True orthophoto generation

Morten Ødegaard Nielsen

Kgs. Lyngby 2004
IMM-THESIS-2004-50

True orthophoto generation

Morten Ødegaard Nielsen

Kgs. Lyngby 2004

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-THESIS: ISSN 1601-233X

Preface

This master thesis is the culmination of my study at the Technical University of Denmark. The thesis investigates methods for creating digital *true* orthophotos.

The thesis is divided into consecutive numbered chapters. Located at the last pages are the appendixes and an index. References are given as numbers in square brackets, and a list of the references can be found at the back. Throughout the thesis, illustrations, tables and figures are labeled with two numbers. The first number references the chapter and the second is numbered consecutive. Unless a reference is provided, illustrations and tables are created by me.

During the project, imagery and surface models are used for testing and illustration. These data have kindly been provided by BlomInfo A/S. In relation to this thesis, I would like to thank the following people, for whom their help and assistance I am truly grateful for.

From Technical University of Denmark, Department of Informatics and Mathematical Modelling:

- Keld Dueholm for general guidance, help and ideas.
- Allan Aasbjerg Nielsen for help with color analysis and color correction.
- Bent Dalgaard Larsen for help with 3D graphics related algorithms and methods.
- Rasmus Larsen and Henrik Aanæs for ideas on image filtering.

From BlomInfo A/S:

- Søren Buch for general guidance, help and ideas.
- Lasse Kjemtrup for help on orthophotos and aerial imagery.
- Morten Sørensen for help with the surface model data.

Morten Ødegaard Nielsen, Kgs. Lyngby, August 2nd 2004

Abstract

This Master Thesis investigates methods for generating true orthophoto imagery from aerial photographs and digital city models.

The thesis starts by introducing the theory for generating orthophotos, followed by a comparison of orthophotos with *true* orthophotos. Methods and problems that arise when extending the theory to true orthophotos are treated. On the basis of the investigation, an overall method for creating true orthophotos is devised. The remaining chapters treat the steps of the method in details, and evaluate the results.

The true orthophoto rectification is divided into four general steps: Rectification, color matching, mosaicking and feathering. Creating the image mosaic is found to be the most crucial part of the process.

Three methods for mosaicking source images are tested. They all rely on simple pixel score techniques used for assigning pixels from the source images. The best method found uses a method where the score is calculated as a combination of the distance to the source images' nadir points and the distance to obscured areas. A histogram matching algorithm is used for giving the source images the same radiometric properties, and feathering is applied along the seamlines to hide remaining differences.

The method is tested on a range of areas, and the overall result shows that the method gives reasonable results, even if the surface model is inaccurate or incomplete. It is furthermore assessed that the method can be applied to large-scale true orthophoto projects.

Keywords: *Orthophoto, Digital Surface Models, Aerial photography, Photogrammetry, Color matching.*

Table of Contents

Preface ..... **i**

Abstract ..... **iii**

Table of Contents ..... **v**

List of Abbreviations ..... **ix**

Chapter 1 Introduction **1**

1.1 Motivation 1
1.2 Problem definition 1
1.3 Outline and structure 2
1.3.1 General overview of the chapters *2*

Chapter 2 Orthophotos **5**

2.1 Creating orthophotos 6
2.1.1 Reprojection *6*
2.1.2 Mosaicking *7*
2.2 Relief displacements 8
2.3 True orthophotos 9
2.4 Accuracy of orthophotos 15
2.5 Summary 18

Chapter 3 Digital Surface Models **19**

3.1 Surface models 19
3.2 Surface representation 20
3.2.1 Triangulated Irregular Network *21*
3.2.2 Grid *22*
3.3 Copenhagen's 3D city model 24
3.4 Summary 25

Chapter 4 Coverage analysis **27**

4.1 Overlapping 27
4.2 Test setup 28
4.3 Test results 30
4.4 Summary 31

Chapter 5 Design description 35

5.1 Limits of other True Orthophoto applications 35
5.2 Creating TOs – Step by step 36
5.2.1 Rectification 37
5.2.2 Locating obscured pixels 37
5.2.3 Color matching 37
5.2.4 Mosacking and feathering 37
5.3 Implementation 39

Chapter 6 The Camera Model 41

6.1 Exterior orientation 42
6.2 Interior orientation 43
6.3 Summary 46

Chapter 7 Raytracing the surface model 47

7.1 Raytracing using a database 48
7.2 Binary Trees 50
7.3 Axis Aligned Bounding Box Tree 50
7.3.1 Creating the tree 51
7.3.2 Intersecting the tree 53
7.3.3 Performance evaluation 55
7.4 Summary 55

Chapter 8 Color matching 57

8.1 Color spaces 59
8.2 Histogram analysis 60
8.3 Histogram matching 62
8.4 Histogram matching test on ortho images 63
8.5 Summary 64

Chapter 9 Mosaicking 69

9.1 Mosaicking methods 69
9.1.1 Mosaicking by nearest-to-nadir 70
9.1.2 Mosaicking by distance to blindspots 71
9.1.3 Mosaicking by distance to nadir and distance to blindspots 74
9.2 Feathering 76
9.3 Enhancing the mosaic 78
9.3.1 Changing the scores 78
9.3.2 Reducing mosaic fragmentation 79
9.4 Summary 81

Chapter 10 Test results 83

10.1 Performance 84
10.2 Pros... 85
10.3 ...and cons 86
10.4 Using simpler DSMs and wide angle imagery 89
10.5 Creating large-scale true orthophotos 92
10.6 Summary 93

Chapter 11 Conclusion 95

11.1 Evaluation 95
11.2 Outlook 96

References 99

Appendix 101

Appendix A Contents of companion CD-ROM 103

Appendix B MATLAB scripts 105

Appendix C True orthophoto generator - Users guide 113

Appendix D Raytracer library 121

Index 125

List of Abbreviations

AABB	Axis Aligned Bounding Box
DBM	Digital Building Model
DSM	Digital Surface Model
DT	Distance Transformation
DTM	Digital Terrain Model
GIS	Geographic Information System
GSD	Ground Sample Distance
IHS	Intensity-Hue-Saturation
LUT	Lookup Table
RGB	Red-Green-Blue
TIN	Triangulated Irregular Network

Chapter 1 Introduction

This chapter gives an overview of the general objectives in this thesis. The basis and goals of the project are presented along with a brief description of the contents of the thesis.

1.1 Motivation

With today's developments in GIS and digital processing, the digital orthophoto has become a very common part of spatial datasets. The demand for greater detail and resolution of the imagery is increasing, which at the same time creates an increasing demand for greater quality and accuracy of the orthophoto.

The orthophoto has some limitations that can cause problems in its everyday use. Displacements in the orthophoto create inconsistent accuracy and scale, which especially shows when combined with vectorized GIS data. The limitations can cause problems for the user who is unaware of them, and incorrectly uses them as a true and accurate map.

The increasing detail of orthophotos makes the limitations more evident to everyone. The demand for greater accuracy therefore involves trying to overcome limitations of the orthophotos. Accurate *true* orthophotos that can be used without considering any reservations are a field of great interest. The ever increasing computer processing power has today made it feasible to create true orthophotos on a larger scale and commercial applications for creating them have already been introduced.

1.2 Problem definition

This master thesis will investigate methods for creating orthophotos and extend this knowledge to true orthophoto imagery. The method for creating true orthophoto imagery on basis of aerial images and a digital city model needs to be as fully automatic as possible.

The overall goals of this thesis are:

- Devise a method for creating true orthophoto imagery.
- Investigate the quality of source data needed and the expected coverage and accuracy of the final true orthophotos.
- Investigate problems and test methods for generating true orthophotos.
- Implement methods in an application that is capable of creating true orthophoto imagery fully automated.

1.3 Outline and structure

This master thesis is partially based on studies from a preparatory thesis [4]. Whenever the preparatory thesis is referenced, the important results are presented, and can therefore be read without the prior knowledge of [4].

In the first chapters the thesis presents the difference between orthophotos and true orthophotos and covers the basic theory needed for generating orthophotos. A method for creating true orthophotos is afterwards devised. The crucial steps in the method are introduced, tested and evaluated independently in the following chapters.

During the study, software has been developed that is able to produce true orthophoto imagery. Code snippets and documentation for using the software are presented in the appendixes. The software is available on the companion CD-ROM.

1.3.1 General overview of the chapters

Chapter 2, Orthophotos: Introduces the concept of orthophotos and the process of creating them. Afterwards this is extended to true orthophotos, and the differences are pointed out. The expected accuracy of an orthophoto is also treated.

Chapter 3, Digital Surface Models: The concept of digital surface models is treated and the different model representations are presented. A description of Copenhagen's 3D city model, that were used during this study, is lastly included.

Chapter 4, Coverage analysis: The detailed surface models are used to identify the expected coverage of a true orthophoto. This is done on basis of different source image setups and different kinds of built-up areas.

Chapter 5, Design description: A step-by-step method for creating true orthophotos is devised and described.

Chapter 6, The Camera Model: A mathematical model of a camera lens system and the colinearity equations are presented. The model is split up in two parts: The exterior and interior orientations.

Chapter 7, Raytracing the surface model: Methods for effectively tracing rays between the camera and the surface model are treated in this section. Performance is an important issue, due to the vast amount of calculations needed.

Chapter 8, Color matching: The concept of color and color adjustment are introduced. Color adjustment techniques are applied to the imagery to make the images share the same radiometric properties.

Chapter 9, Mosaicking: Different methods for mosaicking an image as seamlessly as possible are presented, tested and evaluated.

Chapter 10, Test results: The method devised was tested on a set of data. Pros and cons of the method are illustrated with close-ups and commented.

Chapter 11, Conclusion: The overall results of the master thesis are summarized and the final conclusions are drawn. It furthermore presents suggestions for future work in this field.

Chapter 2 Orthophotos

When a photograph is taken, it shows an image of the world projected through a perspective center onto the image plane. As a result of this, the image depicts a perspective view of the world. For an aerial image - that normally is shot vertically - objects that are placed at the same point but at different heights will therefore be projected to different positions in the photograph (figure 2.1). As an effect of these *relief displacements*, objects that are placed at a high position (and closer to the camera) will also look relatively bigger in the photograph.

