Norwegian University of Science and Technology

Project Thesis

TTK4550

Camera-Based Position Estimation for Autonomous Ships in Elevation Mapped Areas

Author
Fredrik Opeide

Supervisor
Dr. Edmund Førland
Brekke

June 3, 2019

Abstract

In this paper I present a direct visual model-based tracking approach, using edges, to track the position of a ship. A 3D triangle mesh model of a geographical area is created from heightmaps, and is rendered with OpenGL. The OpenGL renderer mimics the real camera's pose in the world, as well as the camera intrinsic parameters. Some small errors are found in the camera calibration data, varying in severity from camera to camera aboard the ship. An image is rendered at the estimated position, and by registering the rendered edges with the real image edges, the estimated pose is improved. The real image edges are found with Canny edge detection, which suffers from clutter. The edge alignment is done using projective ICP where the nearest neighbour distance is minimized, with gauss-newton as the optimization scheme. The tracking is generally successful, but it is discovered that the tracking may fail when the contours of the surrounding terrain aren't distinct enough, as there will be multiple positions at which the contours of the terrain look identical.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Most maritime navigational system use a GNSS system, such as GPS, as their primary means of navigation, with no other systems measuring the position. GPS signals are weak, and subject to interference either intentional or otherwise, meaning the service can be denied over a large area [4]. The measurements could fail completely, or even worse be given hazardous misleading data. A solution to this is to introduce a secondary position estimation system. Since humans manage to navigate based on vision and a prior knowledge about their surroundings, a digital system should be able to complete the same task. The field of computer vision is growing very quickly lately, and systems have already been developed that track a camera's movement relative to an object, based on the camera images combined with a 3D model of that object [16] [12]. Accurate 3D models are available for the terrain in many large geographical areas, and an accurate 3D model of the entirety of Norway's geography is even publicly available [8]. Computer generated imagery for virtual worlds have been studied and developed intently, due to the video gaming industry [18]. A visual positioning system could utilize this data to be able to navigate all along the Norwegian coast, completely based on comparing camera images to the rendered 3D model. By creating such a visual

position estimation system, the navigation becomes more robust, and can navigate despite GNNS failure and jamming. This is especially important for autonomous ships, on which no humans are involved in the navigation who can detect and correct failures.

1.2 Literature review

There has been much work related to model based visual tracking, and many different approaches have been proposed and tested. The tasks is finding the camera pose that best aligns the model with the image. One influential paper on the subject is the RAPID edge tracking algorithm, developed by Chris Harris in 1990 [5]. This is an edge tracker, and has been the basis for many later edge tracking systems, which have aimed to achieve more robustness. Some methods fuses the algorithm with other sources of measurements, such as image features [10], or sensor systems such as an IMU [9]. Some methods use low lever roust features in the pose optimization [13], which is further developed by tracking multiple pose hypotheses, reducing the change of a wrong tracked frame to cause complete tracking loss [17]. The point registration can be done without even cleverly creating point correspondences, but simply minimizing the nearest neighbour distance, know as Iterative Closest Points (ICP) [6]. Other tracking systems are based on region matching, rather than edges. One method renders a textured model in OpenGL, and analytically maximizes the mutual information of the view and the render to find the camera pose [1]. Another region based approach attempts to segment the model from the background, and maximizes the discrimination between statistical foreground and background appearance models, via optimization on the pose parameters [15].

1.3 Assumptions

The ship will only move within a confined known area, for which the terrain is modelled with heightmaps. The project will use an imperfect world model, which does not model buildings and vegetation, and assume that it is perfect. Only the terrain outline contours will be used for the tracking. The tracking assumes that there are no

errors in the data from the ship dataset, such as in the camera's intrinsic parameters, and its pose relative to the ship. Furthermore camera distortion effects are ignored. The ship attitude is known to the tracking system, assuming the INS is perfect, and only the position will be estimated. Also, the ship's true position will be used for the first timestep in the tracking, assuming that the starting position is known. Other than that, no positional data from the dataset can be used in the tracking.

1.4 Background and Contributions

Through Kongsberg Seatex I had access to a ship dataset, with pose and and velocity of the ship, created from fused GNSS and INS data. The data was collected with a system created and placed by Seatex employees aboard the ship Hurtigruten Polarlys. Furthermore images from several cameras aboard the ship are available, with camera calibration matrix and camera pose relative to the ship. The data was accessible through a data loading interface made by Seatex employees, who had also collected the data and calibrated the camera systems. The data is requested with a time argument as all the data is time synchronized. The terrain data is accessed through a WMS server. Seatex already had some code for accessing the terrain data as part of the ship data loader, which I modified to better suit my application. At the Seatex office I also had access to powerful computers to run simulations quickly. Arild Nøkland, and and Torbjørn Barheim from Kongsberg Seatex have been helpful with how to use the datasets and computer systems at Seatex, and for giving feedback to implementation ideas. They have had the main supervisor roles during the project. Edmund Førland Brekke is my supervisor from NTNU and was helpful during the start of the project until i was situated at Seatex, although the ideas from the beginning phase were somewhat departed from in favor of the current implementation.
A particularly helpful tutorial on rendering with OpenGL was available online [2], which taught me everything I needed to know about OpenGL and rendering.

Although similar algorithms, methods and systems as those developed in this project project have been previously implemented by others [16], they are intended

to form the basis of a more complex and robust position estimation system that will be developed in my master thesis during the spring semester of 2019. The system manages to track the position of a ship using only images taken from aboard the ship, and a custom generated 3D model of the environment, for which a rendering system is implemented. The rendering attempts to mimic the real camera pose and intrinsic parameters. Using ICP with gauss-newton for optimization of edge-image similarity between a real image and an image rendered at an estimated position the system manages to improve the position estimate, and thus track a moving target. A critical weakness of the system is proven, namely that there exists points in the world for which moving in two different directions result in the same view-change, making it impossible to track the true position with absolute certainty.

1.5 Outline

The report is organized as follows. In chapter 2 the essential theory related to the project implementation is covered. Chapter 3 describes the implementation of the system, from how models are generated and rendered, to how the image tracking algorithm works. In chapter 4 the tracking system is tested on both real and synthetic data, and the results of these experiments are elaborated. In chapter 5 the report is summarized, and possible improvements as well as future development of the project is discussed. The Appendix A and references are found at the end of the report.

Chapter 2

Theoretical Background

2.1 Coordinate systems

2.1.1 Rotation and translation

Conversion of points from one coordinate frame to another can be done using a rotation and a translation. The rotation can be represented as a multiplication with a rotation matrix, R^{3x3}, and the translation as a summation with a translation matrix/vector, t^{3x1}. To perform both the rotation an the translation as a single matrix multiplication, the matrices from the special euclidean group SE(3), 4x4, is used (shown below). To do this, the points must be converted to homogeneous coordinates by adding an additional element, increasing the dimension, which has the value 1. To convert back from homogeneous coordinates all elements are divided by the last element, and the last element is removed, reducing the dimension. A conversion of point p expressed in frame a to frame b is shown below, using a transformation matrix SE(3), where ~ represents homogeneous coordinates.

$$\tilde{p}_b = \begin{bmatrix} R_a^b & t_a^b \\ 0 & 1 \end{bmatrix} \tilde{p}_a$$

2.1.2 North East Down

North East Down (NED) is a geographical coordinate system in which the world geometry is approximated by a plane intersecting the world at a given point. The farther away from this point, the less accurate the NED coordinates become due to the curvature to the earth. In need coordinates the north direction is the first axis, the east direction is the second axis, and the vector pointing straight down is the third axis.

2.1.3 Body frame

The body frame is a reference frame fixed to an object moving a world frame, such as NED. Conventionally the body frame for a vessel has the x axis pointing forward, y axis pointing right, or starboard, and z axis pointing down. This way a rotation from NED to body is a simple rotation about the z axis. This angle is called the heading. The pose of a body in a world frame is both the position and orientation of the body frame relative to the world frame. The orientation is commonly called the attitude.

2.1.4 Euler Angles

Euler angles are a parametrization of the rotation matrix. The rotation matrix uses 9 parameters, meaning that the 3D pose is overdetermined. The orientation of a frame is therefore represented as 3 rotations in sequence about the axes of one frame required to be transformed to the second frame. The euler angles use a zyx convention, in which the frame is rotated about the x axis, the about the y axis of the resulting rotated frame, and finally about the z axis of the resulting frame from the last step.

2.2 Sensor systems

2.2.1 INS

An inertial navigation system that can consist of motion sensors (accelerometers), rotation sensors (gyroscopes) and sometimes magnetic sensors (magnometers). When the measurements are integrated and added to a previous state estimate this is called

dead reckoning. Since the accelerometer and gyroscopes don't measure position or attitude, but rather changes, the system will drift over time. Inertial measurement unit (IMU) is another name for INS.

2.2.2 Cameras

Cameras can often be modelled using a pinhole camera model, shown in 2.1. The intrinsic matrix K is used to project a point from the camera frame into the image plane. The matrix K is constructed from the camera's intrinsic parameters, focal length, principal point, and pixel density of the image. The principal point is used to move the image origin to the top left corner, while the focal length represents the distance from the lens to the image plane. Often the pixel density is included in the focal length and principal point values. Also there may be different focal lengths corresponding to the x and y directions. Where f_x and f_y are focal lengths, and c_x, c_y is the principal point, the intrinsic matrix K is

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

Pixels in the image may also be back projected into the camera frame, but the depth is then unknown, and must be provided from another source.

2.3 3D models

2.3.1 heigthmap

A heightmap is a discrete representation of the elevation of a geographical area. A heightmap is a 2D array in which the position represents a geographical location, while the value in the array represents a height. A heightmap must have an origin point, as well as a scale, to tie it to the correct geographical location. A model of an area's elevation is called a digital elevation model (DEM), which can model the area including buildings and vegetation as a digital surface model (DOM in Norwegian),

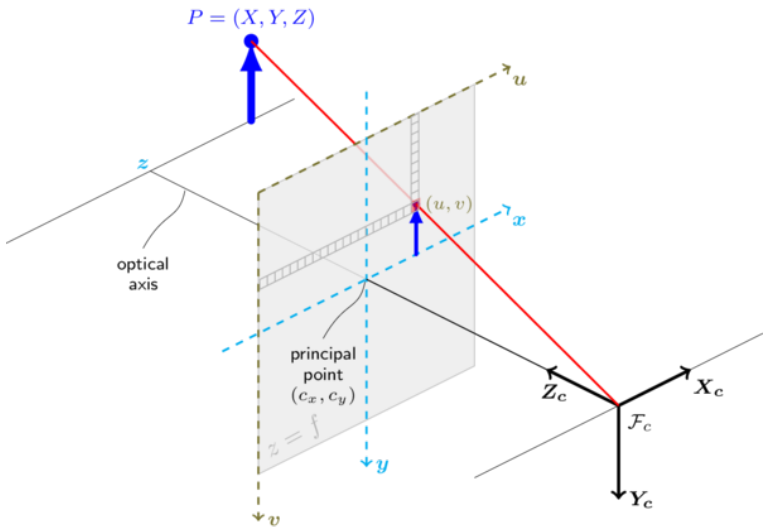


Figure 2.1: Pinhole camera model. Image from the OpenCV documentation

or a digital terrain model (DTM) which only models the underlying terrain, the earth itself.

