Enhancing 3D point clouds with low-end devices

Laura Kreuk, Rutger van den Berg, Joren Hammudoglu

Visual Point cloud Enhancer (VIPE),
Bachelorproject,
Geodelta, June 30, 2016
Coach: Jan van Gemert
Client: Raynor Vliegendhart
Preface

This is the bachelor thesis produced by Laura Kreuk, Rutger van den Berg and Joren Hammudoglu as part of the Bachelor end project course TI3806 of the Bachelor Computer Science at the Delft University of Technology. Over the course of 10 weeks a product to combine low-end device photographs with a 3-dimensional structure generated using photographs taken by a calibrated camera, is produced. The client is the engineering company Geodelta located in Delft. The goal of this report is to inform the reader about the completed work and describe the process we went through during the project.

We would like to thank everyone at Geodelta for the constant support during the project. Special thanks goes to Raynor Vliegendhart for supporting us and contributing throughout the whole project.

Also, we would like to thank our supervisor Jan van Gemert for his guidance and expertise during the project, in addition to his feedback.

Finally we would like to point out that we are looking forward to seeing how Geodelta will use our product in the future.

Laura Kreuk
Rutger van den Berg
Joren Hammudoglu
June 30, 2016
Abstract

Currently, after a traffic accident happens and the police arrive at the scene it takes some time for them to gather the necessary evidence. This means that the road is closed for quite some time. To speed this up our client Geodelta has developed Raida.

The idea is that the police will take photos of the area using a calibrated DSLR camera. They will then use Raida to generate an accurate computer model of the scene. This model can then be used to gather further evidence, such as measurements of tire tracks. This means that less time is needed on the scene, and the road can be reopened sooner.

Our software will be using that structure. The pictures and other data from Raida will be referred to as the Raida data.

Often, the police are able to gather photos taken by bystanders using low-end uncalibrated devices, such as smart phones. Many of these photos are taken before the police arrive, which means that they can contain relevant information that is no longer present when the police take their photos. Currently, the police is uncertain on what to do with these photos. It would be useful to be able to use them to enhance the Raida model.

To this end Geodelta asked us to create a way to use smartphone photographs to enhance the Raida model. In this report we will describe the software product we have created. Our product is called Visual Pointcloud Enhancer, or VIPE. It will take as input an accurate 3D model in the form of a point cloud, the photos used to generate this point cloud, and a number of additional uncalibrated photos. It will show additional information about the scene.

This report will discuss the algorithms and libraries used for developing VIPE. As Raida has a very limited visualisation of the point cloud, VIPE focuses on a clearer and more informative visualisation of the point cloud so the point cloud can also be used to examine the accident scene. The external photographs are linked to the Raida data using a SfM pipeline which produces a 3D point cloud. The point cloud consists of feature points from the pictures and the camera positions of these pictures. We know that the Raida data is relatively accurate as there is a quality control in the program. We calculate a transformation matrix by comparing the coordinates of the Raida data and SfM pipeline data and can add with this transformation matrix the camera positions of the external data to the Raida data. Within the pipeline SIFT is used to extract features from the photographs and match these features using a nearest neighbour algorithm.

While implementing the software some important software goals were taken into account. Geodelta wants to use our software and extend it or use parts of it. Documentation and readability of the code was focused on intensively during the development phase. Also modularity and extensibility were taken in account when writing the software. Testing is also an important part of the code. The software is tested thoroughly using unit tests and continuous integration. These unit tests were executed on Linux and Windows to guarantee VIPE works on both platforms. Also a user test is executed, to check if the software program is clear to the user.

The software product we made is a working beta implementation that includes all the requirements set with the client. The product can be extended or partly used for future products of Geodelta.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>i</td>
</tr>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>2 Problem definition</strong></td>
<td>3</td>
</tr>
<tr>
<td>2.1 Raida</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Problem description</td>
<td>3</td>
</tr>
<tr>
<td>2.3 Requirements</td>
<td>4</td>
</tr>
<tr>
<td>2.3.1 Non-functional requirements</td>
<td>4</td>
</tr>
<tr>
<td>2.3.2 Functional requirements</td>
<td>5</td>
</tr>
<tr>
<td><strong>3 Problem analysis</strong></td>
<td>6</td>
</tr>
<tr>
<td>3.1 Structure from motion</td>
<td>6</td>
</tr>
<tr>
<td>3.2 Feature detection</td>
<td>6</td>
</tr>
<tr>
<td>3.2.1 SIFT</td>
<td>7</td>
</tr>
<tr>
<td>3.3 Image matching</td>
<td>7</td>
</tr>
<tr>
<td>3.4 Low-end device photographs</td>
<td>7</td>
</tr>
<tr>
<td>3.5 Camera view angle</td>
<td>8</td>
</tr>
<tr>
<td>3.6 Geodetic rotation definition</td>
<td>8</td>
</tr>
<tr>
<td>3.6.1 OpenMVG SfM algorithms</td>
<td>9</td>
</tr>
<tr>
<td><strong>4 Software design</strong></td>
<td>13</td>
</tr>
<tr>
<td>4.1 Design goals</td>
<td>13</td>
</tr>
<tr>
<td>4.2 Architectural Patterns</td>
<td>14</td>
</tr>
<tr>
<td>4.2.1 Modular programming</td>
<td>15</td>
</tr>
<tr>
<td>4.2.2 Model–view–controller</td>
<td>15</td>
</tr>
<tr>
<td><strong>5 Software implementation</strong></td>
<td>16</td>
</tr>
<tr>
<td>5.1 Planning</td>
<td>16</td>
</tr>
<tr>
<td>5.2 Scrum</td>
<td>18</td>
</tr>
<tr>
<td>5.2.1 Reflection</td>
<td>18</td>
</tr>
<tr>
<td>5.3 Libraries</td>
<td>18</td>
</tr>
<tr>
<td>5.3.1 Python</td>
<td>18</td>
</tr>
<tr>
<td>5.3.2 WxPython</td>
<td>19</td>
</tr>
<tr>
<td>5.3.3 VTK</td>
<td>19</td>
</tr>
<tr>
<td>5.3.4 OpenMVG</td>
<td>19</td>
</tr>
<tr>
<td>5.4 Tools</td>
<td>19</td>
</tr>
<tr>
<td>5.4.1 Github</td>
<td>20</td>
</tr>
</tbody>
</table>
Introduction

Imagine you drive, in the passenger seat, past a traffic accident on the other side of the road and you take a picture with your smartphone just after it happens. Some of the information seen on the picture could prove to be of great value to the police, either by itself or in conjunction with other data.

Raida is a software product that is being developed for the police. The method of obtaining visual information of the scene of the accident will be that when the police arrives at the scene (this could be a while after it happened), an officer takes pictures of the accident with a calibrated DSLR camera. The police will provide these pictures to the Raida software and make it generate a 3D point cloud of the scene. The potentially valuable information on the smartphone picture however, cannot be used in Raida and the picture is discarded.

If the police could work with low-end device pictures, extra information about the accident might be found. Especially the situation right before or just after the accident could be of interest to the police and could contain a key piece of information.

When waiting for the police to arrive, and also the person that has to take the pictures of the accident scene, a lot of time goes by. The road will be impassible during this time, which means other lanes of a highway have a higher occupation or the road will be closed completely. This can result in traffic jams on the road of the accident and roads nearby, especially when it takes a long time before the accident scene can be cleaned up because the photos have to be taken first.

It would be a lot faster when the photographs can be taken immediately when some agent nearby arrives at the accident. In this case the scene can be cleaned up after the photographs are taken and a lot of time is saved.

Concluding from the examples mentioned before, it would be useful when pictures from smartphones or other low-end devices can be processed to give information about a certain event. This could be in addition to the calibrated Raida photographs or as the only information about a scene. Because smartphone pictures aren’t accurate enough, we decided to combine them with the Raida data. This report will focus on using external data in addition to the Raida data.

In this report we introduce a solution to link low-end device pictures to 3D structure generated from calibrated photographs. We have developed a software product that uses low-end device pictures to give extra information about a scene given that calibrated photographs of that scene are available. The name of our software product is Visual Pointcloud Enhancer, or VIPE for short.

In this report the problem will be explained in further detail as well as our client, in chapter 2. Thereafter the problem analysis will be discussed in chapter 3 which in fact is a short recapitulation of the research report (see appendix 2). Then, in chapter 4 we explain the detail about the software design and what the important aspects were when designing the software. After that the way the software is implemented is explained in chapter 5. A full description of the software product is given after the foreknowledge is introduced in chapter 6. Next we evaluate the results of our project and give suggestions and recommendations for the future work of the product in chapter 7. Finally the
report is concluded in chapter 8. The appendices contain intermediate products and more detailed documentation that is referred to in the report.
Problem definition

To prevent confusion in the rest of the report the term external photographs will be used to indicate photos taken with low-end or uncalibrated devices, such as smart phones. The photographs taken by calibrated DSLR camera's will be referred to as Raida photographs. Finally we will use the term Raida data for the data output by Raida, which includes, among other things, the Raida photographs, 3D point cloud and the camera positions and orientations.

This chapter explains the problem our product was created to solve. The original project description drafted by Geodelta can be found in appendix A.

First we will introduce Raida in 2.1. In section 2.2 we will describe the specific problem we have solved. Finally section 2.3 lists the requirements we created in collaboration with the client.

2.1 Raida

Raida is a software product created by Geodelta. It is designed to be used by the Dutch police, and is currently in the testing phase. It uses photos taken at the scene of a traffic accident to generate an accurate 3D representation of the scene.

The photographs the police provides as input are made using a high-end, calibrated, DSLR camera.

These photographs are loaded into Raida, and used to generate a 3D point cloud. This point cloud consist of keypoints that are detected in at least two pictures. When the police wants to take a closer look at some part of the scene, they can select this part and an orthophoto is generated. An orthophoto is a top down view, without perspective. When you zoom in on a certain area of the orthophoto, the police can see for example skid marks on the road. These can then be measured and used as evidence. These orthophotos are calculated with the use of beacons. These beacons are placed in the scene before the photos are taken. With the use of these beacons the accurate scale can be calculated in the orthophoto.

2.2 Problem description

Our task is to investigate whether or not the external photographs can be used to enhance the model Raida produces. The idea is that a separate program is made to process these low end device photographs. This program can make use of the data produced by Raida.

When an accident has just happened and the police have yet to arrive, bystanders have often already taken photos. These photos can be essential for the investigation. There might be evidence
on these pictures the police didn’t see when taking photos after they arrived at the scene.

We have developed a program that is an addition to the already existing program, Raida, for the police. Is it possible to connect data from low end devices to the already existing point cloud? How can this data be visualised? These are all questions we have answered in order to develop an useful program for the low end device photos or videos.

2.3 Requirements

Based on our research and in collaboration with our client, we have compiled a list of requirements for the software product using the MoSCoW method. These requirements were drafted before the implementation/development phase. The reflection on these requirements can be found in section 7.1.2.

The product is considered finished once at least all must-have’s have been implemented. The software is built with the assumption that the external photographs are relevant (i.e. of the crime scene), the EXIF data, when present, is correct (e.g. the timestamp is in the local time-zone) and that the existing 3D point cloud created by Raida is accurate.