The ortho rectification process is a process that tries to eliminate the perspectiveness of the image. The result is an orthographic projection where the rays are parallel as opposed to the perspective projection where all the rays pass a common *perspective* center.

As a result of the rectification, the orthophoto is an image where the perspective aspect of the image has been removed. It has a consistent scale and can be used as a planimetric map [2]. This makes it useable for combining with spatial data in GIS systems or as part of 3D visualizations, where the orthophoto can be draped over a 3D model. Orthophotos can function as a reference map in city planning, or as part of realistic terrain visualizations in flight simulators. The orthophoto has a reference to a world coordinate system, and

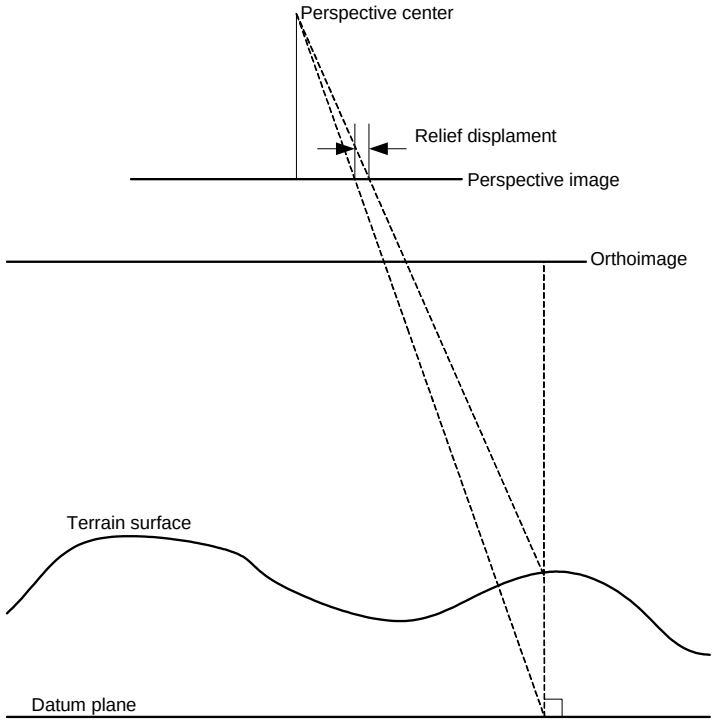


Figure 2.1 - Perspective and orthographic image geometry, illustrating the cause of relief displacements [2]

can therefore function as an un-interpreted map. The orthophotos can be merged into one large photo of an enormous area. An example of this is the company Cowi's *Kortal* (www.kortal.dk) where orthophotos have been merged to form one large mosaic that covers all of Denmark.

2.1 Creating orthophotos

In order to create the orthophotos, knowledge of the terrain is needed. A terrain model can be created in several ways, but the most common is using photogrammetry. Furthermore the position and orientation of the camera during the exposure is needed. These parameters can be derived using either a bundle adjustment or by fitting the image over some known ground control points.

2.1.1 Reprojection

The orthophoto rectification is done by reprojection, where rays from the image are reprojected onto a model of the terrain. The reprojection can be done in two ways: Forward and backward projection.

The forward projection projects the source image back on to the terrain (figure 2.1). The point where the projected point intersect the terrain (X,Y,Z) is then stored in the orthophoto. If the corner of the orthoimage is placed at X_0, Y_0 the pixel coordinate in the orthoimage is found by [2]:

$$\begin{bmatrix} column \\ row \end{bmatrix} = \frac{1}{GSD} \cdot \begin{bmatrix} X - X_0 \\ Y_0 - Y \end{bmatrix}$$

Where *GSD* is the *Ground Sample Distance*, which is the distance between each pixel. This is also referred to as the *pixel size*. Notice that the equation takes into account that a pixel coordinate system has the Y-axis downwards, and the world coordinate system has the Y coordinate upwards / north.

The forward projection projects regularly spaced points in the source image to a set of irregular spaced points, so they must be interpolated into a regular array of pixels that can be stored in a digital image. This is why the backward projection is often preferred. In a backward projection, the pixel in the output image is projected back to the source image. Instead of interpolating in the orthoimage, the interpolation is done in the source image. This is easier to implement, and the interpolation can be done right away for each output pixel. Furthermore only pixels that are needed in the orthoimage are reprojected.

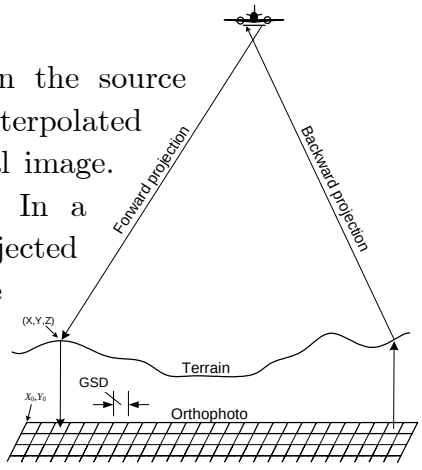


Figure 2.2 - The basic idea of forward and backward projection.

When using the backward projection, a row/column coordinate of a pixel in the orthophoto is converted to the world coordinate system, and the Z coordinate is found at this point in the terrain. The pixel-to-world transformation is given by [2]:

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} X_0 \\ Y_0 \end{bmatrix} + GSD \cdot \begin{bmatrix} column \\ -row \end{bmatrix}$$

The position in the source image that corresponds to the found X,Y,Z coordinate can be found by modeling the camera. A description of the camera model and the equations needed for this calculation can be found in chapter 6.

2.1.2 Mosaicking

Large orthophoto projects will require rectification of several source images, which are afterwards put together. This process is known as mosaicking. Mosaicking images involves several steps:

- Seamline generation
- Color matching
- Feathering and dodging

The seamlines in a mosaic defines where the images are stitched together. The seamline generation can either be done automatically or manually. The goal is to mosaic the images along places where they look very similar. A manual seamline placement can preferable be placed along the centerlines of the roads. If the orthophotos are reprojected onto a surface model that doesn't include the buildings, these will have uncorrected relief displacements, and placing a seamline through a building will create a poor match.

There exist several methods to place the seamlines automatically. One method is to subtract the images and place the lines along a least-cost trace, where the cost is the difference between the two images. A simpler approach places the seamlines along the centre of the overlap.

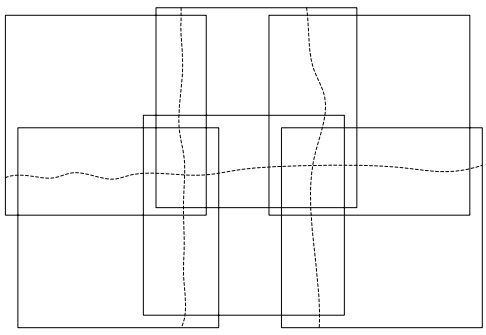


Figure 2.3 - Example of seamline placement in six orthophotos.

The images mosaicked should have the same color characteristics near the seamlines. If the color or brightness of the images are very different, the result of mosaic will be very poor, and the placement of the seamlines visible. There are several tricks that can be performed to hide the seamlines. Color matching and dodging techniques tries to remove the radiometric differences in the images, by analyzing and comparing the overlapping sections. Feathering tries to hide the remaining difference by making a smooth cut that slowly fades from one image to the other.

2.2 Relief displacements

The lower the flight altitude is, the higher are the relief displacements. In the nadir point there are no relief displacements, but these increase with the distance to nadir. If h is the height of an object on the ground (ie. a building), H is the flight altitude above the base of the object, and r_t is the distance to the image centre, the relief displacements in the image is calculated by [2]:

$$\Delta r = \frac{h \cdot r_t}{H}$$

Figure 2.4 illustrates that a high flying altitude results in smaller relief displacements. A real-world example is illustrated on figure 2.5.

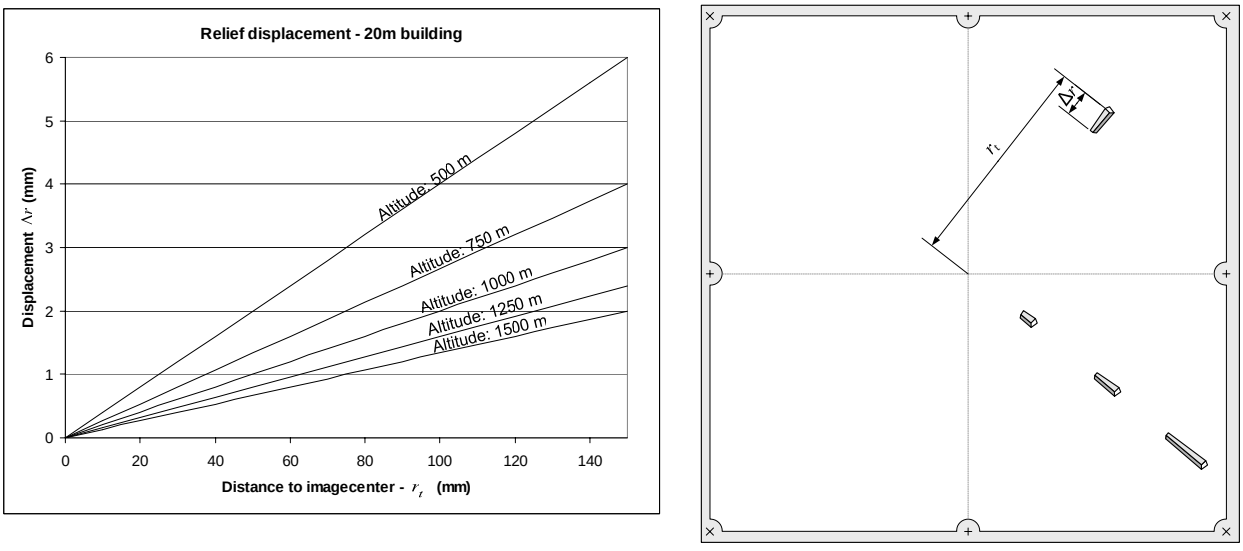


Figure 2.4 – Relief displacements increase towards the edge of the image or when the flight altitude is decreasing. The displacements are always oriented away from the nadir point.