2.3.2 triangle mesh

A triangle mesh is a common way to represent object surfaces used in computer graphics. It is comprised of vertices, which are positions in a space, and triangle faces which are constructed from three vertex points. Conventionally the vertices forming a triangle face is listed such that going from vertex to vertex forms a clockwise movement when viewed from the outside of the surface. Following this convention, it is known in which order to do vector cross products to calculate the surface normal pointing outwards.

2.3.3 the .obj file format

The Wavefront .obj file format is a file format used to specify a 3D model. It was first developed by Wavefront Technologies, and has since been adopted by many graphic application developers. The format specifies a list of vertices, as well as list of any data that can be associated with the vertices, such as vertex normal vectors. Then a list of faces, specifying which vertices constitute the triangle face, and what data is associated with each vertex. An entry into the vertex list is specified as "v x y z", where v is just the letter v, and x, y and z are the coordinates of the vertex. An entry in the normal vector list is specified as "n x y z", where n is just the letter n and x,y,z is the normal vector. A triangle face is then specified by "f v1/n1 v2/n2 v3/n3", where f is just the letter f, while v1,v2 and v3 are indices corresponding to positions in the vertex list, indicating which vertices make up the triangle. While n1,n2,n3 are indices in the list of normals.

2.4 Rendering

2.4.1 OpenGL

OpenGL is an API for rendering 2D and 3D graphics using a computer's graphical processing unit (GPU). It is cross platform (Ubuntu, Windows, etc.) and cross language (python, c++, etc.). GLFW is a window context handler, which can be used to handle some of the low level OpenGL tasks. An OpenGL program will typically load a polygonal mesh into buffers, and bind a compiled shader to the rendering process, and then draw the buffer contents into a buffer, and finally switches buffers with the screen context to display the view.

2.4.2 Shaders

A shader is comprised of a vertex shader, and a fragment shader. The vertex shader transforms each vertex to their correct position within the OpenGL camera view, and the fragment shader specifies how the view is drawn. When rendering a triangle

mesh a the inside of triangle face is split into smaller fragments in a process called rasterization. The normal vector and other values of the fragment is interpolated from the triangle vertices according to the fragment position within the triangle.

2.4.3 VBO indexing

VBO indexing is a rendering method in OpenGL in which a tringle mesh is represented by a list of vertices, and a list of triangle face indices, where the indices correspond to which vertices make up each face. These arrays can be loaded directly into a vertex buffer object (VBO) and an index array, and rendered with a glDrawElements call with GL_TRIANGLES and the index array as arguments. This can save on memory, as vertices can be indexed multiple time, and that way be reused.

2.5 Edge detection

2.5.1 Sobel

Sobel edge detection is done by convolving an image withe sobel operators, which will approximate the x and y gradients of an image. The absolute value of the gradient is then used to determine if there is an edge present in the image or not by comparing it with threshold. The Sobel operators for the x and y directions are

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

2.5.2 Canny

Canny edge detection is a multi stage detection algorithm. First it smooths the image with a gaussian filter. Then it finds the intensity gradients. Then it does non-maximal suppression to remove pixels which most likely are not edges. Then hysteresis thresholding is performed to decide which edges to keep and which to discard. This final step uses two thresholds, max and min. Those above max are definitely strong edges and kept, and those below min are discarded. The remaining edges, between min and max, are kept if they are connected to a strong edge, the rest discarded.

2.6 Optimization

2.6.1 Nearest neighbours, KD-trees

If you have two sets of points, a and b, and you want for each point in a to find the closest point in b, this is called a nearest neighbour search. This is typically computationally expensive, but can be sped up using a data structure known as a KD-tree. It is a type of binary partitioning in K dimensional space. In the case where a and b contain 3-dimensional points, a KD-tree can be constructed for b, which will then be a 3 dimentional tree. For each point in a search in the KD-tree is much quicker than what an exhaustive search would have been.

2.6.2 ICP

ICP stands for Iterative Closest points, and is an algorithm used to register two point clouds. That means finding the rotation and translation that gives the best overlap between the point clouds. Typically the overlap match is measured by the root mean square nearest neighbour distance between the points of the clouds. It is an iterative method, and for each iteration the nearest neighbour distance is minimized through some optimization scheme on the rotation and translation, and then new nearest neighbours are calculated with the rotated and translated cloud. One cloud is static and is represented by a KD-tree for efficiency.

2.6.3 Gauss-newton

Gauss newton is an optimization algorithm used to solve non-linear least squares problems. The error function is presented as a sum of squared residual errors, R, that are functions of some parameters β to be optimized. The jacobians, J of the residuals with respect to their parameters are calculated, and a step in the parameters that further minimizes the error for a step k is calculated as such

$$\beta_{k+1} = \beta_k - (J_k^T J_k)^{-1} J_k^T R(\beta_k) \tag{2.2}$$

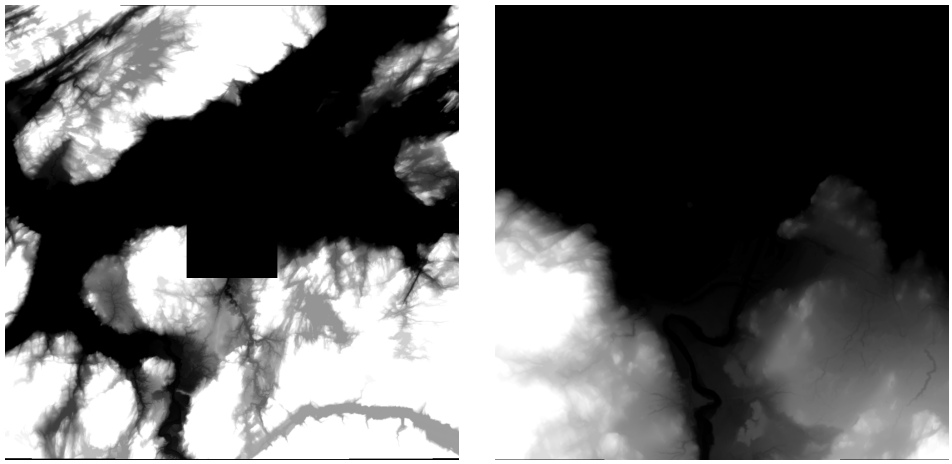
Chapter 3

System Implementation

3.1 World Model Generation

3.1.1 Acquiring Terrain Data

When modeling a huge world the entire model cannot be at the highest resolution, due to memory limitations. Rendering a large area, all with 1m resolution, would quickly become unmanageable, as the memory requirements would grow exponentially. Therefore, some kind of level of detail (LOD) must be implemented. While close objects remain at high resolution, objects that are far away can be lower resolution without impacting their appearance. There exists many methods to exploit this. One widely used method, chunked LOD, splits the map into chunks, and creates different resolution versions of these chunks. When moving around the world the chunks closer to the camera will have their higher resolution versions loaded.[19] We assume in this project that the system will only operate within an area confined to the Trondheim Fjord, and only near Trondheim city will it be close to the shore. Therefore height data for Trondheim city is downloaded at higher resolution, while the rest of the fjord is downloaded at lower resolution. Since we want to combine the models made from these heightmaps, the overlapping area must be removed from the larger model. The terrain data is downloaded in the form of heightmaps from a wms server, wms.geonorge.no, with specified dimensions, scale and elevation model type. The heightmap data can be requested at three different scales, 50m, 10m and 1m, signifying the distance between the height measurements, i.e. meters/pixel. Furthermore there are two different types of elevation data available. There is a digital surface model (DSM) and a digital terrain model (DTM). Since we want a rendering similar to reality, DSM is more suited, as it models buildings and vegetation. However this is only available in 1m scale, and it would take an unreasonable amount of time to download and process, so for this project DTM is used to model the world. 1000x1000 pixels is used, as it's a good trade-off between file size (processing time) and world size. At 1m scale the heigthmap only spans 1km, too small to cover the Trondheim area, and so 10m and 50m heightmaps are used, and it can be seen from fig 3.1 that they cover the areas we want to model.



(a) Scale 50, center cut

(b) Scale 10, center uncut

Figure 3.1: DTM heightmaps of Trondheim created at different scales, where (b) is made to fit into (a)'s cut center

3.1.2 Generating an .Obj Model

From the heighmaps acquired in 3.1.1, we want to create 3D-models in a file format that can be loaded and rendered. The Alias/WaveFront Object (.obj) format is chosen, as it is popular, simple and fairly flexible. The format defines a polygonal mesh comprised of vertices, polygonal faces, and normal vectors for the vertices. vertices are positions in whatever dimension the model is, usually 3D or 2D. Furthermore texture data or in fact any sort of data can be associated with the vertices, which it what makes it a flexible format [3].

A triangle mesh will be created from the heightmap, as a triangle is the simplest polygon. A lot of algorithms exists to create efficient triangle mesh representations of terrain models using triangulated irregular networks (TINs), removing redundant information and reducing the number of polygons. These methods are however computationally expensive and have complex implementations [11]. The alternative is a triangulated regular network (TRN), where every pixel in the heightmap is represented by a vertex,

and all triangle faces have the same size and shape. The easier construction comes at the cost of producing more polygons.

In the rendering we would like flat triangle faces to have the possibility to model sharp edges. Since the rendering will interpolate the surface normal within a face that means that the vertices comprising a triangle face must have identical normal vectors. The only way of doing this in the obj-format is duplicating indices when they are part of multiple faces, so that each has a unique normal vector. A face is comprised of three vertices and three associated normal vectors. A triangle face in .obj is defined as the indices pointing to a position in a list of vertices and a list of normals. The vertices and normals that are pointed to make up the triangle face. To create this we loop over the pixels in the depthmap, creating two triangles from the square defined by the pixel and the pixel a row and column over, illustrated in 3.2. The metric vertex xy coordinates are calculated by scaling the pixel coordinates with the heightmap scale (m/pixel), and adding an offset so that the center of the heightmap is the origin. The matrix vertex z-coordinate is simply set as heightmap value. Since the pixels coordinates are row-col, and the depthmap is aligned north-up east-right, the x-axis will point south, and the y axis will point east. The vertices are added to the vertex list, and faces created from referencing the vertex list such that the sequence is clockwise, shown in 3.2. Normal vectors are created from the cross product of the vectors between the vertices, and are stored and referenced the same way as the vertices themselves. We want to remove water, as it adds nothing useful to the model. If all the vertices of a triangle face have height 0, it is in the open ocean, and is not added to the model. The implementation becomes slightly too convoluted to write pseudo code, and so the code is added in the appendix.
Since .obj was chosen we can use popular tools to visualize the generated mesh. We load the models into Meshlab. The 10m model is shown in figure 3.3 and the 50m model in figure 3.4. Verifying that the metric vertex positions are correct, the models are simultaneously loaded in Meshlab, shown in figure 3.5, where it can be seen that the models fit each other. We see that the ocean is removed from the model, as intended. An illustration of the triangle mesh itself is shown in figure 3.6, where the view is zoomed and the triangle borders are visible.