2.3.1 Non-functional requirements

The required features of VIPE that do not necessarily provide a specific function in the software are listed here.
1. An intuitive user-interface which should be understandable in under five minutes for people who have experience with Raida
2. Well documented, extendable and understandable code and conform to existing style guides
3. SfM algorithm should run in under an hour
4. Should work with up to at least 200 pictures
5. Visualisation should run in at least 18fps
6. Cross-OS (Windows and Unix-like systems) compatibility
7. Library licenses must allow the product to be used as proprietary software

2.3.2 Functional requirements

Must have

1. 3D Visualisation of the generated structure from external photographs
2. Import pictures to be processed (incrementally or in its entirety) with SfM and added to the visualisation
3. Browse through the external photographs on a temporal basis
4. View the imported images by clicking on them in the visualisation
5. Import/Export calculated data
6. Visualisation of the original 3D point cloud

Should have

1. Display the estimated accuracy of low-end photo geographical placement
2. Localizing and displaying the position of the camera based on an image from metadata/calibration
3. Orientation of the external photographs based on the existing 3D point cloud
4. Accurate estimation of the scale in the 3D point cloud

Could have

1. Face detection and extraction
2. Check correctness EXIF data by comparing it with the existing 3D point cloud
3. GPU acceleration
4. Low-end video support

Won’t have

1. Social media pictures
2. Google Maps geographical data integration
Problem analysis

During the research phase, we looked into methods and algorithms required for the development of our project. In this chapter we will briefly discuss our findings. The reasoning behind our choices can be found in the research report, which can be found in appendix E.

We introduce structure from motion (SfM) in section 3.1. In section 3.4, relevant characteristics of low-end photographs in relation with SfM are discussed. After that the algorithms we use for feature detection and extraction are explained in section 3.2. Finally our choice of SfM library is explained in 3.6.1.

3.1 Structure from motion

The technique of creating a 3D structure using a sequence of two-dimensional images is called Structure from Motion (SfM), first proposed by Koenderink and Van Doorn in 1991. With the tremendous increase of computing power and the rise of social media, this technique has become very popular and can even be used to generate a detailed 3D model of entire cities in a short time-frame [4]. We use SfM to generate the model we display in our visualisation.

The basics of SfM are fairly straightforward. A feature detection algorithm is used to extract recognizable points keypoints in each of the images. This produces, for each keypoint, the 2D image coordinates where the point was found, along with a descriptor that describes the characteristics of the keypoint. Feature detection will be explained in more detail in 3.2.

Next a matching algorithm uses the previously calculated descriptors to determine which of the detected keypoints are views of the same object. In other words, it finds out which area of the photographed object, is seen in multiple photos.

Once enough keypoints have been matched, the relative positions of the photos can be calculated. From this information a graph can be created, where the vertices are images and the edges are distances between the images. Next the relative orientation between the images can be calculated. This produces a model with unknown scale. If scale information is known, for example the real distance between some detected points, the scale can be adjusted for the whole model.

3.2 Feature detection

Feature detection algorithms are used to find recognizable points in an image. This makes it the backbone of SfM. We have looked into a number of feature detection algorithms, such as SIFT [13], SURF [6] and ORB [17]. SURF and ORB are fast alternatives to the SIFT algorithm. We came to the conclusion that the SIFT algorithm still produces the best results. For more detail on our choices, see our research report in appendix E. We will explain SIFT in more detail here.
3.2.1 SIFT

The most popular feature detection algorithm is the Scale Invariant Feature Transform (SIFT) algorithm ([13]). SIFT transforms image data into coordinates of scale invariant feature points.

There are four stages in which SIFT computes the most important features of an image. In the first stage the scale-space extrema are detected by searching over all scales and image locations. This is implemented using a difference-of-Gaussian function. After that, at each candidate location, keypoints are selected based on measures of their stability. To achieve invariance to rotation in the next step each keypoint is assigned one or more orientations based on local image gradient directions. The last step of the algorithm is to measure the local image gradients around each keypoint. These are transformed into a representation that allows local shape distortion and change in illumination.

3.3 Image matching

Two affine images can be matched by comparing their keypoint descriptors found by a feature detection algorithm. To do this, the most similar descriptors are paired using a clustering algorithm. An example of this is the $k$ nearest neighbour (kNN) algorithm which identifies the $k$ closest matches in a dataset. In this context it is used to find matching keypoint pairs by finding the closest descriptors. Note however that matching descriptors do not guarantee a good match of keypoints. By using a $k$ of 2 we can, for each keypoint, compare the two closest other keypoints and only keep the matches for which the closest two matching descriptors are dissimilar enough to exceed a certain predefined ratio, as described in [12]. This way the keypoints that are likely to be outliers are not matched.

3.4 Low-end device photographs

In this section the relevant characteristics of the external data are discussed. Usually structure from motion is used with photographs taken with calibrated cameras. For these cameras things like focal length, sensor width and lens distortion are accurately known. Because these properties are known, we can compensate for the effect they have on the photos. However, the external data we use comes from devices that are not calibrated, for which we do not know the relevant properties. Photos generally contain some metadata, stored using EXIF. This can include the required properties. However, smartphones are notorious for inaccurate EXIF data. They also contain a lot of software optimizations which ensures that they produce very pretty pictures, but also that we cannot rely on camera properties even if they are known. Another problem is that some smartphones, when taking photos at lower resolutions, only use a part of the sensor. This means that even if the sensor width is accurately known, we cannot rely on this information to adjust for the effects of the sensor size.

An important property of a camera is the focal length $f$ at the moment the photo was taken. The shorter the focal length, the wider the angle of view and the greater the area captured. The longer the focal length, the smaller the angle and the larger the photographed object appears to be. This information is essential when trying to determine the position of the camera from the photo as described by equation 3.1 with $i$ and $o$ the image and object distances respectively [20].

$$\frac{1}{f} = \frac{1}{i} + \frac{1}{o}$$

(3.1)

The focal length is generally present in the picture’s Exchangeable image file format (EXIF) data. However, a problem with low-end camera’s is that the EXIF data is not always reliable, as
it can be inaccurate or even be tampered with or missing. So another method of determining the focal length is required.

Because an important aspect of the model is accuracy, we don’t add new points to the point cloud based on the external photographs. We only determine the position at which the photo was taken relative to the provided, known accurate, point cloud.

We use three pieces of information provided by the EXIF data. First, we use the time stamp. We assume that this is accurate, because smartphones almost always have the accurate time. We disregard time zones, because we are only interested in pictures taken of the same scene. Additionally, smartphones generally automatically detect the time zone of the area in which they are used.

Second, we use the camera name in the EXIF data. This information is used to decide which photos were taken using the same camera. This is useful because when multiple photos are taken using the same camera, some camera properties may be determined based on the photos alone. Of course the camera name is not unique to one specific camera, but we feel that it is a reasonable assumption.

We also use the focal length contained in the EXIF data. However, this focal length is only used as an initial estimate. The estimate will be automatically refined based on the actual photos. More on this in 3.6.1

3.5 Camera view angle

Visualising the cameras accurately requires a correct representation of the field of view. So the view angle is needed. The information needed to calculate the horizontal angle, are the focal length \( f_{\text{mm}} \) in mm, the width of the CCD chip \( c_w \) in mm and the width of the image in pixels \( i_w \). From this the focal length can be converted to units of pixels

\[
f_{\text{pixel}} = \frac{f_{\text{mm}} \cdot i_w}{c_w}.
\]

With this, the horizontal angle of view \( \alpha_w \) can calculated:

\[
\alpha_w = 2 \arctan \frac{i_w}{2f_{\text{pixel}}}
\]

The calculations for the vertical angle of view are analogous.

3.6 Geodetic rotation definition

Raïda uses the geodetic definition for 3D rotation to describe the orientation of the camera’s. Such a rotation is defined by three numbers \( \omega \), \( \phi \) and \( \kappa \). These are rotations around the \( x \), \( y \) and \( z \) axes respectively. To visualise the camera orientations, a rotation matrix is needed. The rotation matrices around the individual axes are defined as

\[
R_\omega = \begin{bmatrix}
1 & 0 & 0 \\
0 & \cos(\omega) & \sin(\omega) \\
0 & -\sin(\omega) & \cos(\omega)
\end{bmatrix},
R_\phi = \begin{bmatrix}
\cos(\phi) & 0 & -\sin(\phi) \\
0 & 1 & 0 \\
\sin(\phi) & 0 & \cos(\phi)
\end{bmatrix},
R_\kappa = \begin{bmatrix}
\cos(\kappa) & \sin(\kappa) & 0 \\
-\sin(\kappa) & \cos(\kappa) & 0 \\
0 & 0 & 1
\end{bmatrix}.
\]
In photogrammetry, the usual order of the rotations is \( \omega \) first, then \( \phi \), and lastly \( \kappa \). So the rotation matrix is \( R = R_\kappa R_\phi R_\omega \), or

\[
R = \begin{bmatrix}
\cos(\phi) \cos(\kappa) & -\cos(\omega) \sin(\kappa) + \sin(\omega) \sin(\phi) \cos(\kappa) & -\sin(\omega) \sin(\kappa) + \cos(\omega) \sin(\phi) \cos(\kappa) \\
\cos(\phi) \sin(\kappa) & \cos(\omega) \cos(\kappa) + \sin(\omega) \sin(\phi) \sin(\kappa) & -\sin(\omega) \cos(\kappa) + \cos(\omega) \sin(\phi) \sin(\kappa) \\
\sin(\phi) & -\sin(\omega) \cos(\phi) & \cos(\omega) \cos(\phi)
\end{bmatrix}
\]

### 3.6.1 OpenMVG SfM algorithms

To link the external photographs to the Raida data we use the library OpenMVG, this is a structure from motion library (see §5.3.4). We use a SfM pipeline for this, OpenMVG provides a global and a sequential pipeline.

These pipelines only differ in the method used to compute the structure, after keypoints have been found and matched.

**OpenMVG SfM pipeline**

We will now explain the OpenMVG SfM pipeline.

**Image listing**

First, properties of the photos and the cameras used to take them are gathered. This includes photo names, resolutions, and internal calibration information. This information is obtained from the EXIF data. Focal length, if specified in EXIF, is used as an initial estimation. The camera name is used to group photos taken by the same camera, and to attempt to look up the sensor size in an internal database.

**Image description computation**

Secondly, features are generated for all photos. OpenMVG provides a number of algorithms for this purpose, but we only use SIFT.

**Corresponding images and correspondences computation**

After this, matches are computed. Again, several algorithms are provided. We use the default (fast cascade hashing).

This algorithm is an improvement on the cascade hashing algorithm [9]. First the cascade hashing will be explained and then the improvement of the fast cascade hashing algorithm will be explained.

In order to speed up the matching between images the cascade hashing method uses three layers: hashing lookup, hashing remapping, and hashing ranking. In figure [3.3], you can see the steps that are explained hereafter. In the first step, hashing lookup, all the keypoint descriptors of the images are transformed into \( m \) bits binary codes. Then a lookup table is constructed with a set of buckets using the \( m \) bits binary codes. To find a feature point \( f \) from image \( i \) in an image \( j \) you check the bucket from point \( f \), the points from image \( j \) in that same bucket are returned. The candidates from the hashing lookup stage are then set in a new \( n \)-bit (\( n > m \)) hashing function and remapped into Hamming space. Then the candidates are sorted according their hamming distance and the top \( k \) items are selected for the last step. In the last step these \( k \) candidates are hashed according their hamming distance in buckets. The buckets are accessed in ascending order until the top \( k \) candidates are found. The two nearest neighbours are found calculating the Euclidean distance from these \( k \) candidates.
The improvement of the fast cascade hashing method over the cascade matching algorithm is the fact that fast cascade hashing uses precomputed hashed regions. This means it is faster than cascade hashing at the cost of using more memory.

SfM solving
For the actual SfM solver, openMVG provides two alternatives. The global and the incremental method.

The sequential or incremental pipeline is based on [15]. It starts with two photos. For these two photos, the 3D poses (position and orientation) of the cameras are estimated. On the next iteration a photo is added, and a new view and 3D points are added, and the estimated poses and camera intrinsics are refined.

The global pipeline is based on the paper [16] by Moulon et al.. With this pipeline the position and orientation of the pictures is estimated in a 3D coordinate frame.

Datacolor (optional)
Next the colours of the points can be determined.

Compute Structure From Known Poses (optional) Finally, the generated point cloud can be refined by triangulating the keypoints using previously determined camera poses.