Altitude 1500m (Normal angle lens)

Altitude 750m (Wide angle lens)

Figure 2.5 – The two images above are taken from approximately the same position, but at different altitudes. The relief displacements are significant smaller when the flight altitude is higher. The church on the image is approximately 70 meters tall.

2.3 True orthophotos

A general problem for normal orthophoto generation is that it cannot handle rapid changes in elevation. The relief displacements can be so large that they will obscure the terrain and objects next to them (figure 2.6).

A normal orthophoto is made on basis of a model of the terrain. The terrain model doesn't include the buildings, vegetation etc. This results in an image where buildings are leaning away from the image centre, and doesn't get corrected, and only objects that are in level with the terrain are reprojected correctly. Roads running over bridges will look like they "bend down" to follow the terrain below it. A true orthophoto reprojects the source images over a surface model that includes buildings, bridges and any other object that should be taken into account. When the buildings are included they will surely obscure objects close to them, since the walls of the buildings can be thought of as a rapid change in elevation.

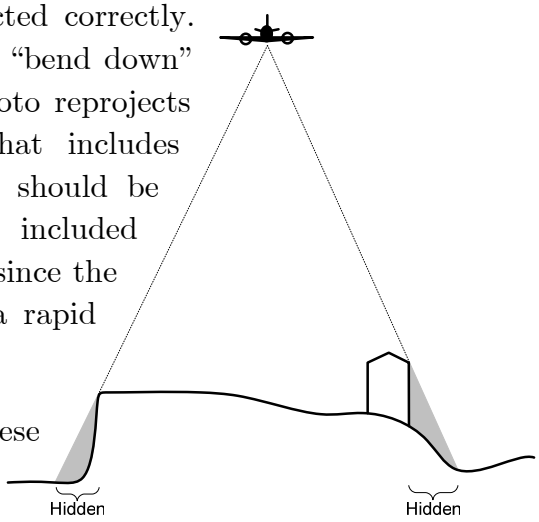


Figure 2.6 - Tall objects and rapid elevation changes will hide objects behind them due to the relief displacements caused by the perspective projection.

An orthophoto application does not detect these obscured areas, and instead creates "*ghost images*". If a building in a DSM is orthorectified, the building will get rectified back to its original position, but it will also leave a "copy" of the building on the terrain. This is illustrated on figure 2.9B. The reason

for this is that during the reprojection, rays are reprojected back to both the obscured area and the obscuring object, without detecting that obscured data is being rectified. Therefore the "wrong" image data is rectified in the obscured areas.

In common terms the true orthophoto is "true" when it tries to restore any obscured objects, and at the same time include as many objects as possible in the surface model. A true orthophoto should be based on a surface model that include anything that is visible in the source images, but it would be an incomprehensible task to create a full model including vegetation, people, cars, traffic lights etc. In general when talking about true orthophotos, they are based on surface models that only include terrain, buildings and bridges. A similar definition is found in [15]:

»[...] *the term true orthophotos is generally used for an orthophoto where surface elements that are not included in the digital terrain model are also rectified to the orthogonal projection. Those elements are usually buildings and bridges.*«

A very different definition is found in [25]. It only defines the true orthophoto on basis of removing ghost-image artifacts caused:

»[...] *the term "True Ortho" means a processing technique to compensate for double mapping effects caused by hidden areas. It is possible to fill the hidden areas by data from overlapping aerial photo images or to mark them by a specified solid colour.*«

In order to restore the obscured areas, or *blindspots*, imagery of these missing areas are needed. These supplemental images can be created by using pictures of the same area taken from different locations (figure 2.7). This will result in different relief displacements in each image, and by combining the images full coverage can be obtained. In aerial photography it is normal to let the images overlap as illustrated on figure 2.8.

The task is to locate the blindspots and automatically fill them with data from other images where the areas are visible. The number of seamlines needed for true orthophotos is therefore much higher than that of an ordinary orthophoto. Seamlines must be generated around every blindspot, and this makes the mosaic process more demanding. It also increases the demand for a good color matching algorithm, since the match must be good around all the numerous seamlines.

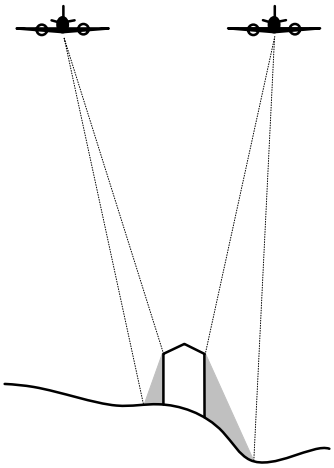


Figure 2.7 – Combining several images to get full coverage.

True orthophotos gives a much better fit when used as backdrop for a digital map. A building outline will match perfectly with the true orthophoto. True orthophotos are

also useful for draping over a complex surface model for use in 3D visualizations. The rooftops in the image match perfect with the surface model. Examples of some of the advantages of true orthophotos over ordinary orthophotos are illustrated at figure 2.10 and figure 2.11.

Aerial images taken from a high altitude resulting in a small scale and low resolution will have relatively smaller relief displacements, or more correctly: less visible relief displacements. Furthermore orthophotos based on small scale source images will usually be used to produce a corresponding low resolution in the orthophoto. Therefore it is up for discussion if the problems of relief displacements are large enough in a low resolution orthophoto, to make it justifiable to create true orthophoto imagery instead. The resolution must either be of high detail, the buildings tall or the terrain very rough. This can be directly related to the pixel size of the true orthophoto. If the relief displacements are at the subpixel level, obscuring relief displacements can clearly be ignored. Displacements of 2-3 pixels will probably not matter either. The relief displacements can be decreased further by using images created with a normal-angle lens shot from a higher altitude, or only use the central part of the images.

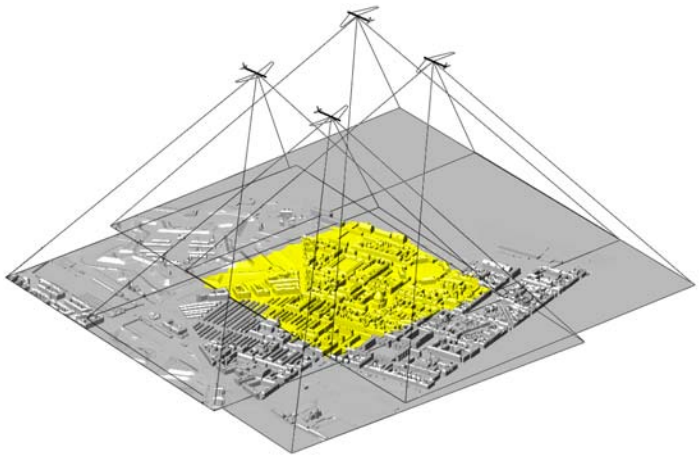


Figure 2.8 – Aerial photography with 60% forward overlap and sidelap. The four images above, all cover the same yellow area, but viewed from different locations. Buildings in the yellow area will all have relief displacements in different directions, so that blindspots hidden in one image is likely visible in another.

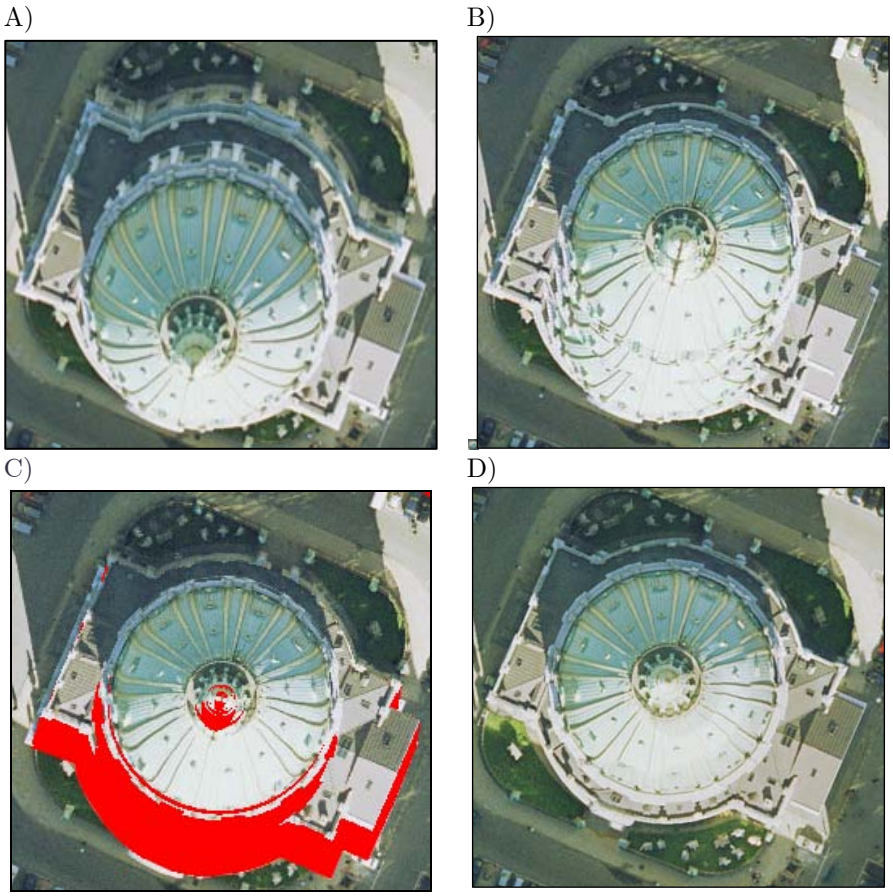


Figure 2.9 – A) An orthophoto rectified over a terrain model. The church is not moved to its correct position. B) Orthophoto based on a city model. The church is rectified to its correct location, but a "ghost image" is left on the terrain. C) Same as B but the obscured area has been detected. D) True orthophoto where the obscured area has been replaced with imagery from orther images.

Figure 2.10 – Orthophoto (top) and true orthophoto (bottom) overlaid with a vector map of the building outlines. The orthophoto has a good fit with objects in level with the terrain, but not with objects like the rooftops that wasn't included in the rectification. With the true orthophoto this is not a problem.