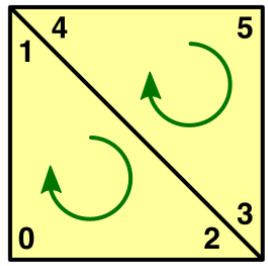


Figure 3.2: vertex indices for a square turned triangle mesh

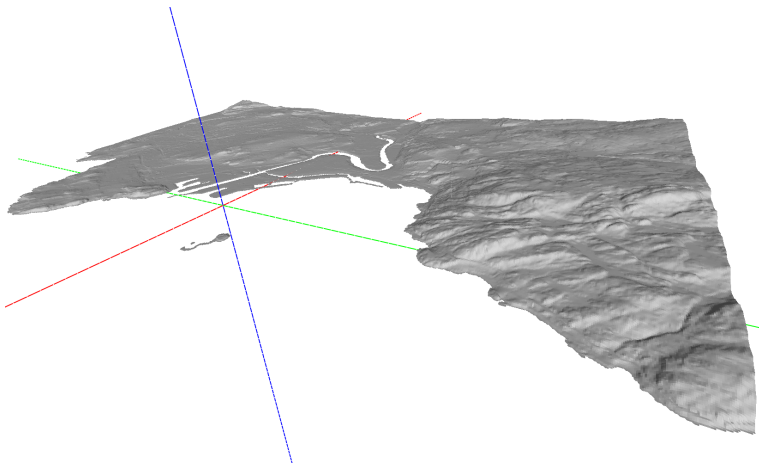


Figure 3.3: Mesh created from DTM heightmap with scale 10, visualized in Meshlab

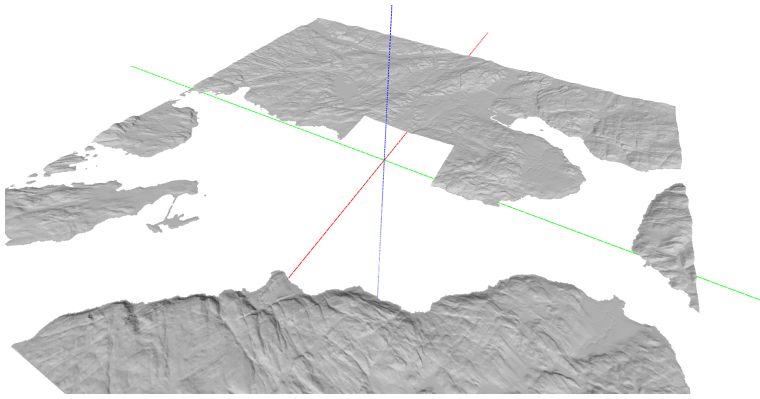


Figure 3.4: Mesh created from DTM heightmap with scale 50 and cut center, visualized in Meshlab

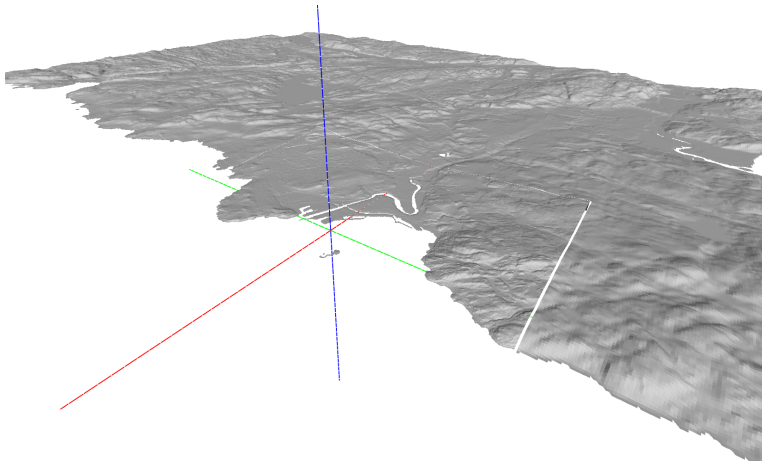


Figure 3.5: Zoomed Meshlab visualization of the combined meshes with scale 50 and 10

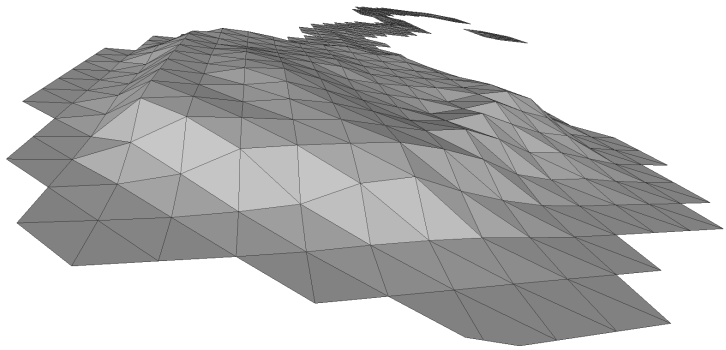


Figure 3.6: Meshlab visualization of mesh model of DTM with scale 10. Zoomed in on Munkholmen

3.2 World Model Rendering

3.2.1 Loading and Combining .Obj Models

In section 3.1 it was shown how multiple terrain models at various scales are generated, and how they are made to fit together. Now we want to combine these models, and render them. All the model files are parsed, and their data loaded into arrays. Since the models were created individually, the vertex and normal indexes start at 0 in each file. This means the files cannot be combined by simply adding together all the data, as the indexing would conflict. Therefore an index offset is added to each file as they are parsed. The index offset is the cumulative number of indexes in the previously parsed files. The vertex, normal and index arrays from each file can now be concatenated in the sequence they were parsed. Since it takes a lot of time to parse a model file, after they are parsed they are cached as binary .npy files, so they can be almost instantly loaded the next time. The pseudo code for this is written in 16. The file parsing itself is straight forward as it just reads each line in the file and adds the data to either the vertex- normal- or index-array. The code for the file parsing and model combining is added in the appendix A.2. The vertices, normals and indices are simple numpy arrays, that will later be loaded into attribute buffers and an index buffer respectively, detailed in the next section 3.2.2.

Algorithm 1 Loading .obj files

Require: $objFiles$
  $indexOffset \leftarrow 0$
  $verticesAll \leftarrow EMPTY$
  $normalsAll \leftarrow EMPTY$
  $indicesAll \leftarrow EMPTY$
  for all $file \leftarrow objFiles$ **do**
    if $hasCache(file)$ **then**
      $vertices, normals, indices \leftarrow getCache(file)$
    else
      $vertices, normals, indices \leftarrow parseFile(file)$
      $createCache(file, vertices, normals, indices)$
    end if
    $verticesAll \leftarrow append(verticesAll, vertices)$
    $normalsAll \leftarrow append(normalsAll, normals)$
    $indicesAll \leftarrow append(indicesAll, indices + indexOffset)$
    $indexOffset \leftarrow indexOffset + size(vertices)$
  end for
  return $verticesAll, normalsAll, indicesAll$

3.2.2 Rendering with OpenGL

3.2.2.1 Initialization

In the previous section 3.2.1 it was shown how .obj files are parsed and combined into a single model, with a vertex-, normal- and index-array. Now this data will be rendered with a custom OpenGL renderer. The GLFW library is used for handling a lot of low-level tasks related to the rendering, such as specifying the window-context. Upon initialization the renderer is given the model files, the window size, and the camera intrinsics. First the model files are loaded. This project uses VBO indexing, and so two attribute buffers are created and filled with the vertex- and normal-arrays. An index buffer is created and filled with the index-array. Then a window context is created with GLFW, setting background color to black, and setting the window size. The window size is set to the same size as the real camera images. Depth tests are enabled by calling glEnable(GL_DEPTH_TEST). This is necessary to later extract a

depth map for the rendering, shown in section 3.2.3. Then the fragment and vertex shader files are compiled into a shader program, which is then linked to the OpenGL context. Finally uniforms, used to send variables to the shader, are initialized. The uniforms will hold the matrices needed to specify the camera view. The shader is described in the next section, 3.2.2.2.

3.2.2.2 Rendering

Since we want to be able to track a real image by analyzing differences to an image rendered at an initial guess position, the images must have some mutual information. Though the real and rendered images are of different modalities, they must be structurally similar in terms of image gradients, as this project implements edge tracking. This challenge consists of two parts.
The first is rendering at the location in the virtual world that corresponds to the exact pose of the ship-camera. When rendering is called, it is given four matrices, each an SE(3) transformation matrix. The model matrix M_{model} defines the transformation from model coordinates to NED coordinates. As mentioned in section 3.1.2, in the model x points south, y east, and z up, and so a rotation of π around the y-axis will align the frames the model to NED. The body matrix M_{body} defines the ship body frame's position and attitude in NED coordinates with the model center as origin. The camera matrix M_{cam} defines the position and attitude of the camera-mount relative to the ship body frame, with x forward and z down. The fourth input is the view matrix M_{view}, which defines the transformation from the camera mount to the camera frame, in which x points down and z into the view. These matrices, when multiplied, define the pose of the camera frame in the world.