Comparing sequential and global
We used 34 pictures (14 calibrated camera photos and the rest external photographs) from the dataset of the blue chair, see 5.5.2, to test the global and sequential pipelines. Both sequential and global pipeline have a normal, high and ultra setting. These settings determine how many keypoints will be found. Of course generating more keypoints will result in longer calculations for the entire pipeline. We tested every combination of these settings for both the global and the sequential pipeline. The results can be found in table 3.1 as well as figure 3.3 and figure 3.3.

The runtime varied greatly, from 4 minutes for the normal setting on the global pipeline, to 2 hours and 8 minutes for the ultra setting on the sequential pipeline. The generated models were similarly different. The ultra setting on both versions of the pipeline was rejected because the
runtime was too long. From looking at the generated models, we found that the sequential pipeline produced significantly better results, while the runtime was only slightly longer. See figure 3.2 and figure 3.3. We also noticed that the global pipeline only used a small subset of the photos, meaning that we could not accurately place all of the photos in our model.

For these reasons, we decided to use the sequential pipeline.

![Figure 3.2: Point cloud of global pipeline, high setting](image)

<table>
<thead>
<tr>
<th>Version</th>
<th>Global</th>
<th>Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>4m23.199s</td>
<td>5m53.281s</td>
</tr>
<tr>
<td>High</td>
<td>7m17.897s</td>
<td>9m40.804s</td>
</tr>
<tr>
<td>Ultra</td>
<td>1h46m25.440s</td>
<td>2h8m44.529s</td>
</tr>
</tbody>
</table>

Table 3.1: Computing times SfM pipelines
Figure 3.3: Point cloud of sequential pipeline, high setting

Figure 3.4: 3D structure of the scene is computed from the estimated motion of the camera [15]
Software design

In this section we will explain general software design principles that we kept in mind while developing VIPE. It also details the design decisions we made.

In section 4.1 our design goals are explained. These are important points we considered while developing VIPE. After that in section 4.2 the patterns we used in the architecture of the software are described in detail.

4.1 Design goals

A number of goals were taken into account while designing the software product. These are based on both general software principles, and the wishes of the client. For each goal we explain why it is important for our project, as well as how we achieve it.

Documentation and readability

On top of actually creating VIPE, our client is very interested in seeing what we can do with the external data. This may be used to extend their own products in the future. For this reason it is important that any code we write is well documented, and easily understandable.

Our code is written in compliance with the Python style guide PEP8 [19]. Following a single (well known) style guide ensures that the code is easily readable for future developers.

Documentation is provided in the form of docstrings, in compliance with the Python docstring conventions defined in PEP257 [10]. This ensures that introspection can be used to find information about the code as needed. Also, automated documentation libraries can generate full documentation based on these docstrings.

Maintainability, reusability and extensibility

Because the client may wish to use parts of our software in the future, it is important that code cohesion is high, and code coupling is low. This means that code that fulfils the same purpose is gathered together, and parts that fulfil separate functions do not depend on each other.

We use modularity for this. Our code is separated into several individual modules. See section 4.2 for details.

Working with modules ensures that code can be implemented and tested separately, and can therefore be used separately. These modules can then be integrated into a full system. In addition to the previously stated goals, modularity also ensures that working in groups is easier to manage. This is because one person can work on a module, while a different developer can independently modify a different module.
Usability
The user interface of our software should be appropriate for our target audience. Our users will be the police, which means that our user interface should be easy to use for a policeman or policewoman. We have tried to accomplish this by using a standardized layout, very similar to most other programs. This ensures that the user will recognize the interface and be able to use their previous experience with different software to learn to use our software. We also made a help screen which explains the interface.

Performance
Since we’re working with large datasets of images, performance is essential. We don’t want the user to have to wait too long for their results.

As stated in our requirements, we want to be able to process 200 photos within an hour. Also, our visualisation should run at at least 18 frames per second.

To ensure this we made an option to save a project. This project saves the result of previous computations so that they do not have to be run again. It is also a lot quicker to load than parsing the original input data. This saves time when you want to look at the same project again later.

Portability
Since our clients are the police, and their systems run Windows, it is important that VIPE runs smoothly on Windows. A user will not wish to run a program that only runs on a different platform than the one they use.

Because a good portion of our project team runs Linux, and it is easier for development work, we also support Linux.

To ensure this portability we specifically looked for multi platform libraries as opposed to platform specific libraries. See 5.3

Reliability
Last but not least we aim for our software product to be reliable. This means in our case that the 3D point cloud that is generated from the pictures accurately describes what is shown in the photos, and that the software does not fail.

To increase model accuracy we try to ensure that the external photos have as little impact on the actual model as possible. This is because we can’t be sure that any data produced using them is accurate.

To ensure that our software does not fail, we have tested VIPE extensively. See 5.4 for details.

4.2 Architectural Patterns
An architectural pattern is a standard solution to a common problem in a software project. It is a template for a correct, recognizable, solution. An architectural pattern is applicable in a given development context and in that context produces beneficial qualities in the resulting system.

Our software product contains two architectural patterns. The most important pattern is modular programming, see subsection 4.2.1. This is our top level architecture design. Inside the GUI and visualisation modules we use the model-view-control pattern, see subsection 4.2.2.
4.2.1 Modular programming

Modular programming is a design technique that emphasizes separation of functionality. With modular programming the functionality of the software are separated into several modules. The modularity in our project will be achieved by creating a package per module. This provides for a clean 'flat' design. It is important to avoid circular dependencies between modules, as it defeats the purpose of modularity by increasing coupling. In figure 4.1a we show the separate modules our project consists of, as well as the dependencies between them. The visualisation module contains the visualisation. The GUI module contains the user interface, for example buttons and menus. The project module describes how we store all data relevant to a project. The raida module parses out input data into a project. Finally the sfm merger module uses an imported pointcloud and external images to produce a new model.

In Python, the programming language we use (see subsection 5.3.1), modular programming is supported. Every file in Python is a module, and a package contains all the modules.

![Diagram](image)

(a) Modules of VIPE

(b) MVC

Figure 4.1

4.2.2 Model–view–controller

An architectural pattern for implementing user interfaces is the model-view-controller. It divides the user interface part of the code into three parts, the model, the view and the controller (see figure 4.1b). The view part is what is shown to the user. The model manages the data and rules of application. The controllers accept user input and send this to the model or the view.

This pattern is used for both the visualisation and the GUI.
Software implementation

While developing VIPE we used a number of programs, libraries and tools. In this section we will explain why we used them and how they helped us while implementing VIPE.

We start by explaining our planning in 5.1. After that 5.2 we explain what Scrum is and why we used it for our short term planning. Next we explain which libraries we used and why in 5.3. After that the tools we used while building VIPE are listed in section 5.4. Furthermore in section 5.5 we explain what kind of tests we used to ensure the product works as intended. Finally in section 5.6 the feedback we received from SIG can be found, as well as how we used this feedback to improve VIPE.

5.1 Planning

In the first days of the project a project planning was made. We divided the planning into two parts, the research part and the development part. The project started with a research phase of two weeks, followed by the development phase from week 3 to 10. For the planning of the research phase see appendix D. In this section we focus on the planning of the implementation phase.

We used the Scrum agile development process to plan which features had to be built every week (see section 5.2). We also made a Gantt chart of the overall planning, see figure 5.1. While creating the planning, we took into account the SIG deadlines. We wanted to have the basic functionality finished before the first SIG deadline and before the second SIG deadline we wanted to have the full functionality implemented. For this last deadline it was important that no big changes in the code should be made after this deadline so the last SIG feedback gives a good overview on the quality of our final code.

One of the problems we had is that we underestimated the time required to get all the tools and libraries to work properly. OpenMVG was easy to set up, but VTK was exceedingly tricky to get working on all our machines. It took us almost two weeks before we had everything up and running. Because of this delay we started writing code later than planned which resulted in an overall delay in the project. This also resulted in not finishing the basic functionality (must haves, see 2.3) before the first SIG deadline. Another reason for the delay in the planning was working with openMVG. It took us a while to figure out how to get useful information out of openMVG, because at first it only output a point cloud and a big binary. The must have features that didn’t relate to openMVG however were all finished before the first SIG deadline. The linking of the external photos to the Raida data and visualising the camera positions of external photos were implemented during the final weeks. The last weeks were also used to improve the visualisation and finish the report. The full functionality was finished before the last SIG deadline, which means that the must haves and most of the should and could have requirements were finished by then.
Figure 5.1: Gantt Chart
5.2 Scrum

Scrum is an agile framework that can be used to develop software in a team. Scrum splits the project duration into sprints. For each sprint a list of features to implement is selected. Every sprint took a week. The sprint planning and a review for the previous sprint happened on the first day of the week. We also had a daily scrum meeting every morning to check the progress of every team-member. The following three question were asked. What did you do yesterday? What problems did you run into? And what are you going to do today?

In collaboration with the client a list of requirements (see section 2.3) was produced at the end of the research phase. It is important that the main features, called must haves in the requirements, are implemented first according to the scrum methodology.

At the beginning of every sprint the features that we wanted to implement during that sprint were chosen and assigned to a team member. The scrum framework means that goals are set and the goals for every sprint were the features assigned to that week. We used the issues tab in Github to manage our features and sprints.

5.2.1 Reflection

We noticed that Scrum worked very well most of the time but it still had some shortcomings. For example some of the tasks we set for a week of work appeared to be a lot more work than we had first assumed, which meant that they were not finished by the end of the week. We moved these tasks to the next sprint, which unfortunately meant that we could not get as many features as we had wanted into the next sprint. Another shortcoming of scrum is the fact that the main features had to be implemented first, while sometimes it was useful to do some other tasks first. For example it took a lot of time to figure out how to get the information we needed out of openMVG, while some of the main features depended on that information. This meant that we could not implement these features first. We also ran into some obscure bugs that took a long time to fix. Although you know that you will run into bugs, you can’t really plan for them, which also messed up our sprint plans.

5.3 Libraries

As for most programs, VIPE uses a number of libraries. In this subsection we will discuss the programming language and libraries we used.

5.3.1 Python

Early on we realised that the only language that our entire team was familiar with was Java, and we did not want to create an image processing related program in Java. This meant that some of us had to learn a new language during the project. We decided to use Python, because it has a relatively short learning curve. It also has bindings for all the libraries we needed. We chose Python 3 (3.4 to be precise) over Python 2 because it is faster, has a more readable syntax and it is a lot newer than Python 2 (nine years). Python 2 is also 9 years older than python 3.

In retrospect we may have been better off using Python 2. This is because while all our libraries worked with Python 3, Python 2 support was much more mature. This means that it took a lot longer to get everything working properly, because we had to compile almost everything from source.
5.3.2 WxPython

WxPython is a set of Python bindings for the wxWidgets GUI (graphical user interface) toolkit. We use WxPython 3.0.3, because this is the newest version of WxPython.

WxPython is open source which means that everyone is free to use it and the source code is available for everyone to look at and modify. Built on top of wxWidgets, WxPython is licensed under a custom made wxWindows License, similar to the GNU Lesser General Public License (LGPL), with an exception stating that derived works may be distributed in binary form under the user’s own terms. wxWidgets has a clean API and is well-documented.

5.3.3 VTK

VTK is a library specifically designed for scientific visualizations and rapid prototyping. It has, in contrast with OpenGL, high levels of abstraction and an object-oriented API. VTK provides Python bindings. Its license is BSD, which is a permissive free software license. VTK is very extensive, it provides a large amount of filters and algorithms. This means that we will not have to implement everything ourselves. A downside of VTK, as we noticed later on, is the limited documentation. We use the latest version of VTK, VTK 7.

The main structure of VTK is a pipeline of data, this begins with a source of information and ends with an image on the screen [7]. Our visualisation contains several of these pipelines. In figure you can see portions of the VTK pipeline.