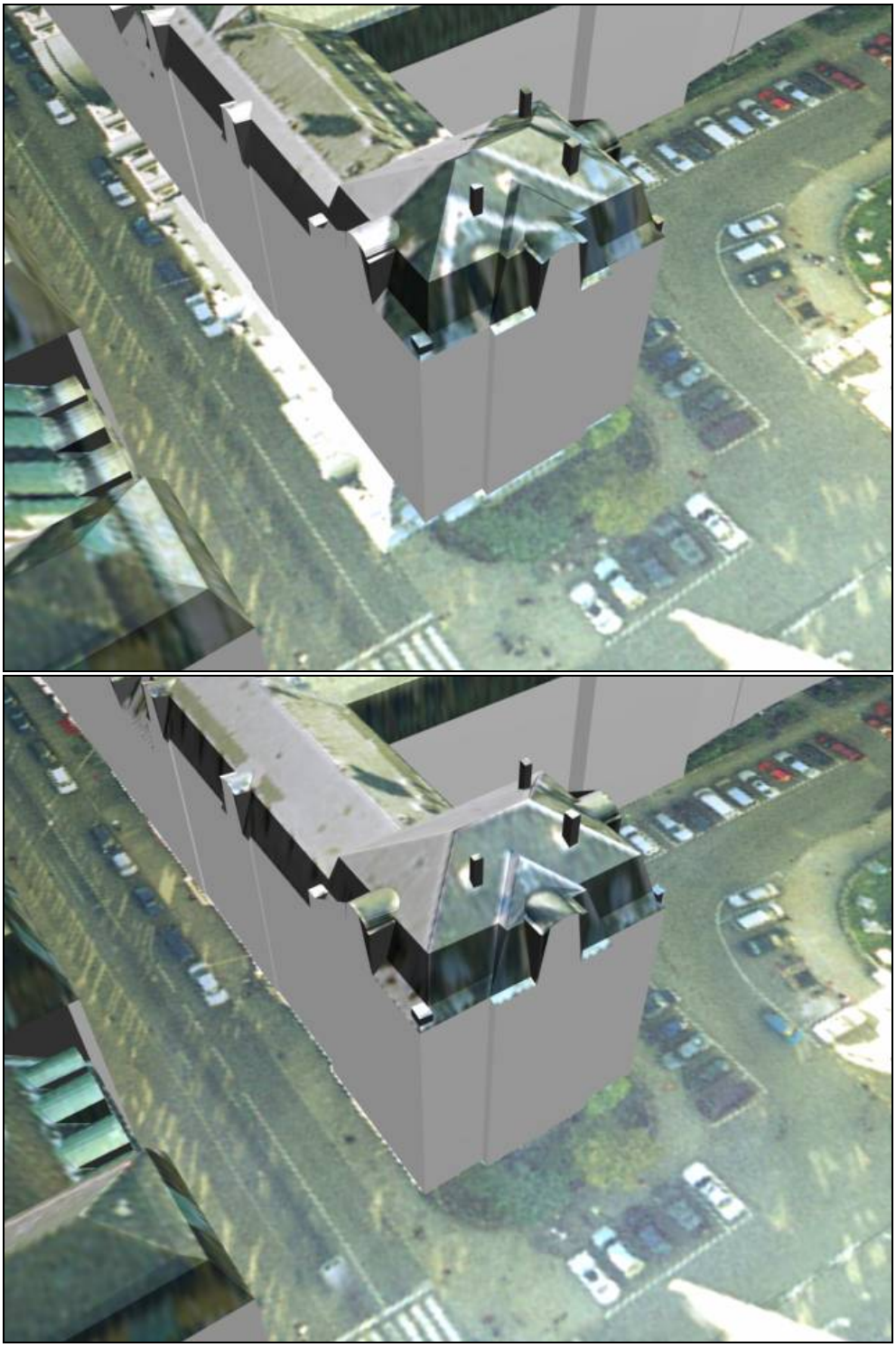


Figure 2.11 - Orthophoto (top) and true orthophoto (bottom) draped over a 3D city model. The orthophoto doesn't fit well with the 3D model. Parts of the roof are visible on the ground and the remaining image of the roof is draped wrongly onto the surface. The true orthophoto has a perfect fit with the roof construction.

2.4 Accuracy of orthophotos

The accuracy of an orthophoto is affected by several different parameters. Since orthophotos are a product derived from other data, they are dependent of the quality of these base data. Specifically these are:

- The quality and resolution of the source images
- The inner and outer orientation of the images.
- The accuracy of the digital terrain/surface model.

The general visual quality of a true orthophoto depends greatly on the source images. Some of the parameters that affect the quality of the images are:

- Quality of the negative (grain size)
- Quality of the camera and lens
- Resolution, precision and overall quality of digital scanning

The metric cameras used today for mapping are of a very high quality, as is the scanners that convert the photographs to digital images. The images used in this project are scanned in 15 microns, which is around twice the grain size of the negative.

The accuracy of the inner orientation is with modern aerial cameras so small that the remaining errors can be ignored. For the outer orientation, the standard deviation remaining from the bundle adjustment is of some significance. The standard deviation in the image σ_{ob} will have an effect in the terrain proportional to the scale M of the image[1]:

$$\sigma_o = M \cdot \sigma_{ob} = \frac{H \cdot \sigma_{ob}}{c}$$

The standard deviation σ_{ob} is normally between 10 and 20 microns, depending on the accuracy needed. BlomInfo, who produced the images and surface model, has a standard demand for the residuals of maximum 14 microns. If the scale is 1:5000 the accuracy on the ground is $14\mu m \cdot 5{,}000 = 0.069$ m.

A few examples of planar mean errors based on the individual specifications are shown at table 2.1.

[1] From personal consultancies with Keld Dueholm

Mapping standard	Scale	σ_o	σ_{ob}
Danish Road Directorate	1:4,000	4 cm	10 µm
TK99	1:10,000	14 cm	14 µm
TOP10DK	1:25,000	49 cm	20 µm

Table 2.1 - Examples of accuracy of bundle adjustments.

Errors in the DSM will introduce horizontal errors which are caused by "uncontrolled" relief displacements. The horizontal error Δ_{hor} can be found by a geometric analysis of a vertical offset Δ_{ver} as illustrated on figure 2.12. From this figure the following relation can be derived:

$$\frac{f}{r_T} = \frac{H}{D + \Delta_{hor}} = \frac{H - \Delta_{ver}}{D}$$

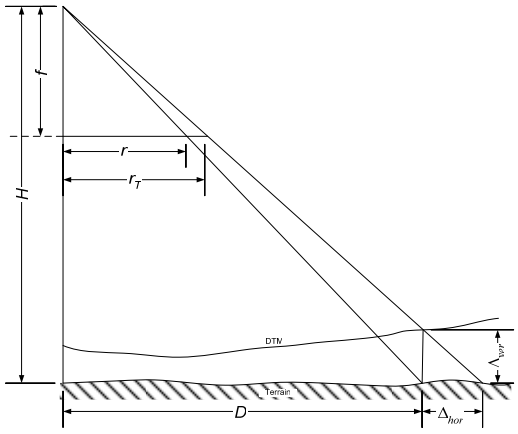


Figure 2.12 - Relief displacements.

Isolating Δ_{hor} gives:

$$\Delta_{hor} = \frac{r_T \cdot \Delta_{ver}}{f}$$

This means that the inaccuracies caused by a poor DSM increase linearly away from the nadir point, and therefore a constant error doesn't apply to orthophotos. Orthophotos will often not use the edges of the image, since the neighbor image will overlap, and it is preferable to use the central part of the image. This reduces the worst part of the effect. With true orthophotos - that are heavily mosaicked - it is hard to give a good overall estimate of the mean accuracy. It will all depend on the final mosaic pattern.

The DSM used in this project has a mean vertical error of 15 cm for well-defined points. At the corner of the image, the error for a normal angle lens is:

$$\Delta_{hor} = \frac{\sqrt{(115mm)^2 + (115mm)^2} \cdot 15cm}{303mm} = 8cm$$

For a wide angle lens where the focal length is approximately 150 mm, the error will be twice as large.

One method to give an estimate of a mean standard deviation integrated over the entire image area is given by [24]:

$$\sigma_{dg} = \frac{\Delta_{ver}}{f}\sqrt{\left(\frac{a^2+b^2}{3}\right)}$$

where $2a$ and $2b$ are the length of the sides of the image. Because of the overlap, an ordinary orthophoto a and b will be smaller than the size of the photograph since only the central part will be used. For a true orthophoto this is not the case as mentioned above, and therefore the effective area is much larger. Chances are that the edges of the image will not be used as much as the central parts, but the final mosaic is not known beforehand, and therefore it is hard to predict a good measure for the standard deviation.

Combining the two errors σ_o and σ_{dg} gives[2]:

$$\sigma_{ogns} = \sqrt{\left(\frac{\sqrt{a^2+b^2}}{f\cdot\sqrt{3}}\cdot\sigma_{ver}\right)^2 + \left(\frac{H}{f}\cdot\sigma_{ob}\right)^2}$$

An estimated accuracy for the project data is given below. The full image area is used as the effective area. As mentioned earlier chances are that central parts of the image will more often be used. This example doesn't take this fact into account, which means that the actual mean standard deviation is probably smaller[3]:

$$\sigma_{ogns} = \sqrt{\left(\frac{\sqrt{0.115^2+0.115^2}}{0.303\cdot\sqrt{3}}\cdot 0.15\right)^2 + \left(\frac{1500}{0.303}\cdot 0.000014\right)^2} = \sqrt{0.0465^2+0.0693^2} = 0.083m$$

When using smaller scale images and wide angles lenses, the errors from relief displacements will usually be larger than the orientation. Example:

Photograph of scale 1:25,000, wide angle lens with a mean vertical error of 1m:

$$\sigma_{ogns} = \sqrt{\left(\frac{\sqrt{0.115^2+0.115^2}}{0.152\cdot\sqrt{3}}\cdot 1{,}0\right)^2 + \left(\frac{3800}{0.152}\cdot 0.000014\right)^2} = \sqrt{0.618^2+0.35^2} = 0.71m$$

[2] Based on personal consultancies with Keld Dueholm

[3] The result is based on the accuracy of a DSM with well-defined points only and therefore some larger errors might occur.