The second challenge of 'realistic' rendering is accurately mimicking the real camera intrinsics in the rendering process, e.g. the field of view. Camera distortion effects are ignored, even though they can have a visible effect, especially on the edges of an image. The cameras are modelled with a pinhole camera model, which is a perspective projection. The intrinsic matrices for the cameras have already been estimated by employees at Kongsberg Seatex, and is given to the renderer in the initialization

process. A perspective projection matrix, P for openGL rendering is defined as in eq 3.1, with *near* and far clip planes, $fovY$ vertical field of view angle and the *aspect* ratio, width/height, of the image [14].

$$P = \begin{bmatrix} \frac{1}{aspect*tan(\frac{fovY}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(\frac{fovY}{2})} & 0 & 0 \\ 0 & 0 & \frac{far+near}{near-far} & \frac{2*far*near}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3.1}$$

The near clip field is set to 1, and the large clip field to the size of the world, so that nothing is clipped for being too far away. $fovy$ and *aspect* can be defined in terms of the camera intrinsics, assuming the principal point is at the image center.

$$fovy = 2tan^{-1}(\frac{cy}{fy}) \tag{3.2}$$

$$aspect = \frac{cx}{cy} \tag{3.3}$$

Where the camera intrinsic matrix is

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

Finally, OpenGL does not follow the camera frame convention of x down z into the frame, and must be told to view the world in this way. A library called pyrr is used to create a matrix converting the opengl view frame to the conventional camera frame, M_{gl}. Combining the transformation from world frame to camera frame, and the projection from camera frame to image plane, the matrix MVP is defined as in eq 3.5

$$MVP = P * M_{gl} * M_{view} * M_{cam} * M_{body} * M_{model} \tag{3.5}$$

This MVP matrix is then passed to the shader as a uniform, telling it how to render an image from the model at the desired pose, with the desired camera parameters. Specifically it will be used in the vertex shader, where it will simply be multiplied with all vertices that the shader is given. The vertices are first transformed to homogeneous coordinates. Since we will do edge tracking, and vegetation and buildings are not part of the model, the only reliable edges are the mountain and shore contours. Therefore the rendering can set the color of all vertices to anything but the background color, which is black. In the fragment shader, the color of all fragments is set to red, an arbitrary choice. This will render the model red with black background, making the mountain and shore contours easily extractable, as shown in section 3.3.1. An example of a mask rendering is shown in fig 3.7. To render the model the normal and vertex data is loaded into GL_ARRAY_BUFFERs, and the index data is loaded into the GL_ELEMENT_ARRAY_BUFFER. glDrawElements is then called with the GL_TRIANGLES argument as well as the size of the index array. The rendered image is then accessed trough a glReadPixels call to GL_RGB with the window size.

3.2.3 Retrieving True Depth Map

In the coming chapter 3.3.1 pixels are back projected into the 3D camera frame. To do this depth information about the pixels are needed. OpenGL uses a z-buffer during rendering, which contains depth-data of each rendered pixel, but these depth values are not metric. Using knowledge of the OpenGL projection matrix, the depth-buffer is converted to a map of each pixels true depth in meters using eq 3.6, where D_{GL} is the z-buffer data, D_{metric} is the metric depth map, with $near$ and far clip planes. [14] The final depth value is the distance from the projective plane to the point, i.e. the z-coordinate in the camera frame. The z-buffer is accessed though a glReadPixels call to the GL_DEPTH_COMPONENT, after the rendering with glDrawElements. Before applying the transform the buffer data is loaded into a numpy array using numpy's frombuffer function.

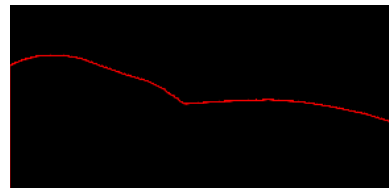
$$D_{metric} = \frac{(\frac{near*far}{near-far})}{(\frac{D_{GL}-far}{far-near})} \tag{3.6}$$

3.3 World Model Tracking

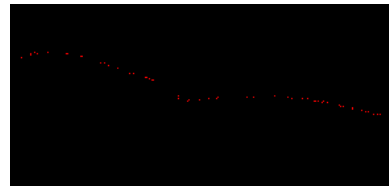
3.3.1 Contours in 2D and 3D

To do edge matching, first the terrain-contours of the synthetic and real views must be calculated. The tracking will be sensitive to outliers, so we want only the strongest contours, which are likely to be found at the transition between the terran and the sea and sky. We could have used a normal map rendering as in 4.1 to get contours from the terrain itself, but this would have given many contours that would not be found in the real image, outliers. Furthermore since the model is DTM and not DOM the model might be too differnt from reality to have accurate edges within the terrain. The rendering gives a semantic mask, where terrain is red, and everything else is black. The only edges present in the rendering are the terrain contours themselves, therefore a simple method such as Sobel edge detection is sufficient. The Sobel operators are convolved with the red image channel to approximate the image derivatives, and for all pixels with gradient magnitude greater than 0, the image-frame coordinates are stored. The result is shown in figure 3.7. These edge pixels, when expressed in image frame coordinates, are called edgels. The implementation uses the python imaging library (PIL), where images are accessed with the a row-column convention. In the image frame the origin is still top left corner as with row-column, but the axes are flipped, meaning that x goes right and y goes down. A conversion from row-col to x-y image frame coordinates is done when receiving an image.

(a) mask



(b) edges (zoomed in)



(c) sampled edges (zoomed in)

Figure 3.7: The terrain mask rendering (a) subjected to Sobel edge detection (b) and then sampling (c). (b) and (c) are zoomed in on the top left corner

In order to later re-project the rendered contour with some world-movement, the edgels must first be back-projected from the image plane into the 3D camera frame. For this the camera intrinsic matrix and the calculated true depth map from 3.2.3 is used. In the depth-map, the pixel locations of the depth-values do not perfectly correspond to the view. The reason for this in unclear, but it results in some pixels erroneously being assigned the depth value of a nearby pixel. Some of the pixels at edges of objects may then be assigned the background depth, which is the maximum depth. To remedy this the depth-value of each pixel is set to the minimal depth-value in the neighbouring pixels. This is only done for the edgels, as they are the only pixels that will be back-projected.

An edgel, p_i, is transformed to homogenous coordinates, $\widetilde{p}_i$, and then multiplied with the inverse of the camera intrinsic matrix, K^{-1}, to produce a 3D ray with depth 1 in the camera frame. The edgel-ray is multiplied with it's depth-value, D_i, to give the 3D position of the edgel in the camera frame, x_i. This operation is expressed in equation

3.7.

$$x_i = D_i K^{-1} \widetilde{p}_i \tag{3.7}$$

The real image, due to textures and lighting, have many edges that are not part of the terrain contours. A more involved edge detector, the canny edge detector, is selected for the real images. Values of 100, 200 were found to be a good tradeoff between suppressing undesired edges, while still detecting the terrain contours. It is most important that there is little noise near the real contours. The Canny values suppress edges due to clouds, which would otherwise severely clutter the real contour lines, and there is little clutter in the water near the shore, although buildings cause clutter above the shoreline.

Figure 3.8: An image from one of the ships's cameras

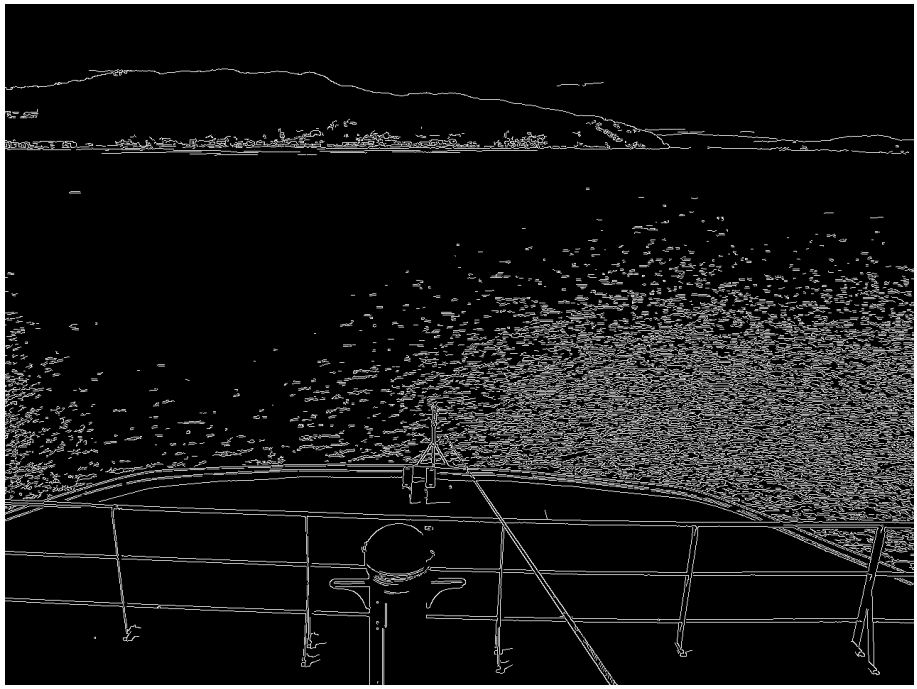


Figure 3.9: An image from one of the ships's cameras, after canny edge detection

3.3.2 Analytical Gauss Newton for Projective Registration

The projective registration algorithm is given a set of 3D edgels, and their corresponding desired positions in the image plane. Their desired positions are the nearest points on the real image contours. We want to find the camera-translation that minimizes the square of the re-projection error. This minimization problem is formulated as

$$t* = \arg\min_t S(t) \tag{3.8}$$

$$S(t) = r(t)^T r(t) = \sum_i r_i(t)^T r_i(t) \tag{3.9}$$

Where t is a translation in the camera-frame, and the residual errors, $r_i(t)$, are

$$r_i(t) = p_{nn,i} - Proj(x_i - t) \tag{3.10}$$

x_i is the 3D positions of the i'th edgel in the camera-frame, and $p_{nn,i}$ is the corresponding desired image-plane position, the nearest neighbouring point on the true-image contours. The projection function $Proj(\cdot)$, represents a perspective projection with the camera intrinsic matrix K, and a conversion from homogeneous to Cartesian coordinates. The residual error for one point correspondence, i, becomes

$$r_i(t) = \begin{bmatrix} p_{nn,i,0} \\ p_{nn,i,1} \end{bmatrix} - \begin{bmatrix} \frac{(K(x_i - t))_0}{(K(x_i - t))_2} \\ \frac{(K(x_i - t))_1}{(K(x_i - t))_2} \end{bmatrix} = \begin{bmatrix} p_{nn,i,0} \\ p_{nn,i,1} \end{bmatrix} - \begin{bmatrix} \frac{f_x(x_{i,0} - t_0)}{x_{i,2} - t_2} + c_x \\ \frac{f_y(x_{i,1} - t_1)}{x_{i,2} - t_2} + c_y \end{bmatrix} \tag{3.11}$$

Where the focal length values f_x and f_y, and the principal point, c_x and c_y come from the camera intrinsic matrix

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.12}$$

The problem is formulated as a non-linear least squares minimization problem, and so a non-linear iterative least squares solver is implemented, Gauss-Newton [7]. Using the Gauss-Newton algorithm, an update to t at iteration number k that further minimizes