The source simply provides the data that is sent into the pipeline, this can be read from files or it can be generated. For example, we use cones in our project, these are generated. Within VTK filters are components that receive data from other components and modify it in some way, they are an optional component in the VTK pipeline. After the data is modified by the filter the data is given as output to be used by other components. Mappers ‘map’ data to some form so it can be rendered. The difference between mappers and filters is subtle. A mapper outputs to an actor, while a filter outputs to a different pipeline component. Actors are VTK components which can modify the appearance of the data rendered on the screen, such as transparency or colours. Finally at the end of the VTK pipeline the renderers and windows show the user the actual visualisation.

Unfortunately, VTK is rather difficult to get working properly. We had some problems figuring out the exact cmake parameters to use, which was especially frustrating since the only way to test which set of parameters worked was to compile VTK and see if it works. Since VTK is a rather large library, this takes some time. The commands we used to get it working can be found in appendix G.

5.3.4 OpenMVG

We use openMVG[14] for the actual structure from motion calculations. It provides a number of structure from motion and other image processing algorithms. OpenMVG provides a SfM pipeline that we use to generate a point cloud using the photographs. Simple installation instructions for both windows and Linux can be found on the openMVG github page.

5.4 Tools

The tools described in this section simplify the development of VIPE. By using these tools the quality of the end product was improved significantly.
5.4.1 Github

In the development phase of the project we decided to work with Github as this is a web based host service for Git repositories. Github is the most popular hosting site for Git repositories. Git is a version control system that makes use of branching and merging. This means that you can work on one branch without affecting other branches. This means that the main development branch is always working properly, and several people can work on different features simultaneously. When a branch has a fully working and tested addition to the main branch they can be merged.

We used a master branch which always working and well tested code. We also had a dev branch. Every new feature was first merged into dev, and whenever we wanted to properly release a new version we tested the dev branch extensively before merging it into master.

We also used Github to list the features we wanted to implement. A sprint lasted a week and at the beginning of the sprint we discussed the features that had to be implemented that week. These features were added as issues in Github. When a feature was finished the issue could be closed. Github supports communication with other web-hosted tools, like Codecov and Wercker. This means that every time we created a new pull request on github, the code was automatically
tested by wercker (see 5.4.2), and then codecov (see 5.4.3) calculated our test coverage and posted the results to github.

5.4.2 Wercker

Wercker is a continuous integration service that can automate the development and deployment cycle. Every time a pull request is made on github, wercker clones the code, builds it, runs all the tests, and posts the results on the github pull request. Wercker then posts the test results to Codecov 5.4.3. It uses containers as a base for every build. This means that we can simply create a container with all our dependencies, and use that to run our tests. This cuts down on compilation time, which would otherwise be too long for our program be tested using continuous integration.

5.4.3 Codecov

Code coverage [[3]] is a measure that describes the degree to which the source code of the program was tested. A high code coverage indicates that most of the code was tested, which means that the program has a lower chance to contain bugs than a program with lower coverage.

We use Codecov because it is supported by Github and can be used for free on private repositories. Codecov gathers the coverage reports generated by wercker and displays the coverage on github. It also has more detailed coverage information on their website, and a browser plugin that can show which lines were covered in github, using a colour overlay.

We aim to have a code coverage of at least 70%. A big part of our software product is the visualisation, unfortunately we are not able to test all of this code, because it cannot be (easily) automated. This goes for the GUI as well, although we were able to check for errors while creating panels and such by using a headless X server to simulate a graphical platform.

5.4.4 Docker

Wercker uses docker containers to run its builds. As stated before in 5.4.2 we were able to install all our dependencies on this container. This took some time, because none of us had worked with Docker before. But it allowed us to use continuous integration, so it was worth it. The sequence of installation commands to create a docker container is stored in a Dockerfile. This means that it can be reproduced or modified easily. Our Dockerfile can be found in G.

5.5 Testing

The main goal of testing is to check if VIPE satisfies the software requirements. When writing a test it is important to know what feature you are going to test, and what requirements this feature fulfills. There are several reasons why a software project should be tested. First of all to check if the requirements are satisfied. It also ensures that the risk of running the program is lowered. Also, when a problem is identified by testing, it is identifier earlier. This means that the cost to solve the problem is lower than it would be had the problem been identified later.

5.5.1 Continuous integration

To automate testing and deployment, we use the continuous integration service Wercker (see subsection 5.4.2) in conjunction with Github. During the development process we wrote tests. Every time a new feature is added and a pull request is created for this, all the tests are run in combination
with this new code. When all the tests pass the new feature probably doesn’t break any existing code.

Unit testing is a method to test small pieces of the source code apart from the rest of the code. With unit testing for every unit one or more test cases are developed. In the best case all the test cases are independent of other tests. Unit tests also find flaws early in the development phase, because when writing tests you check if the functionality works as you expect it to work. The unit test are executed on both Linux and Windows to guarantee the cross-OS platform requirement.

5.5.2 Test photographs

While developing the software we needed some test sets of photos. Testing on only one test set of pictures does not tell us whether our program works with other datasets. This is why we have multiple test sets.

The first data set we used contains photos of the blue chair in Delft. This was an easy choice, because GeoDelta provided us with a Raida data set of it, and it was very close. This means that we could easily walk there and take some photos of the blue chair with our smartphones to complete our testset. We tried to take photos with the same lighting conditions as the Raida photos. In this case this meant it wasn’t sunny and they were taken in the afternoon. In figure 5.3 a few smartphone photos in this dataset are shown.

5.6 SIG

Four weeks into the implementation phase we sent our code to the Software Improvement Group (SIG). Sending the code to SIG is a requirement of the bachelor end project. The feedback received can be found in appendix C.

SIG performs a static code review, without actually executing the code. They check the quality characteristics of maintainability by looking at properties such as volume, code duplication, unit complexity, unit size, unit interfacing, module coupling, component balance and component independence [1].

5.6.1 Feedback

On May 27th the first code was submitted and on June 3rd we received feedback on the first SIG evaluation. The code scored 4.5 out of 5 stars on maintainability. The following aspects of our code could still be improved:

- A file name loadDatasets.py was mentioned to be our longest method, data and logic should be separated. However the file loadDatasets.py doesn’t exist in our project.
- There were also other long methods that contain logic (not extremely long otherwise we would not have this high score).
- The ratio between code and test code is 1:3, this is reasonable but hopefully we can improve this.
5.6.2  Improvements

The focus of the feedback on our second submission will be on whether or not we have processed the feedback. It is important that we improve the code on the points found by SIG.

We sent SIG an email immediately after the feedback to find out why they had mentioned a file that does not exist. After a week we got an email back that explained our examples where switched with those of another group. The improvements for us could be found in parsers.py, which contained some long methods and some data structures were defined in code that are hard to fix. The parsers.py file is a file we got from our client to use the Raida data in our software. Because parsers.py is a provided method and not our own code we won’t improve this as this is actually external code.

To improve the long methods, we searched in the code to find where data and logic could be separated and split these methods.

Also to improve the code more tests were written so the ratio between code and tests should be higher for the second feedback.

5.6.3  Second feedback

The second round of SIG feedback mentioned that we had improved the main issues that they mentioned in the first round of feedback. They further stated that we scored lower in other areas, without specifying which areas they meant. This is not particularly useful. They also mentioned that the ratio between the volume of regular code and the volume of test code was too low. We believe that this is a poor way to judge the degree to which our code was tested. While testing we focused on using as little test code to test as many things as possible. For example, by using parameterised tests to test various classes using the same test functions. Also, the majority of our untested code is visualisation code, which is notoriously hard to automatically test. This does not mean that we did not test it, just that it could not be automated, which in turn means that it is invisible to SIG. We continuously monitored how much of our code was tested using a code coverage tool. Because our program contains a lot of visualisation code, we believed 70% coverage was a reasonable goal. We ended up with 69.4% coverage.

To conclude, we believe that our code is sufficiently tested.
Figure 5.3: Test set of mobile pictures
Final product

The final software product is a prototype, it is not going to be used in this form if Geodelta decides to use it. Geodelta will or expand it or use parts of the software product. In this chapter the product is described in detail. The previous chapters explained the description of the problem, the research done and algorithms used, the design and the libraries and tools used. Now that information is known, it is time to explain the software in total.

The main addition that VIPE brings to Raida is the extensive visualisation of the 3D point cloud and the link of external photographs to the Raida data. The visualisation in VIPE can also be used to investigate the accident scene. Also the addition of the external data to the Raida data should give a better understanding of the accident. This is because the external photos were likely taken before the photos taken by the police.

The program consists of three parts, the start screen, calculating the data and the visualisation. You begin in the start window, where you can load the data into the program, see section 6.1. When the data is loaded, VIPE can calculate the new point cloud. This is explained in 6.2. In the last step a visualisation of the point cloud is shown. See section 6.3.

6.1 Start window

When you start VIPE the first thing you see is the start window. It has an ‘import Raida project’ button and an ‘open project’ button. The ‘import Raida project’ button loads a Raida project into VIPE and shows a visualisation of the project. The ‘open project’ button opens a previously created VIPE project. Figure 6.2 shows the menu bar. This menu bar contains some basic options. The file menu can be used to open and close projects, import a new Raida project, or close VIPE. The help menu has an about and a help option, both give information about VIPE. The about button shows some basic information about VIPE, and the help option shows information about the visible window.
Figure 6.1: Start screen VIPE

(a) File menu VIPE

(b) Help menu VIPE

Figure 6.2: Menu bar VIPE

26
6.2 Calculating data

When the Raida data and some external photos have been loaded the next step is to actually use this information to calculate a new point cloud.

The Raida photos as well as the external photos are loaded into the openMVG pipeline. OpenMVG then produces a point cloud and for each photo a camera position and orientation (pose). Because we don’t want the (possibly inaccurate) information gathered from the external photos to influence the accuracy of the input model, we want the openMVG point cloud to be added to the original, accurate, Raida point cloud. To achieve this we take the camera poses of the Raida photos in the Raida dataset, and the camera poses openMVG calculates using the same photos, and use them to calculate a similarity transformation between the two coordinate systems. We can then use this transformation to transfer any point in the openMVG coordinate system to an equivalent point in the Raida coordinate system, see figure 6.3. We use this to add all the new points to the original point cloud. By doing this we retain the accuracy of the original model, and use the openMVG dataset to enhance it.

Figure 6.3: Transformation of the openMVG camera poses to the Raida coordinate system. The transformation matrix is calculated by using the Helmert transformation on the camera poses of the calibrated high end photographs in the Raida data and in the openMVG data. This transformation can be used on any point (or camera pose) in the openMVG data to transfer it into the Raida coordinate system.
6.3 Visualisation

The visualisation is a view of the combined dataset created by the calculating data step. While making the visualisation we stumbled upon new ideas on how to represent the data. The main idea was to visualise a point cloud and show the camera positions.

![Image of VIPE camera selected and viewpoint](image)

Figure 6.4: VIPE visualisation

Camera poses are represented by a marker in the visualisation. When a marker is clicked, the photo taken from that pose is shown. The marker also shows the time the photo was taken. In addition to showing the photo, left clicking shows the field of view corresponding to that photo. Essentially, it shows you what part of the point cloud is visible in the photo (see 6.4a). Right clicking, instead of only showing the field of view, actually moves the visualisation so that it matches the photo. So you will have the original photo and the generated point cloud showing the exact same view of the scene (see 6.4b).

6.4 Processing external pictures

Based on the work of [Tomasi and Kanade](#), we have devised a method to incrementally add external pictures to the visualisation. This can be done by extending the already known shape motion.