2.5 Summary

This chapter introduced the concept of orthophotos, along with an explanation of the cause of relief displacements and the problems they create both before and after orthophoto rectification. True orthophotos was then defined as an ortho rectification that not only includes the terrain for reprojection, but also at least the buildings and bridges. Orthophotos was then compared to the advantages of true orthophoto imagery. At last the expected accuracy of an orthophoto was described, though only a rough estimate of the standard deviations could be obtained.

Chapter 3 Digital Surface Models

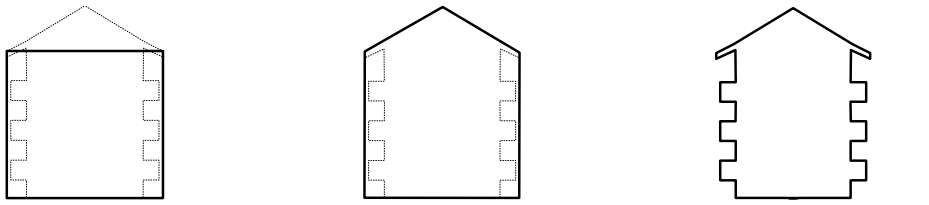
The ortho rectification requires a good knowledge of the geometric shape of the objects that the photograph contains. The photograph and knowledge of how the camera was oriented during the exposure is only enough to reconstruct in which direction the objects in the image are, but not how far they are located from the camera. With a model of the objects, the distance to the objects can be found by intersecting the rays with the model.

3.1 Surface models

The digital terrain model (DTM) is a representation of the shape of the earth, disregarding buildings and vegetation. It is the most common form of elevation model, and is available in both global sparse datasets or in local often denser and more accurate datasets.

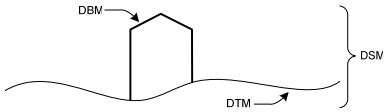
The digital building model (DBM) contains the surfaces of the buildings. The level of detail in the DBM varies. Some only contain the roof edges and therefore the roof construction is missing. Most maps already contain the roof edges and a height of the roof. This can be used for creating the simple DBM. More advanced DBMs also contain the ridges on the roof and is thus a more exact representation of the surface.

Eaves and details on the walls would require terrestrial photogrammetry, so a large DBM with this amount of detail would be very expensive. Furthermore the wall details are somewhat unimportant for creating true orthophotos, since only the top most objects would be visible. For instance if the eaves cover objects below them, these objects will not be visible in a correct true orthophoto. The DBM can therefore be simplified as illustrated on figure 3.1-center.

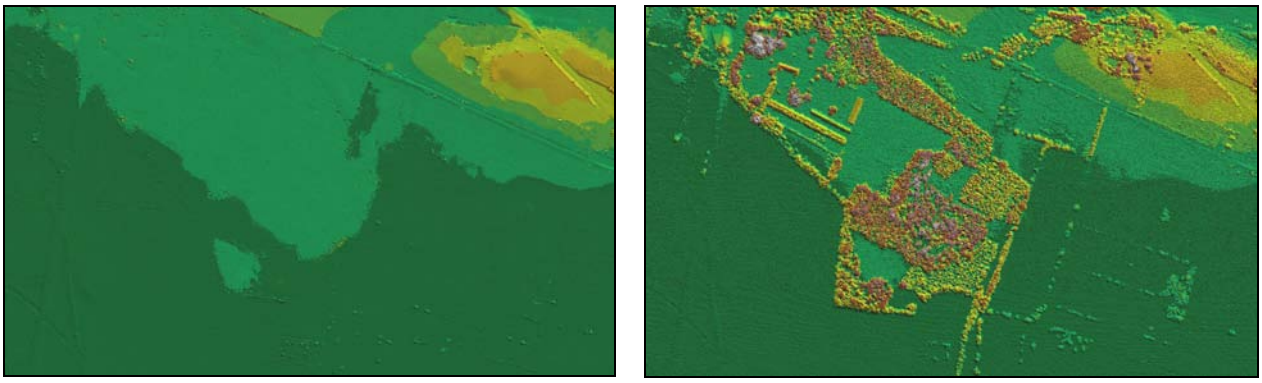


Roof edges | Roof ridges and edge of eaves | Wall details and eaves

Figure 3.1 - Three levels of detail in the DBM.



In theory the digital surface model (DSM) should contain all static objects on the ground, including terrain, vegetation and buildings. The vegetation can be very hard to model, so often a DSM will only contain terrain and buildings as is the case with the DSM used in this project. Thus the combination of the DTM and DBM is a DSM. Using a laser scanner that sweeps the surface, a DSM resembling the real surface more closely can be attained, but the density and accuracy is normally not comparable to what can be attained by standard photogrammetric measurements. Furthermore there is no automatic edge detection which makes the generated DSM poor along sharp edges like the edge of a roof.



DTM | DSM

Figure 3.2 - Example of a DSM generated with a laserscanner [14]. It is possible to "shave off" buildings and vegetation and thereby create a DTM.

3.2 Surface representation

There are several methods of representing the shape of a terrain. The most well-known is contour lines in a map, where each line follows a constant height in the terrain. The closer the lines are to each other the hillier are the terrain.

For data processing purposes two surface representations are the most common: The Triangulated Irregular Network and the grid.

3.2.1 Triangulated Irregular Network

A Triangulated Irregular Network (TIN) consists of a series of height measurements throughout the surface. These points are afterwards connected to a network of triangles. This means that the height at a given point is found by interpolating between the vertices of the enclosing triangle. This gives a rough description of the surface as illustrated on figure 3.3.

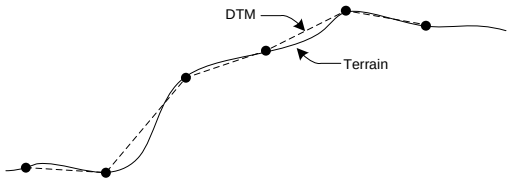


Figure 3.3 - By interpolating points on the terrain, a rough representation of the terrain can be obtained.

There is a large amount of ways that the points can be connected to form a network of triangles. The Delaunay triangulation algorithm tries to connect the points so that it maximizes the minimum angles in all the triangles [6]. This triangulation will also have the smallest possible total edge length. Delaunay triangulation has the benefit of not creating triangles that are long and narrow, but triangulates between the nearest points.

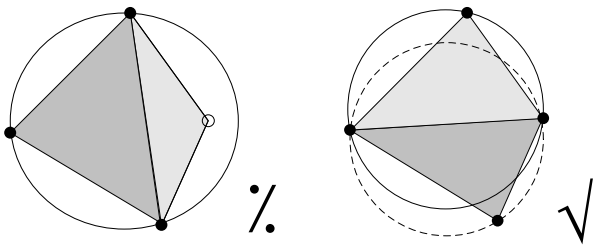


Figure 3.4 –The circumcircle in any triangle in a correct Delaunay triangulation does not contain any points within the circumcircle. The triangulation on the left is therefore not a valid Delaunay triangulation [6].

The basic Delaunay triangulation cannot handle abrupt changes in the surface like cliffs, without a very dense network. A modified algorithm is able to handle *breaklines* which supplements the points in the surface with lines. The breaklines are placed along “edges” in the terrain where the sudden elevation changes runs. The vertices of the breaklines are included as points in the triangulation. A constraint is added that prevents edges of the triangles to traverse the breaklines. This will generate triangles whose edges will only follow the breaklines without crossing them.

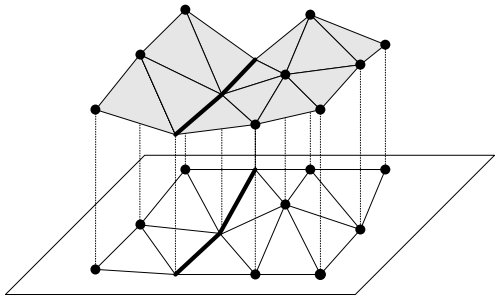


Figure 3.5 - An example of a TIN created with points and breaklines.

A limitation for triangulation algorithms is that it can't handle vertical objects, since this would require more than one height at the same point. It cannot handle overhanging surfaces either as illustrated on figure 3.6. This poses a problem for DSMs that include buildings where walls and eaves can't be triangulated correctly. There is nothing that prevents a TIN from containing these surfaces, but standard triangulation algorithms cannot create these TINs. A TIN that only has one height in any point is often referred to as a 2½D TIN, as opposed to a 3D TIN that can be of any complexity.

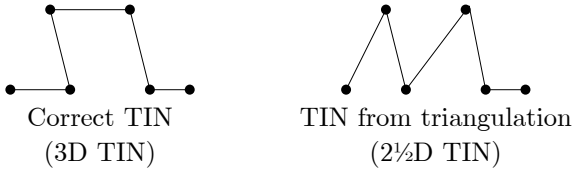


Figure 3.6 - Cross sectional view of two triangulations of the same points. The Delaunay algorithm (right) triangulates to the closest points (measured horisontally), and therefore can't handle overhanging surfaces like the one to the left.

Another aspect of TINs is *thinning*. In very flat terrains a very dense network of points is not necessary. For instance can a football ground be represented by four points and any points inside it can be removed. This is relevant for laser scanned data where the surface is covered with a huge amount of dense points. Thinning can typically reduce laser scanned data with 80-90 % depending on the type of terrain and the required accuracy of the DSM.

3.2.2 Grid

The grid is, oppose to the TIN, a regular net of points. It consists of points with regular spacing in both the x and y direction. The grid can be compared to a matrix, where each cell represents the height in the cell. It can be overwhelming or impossible to measure all the heights in a regularly spaced grid, so the grid is usually created on basis of other datasets like TINs or contour lines in existing maps. Missing grid points can also be interpolated by various methods, for instance linear interpolation or *kriging*.

The grid has some computational benefits since it can be handled like a matrix or processed as a raster image. It does have limitations in accuracy since the detail level is dependent of the grid size. For instance rapid elevation changes cannot occur within the grid size, and has the same limitations that a 2½D TIN has compared to a 3D TIN.

The grid can easily be converted to a TIN by triangulating the pixels and if necessary reduce the points with a thinning. A TIN can also get converted to a grid by sampling the points on the TIN. An example of the 3D city model converted to a grid is illustrated on figure 3.8.

An advantage over the TIN is that it is easier to find the height at a given location. Only a simple calculation is required to locate the nearest grid points and interpolate between them. In a TIN the triangle that surrounds the location must first be identified followed by an interpolation between the triangle's vertices [2].