3.9 is calculated by

$$t^{(k+1)} = t^{(k)} + (J^{(k)T} J^{(k)})^{-1} J^{(k)T} r(t^k) \tag{3.13}$$

Where the elements of the jacobian $J^{(k)}$ are the partial derivative of the i'th residual with respect to the j'th element of t

$$J_{i,j}^{(k)} = \frac{\partial r_i(t)}{\partial t_j}\bigg|_{t^{(k)}} = \frac{\partial(\begin{bmatrix} p_{nn,i,0} \\ p_{nn,i,1} \end{bmatrix} - \begin{bmatrix} \frac{f_x(x_{i,0}-t_0)}{x_{i,2}-t_2} + c_x \\ \frac{f_y(x_{i,1}-t_1)}{x_{i,2}-t_2} + c_y \end{bmatrix})}{\partial t_j}\bigg|_{t^{(k)}} \tag{3.14}$$

Meaning the jacobian $J_i^{(k)}$ related to each 3D-2D correspondence i is

$$J_i^{(k)} = \begin{bmatrix} \frac{f_x}{x_{i,2}-t_2^{(k)}} & 0 & \frac{-f_x(x_{i,0}-t_0^{(k)})}{(x_{i,2}-t_2^{(k)})^2} \\ 0 & \frac{f_y}{x_{i,2}-t_2^{(k)}} & \frac{-f_y(x_{i,1}-t_1^{(k)})}{(x_{i,2}-t_2^{(k)})^2} \end{bmatrix} \tag{3.15}$$

The algorithm:

Algorithm 2 Gauss-Newton for position estimation

Require: $GNIterations$
Require: $edgels3D$
Require: $edgels2D$
Require: $intrinsics$
  $position \leftarrow [0, 0, 0]$
  for $i := 0$ **to** $GNIterationss$ **do**
    $residuals \leftarrow edgels2D - project(edgels3D - position, intrinsics)$
    $jacobians \leftarrow calcJacobians(edgels3D, position, intrinsics)$
    $position_{step} \leftarrow (jacobians^T jacobians)^{-1} jacobians^T residuals$
    $position \leftarrow position + position_{step}$
  end for
  return $position$

3.3.3 Pose Estimation with Iterative Closest Points

We will estimate the position of the ship by matching the contours of a real image from the true position, and an image rendered at a guessed position. This is achieved by minimizing reprojection error with an approach like Gauss-newton, as presented in 3.3.2. The reprojection error calculated for a set of correspondences between the contour points in the rendered and real image. There are multiple ways of finding these correspondences. An edge image has few distinct features, and the sensitivity to clutter makes a feature based matching approach unsuited. The corresponding real-edgels for each edgel are therefore simply selected using nearest neighbours, implemented with KD-trees for fast performance. These matches will most likely be incorrect, but iteratively minimizing the nearest neighbour distances while for each iteration calculating new nearest neighbours, is shown to be suitable for pose estimation under many circumstances [12]. This is a basic iterative closest points (ICP) method, and the pseudocode is written in alg 3. While the nearest neighbours are recalculated at each iteration, the guessed 3D edgels are not. Going one step further and iteratively doing ICP while recalculating the 3D edgels between runs is covered in the next section, 3.3.4. There are potentially many local minima when doing ICP. It is important to remember that the algorithm optimizes the view similarity, and that there may be multiple poses resulting in very similar looking views, especially when all that is seen by the algorithm is contour points.

Algorithm 3 Pose estimation with ICP

Require: $ICPIterations$
Require: $GNIterations$
Require: $edgelsInit$
Require: $edgelsTrue$
Require: $intrinsics$
Require: $depthMap$
  $position \leftarrow [0, 0, 0]$
  $edgels3DInit \leftarrow backproject(edgelsInit, intrinsics, depthMap)$
  for $i := 0$ **to** $ICPIterations$ **do**
    $edgels3D \leftarrow edgels3DInit - position$
    $edgels \leftarrow project(edgels3D, intrinsics)$
    $edgelsNN \leftarrow nearestNeighbours(edgels, edgelsTrue)$
    $position_{step} \leftarrow gaussNewtonPosition(edgels3D, edgelsNN, GNIterations)$
    $position \leftarrow position + position_{step}$
  end for
  return $position$

3.3.4 Tracking Position Over Consecutive Frames

This section describes how the algorithms and methods of the previous sections are put together to track the position of a ship using images taken from aboard that ship. First the world model files are created for the operating area as described in section 3.1. One of the cameras is chosen to be used in the tracking, and it's intrinsic matrix is loaded from the dataset. The renderer described in 3.2 is initialized using the intrinsic matrix. The position in the dataset is in lat long format, but the optimization is based on meters. The NED offset from the origin, whose lat long coordinates are known, is calculated using a utility function associated with the data set. Together with the true ship position the ship attitude is also loaded. Next a true view image is acquired, either loaded as a real image from the dataset, or as an artificial image rendered using the true ship pose, as well as the camera pose. Then the initial guess position for the time step is used, with the known camera pose, to render a guessed view. The real image edges are calculated using canny edge detection, and the edges for the guessed view are created using Sobel, as described in 3.3.1. The edge images of the real and

guessed view are then passed to the ICP algorithm, described in 3.3.3, which refines the position estimate by finding a position for which the real and guessed view are more similar. The system has no relocalization capabilities, so the estimated position is initialized to the true position for the first time step, assuming that the ship knows where it starts. The initial position guess for subsequent steps is the refined position from the last step. The systems assumes that the attitude i perfectly estimated from the INS, and uses the true ship attitude at the initial guess pose. Furthermore no filtering techniques of any kind have been implemented. This process of improving the position estimate by comparing real images to a render at the guessed position is run at each time step, making the estimated position follow the path of whatever is taking the images. A figure representing the process is shown in figure 3.10.

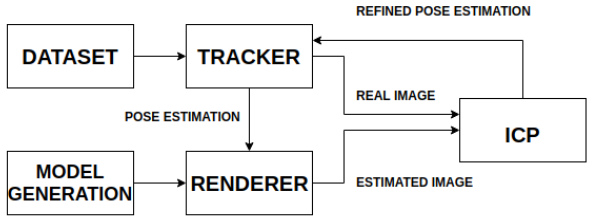


Figure 3.10: Simplified system overview, showing how the tracking process works. Edge detection is included in the ICP block

Chapter 4

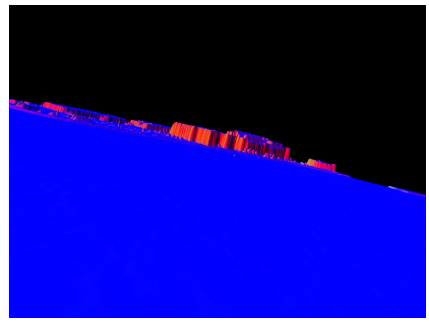
Experiments and Results

4.1 Model and Rendering

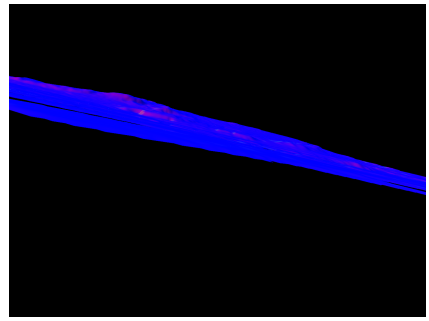
4.1.1 model

By setting the color of a rendered vertex equal to the vertex normal, the surface color becomes dependent on the direction the surface is facing. This reveals more information than the mask rendering, and is used to inspect the rendered models. In figure 4.1 two renders are compared to a real image at the same location. One render uses the surface model, DOM, while the other render uses the terrain model, DTM. Due to the problems with getting large DOM models, detailed in 3.1.1, only a small area is renderable, an if the ship were to move further away from the center, it would quickly be moving into a completely unmodeled area. It is still apparent, even though background mountains are not modelled, that the DOM model rendering gives a view that is much more similar to the real image. The DTM rendering, while it captures the shore line and mountain contours gives a very undetailed view, with few features tying it to the real image. The meshing algorithm seems to have kept the sharp edges of where the buildings' roofs meet their walls. However we can see that the water is still represented in the model. This is because the DOM heightmap is created using different measurement techniques, and the sea is not exactly at height 0 in the model. This causes the DOM model to be much larger than the DTM version, as there is no decimation on the sea-polygons, even though it is approximately a flat surface.

(a) real image



(b) DOM render



(c) DTM render

Figure 4.1: A real image (a) compared to normal-map rendering using surface model (b) and terrain model (c)

4.1.2 rendering

The rendering is designed to be as similar to the real view as possible. This means accurately modeling the camera pose, as well as modelling the camera itself. There are multiple cameras on the ship, for which the dataset specifiy the poses relative to the ship, as well as the camera intrinsic parameters. The result of using these values, and a pinhole camera model is shown in figures 4.2 and 4.3, where the rendering using different camera specifications is overlain the real image locations. In fig 4.2, the camera is at the fron of the ship, and it can be seen that the rendering matches the

real view quite well. The center matches particularly well, and it can be inferred that the extrinsics, pose in the world, is quite good. It is also seen that the match becomes poorer at the edges of the image, meaning that the intrinsic camera values are not quite right. It appears that a the field of view for the real image is wider than for the rendering. Furthermore the rendering has ignored distortion effects, whose effect may be significant. For fig 4.3 the same may be said for the intrinsics as for the from camera, but here the camera attitude is obviously wrong. A slight rotation would result in a much better match. The ship attitude is ruled out as the causing effect, as the offset is different for each camera, and also by inspecting a sequence of overlay images, for the offset is constant. Distortion effects are also ruled out as the offset is the same over the entire image. This means that there is a slight error in the mounting angles of the cameras. The camera position aboard the ship, as well as the ship attitude and position seem to be captured accurately by the model and the rendering system, as it can be seen in the right side of the image in fig 4.3 that the islet, Munkholmen, which is relatively close to the camera, is rendered correcly, beside the discussed camera rotaion error.

