need something to relate these to — a coordinate system or frame of reference.

4.3.1 Frame of reference

Based on the theory from the previous section, there are two ways to estimate the projection of a frame $I_j$. First, we need to realize that the projection of a given frame is necessarily expressed in relation to some other projection. This is a necessity because we have no absolute frame of reference. The projection
of the very first frame $I_1$ is therefore selected as the frame of reference for all other projections by defining it as $P_1 = [I \overrightarrow{0}]$. This puts its focal point at the origin in the coordinate system, and all projections and reconstructed structure will be relative to this projection. A frame $I_j$ can then be estimated relative to $I_1$ either via the fundamental matrix $F_{1j}$ as explained previously, or, once some 3D structure is known, via solving $\overrightarrow{x_j} = P_j \overrightarrow{X}$ for $P_j$.

4.3.2 Bootstrapping

We need to know at least one more projection before we can begin estimating the 3D positions of the points. A second projection $P_j$ can be estimated by calculating the fundamental matrix $F_{1j}$ and extracting $P_j$ from it. Now the small baselines between successive frames come into play. We could choose to estimate $P_2$ of the second frame $I_2$ next — it would seem to be a natural choice — but the small baseline between $P_1$ and $P_2$ could have an adverse effect on the accuracy of the estimated $P_2$. Instead of $P_2$ we find the last frame $I_j$ that still has enough points in common with $I_1$ to calculate $F_{1j}$: that is, we find the last frame $I_j$ where

$$|\overrightarrow{x_1} \cap \overrightarrow{x_j}| \geq k \geq 7.$$

Calculating $F_{1j}$ boils down to solving $A \overrightarrow{f_{1j}} = 0$ for $\overrightarrow{f_{1j}}$ as seen previously. The equation can be over-constrained by selecting a large $k$, such as 100, and a total least squares solution for $\overrightarrow{f_{1j}}$ can then be found. This is accomplished by constructing the matrix $A$ from the points $|\overrightarrow{x_1} \cap \overrightarrow{x_j}|$ and then finding a Single Value Decomposition (SVD) $A = U \Sigma V^*$. A property of SVDs is that the total least squares solution to $A \overrightarrow{f} = 0$ is found in the column of $V$ corresponding to the smallest singular value (values on the diagonal of $\Sigma$). The implementation utilizes the JAMA/C++ Linear Algebra Package library [Poz04] on the host to calculate the SVD of $A$. Using a large value for $k$ has the advantage of both reducing the effect of noise and increasing the generality of the point set.

Having determined $\overrightarrow{f_{1j}}$ and hence $F_{1j}$, we can now calculate the epipole $\overrightarrow{e_1}$.
and then generate the projection $P_j = [[\vec{e}_1] \times F_1][\vec{e}_1]$. As was detailed in Section 4.2.1, this projection is but one of infinitely many possibilities. Associated to this particular solution for $P_j$ is a particular structure $\vec{X}$. Since the reconstruction currently is only projective, not metric, if we had chosen to generate a projection $P_k$ from a different image $I_k$ instead, then we could have gotten a very different associated structure. In other words, generating projections from the fundamental matrix results in projections tied to different 3D structures. The 3D structures will be projectively equivalent, but not identical. If we are to combine the information from all images, then we need a way to make sure all the projections we generate are associated with the same 3D structure. For this reason we cannot keep generating projections using the fundamental matrix since that method generates arbitrary projections within a projective similarity; we need projections within metric similarity.

This is accomplished by finding further projections using the second method, solving $x_j^* = P_j \vec{X}$ for $P_j$. This ensures that the projections we find are within a metric similarity to the structure $\vec{X}$. To be clear, this does not mean that $\vec{X}$ is a metric reconstruction of the actual scene, it just means that all the projection matrices we generate are metric in relation to the “unmetric” $\vec{X}$. We are not yet ready to upgrade $\vec{X}$ to a metric reconstruction because we do not yet have enough projections to solve for the absolute quadric $Q^*_\infty$. First we need to generate more projections. In order to do this though, we first need to estimate $\vec{X}$. The first estimate for $\vec{X}$ is calculated by solving the set of equations

$$x_1^* = P_1 \vec{X}$$
$$x_j^* = P_j \vec{X}$$

for $\vec{X}$. The resulting structure $\vec{X}$ is stored on the device as a list $L^{3D}$ of 3D points where

$$L^{3D}_i = \vec{X}_i$$

With an initial estimate of the structure $\vec{X}$ we can start generating more
projections, and in turn improving the reconstruction.

4.3.3 Iterative reconstruction

While frames are being received from the previous stage, the 2D position of each point is copied from the known interest point table into new, separate table $T^{2D}$ on the device. This also happened during the bootstrapping process, such that $T^{2D}$ now contains the history of positions in images $I_k$, $k \in 1, 2, \ldots, j$ for all points. Table $T^{2D}$ is constructed such that $T^{2D}_{ij} = \vec{x}_j$ where $i$ is a point identifier and $j$ is the frame number. For every point $\vec{x}^i$ in the list $L^{3D}_i$ of 3D positions $\vec{X}^i$ there is a row in the table $T^{2D}$. In addition, a list $L^P$ of projection matrices is constructed where $L^P_j = P_j$.