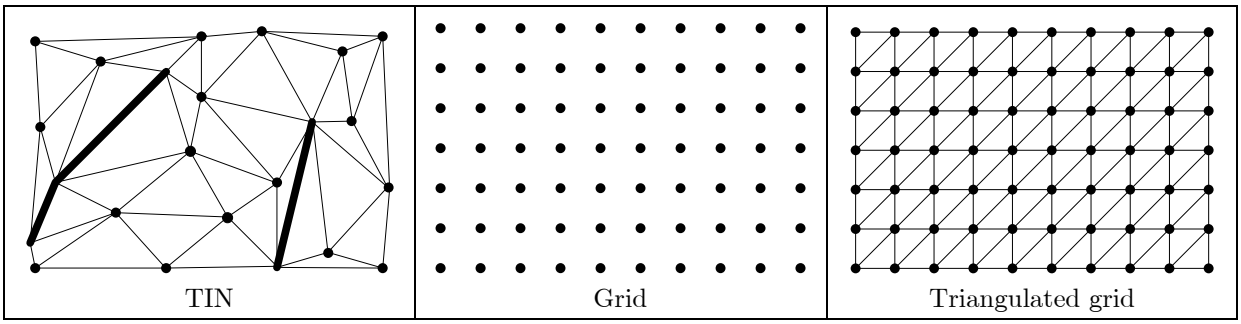


Figure 3.7 – Converting from a TIN to a grid can cause a loss of accuracy. The grid to TIN is on the other hand exact.

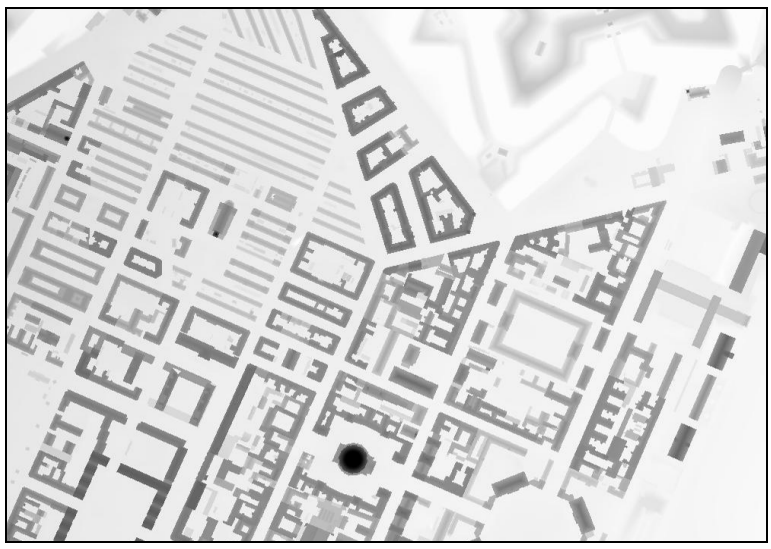


Figure 3.8 – Grid of a part of Copenhagen, where the intensity of each pixel corresponds to the height at the centre of the pixel. A low intensity (dark) represents taller buildings. The grid is created from a TIN by sampling pixel by pixel.

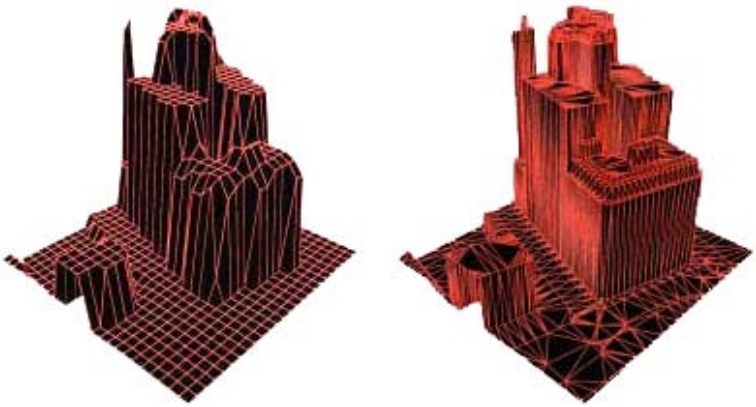


Figure 3.9 – A Grid DSM (left) is not sufficient to give a good description of the buildings, without using a very dense grid. The TIN (right) gives a much better approximation [9].

3.3 Copenhagen's 3D city model

BlomInfo A/S has created a detailed 3D city model of Copenhagen that is a good foundation for creating true orthophotos. It contains all buildings larger than $10m^2$, and has most of the details on the rooftops. The model does have some limitations in relation to completeness, detail and accuracy. These limitations are related to the production method and production cost. For instance small bay windows are only included if they extent out to the edge of the wall. The edges of the roof are measured at the position of the wall. This is because the buildings in the existing map are measured to the walls by a surveyor and a direct spatial relation between the base and top of the buildings was wanted. The roof construction is traced along the outline of the existing map. This results in a model where the eaves are not included, and the roof only extends out to the walls of the building (figure 3.10). Since these details aren't included, the eaves will risk being "left" on the terrain during the rectification, and smaller objects on the rooftops will not get rectified either. The missing eaves are only a problem for a few of BlomInfo's 3D city models, since most of the models are measured to the actual edge of the roof.

The trees are also included during the registration. They are represented by a circle, where the centre of the circle is placed at the highest point in the centre at the tree, and the radius defines an approximate width. This is a rough approximation and since they are only represented as circles and not as surfaces, they aren't part of the DSM. An example of the level of detail in the model can be seen on figure 3.11. Some variations do occur on request of the client. BlomInfo has also produced city models where the walls are placed at the edge of the eaves, and where all roof windows are included. The 3D city model standard is based on the Danish TK3 mapping standard. The accuracy and registration detail is therefore comparable to that of TK3.

The Copenhagen city model is created photogrammetrically by digitizing the contours of the rooftop construction as lines. BlomInfo has developed a method that is able to handle a triangulation to 3D TINs. The lines are used as breaklines in a specially adapted TIN triangulation based on the Delaunay triangulation.

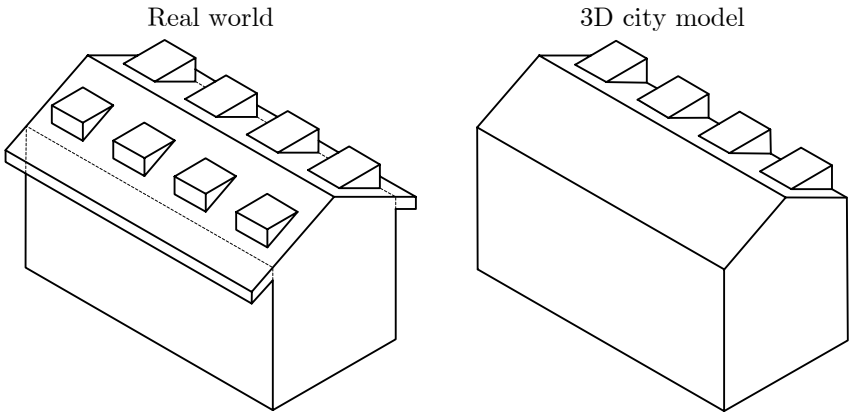


Figure 3.10 – The building on the right illustrates some of the simplifications that are done to the model. Eaves are removed, and small constructions on the roof is only included if it is a part of the edge of the building.

The breaklines in the model are categorized, and on the basis of the categories, the triangulation method is able to detect what kind of surface a triangle belongs to. The triangles in the city model are categorized in four object types:

- Building bottom
- Building wall
- Building roof
- Terrain (excluding building bottom)

The categories can be used for enhancing a visualization. It is easier to interpret the model if the categories are assigned different colors, and when orthophotos are draped on the model, only roof and terrain are assigned the colors of the orthophoto leaving the walls with a neutral color.

3.4 Summary

This chapter introduced the concept of digital surface models. Two basic kinds of surface models were described; the irregular triangulated network and the grid. The surface model was described as a combination of a terrain model and the objects located on the terrain, for instance buildings and vegetation. Lastly the 3D city model of Copenhagen was introduced and analyzed. Simplifications and limitations of the city model were described where it was pointed out that certain features were left out of the model, such as eaves and roof windows.

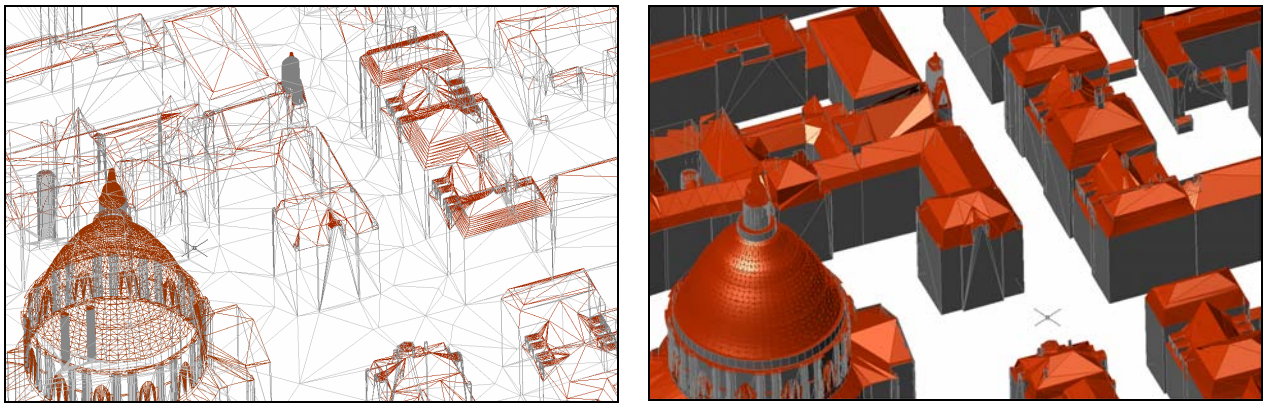


Figure 3.11 - Example of the detail in BlomInfos 3D city model. On the left a view of the TIN. On the right the triangles in the TIN have been shaded.