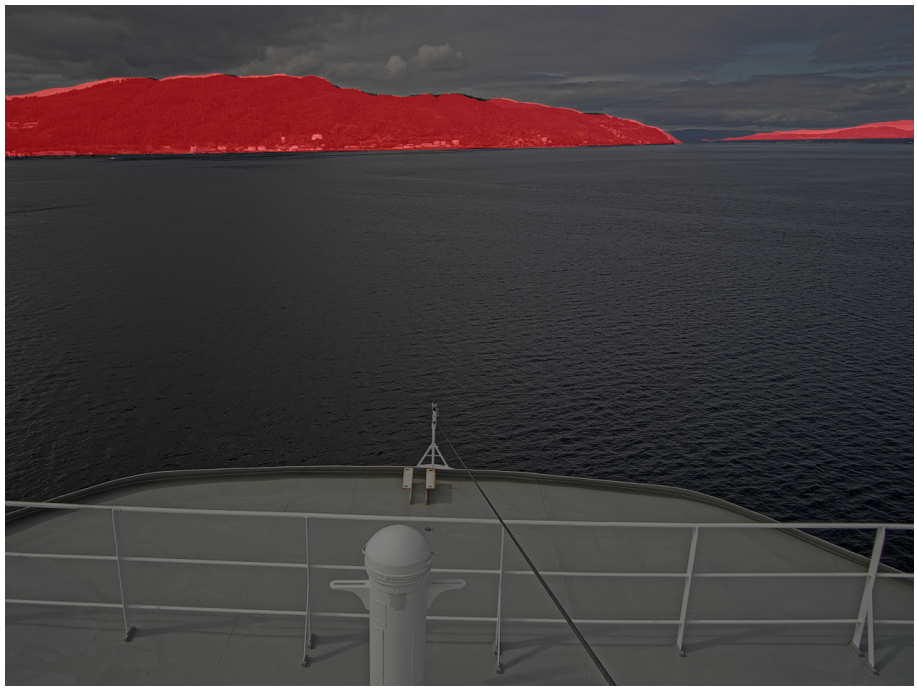


Figure 4.2: Real image from camera 1 from cluster 0 with rendering overlay

4.2 Tracking

4.2.1 Tracking Synthetic Images

4.2.1.1 Stationary Target

We want to inspect the convergence properties of the position estimation system presented in 3.3.4. In stead of using real images for the tracking we test the system on synthetic data. The image used at the view for the true position is a rendering at the true position. To track a stationary target the true position and true view image is simply held constant for every time step. The system is doing 1 ICP cycle per second, each doing 100 iterations of calculating nearest neighbours and doing 1

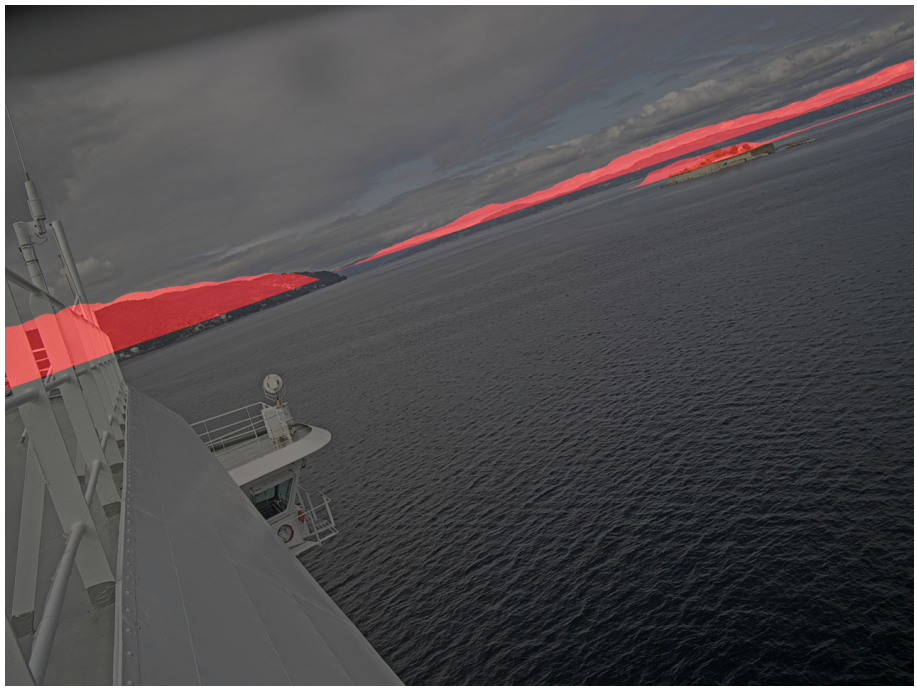


Figure 4.3: Real image from camera 0 from cluster 1 with rendering overlay

iteration of projective registration for each correspondendce set. The true position is set approximately 5km off the coast, the the initial guess position is a bit over 1km away from the true position. The tracked position in the fjord is shown in 4.4, as well as a zoomed in version in 4.5. A visualization of the tracked view is shown in 4.7, where it can be seen that the guessed view and true view become more and more similar. After 50 iterations the projective registration is pixel perfect, and the positional error about 7m, which is seen in the error graph in fig 4.6.
Looking zoomed in on the estimated positon, is is clear that the ICP algorithm manges to move towards the correct position for most of the steps. This indicates that the tracking system might have a lot to gain from implementing a line search, first calculating a direction and then testing view correspondence along that direction. This assumes

that the world points at which the contours originitate do not change too drastically between different positons. A line search culd be implemented in the Gauss Newton optimization, but the potential here is limited, as the algorithm operates using the same edge correspondence set, which won't give a good match unless really close to the true position. Therefore the line search could be implemented in the ICP itself, calculating new nearest neighbours for each match evaluation along the search direction.

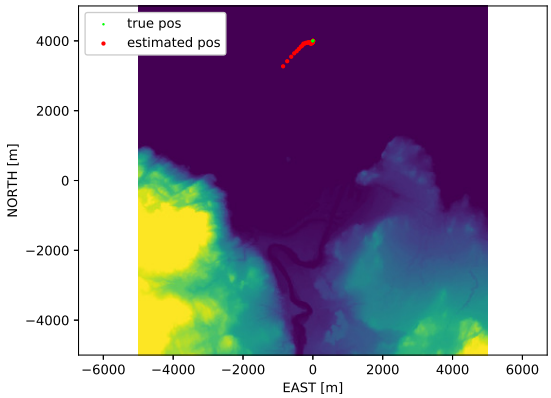


Figure 4.4: Plot of position convergence when tracking stationary target with synthetic view

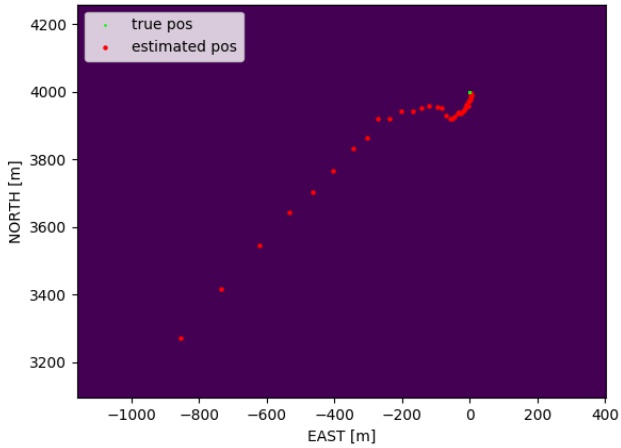


Figure 4.5: Zoomed plot of position convergence when tracking stationary target with synthetic view

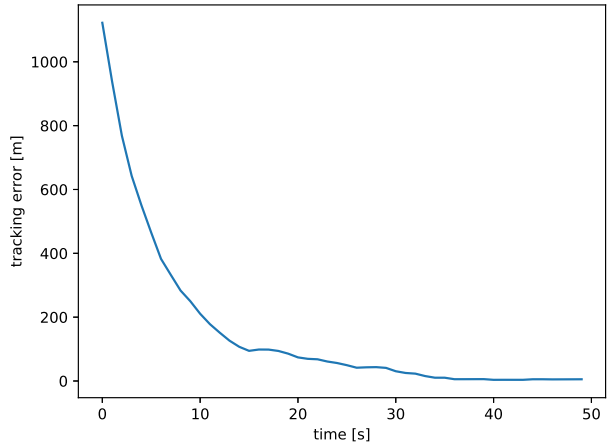


Figure 4.6: Absolute positional error of stationary target image tracking

(a) time 0

(b) time 5

(c) time 10

(d) time 49

Figure 4.7: View during convergence at different times when tracking stationary target. Green is view at true position, red is view at estimated position, yellow is overlap

4.2.1.2 Moving Target

Instead of a constant position, now the true position and ship attitude is sampled from the real ship's path at one sample per second. This is the setup from 3.3.4. The real camera extrinsics and intrinsics are loaded and used in the rendering of both the true position image and the guess position image. The real view is simulated by rendering at the true ship pose. The front camera's parameters are used. At each step the ICP algorithm is run for 100 iterations. As seen in figures 4.8 and 4.9 the system manages to track the true position of the ship. The accuracy is shown in 4.10. More iterations would result in higher accuracy at the cost of being more computationally demanding. This is seen in the last section, 4.2.1.1, where the system used 50 steps to reduce the position error to under 10m, far away from the shore. The positional error is somewhat higher towards the end of the tracking sequence. This may be due to the way the contour correspondences are found. When the ship turns, the contours get a larger sideways displacement. With a sideways displacement the nearest neighbour matches will also have more sideways displacement compared to the true match. Since the terrain contours are mostly vertical, large parts of the contours still have a close neighbour, although erroneously matched, and the ICP will have slower convergence, since the distance between the nearest neighbours is already small. When moving straight towards the terrain however, the closer distance will mostly cause a vertical displacement in the contours. Since the contours are mostly horizontal, the nearest neighbour matches will be closer to the true match, resulting in faster convergence, thus more accurate tracking. Since the initial guess position is the last refined position and no filter methods are used, the estimated position will lag behind the true position, since it doesn't manage to reduce the error enough in one iteration of ICP, as discussed in the last section. This can be seen by looking at the end of the tracked path in fig 4.9.