With the data structures in place, 3D positions and projection matrices can now be continuously estimated. Maintaining a history of 2D point positions and related projections makes it possible to iteratively improve the estimations by re-estimating $\vec{X}$ and prior projections. An overview of the algorithm is provided next, before challenges and special considerations are discussed:

1. A column of sampled positions $\vec{x}$ is appended to table $T^{2D}$.
   (a) Positions of points which were not observed in this frame are set to 0.
   (b) Brand new points effect new rows in the table.

2. The projection matrix $P_j$ for the current frame is estimated by finding a total least squares solution for $\vec{x}_j = P_j \vec{X}$.

3. The new projection matrix $P_j$ is appended to the list $L^P$.

4. The structure, $\vec{X}$, including any new points is recalculated.

5. Repeat for the next frame, starting at Step 1.
This is an iterative process where the estimations for the projections and the reconstruction is repeatedly improved as more data — 2D points $\vec{x}$ — is made available. There are however a few devils in the details here.

**Upgrading from projective to metric** The reconstruction $\vec{X}$ is not yet metric. It is currently some arbitrary projective reconstruction. This is solved by waiting until we have calculated enough projections, and then the cameras are calibrated as explained previously. The old, projective reconstruction and projections are then replaced by their new, upgraded versions before the iterative process continues. Any subsequent projections that are estimated based on the new reconstruction $\vec{X}$ will then automatically be calibrated and hence metric.

**Memory usage** With tens of thousands of 2D points $\vec{x}$ per frame, the memory usage on the device quickly skyrocket as frames are loaded and points added to the table $T^{2D}$. Limited memory on the device forces us to find a more sustainable way to manage the memory.

This is solved by limiting the length of the history and the amount of 2D points that are retained. Only data about the most recent frames and observations is retained instead of keeping the entire set of all observations of all points in memory all the time. This is implemented as a sort of sliding window which slides in two directions:

- It slides through time in the sense that it is always aligned with the most recent input frame.
- It slides through space in the sense that it only retains the most recently observed points, which will tend to be the 3D points that the latest projections caught. Old points are discarded as new points are added.

This fixes the size of the table $T^{2D}$ and thus the memory requirements. The list $L^{3D}$ of reconstructed points and the list of projections $L^P$ will also grow indefinitely, but much slower than the table $T^{2D}$ did. By limiting the processing to the sliding window it is now possible to dump old projection matrices and
reconstructed points to a file, since they are not being used anymore. That way the size of the lists $L^{3D}$ and $L^P$ are also fixed. Videos of any length may now be processed without worrying about memory usage.

**Shuffling data** Solving sets of equations for projection matrices and for 3D points involves constructing large matrices for every set of equations. With potentially hundreds of thousands points and iterative computations, this results in extremely large amounts of shuffling data around, much of it redundant. We can minimize the amount of shuffling.

This is accomplished by simply assuming that all points within the sliding window were observed in all images within the sliding window. This assumption means that any projection $P_j$ can be calculated by simply feeding all 2D points $\vec{x}_j$ and 3D points $\vec{X}^i$, where $i$ is within the sliding window, straight into the library routine. Likewise, any 3D point $\vec{X}^i$ can be calculated by simply feeding all 2D points $\vec{x}_j$ and projections $P_j$, where $j$ is within the sliding window, straight into the library routine. This eliminates all shuffling since we are just providing the library routines with pointers into our table and lists.

**Missing 2D point observations** A known interest point is not necessarily observed in every frame for the duration of its lifetime. There will be “gaps” where it disappears, just to reappear a few frames later. This needs to be accounted for.

If we were still shuffling data around and constructing a new matrix every time we wanted to estimate a projection or a 3D point, then we could account for this by only involving the appropriate intersections of points. A projection would be estimated from 2D points $\vec{x}_j^i$ and 3D points $\vec{X}^i$ where $i$ is intersection of points that were observed in $I_j$ and 3D points which have been estimated already. Similarly, a 3D point $\vec{X}^i$ would be estimated from only the projections $P_j$ and 2D images $\vec{x}_j^i$ where $\vec{X}^i$ was observed in $I_j$.

This approach is unfortunately not an option anymore since the data from the table and lists are being fed straight into the library functions, without
constructing intermediate matrices. This means that undefined values \( x^j_i \) will be affecting the calculations. Assuming that most points are in fact observed in a given image \( I_j \) then the projection \( P_j \) may still be estimated, if poorly. The solution is to first estimate \( P_j \), and use that estimation to actually project the 3D points \( \hat{X}^j_i \) that were not observed in \( I_j \). This way the missing 2D points \( x^j_i \) are approximated. The next time the projection matrix \( P_j \) is estimated it will be a more accurate estimation. The observed points and the 3D points are kept fixed, so that repeating the process iteratively will cause the projection matrix and the unobserved points to converge to a state that agrees with the known data.