From the Raida project, the motion (camera positions) matrix $M (2F \times 3)$ and the shape matrix (point cloud) $S (3 \times P)$ can be obtained. Assuming there is no noise, the original single value decomposition (SVD) can be computed by noting that $M = L \Sigma^{1/2}$ and $S = \Sigma^{1/2} R$ for the SVD of the original measurement matrix $W = L \Sigma R = MS$, without actually recomputing the SVD of $W$. By decomposing the measurement matrix of the external data with SVD, we obtain $W^* = L^* \Sigma^* R^*$. With SVD revision [8], the decompositions of $W$ and $W^*$ can be merged and processed into the shape and motion matrices $S^*$ and $M^*$. This way the original shape and motion can be extended with external data while keeping their original structure.

Due to the absence of existing implementations of the method discussed and the vast amount of time required to implement this method, we cannot apply it to our project. That is why we decided to use an existing structure from motion library to calculate shape and motion from the
Raida data and external data together. The results are then matched with the original shape and motion calculated by Raida using a similarity transformation \[ 5 \]. This ensures that the coordinate systems of Raida and OpenMVG match and the shape and position calculated from the external data can be added to the original Raida data.

The latter method essentially has to do double work by recomputing the shape and motion from the Raida data, which has already been done by Raida. But this method is a lot easier to implement, because the complicated calculations have already been implemented in OpenMVG. This makes it the best option for the given time-frame.
Evaluations and recommendations

In this chapter we evaluate the final software product and give recommendations to our client based on our knowledge after developing the software. We have done a lot of research, even during the development phase about the combining of Raida data and external photographs. Our knowledge will be shared in this chapter and recommendations will be given about our experience in the field.

In section 7.1 we will discuss if the final product is what we aimed for and the obstacles in the process are listed. After that in section 7.2 we will explain what extensions could be made to Raida, which we have already implemented in VIPE. Finally we will give advice for future product developers in section 7.3. As Geodelta will use our product or parts of it, we will explain what could make the product even better or more useful in the future.

7.1 Final product evaluation

During the development we had some delays because of unforeseen difficulties. In this section the final product and the development process are evaluated. We will answer questions such as is the product what we had intended it to be? Did we meet the requirements and design goals?

7.1.1 Evaluation of progress

When making the final product some parts of the project were harder and took longer than we planned. We lost a lot of time because of these obstacles.

It started after the development phase with installing the libraries and setting up the tools. Docker and Wercker took some time to figure out how to set up.

The first problem was getting all the tools and libraries working. It took us a while to get VTK and, to a lesser degree, wxPython and openMVG working. Especially considering the fact that we also had to get it working on a clean installation in a Docker container.

Installing VTK was especially problematic. First of all it was unclear what options we had to set in CMake. We initially had BUILD_SHARED_LIBS turned off, because we intended to pack VIPE and all of its required libraries into a single distributable. We ran into some strange, badly explained, errors while building VTK. After some attempts to fix it we forgot to turn off BUILD_SHARED_LIBS and the problems disappeared. We still don’t know what actually caused the error.

A second problem was that CMake could not always correctly identify the python library folders to install VTK bindings into. We first tried to simply move the created bindings into the correct location. Unfortunately it seems that there were some absolute paths used in the bindings, breaking them if moved. We eventually solved this issue by manually setting the python library folders using the CMake GUI.
Another issue was that not all of our laptops were able to run VTK. Specifically, one of our laptops did not support OpenGL 3.2. There was a CMake option to set the OpenGL version used by VTK, but setting this to lower versions did not appear to make a difference; it certainly did not resolve the issue. Like many things, this was not properly documented by VTK.

If we would start the project with the knowledge we have now we would probably use python 2 instead of 3. This would allow us to simply download binaries for VTK, or even pull them from repositories on Linux.

Next we had to work with OpenMVG. In contrast with VTK, installing OpenMVG was easy. There were clear instructions on how to install it on Windows, Linux and even OSX, although we did not test the latter. OpenMVG even provides a Dockerfile, which shows exactly which dependencies should be installed and how to install them on a clean Ubuntu installation.

Unfortunately, actually using OpenMVG in the way we had intended was not as simple. We first intended to take the provided features calculated by Raida, and match their descriptors to new ones generated by OpenMVG. This was obviously not supported by OpenMVG, so we modified the OpenMVG source. After some attempts we came to the conclusion that this method could not work. The descriptors created by different implementations of the same algorithm were incompatible.

Another problem was that while using the OpenMVG pipeline to generate a point cloud from a set of photos was very easy, it was not clear how to extract intermediate results from the OpenMVG output. For example, we wanted to know which points were found in which photo, and the camera poses of the photos. By default, OpenMVG only outputs a point cloud as a .ply file, and a binary blob. After some time we more or less accidentally figured out that if we changed the name of the output file from output.bin to output.json, we actually got a json representation of the data.

Another problem we ran into was that the .ply point cloud file produced by OpenMVG contained some strange formatting. Instead of a single space delimiter between point coordinates, there were often several. Some lines even had one or more leading spaces. Additionally, the decimal mark was based on the system locale, which meant that on some systems the file had comma separated values, while the viewer we used expected dots. We used the Linux utility sed to apply a simple regular expression replace function to the files as a workaround. We also opened an issue on Github. Fortunately, openMVG has some pretty active support, so the formatting was quickly fixed.

The next problem we ran into was that OpenMVG updated the entire generated point cloud when a single photo was added. This meant that our plan of incrementally adding the external uncalibrated photos to the point cloud without modifying the existing, accurate, points did not work.

We ended up using a different approach. We now calculate a point cloud using OpenMVG on all the photos, both the original calibrated photos and the additional external photos. We then use the calculated camera poses of the calibrated photos to generate a similarity transformation from the coordinate system generated by OpenMVG to the coordinate system used by Raida. We then use this transformation to add the newly calculated points to the original point cloud.

A problem for both OpenMVG and VTK is that they have very limited documentation, mostly limited to very basic functionality. This means that in order to implement certain features using these libraries, you first have to figure out exactly how they work. This makes it a lot harder to use them.

### 7.1.2 Evaluation of requirements

All the must haves [2.3] were implemented as planned. Half of the should have requirements are also implemented. Table [7.1] lists the requirements and shows whether or not they have been met. The could have requirements turned out to be less useful to our product, so we did not implement.
<table>
<thead>
<tr>
<th>Requirements</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Must Have</strong></td>
<td></td>
</tr>
<tr>
<td>1. 3D visualisation of the generated structure from external photographs</td>
<td>✓</td>
</tr>
<tr>
<td>2. Import pictures to be processed with SfM and added to the visualisation</td>
<td>✓</td>
</tr>
<tr>
<td>3. Browse through the external photographs on a temporal basis</td>
<td>✓</td>
</tr>
<tr>
<td>4. View the imported images by clicking on them in the visualisation</td>
<td>✓</td>
</tr>
<tr>
<td>5. Import/export calculated data</td>
<td>✓</td>
</tr>
<tr>
<td>6. Visualisation of the original 3D point cloud</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Should Have</strong></td>
<td></td>
</tr>
<tr>
<td>1. Display the estimated accuracy of the low-end photo geographical placement</td>
<td>×</td>
</tr>
<tr>
<td>2. Localizing and displaying the position of the camera based on an image from metadata/calibration</td>
<td>✓</td>
</tr>
<tr>
<td>3. Orientation of the external photographs based on the existing 3D point cloud</td>
<td>✓</td>
</tr>
<tr>
<td>4. Accurate estimation of the scale in the 3D point cloud</td>
<td>×</td>
</tr>
<tr>
<td><strong>Could Have</strong></td>
<td></td>
</tr>
<tr>
<td>1. Face detection and extraction</td>
<td>×</td>
</tr>
<tr>
<td>2. Check correctness EXIF data by comparing with the existing 3D point cloud</td>
<td>×</td>
</tr>
<tr>
<td>3. GPU acceleration</td>
<td>×</td>
</tr>
<tr>
<td>4. Low-end video support</td>
<td>×</td>
</tr>
<tr>
<td><strong>Won’t Have</strong></td>
<td></td>
</tr>
<tr>
<td>1. Social media pictures</td>
<td>×</td>
</tr>
<tr>
<td>2. Google maps geographical data integration</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 7.1: Summary of the requirements met

most of them. We did come up with a good number of more useful features not in the original requirements. This is because when you actually use the product you will notice things that you had not considered while creating the requirements. Because we used Scrum we could easily change our plans to account for these new ideas. To see all the features we implemented go to appendix F.

Some of the features we thought of during the development can be implemented in the future and will be listed in section 7.3. Because Raida has a limited visualisation of the 3D point cloud we extended the visualisation a lot to so the user gets a good impression of the scene.

So far only the functional requirements have been mentioned, but we also composed some non-functional requirements. All of the non-functional requirements were met, but there were a few issues related to them.

The biggest issue was the requirement that the product worked cross-OS. Even though we used cross-OS libraries, there were subtle differences between the results on different systems. Primarily this means that some things looked different on Windows than it did on Linux, but there were even subtle differences between various Linux systems. For example, at first Windows could not display all the photos. Any TIFF photos simply could not be opened using wxWidgets. We solved this by opening the photos using the python library Pillow, and then converting them to the wxWidgets format. We also had some problems when resizing windows that only appeared on Windows systems. Fortunately we were able to fix all these compatibility issues.

We required the SfM algorithm to run in under an hour for 200 pictures. Unfortunately we don’t have a sufficiently large dataset so we weren’t able to test this. We instead looked at 3.1
which used a dataset containing 35 photos. Using the high setting sequential pipeline, and assuming that the algorithm scales linearly with the amount of photos, we conclude that we likely meet the requirement.

7.2 Recommendations for Raida

In this section we will list the VIPE features we believe will be a useful addition to Raida. For these we will provide an estimate on how long it would take to implement them, and how they would be useful.

Currently Raida visualises a point cloud using red points. In VIPE we noticed that adding the actual colour to the points makes the visualisation much more recognizable. It should be very easy to implement, because the colours can be taken from the original photos.

Showing camera poses is a useful way to link the point cloud back to the photos. When the user wants to see more detail on the object shown, it is easy to open a photo that shows the object. They should be easy to implement as well, since the camera poses are already calculated as part of the structure from motion process.

Adding the field of view for a photo is a good way to show what parts of the scene are visible on the photo. It is also possible that when looking at a photo, something is partially shown, with the rest being hidden behind a different object. The field of view makes it much easier to identify what should be in the photo, but is hidden. It is fairly easy to implement using the focal length and the sensor size.

When clicking on a camera position the view 'flies' to a mode where only the points in the area of the picture are visible. In this view it is easier to compare the original picture with the points. This is a particularly useful way to quickly validate the point cloud. It should not be much work to implement this, especially if the field of view is also implemented.

7.3 Recommendations for future product

Since the software product is a prototype, there is a lot of room for improvement. In this section we will give some general advice on using the software, and list some features we thought of during the development that we did not get around to implementing.

If only parts of VIPE will be used, we recommend not to use VTK. Primarily because it is very large, making shipping it as part of a product inconvenient. Furthermore it is not very well documented which means that new developers will need a lot of time to familiarize themselves with it before they can make meaningful changes.

We also have some ideas for future features.

When the camera pose of a photo is selected, it might be useful to add an option to only show the points that describe the photo. Another idea is to add scale information to the point cloud. This allows the user to see how big an object actually is. This could be done by specifying the known size of an object in the model. This would also allow for measuring distances between points. Another useful addition would be to identify a ground plane. A plane or grid could be used to visualise this. It can also be used to determine an initial view for the visualisation.
Conclusion

During the bachelor project we tried to link external photographs (smartphone / low end device pictures) with the data from Raida. Raida is a software product being developed by our client Geodelta, to be used by the police for investigating accidents using high-end, calibrated cameras. An addition to Raida was produced to link external photographs to the Raida data. The software product is called Visual Pointcloud enhancer, or VIPE for short. As the visualisation of the 3D point cloud in Raida is very limited, VIPE also improves the visualisation of the point cloud extensively.

The open-source software used did not support the use of our optimal solution, explained in section 6.4. We constructed a work-around for this by using a similarity transformation to link the newly generated openMVG point cloud to the existing Raida point cloud and use this transformation to add new points to the original point cloud. This retains the accuracy of the original model, while still filling in the gaps using the new photos.