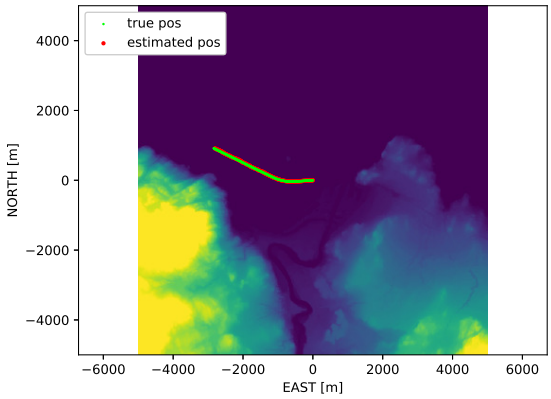


Figure 4.8: Plot of position tracking of synthetic images, movement from east to west

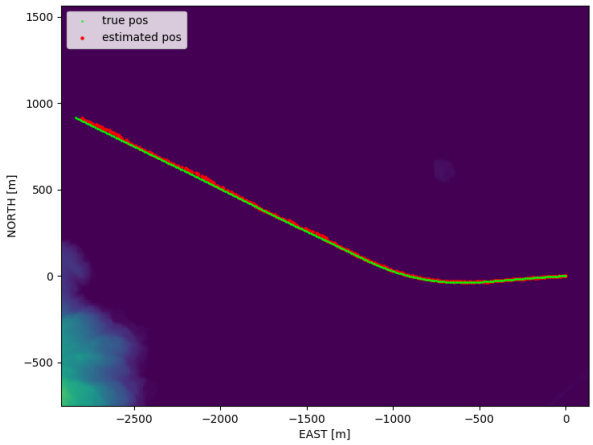


Figure 4.9: Zoomed plot of position tracking of synthetic images, movement from east to west

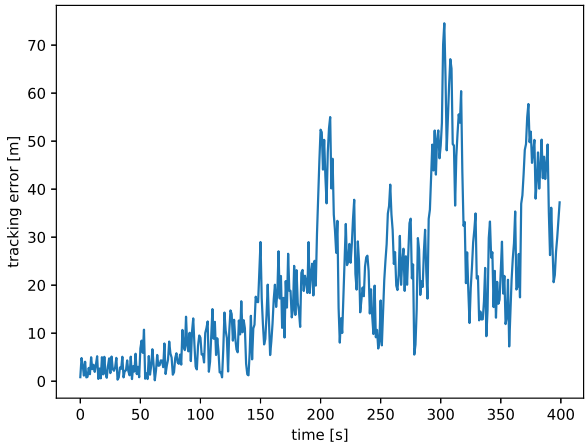


Figure 4.10: Absolute positional error of image tracking

4.2.2 Tracking Real Images

Now instead of rendering an image using the camera parameters at the true position to simulate the real view, real camera images are loaded from the dataset. The tracking test were carried out on camera cluster 0, with camera 1, as out of all the cameras it seemed to have the most accurate extrinsic and intrinsic data. Camera and rendering comparison for this camera is seen in fig 4.2. Furthermore, as can be seen from the path overlaid the area map in fig 4.8, the ship is heading towards the highest terrain in the area, and so it makes sense to track using the front facing camera, as the seen contours will be closer to the ship. As for the previous tracking sections the pose estimation is as described in section 3.3.4. The resulting true and estimated paths are compared in fig 4.11, and a zoomed in version in fig 4.12. For the first 150 seconds the system manages to track the position purely based on the images taken from the ship. While the accuracy is not as high as for the artificial view tracking in 4.2.1, the error is still below 100m, while travelling over 2km from the closest shore point. The tracking accuracy is shown in 4.13. One must also remember that there exists a pixel

perfect match for the artificial tracking, while for the real images all the edges are not detected, and there is also other edges than just from the contours, such as textures and unmodelled elements such as buildings and clouds. Furthermore there is also the inaccurate camera modelling, where the field of view is incorrect, in addition to ignoring distortion effects, and not correcting the error in the camera mounting angle. Between 150 and 175 seconds the tracking is lost and not recovered. the estimated ship path suddenly changes direction, heading almost straight east. The system has found a local minima for the contour match, but unfortunately not the minima corresponding to the correct position. What has happendes is that there is an ambiguity in the view. There are multiple positions resulting in views that are similar to the true view. This is seen in figure 4.14, where the estimated and the real views match well, but the position is wrong nonetheless. Moving east is results in similar views as moving north-west since the only terrain contours visible that are close to the ship are just two straight lines forming a trinangle . As long as the point of the tringle ramains at the same place in the image, moving towards and away from it or standing still does not change the angle between the lines, thus not changing the terrain contours. The cause for the tracking loss is that the ship moves along a path for which the terrain contours do not change, even the terrain is relatively close. The reason such a path exists is due to the relatively smooth and flat terrain, which results in long straight contour lines, which are difficult to track. This would not be as big a problem in an area with more distinct mountains and hill tops, but it is nonetheless the biggest weakness of this camera based position tracking system.

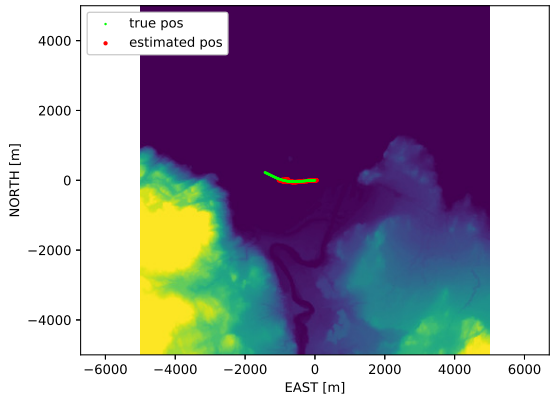


Figure 4.11: Plot of position tracking of real images

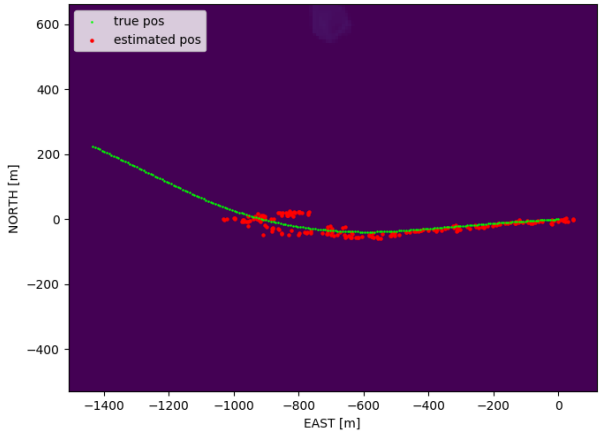


Figure 4.12: Zoomed plot of position tracking of real images

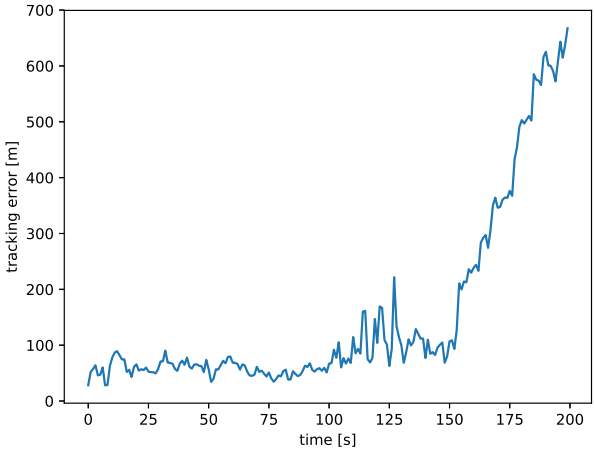
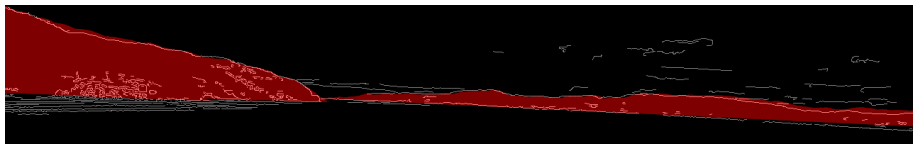
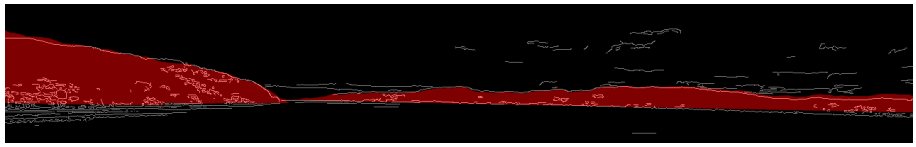


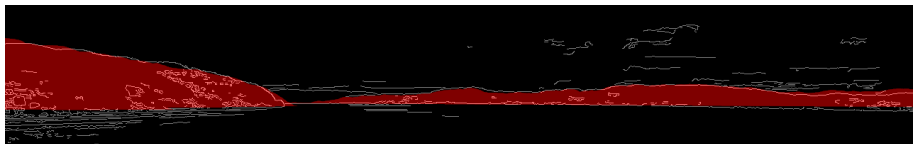
Figure 4.13: Absolute positional error of real image tracking



(a) time 150



(b) time 175



(c) time 199

Figure 4.14: Cropped view during convergence at different times when tracking moving target with real images

Chapter 5

Conclusions and future work

5.1 Report Summary

The report has detailed the implementation of a camera based position tracking system for ships, and investigated the effectiveness and the weaknesses of the system. The system uses a model of the terrain, which is created from a heightmap that is converted to a triangle mesh and stored as an .obj file. The heigmap is a digital terrain model (DTM) which is not as realistic as a Digital Surface Model (Digital Overflate Modell, DOM). The DOM data was not fully available. A rendering system is created in OpenGL to render the model intending to mimic a real camera, both in pose and intrinsic camera parameters. This was mostly successful apart from some small errors in the ship relative camera attitude, as well as the field of view. The effect of camera distortion effects remains untested. An image is rendered at an initial guess position. The position tracking is done using an iterative closest points algorithm (ICP), using Gauss newton for the optimization, to find a change in the guessed position for which the rendered image's contours become more similar to to the real image's contours. There are no filtering methods used to guide the initial guess, it simply guesses it's last refined position. This causes the system to lag behind the real path. The tracking system is tested on data that Kongsberg Seatex has generated from placing cameras

and navigation data loggers on a ship. The data is available through an interface made by Seatex employees. The testing is confined to data collected in the Trondheim Fjord. The experiments found that position estimation convergence is rather slow. The convergence is faster when the position error is greater, since the gradients of the reprojection errors are greater. This enhances the lagging effect of the tracking. The tracking with the synthetic images was much more accurate than the tracking with real camera images, indicating that if the edge image for the real and guessed images were more similar the tracking would perform better. When tracking on synthetic data the tracking was not lost even when subjected to a view with sparse information in the contours. The tracking on real data was lost after some time due to multiple movement directions yielding the same change in view. This failure mode for ambiguous views is by far the biggest weakness of the system. The tracking of synthetic images could also fail to this, but the additional errors in the real data making the tracking even more difficult. In addition the Canny edge detection yields many outlier edges and fails to detect some contour edges, which the system does not account for, it simply tries to move any edges closer to each other. It is apparent that terrain contours alone are not sufficient to robustly visually track the ship position due to the high likelihood of scenes having too sparse and indistinct contours.

5.2 Future Work

Many of the systems and methods devloped in this project will serve as a basis for a master thesis. Especially the model creation and rendering system is very reusable. Figuring out how to use the DOM data instead of the DTM will make the model much more similar to the real world. To remove the ocean from the DOM the DTM can be used as a mask, since the ocean here is at always at height 0. As of now the world resolution is simply higher in a confined area at the center of the map. A chunked LOD'ing technque will be considered for making the resolution dependent on the position of the ship, making the operation area of the system much higher. Reduction of redundant polygons, such as those in flat areas, will be considered. This would give the model a lower polygon count, letting us use high resolution for larger areas, as well as load-

ing a bigger part of the world, making objects on the far horizon visible if that is desired.