Chapter 4 Coverage analysis

The imagery that are available for use in this project is somewhat ideal to use for true orthophotos, it is not what is normally used for photogrammetric mapping. It would be appropriate to reuse existing imagery instead. The imagery that was used for producing the 3D city model was taken with a different camera lens and from a different altitude as shown in table 4.1. This section tests different photo shooting setups, to determine what amount of coverage that can be expected from different setups.

	Production images	**Project images**
Flight Altitude	750 m	1,500 m
Lens type	Wide angle	Normal angle
Focal length (field of view)	153 mm (94°)	303 mm (56°)
Forward overlap	60 %	60 %
Sidelap	20 %	60 %

Table 4.1 – The production image column describes the images that were used for creating the 3D city model. Some additional normal angle images (right column) were made for testing their practical uses for true orthophoto creation.

4.1 Overlapping

Since the relief displacements of buildings cause *blindspots*, especially in dense suburban areas, it is important to have images that sufficiently overlap. The overlapping images will have relief displacements in different directions and often of different magnitude. By combining the images, they will supplement each other so that data obscured in one image most likely is visible in another.

For photogrammetric uses, the most common imagery is made with a 60 % overlap in the flight direction (forward overlap) and 20 % to the sides (sidelap) with a wide angle lens. This has some benefits when compared to imagery taken from a higher altitude with a normal angle lens. The lower flight altitude results in better measuring accuracy in Z, but as shown in figure 2.4 also larger displacements.

Since the sidelap is only 20%, the area just outside the overlap will have a lot of obscured areas that aren't supplemented by the neighboring images. This is worst farthest from the flight lines and outside the overlaps, since the relief displacements increase away from the nadir point. This can be reduced by adding additional flight lines, and thus increasing the sidelap to 60 %. This is more expensive to produce, but should result in lesser obscured areas.

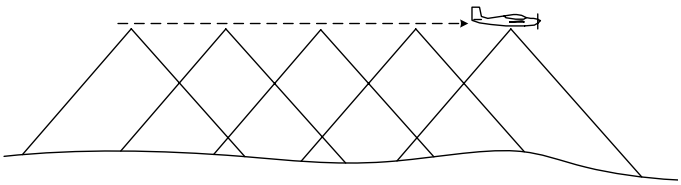


Figure 4.1 - Overlapping in the flight direction is usually around 60 % to provide sufficient data for stereo measurements. The sidelap is commonly 20 %, but can be supplemented by a flight line in-between, thus increasing sidelap to 60 % [7]

4.2 Test setup

To test the amount of obscured data, a comparison was made between seven scenarios, each in two types of built-up areas. These scenarios are based on the previous reflections on relief displacements, sidelap and cost. Therefore different sidelap, lenses and flight altitude are tested. Except for scenario 3, the forward overlap is 60 %. The seven scenarios are:

Wide angle lens (f=152mm):

1. 20 % sidelap
2. 60 % sidelap
3. 20 % sidelap and 80% forward overlap

Normal angle lens (f=304mm):

4. 20 % sidelap
5. 60 % sidelap

Normal angle lens (f=304mm) – Double flight altitude:

6. 20 % sidelap
7. 60 % sidelap

Scenario 1 is the most common used setup, and scenario 7 corresponds to the imagery that is available in this project. The coverage of each image is the same in scenario 1, 2, 6 and 7. Case 4 and 5 requires many more images, since the coverage is smaller. Adding extra images on a flight line is more cost effective than adding extra flight

lines. Scenario 3 tests if adding extra images along the flight line will have any profitable benefit.

The test areas are of very different built-up character; especially regarding building heights and density. One area is the central part of Copenhagen, which mostly consists of five story buildings and narrow backyards. The other area is the central part of Ribe, which in general consists of smaller buildings and larger backyards.

The image data is constructed so that the overlapping is exact, and that case 6 and 7 perfectly aligns with case 1 and 2. This is due to the fact that the focal length is doubled as well as the flying height. The area selected for processing is the part of the overlapping pattern that is repeated in both the flight direction and across it, if the flight lines were extended or additional lines were added. This area is illustrated at figure 4.2, where the grey area, is the area that will be processed. Because of the smaller coverage of the imagery in case 4 and 5, many more images are needed to cover the same area with this setup.

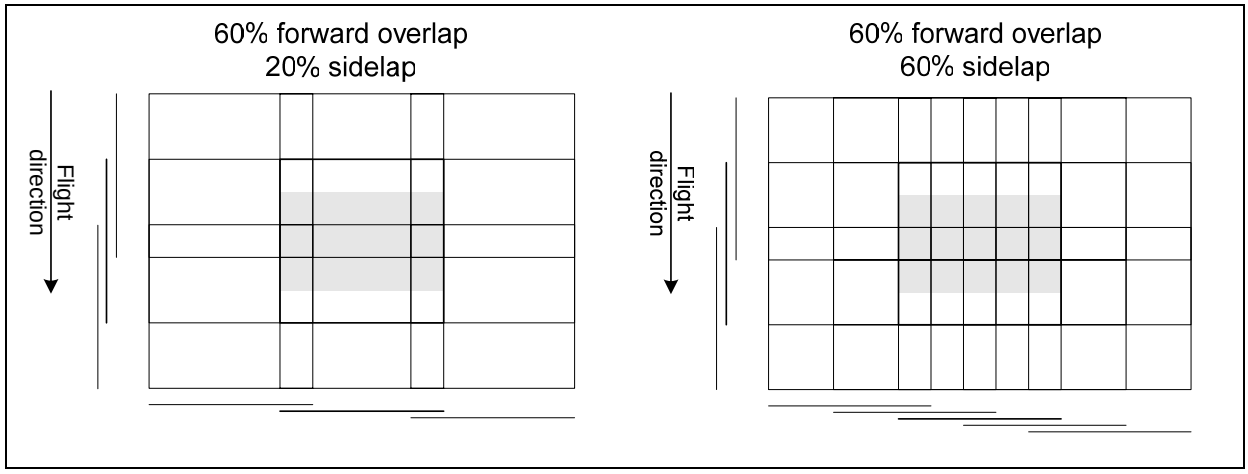
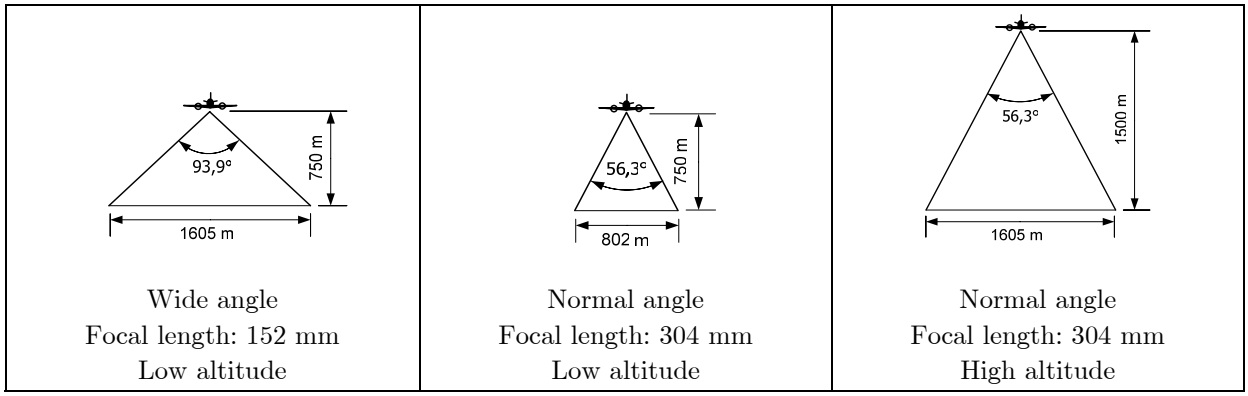


Figure 4.2 – Illustration of the overlapping pattern used for the test. The gray area is to be processed for visibility. The left image have three flight lines with 20 % sidelap; the right, five flight lines and 60 % sidelap. The center image is shown with thick lines.



Wide angle Focal length: 152 mm Low altitude	Normal angle Focal length: 304 mm Low altitude	Normal angle Focal length: 304 mm High altitude

Figure 4.3 – The different flight altitudes and focal lengths give different coverages. The right and left setups results in the same coverages. Angles and horisontal distances are in the diagonal of the image.

The testing was done by checking for the visibility from every camera. This results in a visibility map for each camera position, which shows the obscured areas as black parts in the map (figure 4.4). This actually resembles the shadows that would be thrown if a light source would be placed in the camera point. By combining all the visibility maps generated, the resulting image will only contain black pixels where they are obscured for all the cameras. The number of black pixels remaining will give an idea of how good or bad a given case will be compared to the other cases. Again if a light source were placed in all the camera points, only those areas that are not lit up by any light source are completely obscured areas.

Figure 4.4 – Visibility map. The black areas are obscured from the camera. They resemble shadows from a light source placed at the camera point.

4.3 Test results

The test area is 1135 x 681 meters. The resolution used for this test is 0.25 meters, resulting in approximately 12.4 million samples per source image. Table 4.2 shows some statistical results.

	Focal length	Forward overlap	Sidelap	Altitude	Images	Obscured pixels			
						Copenhagen		Ribe	
1	152 mm	60 %	20 %	750 m	9	592,016	47.90 ‰	90,032	7.28 ‰
2	152 mm	60 %	60 %	750 m	15	95,707	7.77 ‰	4,356	0.35 ‰
3	152 mm	80 %	20 %	750 m	15	420,799	34.05 ‰	69,742	5.64 ‰
4	304 mm	60 %	20 %	750 m	15	183,885	14.88 ‰	42,634	3.45 ‰
5	304 mm	60 %	60 %	750 m	35	12,864	1.04 ‰	1,029	0.08 ‰
6	304 mm	60 %	20 %	1500 m	9	197,752	16.00 ‰	42,665	3.45 ‰
7	304 mm	60 %	60 %	1500 m	15	12,980	1.05 ‰	1,090	0.09 ‰

Table 4.2 – Results of the visibility tests. Results are given in number of obscured pixels and per mille of total.

It is likely that there will be more occluded pixels far from a flying line where there is only little overlap. Since the flight lines have been placed so that they are parallel to the columns in the image, summarizing the columns gives an estimate of the obscured areas with respect to the flight lines and the sidelap. The results of this are visualized in Figure 4.7. Figure 4.7c particularly illustrates this problem. The obscured pixel count falls significantly around the 20 % sidelap. Furthermore the number of obscured pixels is also very low close to the flight line. In figure 4.7e, this is also illustrated, where the image centers are at 10 %, 50 % and 90 %, and the overlaps around 30 % and 70 %. The extra forward lap in scenario 3 didn't give any significant decrease in obscured areas.