We showed that it is possible to link the external photos to the Raida data. The method we use can be improved using the solution explained in section 6.4. Also the accuracy could be improved further. Due to the time constraints and the open-source software used we didn’t get the chance to do this yet. Also the visualisation of the 3D point cloud can still be extended to give extra information.

We will conclude this report by stating that it is possible to use low end device pictures to extend the Raida data.
Bibliography


Original project description

The name of our project is 'Enhancing 3D pointclouds with low-end devices'. The original project description can be found on BEPsys. [2] All the information in this chapter is achieved from the BEPsys site and drafted by Geodelta.

Company description

Geodelta is small, but highly specialized technical company in the fields of geodesy, photogrammetry, lidar, terrestrial laser scanning, geo information and GIS. With more than 30 years of working experience, the strength of Geodelta lies in the synergy between groundbreaking technology, consultancy, application development and production. Our projects – often varied from R&D to quality control - require an agile mindset. We love quality, but are not afraid to be sceptic and pragmatic when needed. The client however, will always come first. All geodetic and photogrammetric software of the company have been developed in-house to ensure quality, creativity and innovation.

Project description

Can data spread by low-end devices be used to improve existing 3D models?

Nowadays, news gets spread in a blink of an eye. With low-end devices as smartphones, it has become incredibly easy to share videos and pictures of noteworthy events. The field of photogrammetry focuses on obtaining geometric information from pictures. Geodelta recently developed a software package which can generate 3D geometric information out of DSLR pictures for use in crisis management (through drones) and road accident registration. One of the results is a 3D pointcloud. What if pictures or even videos from smartphones could be automatically positioned within these pointclouds?

This project will involve designing and implementing prototype software. We will mainly support you in the geometric and photogrammetric aspects of the work. Typical steps will involve (but are not limited to):

- Choosing the data types and setting the requirements for their use.
- Researching image matching and/or point tracking techniques.
- Finding ways to automatically combine data sets (e.g. purely visual enhancement, or fully co-registering different 3D sets, etc.).
- Assessing the quality and added value of the end product.
- Designing a user friendly interface.
Additional information

**Company** Ingenieursbureau Geodelta

**TU Delft coach** Jan van Gemert

**TU Delft coach email** j.c.vangemert@tudelft.nl

**Company coach** Raynor Vliegendhart

**Company coach email** raynor@geodelta.com

**Company contact phone** +31 15 215 8188
Infosheet

Title of the project: Enhancing 3D pointclouds with low-end devices
Name client: Ingenieursbureau Geodelta B.V.
Date of the final presentation: June 24, 2016

Description

When a traffic accident happens, the first photos are usually taken by bystanders using their smartphones. The information contained in these photos could prove to be of great value to the police when they are investigating the cause of the accident. Raida is a software product being developed by our client Geodelta, to be used by the police for investigating accidents using high-end, calibrated cameras. Our program, called VIPE, is an addition to this by adding the camera positions of low-end device photos to the point cloud of the scene and visualise the data of the point cloud.

Feature detection finds the key points in a photo, these key points are rotation invariant. We used an algorithm called structure from motion to generate a 3D structure of these key points.

Figure 1: Structure from motion. Moulon et al. IEEE International Conference on Computer Vision, pages 3248{3255, 2013.}

The Scrum methodology helped us with planning the project and completing it on time. Even though implementing certain functionality took a lot longer than expected, Scrum allowed us to adapt accordingly. The biggest challenge was working with open source software libraries. They were generally not documented properly, and were difficult to get working at all. Our final product is a prototype and was tested using continuous integration and unit tests. All the must-have’s and most should-have’s have been implemented. Since the project’s end result is a prototype, additional functionality could be implemented to enhance it. In the report we describe how the prototype could be extended with such additional functionality and offer recommendations on how the prototype could be incorporated into Raida.

Figure 2: Matching of low-end device photos and calibrated photos

Members of the project team and contribution

Laura Kreuk  
front-end developer, research and development, communication

Rutger van den Berg  
back-end developer, quality control

Joren Hammudoglu  
core developer, code quality and testing

TU Coach (Pattern Recognition & Bioinformatics Group)
Jan van Gemert j.c.vangemert@tudelft.nl
Client: Ingenieursbureau Geodelta B.V.
Raynor Vliegendhart raynor@geodelta.com
Contact persons:
Joren Hammudoglu joren@linux.com
Rutger van den Berg rutgervandenberg@hotmail.com
Laura Kreuk laurakreuk@hotmail.com

The final report for this project can be found at http://repository.tudelft.nl
SIG feedback

[Analyse]

De code van het systeem scoort 4,5 ster op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt.

De langste methode in jullie project, in het bestand loadDatasets.py, komt doordat data en code daar door elkaar staan. Over het algemene is het goed voor de onderhoudbaarheid om logica en data zo strikt mogelijk te scheiden. Vandaar ook de populariteit van dataformaten als XML, JSON, of YAML.

Daarnaast hebben jullie ook lange methodes die logica bevatten, zoals bijvoorbeeld.GUIController._make_bindings in controllers.py. Die methode is niet extreem lang (dat kan ook niet, want dan zouden jullie nooit aan zo’n hoge score zijn gekomen), maar je zou hem nog verder kunnen verbeteren door het aanmaken van de data structuur met menu bindings naar een nieuwe methode (bijvoorbeeld create_menu_bindings) te verplaatsen.

Tot slot is het goed om te zien dat jullie naast productiecode ook unit tests hebben geschreven. De verhouding tussen testcode en productiecode is met 1:3 redelijk. Hopelijk lukt het jullie nog om die verhouding tijdens het vervolg van het project nog wat op te krikken.

[Naar aanleiding van de vraag over loadDatasets.py die niet in ons project voorkomt]

Zoals jullie al gemerkt hebben zijn de voorbeelden van jullie en een andere groep verwisseld. De mogelijke verbeteringen zitten in jullie geval in parsers.py. Jullie hebben daar een aantal datastructuren in de code gedefinieerd. Die zijn lastig te repareren, maar daarnaast zijn er ook een aantal lange "gewone" methodes: _images_section_parser en _points_section_parser zouden nog opgesplitst kunnen worden, bijvoorbeeld door aparte methodes te introduceren voor 2-parts en het 3-parts geval. Dit zijn relatief kleine aanpassingen, want zoals gezegd zitten jullie al aan de bovenkant van het spectrum.

[Hermeting]

In de tweede upload zien we dat de omvang van het systeem flink is gestegen, terwijl de score voor onderhoudbaarheid ongeveer gelijk is gebleven.

Het is goed om te zien dat de score voor Unit Size (die de kritische punt in de vorige upload was) is nu aanzienlijk verbeterd. Unit Score scoort op dit moment bovenmarktgemiddeld (4 sterran). Met de verbetering van de score voor Unit Size is ook de score voor Unit Complexity licht gestegen. De reden dat de totaalscore alsnog niet is gestegen heeft ertoe te maken dat jullie op andere aspecten nu wat slechter hebben gepresteerd dan voorheen.
Het is goed om te zien dat jullie testcode hebben geschreven. Vergeleken met het volume van productiecode is het volume van de testcode op dit moment lager dan wenselijk. De verhouding tussen testcode en productiecode is nu 1:4.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject.
PROJECT PLAN

LAURA KREUK, RUTGER VAN DEN BERG, JOREN HAMMUDOGLU

Enhancing 3D pointclouds with low-end devices,
Bachelorproject,
Geodelta,

June 13, 2016
Coach: Jan van Gemert
Client: Raynor Vliegendhart
1 Research phase

At the start of every day we will discuss what has been done the day before and what will be done this day. The first week we will primarily use to research the theory of the subject, the second week to research the available tools and how we are going to develop the software product.

Week 1

1. Structure from Motion (SfM) theory and algorithms
2. Feature detection theory and algorithms
3. Research the basics of photogrammetry
4. Issues with photographs taken with low-end devices as opposed to calibrated high-end camera’s
5. Compose suggestions for the scope and the specific requirements of the project. Look for existing solutions to similar problems.

Week 2

1. Discuss the scope of the project and the requirements with the client. Do existing solutions to the specified problem exist?
2. The specific data we will use from the low-end device photo’s
3. Choosing existing SfM libraries/frameworks, which we will base our programming language decision on
4. Research the existing 3d pointcloud dataset and how to combine it with the chosen SfM library/framework
5. Discuss the software architectures to be used

Our findings will be discussed in detail in the research report.

2 Development phase

The development phase will be from week 3 untill week 10 of the project. For the development of the software, we will be using the agile development method Scrum with sprint length of one week. We are going to use Git with the pull-request workflow on Github. The scrum backlog and other data will be integrated in Github by using its issue and release system. To automate testing and deployment, we will be using a continuous integration service (e.g. Travis) in conjunction with Github. During the development process we will write (unit and regression) tests according to the test-driven development paradigm. For this phase the global planning is as follows:

1. Set up a working example using the libraries/framework(s) chosen in the research phase, as well as the toolchain (IDE/projects, version control, continuous integration).
2. Obtain/create test data
3. Create a basic GUI
4. Before the first SIG deadline, finish the basic functionality
5. Finish the entire product before the final SIG deadline.
6. Use the last week to prepare a demo for the presentation and potentially process SIG’s feedback
3 Gantt Chart

![Gantt Chart](image)

Figure 1: Gantt Chart
4 Deadlines

- 2016-04-29: Research report
- 2016-05-27: SIG first upload
- 2016-06-17: SIG second upload
- 2016-06-17: Infosheet
- 2016-06-17: Final Report
- 2016-06-24: Presentation
1 Introduction

In this report we’ll discuss the findings of our research into the subject of our project. We started the project with a two week research phase. This will be followed by a research phase from week 3 to week 10. The main goal of the project is to link photos of low-end devices to an existing point cloud produced by analysing photos taken with a calibrated DSLR camera. In this report we’ll discuss how we intend to achieve the aforementioned goal. We have chosen the name Visual Pointcloud Enhancer for the software product, or VIPE for short.

The assignment will be explained in section 2. We will then briefly discuss some existing products that solve similar problems in section 3. In section 4 the characteristics of low-end device photographs as well as the difficulties related to them are presented. Thereafter, in section 5 the different methods for feature detection will be described. Image matching is explained in section 6. Next, Structure from Motion (SfM) is introduced in section 7. This is followed by a comparison of multiple visualisation libraries, and the most appropriate of these is chosen in section 8. In section 9 several user interface libraries are discussed. Finally in section 10 we’ll explain how we intend to build the product.

2 Project description

In this section we’ll discuss the problem that we’re going to solve. Section 2.1 explains some details about the client. After that in section 2.2 we will give a brief overview of our client’s product Raido, which our product will be designed to work with. Finally, the assignment is explained in detail in section 2.3.

2.1 Company description

Our client is the company Geodelta, situated in Delft. Geodelta is a small, but highly specialized technical company in the fields of geodesy, photogrammetry, lidar, terrestrial laser scanning, geo information and GIS. The customers of Geodelta are private and public organizations for whom the quality and actuality of geographical data is important.

With more than 30 years of working experience, the strength of Geodelta lies in the synergy between ground breaking technology, consultancy, application development and production. All geodetic and photogrammetric software of the company have been developed in-house to ensure quality, creativity and innovation.

2.2 Raido

Raido is software built by Geodelta, used by the Dutch police. It uses photos taken at the scene of an accident to generate an accurate 3D representation of the scene. The photographs the police provide as input are made using a high-end, calibrated, DSLR camera.