4.3.4 Output

The output of this process is the two lists \( L^{3D} \) and \( L^P \). The list \( L^P \) contains the projection of every frame in the video relative to the first, \( P_1 \), which is the identity projection \([I|0]\). These projections are the extrinsic parameters of the camera, which describe how it move through 3D space. The list \( L^{3D} \) contains the reconstruction of every point that was observed in at least two frames. The coordinate system is relative to the first projection such that \((0, 0, 0)\) is located at the focal point of the first image \( I_1 \), and the focal length of the camera determines the scale of the coordinate system. The reconstruction, a point cloud, can now be the input to further processing where for example a triangle mesh is generated.
5 Conclusion

Computer vision was once thought to be an easy problem, suitable as a student summer project. As it turned out, however, computer vision was not a single, well-defined, easily solvable problem. Over the years it has grown into a large and diverse field where research is still very much ongoing. Vision is a very rich sense, with large amounts of data. This has been, and continues to be, a core challenge of applying computer vision theory to real world problems. The increasing availability and performance of hardware such as GPUs is making it more and more realistic to integrate computer vision techniques in real world applications. Computer vision techniques are already being employed extensively in real world factory-, surveillance-, information gathering- and modelling-applications.

This study has experimentally applied state of the art techniques in image processing and multiple view geometry to the problem of reconstructing 3D models from real world structures by means of video. This is a very computationally intense process, so, naturally, implementation on modern high performance GPUs has been a central issue. The result of this study is a revelation of some of the special concerns when applying the various theories and techniques to the problem of reconstruction from video in CUDA.

For simplicity, the implementation was divided into two distinct stages, the first dealing with extracting the necessary information from the input images, and the second dealing with using that information to reconstruct the real world structure.

5.1 Interest point tracking

Reconstruction of a 3D structure by means of multiple view geometry depends on knowing something about the geometry involved in the scene. Knowledge about the projections of the the scene from a sequence of viewpoints was extracted from the input video stream by identifying and tracking interest points
— specific points in the 3D structure — as they moved from frame to frame in the video. This was the first of two stages.

The identification and tracking of interest points was based on SURF. SURF is a state-of-the-art approach to detecting and describing interest points. It exhibits high invariance to most transformations, and performance equal or even better than other approaches to boot. It was however designed for the generic case of interest point tracking, and not specifically for application to video data. Video data brings with it some advantages and disadvantages compared to independent image sources, so several modifications to SURF were therefore made in this study in order to leverage those advantages and deal with the disadvantages.

The high similarity between successive frames was shown to be the primary advantage of using video as a data source. The similarity made the location of interest points more predictable, allowing the matching step to be more localized. This reduced the search space for interest points significantly; instead of searching the entire image space, only a small area around the expected position of the interest point needed to be searched. The computational cost of the matching stage was thereby reduced significantly.

Reducing the search space brought about its own advantage. Reduced search space meant that each interest point was compared to fewer other interest points. This reduces the chance of false matches, and therefore allowed the descriptors of the interest points to be less distinctive. This lowered requirement to distinctiveness was shown to significantly reduce computation and memory accesses when generating descriptors.

A disadvantage of video as input is the sheer amount of data that needs to be processed. The implementation in this study does nothing special to deal with this problem other than reducing complexity where possible.

CUDA lends itself very well to processing images, and most of the steps were straightforward to implement in CUDA. Since transferring data between the host and device is considered a costly operation, the implementation was
designed such that image data was only transferred to the device once, where all computations then were performed without transferring anything back to the host.

The final product of this stage was a table tracking the 2D location of known interest points. The information in this table was the processed further by the second and last stage, reconstruction.

5.2 Reconstruction

Starting off with a table describing the changing projections of known interest points in the video, this stage calculated the motion of the camera and reconstructed the scene present in the video. Leaning on the theory of multiple view geometry, the projection matrices of cameras were generated both from the motion of known interest points and from the 3D location of known interest points and their corresponding projections in the 2D image plane. Some special considerations had to be made though.

Successive frames in a video typically have a very small baseline. Reconstruction based on images with a small baseline will produce inaccurate estimates, so care was taken to maximize the baseline used for calculations. Estimating the 3D location of a point was done by taking into account as many of its projections as possible, ensuring maximum accuracy and robustness towards noise.

The large amount of interest points that were tracked, in the range of tens of thousands in each frame, meant that it was not feasible to store a complete history of all points in device memory. This was solved by means of a sort of sliding window such that only the most recent points and frames were retained. The fixed size of the window effectively made the memory usage constant, allowing for a video of any length to be processed without worrying about memory limitations.
6 References