The camera calibration, both internal and the rotation relative to the ship will have to be corrected. The camera will possibly have to be recalibrated, after the camera distoriton effects are ruled out. The camera mounting angles could be adjusted by aligning the renderings with the real images.

The system will be more accurate if the position estimation is fused with IMU-data. The initial error would be smaller for each step, making the search easier and possibly less prone to tracking loss. A visual odometry system could also be implemented, tracking the velocity by comparing real images frame to frame. This is complex to do with a monocular camera however, since scale is not observed, and would have to be estimated. IMU fusion will be prioritized for sensor fusion, as the data is already available through INS systems developed at Kongsberg Seatex. IMU data and positonal data compliment each other well when fused. For the fusion the uncertainty of the position estimation might be approximated according to how close the ship is to land. Both a Kalman filter approach and factor graphs will be considered. Their main drawbacks and advantages will be compared before making a decision about which to use.

It was determined that the tracking convergence was relatively slow, and can be improved, potentially drastically, by implementing some kind of line search. Even though edge tracking might not be used in future work, any gradient descent related methods will be tested with some line search implementation. Furthermore a robust estimator, such as Tukey, should be used in the optimization process to make it more robust. Insead of minimizing a distance between points, a projective distance could be minimized in stead, minimizing the distance from a point to a plane created with the other points normal. This might reduce some of the problems with tracking sideways motion relative to the terrain.

The tracking system has a problem tracking ambiguous view, and so other methods than pure terrain contour tracking must be considered. If a DOM model is acquired

edges could be created within the terrain, providing more edges to track, thus making the view less ambiguous. Another idea might be to backproject strong features detected in the real images back onto the 3D model, which would provide additional tracking points, also reducing the ambiguous view problem. One must be wary of introduction a a possibility for drifting to the system with this approach, as the approach is similar to feature based monocular visual odometry, which drifts. One problem with the system was creating an accurate understanding of the image scene, as the canny edge detection was not smart enough to know which edges we desired. One idea is to use a neural network to do semantic segmentation on the image, detecting moutnains, buildings, vegetation, ocean, sky and so on. Transfer learning could be used by retraining an existing net on data generated with the rendering system, which is able to render a semantic mask, which can be associated with a real image from the dataset. The semantic data could be added in the .obj model, and be encoded as different colors in the rendering. Some kind of region based matching approach can then be implemented, where the semantically segmented real image is matched to the guessed rendered segmentation. An entropy or correlation based matching approach could be used. A region based matching approach based on semantic segmentation will be strongly considered for the main task of the master thesis.

A weakness of the system is that if the tracking is lost, there is no mechanism attempting to detect and correct this. A relocalization system should therefore be looked into, using visual place recognition to make a position guess based on a search in an image database created offline. Some dimensionality reduction method for the images should considered to make the search faster, possibly utilizing the segmentation network proposed in the previous paragraph.

Appendix A

Appendix

Listing A.1: Heightmap to mesh code

```
import numpy as np


class IndexTable:
    def __init__(self, shape):
        self.default_val = -1
        self.table = np.full(shape, self.default_val)
        self.next_index = 0
        self.sorted_list = []

    def get_create(self, row, col):
        if self.table[row, col] == self.default_val:
            self.table[row, col] = self.next_index
            self.next_index += 1
            self.sorted_list.append((row,col))
        return self.table[row, col]
```

```
    def get(self, row, col):
        return self.table[row, col]

    def num_indices(self):
        return self.next_index



def heightmap_to_obj_mesh(heightmap, scale, origin_offset
    , save_path):
    print('creating␣OBJ␣mesh␣from␣heightmap')


    vertices = ''
    normals = ''
    faces = ''

    vertex_indices = IndexTable(heightmap.shape)
    face_normals = np.empty(heightmap.shape, dtype=object
        )
    for row, col in np.ndindex(heightmap.shape):
        face_normals[row, col] = []

    tri1 = [(0, 0), (-1, -1), (0, -1)]  # offsets that
        form two triangles in a square, counter clockwise
    tri2 = [(0, 0), (-1, 0), (-1, -1)]

    index = 1 #OBJ is 1-indexed
    print('creating␣triangle␣faces␣with␣vertex-␣and␣
        normal-indices')
```

```
for (row, col), h in np.ndenumerate(heightmap):
    if col == 0 or row == 0:  # square origin (2xtri)
        is bottom right corner of square
        continue

    for tri in [tri1, tri2]:
        tri_heights = [heightmap[row + r, col + c]
            for (r, c) in tri]
        if all(h == 0 for h in tri_heights) or any(h
            < 0 for h in tri_heights):  # check if
            square is all ocean, or if any value
            invalid todo: bitmap
            continue

        face_indices = [1+vertex_indices.get_create(
            row + r, col + c) for (r, c) in tri] #+1
            since OBJ 1 indexed
        faces += 'f {f[0]}//{f[0]} {f[1]}//{f[1]} {f
            [2]}//{f[2]}\n'.format(f=face_indices)  #
            face consists of vertex and normal indexes
            , which are identical
        face_vertices = [np.array([(row+r)*scale, (
            col+c)*scale, heightmap[row+r, col+c]])
            for r,c in tri]
        face_normal = np.cross(face_vertices[1]-
            face_vertices[0], face_vertices[2]-
            face_vertices[0])
        for (r,c) in tri:
            face_normals[row+r,col+c].append(
                face_normal)
```

```
    print('indexing␣vertices␣and␣creating␣normals')


    for (row, col) in vertex_indices.sorted_list:
        vertices += 'v␣{}␣{}␣{}\n'.format(row * scale +
            origin_offset[0], col * scale + origin_offset
            [1],
                                           float(heightmap[
                                               row, col]))
        #combine face normals
        normal = sum(face_normals[row,col])
        normal /= np.linalg.norm(normal)
        normals += 'vn␣{}␣{}␣{}\n'.format(normal[0],
            normal[1], normal[2])

    with open(save_path, 'w+') as f:
        f.write('#␣OBJ␣\n#␣Terrain␣triangle␣mesh\n')
        f.write(vertices)
        f.write(normals)
        f.write(faces)

    print('completed!')
```

Listing A.2: Loading .obj files

```
import numpy as np
import os


class ModelContainer:
    def __init__(self):
        self.indexOffset = 0
```

```
def _add_objData(self, vertices, normals, indices):
    if self.indexOffset == 0:
        self.indices = indices
        self.vertices = vertices
        self.normals = normals
        self.indexOffset += np.shape(vertices)[0]
    else:
        indices += self.indexOffset
        self.indices = np.concatenate((self.indices,
            indices))
        self.vertices = np.concatenate((self.vertices
            , vertices))
        self.normals = np.concatenate((self.normals,
            normals))
        self.indexOffset += np.shape(vertices)[0]

def load_obj(self, file_path, use_cache):
    print('loading␣file:␣{}'.format(file_path))

    if file_path.split('.')[-1] != 'obj':
        print('file␣is␣not␣.obj')
        return

    cache_path = file_path.split('.')[0] + '_cache_.
        npy'

    if use_cache and os.path.isfile(cache_path):
        print('found␣cache')
        vertices, normals, indices = np.load(
            cache_path)
```

```
            self._add_objData(vertices, normals, indices)

        else:
            print('no cache. parsing file')

            vertices = []
            normals = []
            indices = []

            with open(file_path, 'r') as f:
                i = 0
                for line in f.readlines():
                    if line.startswith('#'):
                        print('\t{}'.format(line))

                    if line.startswith('v '):
                        line = line.strip().split()
                        vertex = np.array(line[1:], dtype
                            =np.float32)
                        vertices.append(vertex)

                    elif line.startswith('vn '):
                        line = line.strip().split()
                        normal = np.array(line[1:], dtype
                            =np.float32)
                        normals.append(normal)

                    elif line.startswith('f '):
                        line = line.strip().replace('/',
                            ' ').split()
                        vertexIndices = np.array(line
```

```
                                [1::2], dtype=np.uint32)-1
                            indices.append(vertexIndices)

                vertices = np.array(vertices)
                normals = np.array(normals)
                indices = np.array(indices)

                np.save(cache_path, [vertices, normals,
                    indices])

                self._add_objData(vertices, normals, indices)


if __name__ == '__main__':
    obj = ObjContainer()
    obj.load_obj('models/square.obj', True)
```

References

[1] E. Marchand Amaury Dame. Accurate real-time tracking using mutual information. 2010.

[2] Calvin1602@github. Opengl tutorial, 2015.

[3] fileformat.info. Wavefront obj file format summary, 2012.

[4] Alan Grant, Paul Williams, Nick Ward, and Sally Basker. Gps jamming and the impact on maritime navigation. *Journal of Navigation - J NAVIG*, 62, 04 2009.

[5] Chris Harris and Carl Stennett. Rapid - a video rate object tracker. In *BMVC*, 1990.

[6] Hani Javan Hemmat, Egor Bondarev, Gijs Dubbelman, and Peter H. N. de With. Improved icp-based pose estimation by distance-aware 3d mapping. *2014 International Conference on Computer Vision Theory and Applications (VISAPP)*, 3:360–367, 2014.

[7] M Jalloul, Mohammed Baydoun, and Mohamad Al-Alaoui. Gauss-newton image registration with cuda. pages 305–309, 12 2011.

[8] kartkatalogen. Norgeskart, 2018.

[9] G. Klein and T. Drummond. Tightly integrated sensor fusion for robust visual tracking. 2004.

[10] V. Lepetit L. Vacchetti and P. Fua. Combining edge and texture information for real-time accurate 3d camera tracking. 2004.

[11] Peter Lindstrom, David Koller, William Ribarsky, Larry Hodges, N Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. pages 109–118, 01 1996.

[12] Manolis Lourakis and Xenophon Zabulis.

[13] E. Malis and E. Marchand. Experiments with robust estimation techniques in real-time robot vision. 2006.

[14] olivers posterous. linear depth in glsl, 2010.

[15] Victor Prisacariu and Ian D. Reid. Pwp3d: Real-time segmentation and tracking of 3d objects. volume 98, 01 2009.

[16] G. Reitmayr and T. W. Drummond. Going out: robust model-based tracking for outdoor augmented reality. In *2006 IEEE/ACM International Symposium on Mixed and Augmented Reality*, pages 109–118, Oct 2006.

[17] C. Teulière, E. Marchand, and L. Eck. Using multiple hypothesis in model-based tracking. In *2010 IEEE International Conference on Robotics and Automation*, pages 4559–4565, May 2010.

[18] Naty Hoffman Tomas Akenine-Moller, Eric Haines. *Real-time rendering*. CRC Press, 2002.

[19] Thatcher Ulrich. Rendering massive terrains using chunked level of detail control. 01 2002.