These photographs are loaded into Raido, then Raido processes these images and generates a 3D point cloud. When the police wants to look at some part of the scene closer, they can select this part and an orthophoto is generated. An orthophoto is a top down view, without perspective. When you zoom in on a certain area of the orthophoto, the police can see for example skid marks on the road. These can then be measured and used as proof.
2.3 Assignment

Our task is to investigate if the data from low end devices can be used to enhance the model Raido produces. The idea is that a separate program is made to process these low-end device photographs. However the new software can make use of the data Raido produces.

When an accident has just happened and the police have yet to arrive, people have often already taken photos. These photos can be essential in the investigation. There might be evidence on these pictures the police didn’t see when taking photos after they had arrived at the scene.

Our task is to develop a program that is an addition to the already existing program for the police. Is it possible to connect data from low end devices to the already existing point cloud? How could you represent this data? These are all questions we need to answer in order to develop an useful program for the low end device photos or videos.

3 Related work

In this section we discuss existing projects that are in some way similar to our own project.

3.1 Photo Tourism

The paper of Snavely et al. [2006] presents a system for browsing in a large unstructured collection of photos of a scene using a 3D environment. Their approach is based on computing the photographer’s location and orientation. Also the sparse 3D geometric representation of the scene is used in this system. The pictures used for this system were found on the internet, collected from sites like Google and Flickr. The paper describes how the collection can be used to produce a 3D model of popular tourist attractions. It also explains Object-Based photo browsing as showing multiple photos that contain the same partial scene or object. By knowing where a picture was taken, it can be placed in a 3D coordinate system. The user can then explore the scene by moving in 3D space from one image to another. Beyond displaying detected image features as a 3D point cloud, techniques to provide a scene appearance are introduced in this paper. The core of the system is the use of structure from motion.

The correlation between this system and our project is the visualisation of the scene and how the low end device photos can be used to reconstruct this scene.

3.2 Building Rome on a cloudless day

Frahm et al. [2010] introduce a method for dense 3D reconstruction in one day on a single PC (“cloudless”) from Internet-scale photo collections containing around 3 million photos. For example a query for “Rome” returned 3 million images on Flickr, these are images from different angles, made using a variety of cameras and viewing positions. The paper proposes a method for city-scale reconstruction using dense geometry estimation and efficient camera registration.

There are a few big steps to get this dense 3D reconstruction. The first step is appearance based clustering using gist features. After that the images are clustered and a single iconic view is chosen from every cluster as the best representative. Also the geo-location of the pictures is used if it is available. Then neighbouring iconic photos are identified by using a vocabulary tree, clustering based on geo-location, and image appearance. Finally depth maps are obtained from iconic clusters, a watertight scene representation from the depth maps is found.

The similarities between this system and ours lie in the use of low end device photos, and the visualisation of the obtained data. However, it differs in scope and focus. We do not use nearly as
many photos. The focus of our project is more on accurately matching the photos to the existing scene.

4 Low-end device photographs

We researched the effectiveness of image matching using feature detection algorithms on pictures taken with low-end devices. The purpose of this research was to see whether or not the algorithms can handle the low-end data as well as the pictures taken with calibrated high-end camera’s. We have looked at the matching between high- and low-end camera’s and the matching of blurry or otherwise low-quality pictures.

An important aspect of a camera is the focal length $f$ at the moment the picture was taken. The shorter the focal length, the wider the angle of view and the greater the area captured. The longer the focal length, the smaller the angle and the larger the subject appears to be. This information is essential when trying to determine the position of the camera from the photo as described by equation 1 with $i$ and $o$ the image and object distances respectively [Young [1971]].

$$\frac{1}{f} = \frac{1}{i} + \frac{1}{o}$$

(1)

The distortion as consequence of the focal length setting can be corrected using the method described by Brown [1966], which is implemented in OpenCV.

The focal length is generally present in the picture’s Exchangeable image file format (EXIF) data. However, a problem with low-end camera’s is that the EXIF data is not always reliable, as it can be inaccurate or even be tampered with or missing. So another method of determining the focal length is required.

Other important metadata is the time at which a picture is taken. This can not always be reliably extracted from EXIF data, partially due to the fact that EXIF does not include a time zone definition.

5 Feature detection

Feature detection algorithms are fundamental for this project. The purpose of such algorithms is to find key points in an image, regardless of illumination, rotation, scale, distortion or noise. The output of such an algorithm consists of a set of keypoints and descriptors. The keypoints are the locations of the features and the descriptors are extra data about the keypoints such as orientation or information about the surroundings.
5 Feature detection

(a) SIFT
(b) SURF
(c) ORB

Fig. 1: Detected features, original image from https://no.wikipedia.org/wiki/Vipe

In order to see some feature detection algorithms in action, we have used the computer vision library OpenCV 3.1.0 with Python. We used the k nearest neighbour algorithm for the matching of the found features with \( k = 2 \).

In the next subsections three of these feature detection algorithms will be explained. SIFT and SURF are the most popular of these algorithms and ORB is a faster alternative. The first algorithm to be explained is SIFT in section 5.1, after that you can find SURF in section 5.2. In section 5.3 ORB is introduced. In the conclusion 5.4 we will explain which algorithm we are going to use in our program and why.

5.1 SIFT

The most popular feature detection algorithm is the Scale Invariant Feature Transform (SIFT) algorithm (Lowe [2004]). SIFT transforms image data into coordinates of scale invariant feature points.

There are four stages in which SIFT computes the most important features of an image. In the first stage the scale-space extrema are detected by searching over all scales and image locations. This is implemented by using a difference-of-Gaussian function. After that, at each candidate location, keypoints are selected based on measures of their stability. To achieve invariance to rotation in the next step each keypoint is assigned one or more orientations based on local image gradient directions. The last step of the algorithm is to measure the local image gradients around each keypoint. These are transformed into a representation that allows local shape distortion and change in illumination.

5.2 SURF

Speeded-Up Robust Features (SURF) Bay et al. [2008] is a feature detector and descriptor inspired by SIFT. SURF, in comparison with SIFT, claims to be several times faster and more robust against different image transformations.

The main reason SURF is faster than SIFT is the use of integral images, this decreases the number of operations for simple box convolutions. The detector of SURF is based on the Hessian matrix, it detects blob-like structures where the determinant is maximised. The SURF algorithm has three main parts, the detection of interest points, neighbourhood description and matching.

Bay et al. [2008] showed SURF to be 3 times faster than SIFT, while the performance is said to be comparable to SIFT. Images with blurring and rotation are handled well by SURF, while the handling of viewpoint and illumination change is less accurate.
5.3 ORB
A very fast binary descriptor based on BRIEF, is called ORB. (Rublee et al. [2011]) ORB is rotation
invariant and resistant to noise. This method builds on the FAST keypoints and BRIEF features,
this is why it is called ORB, Oriented FAST and rotated BRIEF (ORB).

For the matching, the Hamming distance is used. ORB aims to be faster than SIFT and SURF.
When comparing is with the same data to SIFT and SURF, it is an order of magnitude faster than
SURF, and over two orders faster than SIFT. So for computation time ORB is a good alternative
to SIFT and SURF. However in the paper of Rublee et al. [2011] scale invariance is not adequately
addressed.

5.4 Conclusion
We compared the feature detectors and checked the keypoints they gave for multiple images, as one
can see in figure 1 and 2. In figure 2 you can see the largest amount of outliers are in the ORB
algorithm. From this is would appear that ORB is the least accurate.

It seems logical to select the feature detection algorithm based on the size of the dataset. For a
large dataset ORB is more appropriate and for a smaller dataset SIFT or SURF are more useful.

SIFT produces the best results. Also, We don’t expect particularly large datasets, which means
runtime-wise SIFT should not be a problem. For these reasons we chose to use SIFT.

6 Image matching
Two affine images can be matched by comparing their keypoint descriptors found by a feature
detection algorithm. To do this, the most similar descriptors are paired using a clustering algorithm.
An example of this is the $k$ nearest neighbour (kNN) algorithm which identifies the $k$ closest matches
in a dataset. In this context it is used to find matching keypoint pairs by finding the closest
descriptors. Note however that matching descriptors do not guarantee a good match of keypoints.
By using a $k$ of 2 we can compare the closest two and only keep the keypoints for which the closest
two matching descriptors are dissimilar enough to exceed a certain predefined ratio, as described in
Lowe [1999]. This way the keypoints that are likely to be an outlier are not matched. The kNN
matching of features is demonstrated in figure 2. What immediately becomes clear is that SIFT
yields the best results, in the sense of number and ratio of correct matches.

There exist multiple algorithms to approximate the nearest neighbour. Depending on the library
we further use in the project we choose a nearest neighbour algorithm. An example of a nearest
neighbour library is FLANN, where multiple algorithms are available to find the nearest neighbour.
Fig. 2: \( k \)NN feature matches with \( k = 2 \)
7 Structure from Motion

The technique of creating a 3D structure from a sequence of two-dimensional images is called Structure from Motion (SfM), first proposed by Koenderink and Van Doorn in 1991. With the tremendous increase of computing power and the rise of social media, this technique has become very popular and can be used to generate a detailed 3D model of entire cities in a short time-frame [Agarwal et al. [2009]].

This is generally implemented as follows. Features are extracted from the image sequence. The amount of features matching between two images can be seen as a distance unit. The images then are mutually matched to create a graph, where the vertices present the images and the edges present the distances between them. The closest images can then be compared using the SfM algorithm to determine their mutual orientation. From that data the three-dimensional structure can be approximated.

We will now introduce libraries for structure from motion we might be able to use in our project. We explain how these libraries work and which one is most suitable for our project.

OpenSfM

An open-source SfM implementation is OpenSfM [Mapillary]. We used this to research and the capabilities of the SfM algorithm. OpenSfM takes into account the EXIF metadata of the input photo’s to reconstruct the scene and determine the positions from where the photo’s were taken. It has a simplified BSD license so we can use it in our proprietary software. The problem with OpenSfM however, is that it is not compatible with the Windows operating system. For our project we do want it to be compatible with Windows, because that is the operating system the police uses.

Theia

A structure from motion library that is compatible with Windows, is the Theia vision library [Sweeney]. It is licensed under the new BSD license. Theia will determine the vocal lengths from the EXIF data. It wants images to have a good parallax otherwise the scene is more difficult to reconstruct. At last it needs a sufficient number of features extracted and matched. Unfortunately, although it does work on Windows, there seem to be quite a few windows-specific bugs.

OpenMVG

Also OpenMVG [Moulon et al.] is a SfM library that is compatible with windows. It has a MPL2 license so we can use it in our software. OpenMVG provides pipelines for solving sequential/incremental SfM and solving global SfM. The incremental pipeline process is a growing reconstruction process. It starts from an initial reconstruction with two views and this is iteratively extended by adding 3D points and new features. The global SfM makes a multi-view structure by proposing a 3D coordinate frame and positioning the pictures in this frame by estimating the position and orientation.

We have chosen to use the OpenMVG library in our software. OpenSfM was the first for us the reject because it isn’t compatible with the Windows operating system. We chose OpenMVG over Theia for several reasons. OpenMVG is a little older and more mature than Theia. It also seems to be better tested and more widely used than Theia.
8 Visualisation

An important part of the product is the visualisation of the 3D point cloud in combination with the low-end data. Due to the large amount of data, rendering on the CPU will not suffice. So in order for the visualisation to run smoothly, GPU acceleration will be required.

We have searched for visualisation libraries and studied two of them. In the next subsections these visualisation libraries will be introduced and compared. At the end of this section you can find the chosen visualisation library and the reason why we chose it.

**OpenGL**
A cross-platform, cross-language API that achieves this hardware-accelerated rendering is OpenGL. It focuses on performance and is for that reason often used for scientific purposes or gaming. It provides a highly configurable graphics pipeline for communication between the CPU and the GPU. This results, however, in a lot of boilerplate code when building a simple visualisation.

**VTK**
VTK is a library specifically designed for scientific visualizations and rapid prototyping. It has, in contrast with OpenGL, high levels of abstraction and an object-oriented API. VTK provides bindings to Python. Its license is BSD. VTK is very extensive, it provides a large amount of filters and algorithms. This means that we will not have to implement everything ourselves.

Since we’ve only got about two months to complete our product, there is no time (or need) to reinvent the wheel. Additionally, the high customizability of the pipeline in OpenGL might be useful for advanced complex visualisations but in our case it will unnecessarily complicate the code. For these reasons, and the fact that VTK is specifically built for scientific visualisation we have chosen to use VTK.

9 User Interface

For the user it is important that the software has a graphical user interface (GUI). Since one of the constraints (see Appendix A) of the product is cross-platform compatibility, this constraint must also hold for the GUI library. On top of that, the (open-source) license of the library has to allow our product to be distributed as proprietary software. Because we will be using VTK for the visualisation, support for that will have to be present in the library. Based on these constraints we have made a selection of eligible GUI libraries for further mutual comparison.

**QT**
QT is a widely used GUI framework for C++ and the de-facto standard for large projects. It has a LGPL license. There are two libraries containing python bindings for QT: PyQT and PySide. Unfortunately PyQT has a GPL license, so PySide with its LGPL license is more appropriate.

**wxWidgets**
A more lightweight than QT, wxWidgets is a very popular and mature GUI library. Built on top of wxWidgets, wxPython Phoenix is licensed under a custom made wxWindows License, similar to the GNU Lesser General Public License (LGPL), with an exception stating that
derived works in binary form may be distributed on the user’s own terms. wxWidgets (and therefore wxPython Phoenix) has a clean API and is well-documented.

GTK+
The C++ GTK+ library excels in multilingual and bidirectional text and localization. The license is LGPL and it has python bindings in the PyGTK library. Although it has cross-platform support it is mostly aimed at linux, and considered somewhat outdated.

GLFW
Licensed under a LGPL-like zlib/libpng license, GLFW is a C++ library. It focuses on creating windows with OpenGL contexts and receiving input and events. Python bindings are available.

We have chosen for wxWidgets, for it is more applicable to our project scale than QT, has a cleaner syntax than GTK+ and is better suited for building a GUI than GLFW.

10 Conclusion

In this section our idea of how to make the software product is presented. The conclusion of all the research is summarized. The software product that we are going to make is a separate program that uses the output of the already existing program Raido. The addition of the program is the use of data from low-end devices.

The idea is to make a program, named VIPE, that can process low-end device photographs. The generated data will then be combined with the high-end data produced by Raido (from the police) to reconstruct the scene. In order to do this we investigated the characteristics of the low-end photos. An important aspect here is the focal length of the camera. In the low-end pictures the focal length is generally presented in the EXIF data, however this is not always reliable.

In order to get a good understanding of the problem some papers and algorithms were researched. We have come to the conclusion that we will make use of SIFT for the feature detection in the pictures. When SIFT appears to be too slow we will look into the use of SURF or ORB. We make use of the OpenMVG library to apply these algorithms on our data. The points of the SIFT descriptor are matched to the keypoints of the point cloud. There are multiple image matching algorithms that can be used, for example FLANN which contains a collection of algorithms that can find the nearest neighbours.

Also structure from motion is used to visualize the scene of the photos that are taken. It is a technique to create a 3D structure from two-dimensional images. Conveniently, OpenMVG supports SfM.

Another important part of the product is the visualisation of the 3D point cloud and the low-end data. In order to do this we will use a visualisation library. The one we will use is VTK, a library specifically designed for scientific visualisation and rapid prototyping.

For the user it is necessary to have a graphical user interface. To build this a GUI library will be used. Because we use VTK for the visualisation a library that supports that is needed. We have chosen wxPython because this is the most applicable to our project.

The components named above are the main building blocks of VIPE. With these working together it is possible to build a good working end product that fulfils all the requirements.
Bibliography


Pierre Moulon, Pascal Monasse, Renaud Marlet, and Others. Openmvg. an open multiple view geometry library. https://github.com/openMVG/openMVG.


A Product requirements

Based on our research and in collaboration with our client, we have compiled a list of requirements for the software product using the MoSCoW method. The product is considered finished once at least all must-have’s have been implemented. The software is built with the assumptions that the low-end photo’s are relevant (i.e. of the crime scene), the EXIF data is correct (e.g. the timestamp is in the local timezone) and that the existing 3D point cloud is of high accuracy.

A.1 Non-functional requirements

The required features of VIPE that do not necessarily provide a specific function in the software are listed here

1. An intuitive user-interface which should be understandable in under five minutes for people who have experience with RadiO
2. Well documented, extendible and understandable code and conform to existing style guides
3. SfM algorithm should run in under an hour
4. Should work with up to at least 200 pictures
5. Visualisation should run in at least 18fps
6. Cross-OS (Windows and Unix-like systems) compatibility
7. Library licenses must allow the product to be used as proprietary software

A.2 Functional requirements

Must have

1. 3D Visualisation of the generated structure from low-end photographs
2. Import pictures to be processed (incrementally or in its entirety) with SfM and added to the visualisation
3. Browse through the low-end photo’s on a temporal basis
4. View the imported images by clicking on them in the visualisation
5. Import/Export calculated data
6. Visualisation of the original 3D point cloud

Should have

1. Display the estimated accuracy of low-end photo geographical placement
2. Localizing and displaying the position of the camera based on an image from metadata/calibration
3. Accurate estimation of the scale and orientation of the low-end photo’s based on the existing 3D point cloud
Could have

1. Face detection and extraction
2. Check correctness EXIF data by comparing it with the existing 3D point cloud
3. GPU acceleration
4. Low-end video support

Won’t have

1. Social media pictures
2. Google Maps geographical data integration
Based on our selection of libraries, two choices of programming languages are available: C++ and Python. Unfortunately neither language is known to all of us, so we decided to choose the language that is easiest to learn, which is Python. Additionally, software development in Python is generally quicker than in C++ and Python code is often 5−10 times shorter than C++ code, see https://www.python.org/doc/essays/comparisons/. The main reason to have chosen C++ over Python is speed. But by using libraries like Numpy, which is written mostly in the Fortran, Python can match C++’s speed.

The main programming language of the software will be Python. Since our client will be building on top of our written code, readability and documentation will be a main focus point during the development progress. In Python readability is one of its main features, which makes it a well-suited language. The code will be written according to the Python styleguide PEP8 (https://www.python.org/dev/peps/pep-0008/), and the wxPython style guide (http://wiki.wxpython.org/wxPython%20Style%20Guide). Documentation will be provided mainly in the form of python docstrings (https://www.python.org/dev/peps/pep-0257/). This way, one can use introspection to find information about the objects when needed and automated documentation libraries can generate the full documentation for us.

The modularity will be achieved by creating a package per module and defining its public API in the init file. This provides for a clean ’flat’ design and clear API. Key to proper modular software architecture is to avoid circular dependencies as it defeats the purpose of modularity.

Based on the software requirements and the software architecture constraints, we have made a diagram specifying the modules and their dependencies of VIPE in figure B1. The raido module provides models and parsing logic for the data obtained from Raido, the sfm merger module is meant for processing the low-end images using structure from motion and combining it with the raido data. The visualisation module contains all models, views and controllers for the VTK visualisation and the gui module is reserved for all user-interface-related logic. To minimize code duplication, a separate utilities module has been appointed to contain shared helper logic.

Fig. B1: Module relationship diagram
C Development workflow

For the development of the software, we will be using the agile development method Scrum with a sprint length of one week. We are going to use Git with the pull-request workflow on Github. The scrum backlog and other data will be integrated in Github by using its issue and release system.

To automate testing and deployment, we will be using a continuous integration service in conjunction with Github. During the development process we will write (unit and regression) tests according to the test-driven development paradigm. We have researched several continuous integration services.

Travis CI

Travis CI is the most popular Continuous integration service, has seamless Github integration and support a large amount of languages. However Travis CI is only free for open-source projects, which VIPE is not. https://travis-ci.org/

Wercker

A highly customizable development automation service and free for private Github projects. It offers a command line tool for building locally, essentially speeding up the development workflow. http://wercker.com

Shippable

Customisable workflow using a REST API and SSH build debugging. Shippable is free for one pipeline. https://app.shippable.com/

CircleCI

Free for 1500 build minutes per month and integrates with private Github projects. CircleCI provides SSH access to debug failed builds. https://circleci.com/

Additionally we will be using a code coverage service that integrate with Github. An example of this is Codecov (https://codecov.io/). We will aim to have code coverage of at least 70%.
Features VIPE

In this appendix a list with all the features of VIPE is given. This is a short overview of the software. The features are filtered on time of implementation, the features implemented first can be found at the top of the list.

- Import Raida (.3dias) project
- Import external pictures directory and match the calculated structure with that of Raida
- Help dialog (dependent on the panel shown at the moment)
- 3D point cloud visualisation
- Colour the points of the point cloud in the visualisation
- Intuitive navigation in the visualisation
- Visualise the position and direction of the camera poses based on data from the picture
- Show the picture (in the corner of the visualisation) when left-clicking on the camera position
- Display the viewpoint of the camera when user right-clicks on the camera marker
- Setting to filter points based on how many pictures they appear in
- Settings for point and camera marker size
- Save / open a VIPE project
Dockerfile

This shows the installation instructions we used to build all of VIPE’s dependencies on Linux for our continuous integration tests.

FROM ubuntu:16.04

RUN apt-get update

#install apt-get 'able dependencies
RUN apt-get install -y \
    build-essential \
    cmake \
    cmake-curses-gui \
    git \
    libinsighttoolkit4-dev \
    libxt-dev \
    make \
    ninja-build \
    subversion \
    libqt4-dev \
    python3-dev \
    python3-pip \
    graphviz \
    gcc-4.8 \
    gcc-4.8-multilib \
    libpng-dev \
    libjpeg-dev \
    libtiff-dev \
    libxxf86vm1 \
    libxxf86vm-dev \
    libxi-dev \
    libxrandr-dev \
    dpkg-dev \
    libwebkitgtk-dev \
    libjpeg-dev \
    libtiff-dev \
    libgtk2.0-dev \
    libssl1.2-dev
libgstreamer−plugins−base0.10−dev \
libnotify−dev \
freeglut3 \
freeglut3−dev \
xvfb

# Get VTK source
RUN mkdir /projects/ && git clone https://github.com/Kitware/VTK.git /projects/VTK \
   && cd /projects/VTK && git checkout v7.0.0

# Set up VTK build settings.
RUN mkdir /projects/vtk−build && cd /projects/vtk−build \\n   && cmake -DVTK_WRAP_PYTHON=ON\n      -DVTK_USE_CXX11_FEATURES=ON\n      -DVTK_PYTHON_VERSION=3\n      -DBUILD_TESTING=OFF ../VTK

# Build VTK
RUN cd /projects/vtk−build && make -j8 && make install

# Set up path so that VTK can be imported in python
ENV PYTHONPATH=/usr/local/lib/python3.5/site−packages/
ENV LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH

# Add openMVG binaries to path
ENV PATH=$PATH:/opt/openMVG_BUILD/install/bin

# Clone the openMVG repo
RUN git clone https://github.com/openMVG/openMVG.git /opt/openMVG \
   && cd /opt/openMVG && git submodule update --init --recursive \
   && git checkout v1.0

# Build
RUN mkdir /opt/openMVG_BUILD && cd /opt/openMVG_BUILD && \
    cmake -DCMAKE_BUILD_TYPE=RELEASE \
    -DCMAKE_INSTALL_PREFIX="/opt/openMVG_BUILD/install" -DOpenMVG_BUILD_TESTS=ON \
    -DOpenMVG_BUILD_EXAMPLES=ON .../openMVG/src/ && make

RUN cd /opt/openMVG_BUILD && make test
RUN pip3 install --upgrade --trusted-host wxpython.org --pre \n    -f http://wxpython.org/Phoenix/snapshot-builds/ wxPython_Phoenix