One thing to note is that a big overlap with a wide angle lens (scenario 2) is better than a normal angle high-altitude flight with lesser sidelap (scenario 6). In general the extra sidelap is much more effective than using a combination of lenses and flight altitudes that cause less relief displacement.

The areas that were obscured were compared to a map, in order to determine what kind of areas that were left obscured. Some typical examples are illustrated on figure 4.8. When comparing figure 4.8a-d the influence of the extra overlap is significant. The normal angle lens only decreases the size of the obscured area. Figure 4.8e-f illustrates the problem with a wide angle lens, where the relief displacements are so large that each photograph only barely covers the sidewalks on each side.

Though the two areas show the same tendencies for each scenario, they also display a significant difference. The Ribe test area is much "better", which is something that could be expected. Therefore the density and height of buildings in the area that is to be generated true orthophoto for should also be taken into account when planning the photographing. Scenario 1 would often be sufficient in an area like Ribe with less than 1 % obscured areas, but in Copenhagen the corresponding number is ≈ 5 %.

4.4 Summary

On the basis of DSMs of Copenhagen and Ribe, an analysis of the expected coverage where generated. Tests results were based on seven combinations of forward overlap, sidelap, low/high altitude and normal/wide angle lenses. It was concluded that an increased sidelap provides better coverage than shooting from a higher altitude with a narrower lens.

The test showed that with the standard imagery used for photogrammetric mapping, using wide angle lenses and 20 % side lap, an area of almost 5 % can be expected to be obscured for the central part of Copenhagen, but only 0.7 % of Ribe. If sufficient coverage should be obtained for Copenhagen, 60% sidelap should be considered.

Figure 4.5 Overview map of the Copenhagen test area.

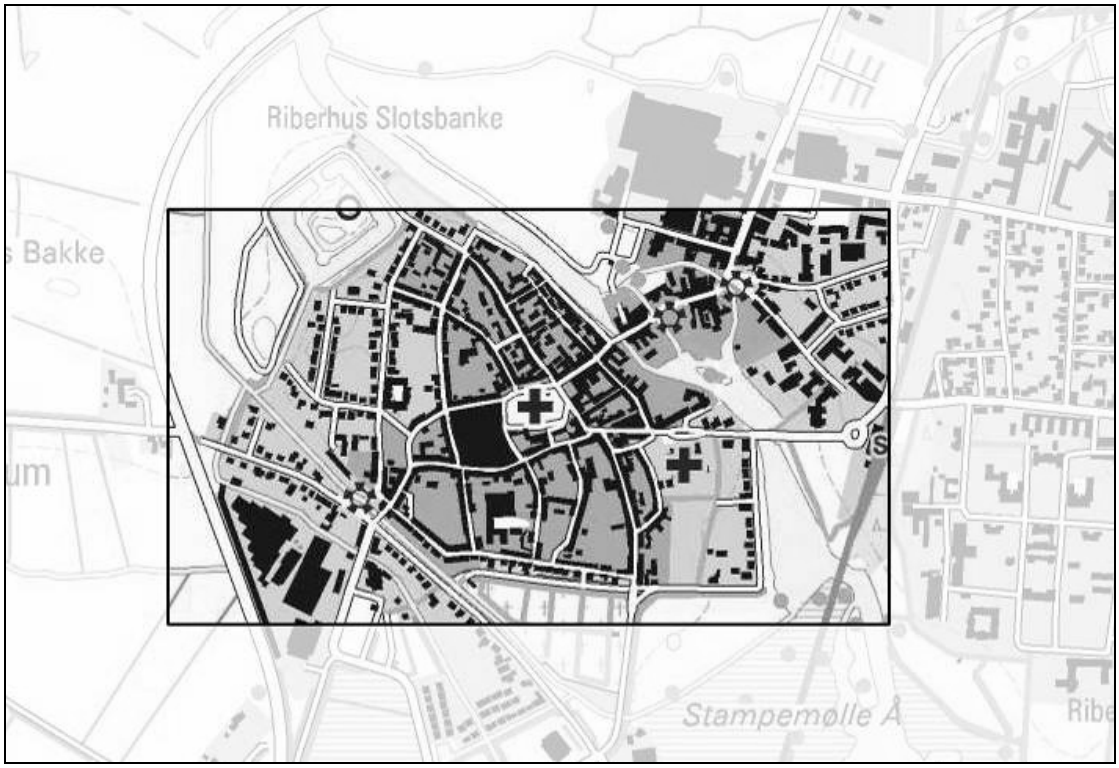


Figure 4.6 - Overview map of the Ribe test area.

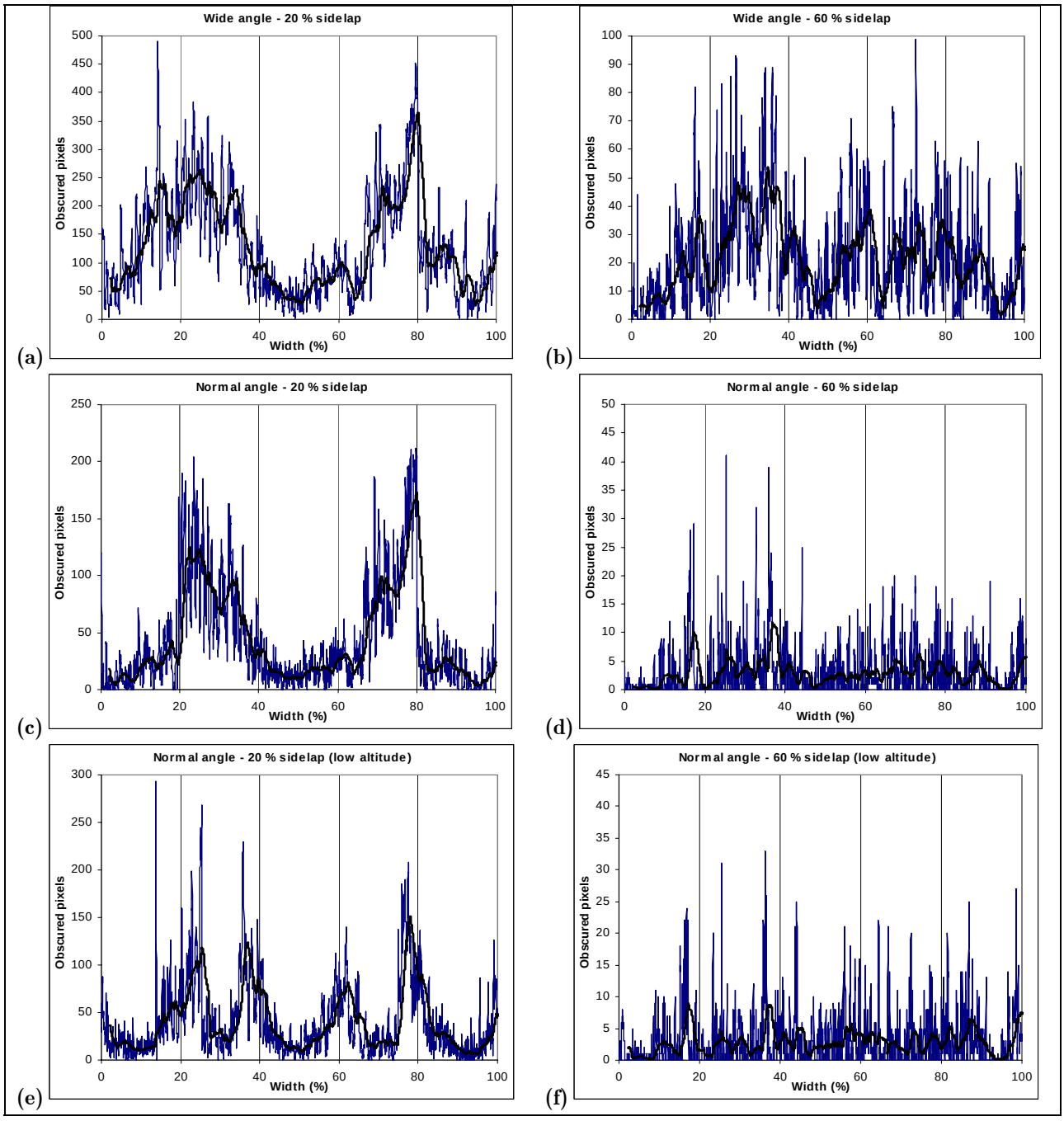


Figure 4.7 - Cross sectional view perpendicular to the flight directions of the Copenhagen test area. The sidelap is very evident in the 20% sidelap images, where the pixel count is much higher in the non-overlapping areas. The width corresponds to the width of the full test area in percent. (a) and (c) have overlap between 0-20 and 80-100. (b) and (d) have overlap everywhere and double overlap between 40-60. (e) have overlap between 25-35 and 65-75. (f) has six double overlaps and five triple overlaps. The Ribe test area shows the exact same tendencies but at a much lower scale.

(a) Wide angle camera, 20% sidelap (outside sidelap)

(b) Wide angle camera, 60% sidelap

(c) Normal angle, 20 % sidelap (outside sidelap)

(d) Normal angle camera, 60 % sidelap

(e) Wide angle, 20 % sidelap (inside sidelap)

(f) Normal angle, 20 % sidelap (inside sidelap)

Figure 4.8 – Map of the buildings overlayed with the completely obscured areas (Copenhagen test area)

Chapter 5 Design description

This section outlines the general method for creating true orthophotos that is used in this project. The method is a step-by-step procedure, and each step is further explored in other chapters. The method is partly based on approaches of other true orthophoto applications, while their limitations are sought to be overcome. Several of the applications capable of creating true orthophotos to different extents were described in [4].

5.1 Limits of other True Orthophoto applications

In [4] it was pointed out that many true orthophoto applications were based on orthophoto software that was extended to be able to create true orthophotos. This caused a general limitation in that they were only able to handle 2½D TINs. The limitation rules out any vertical surface; for instance walls. It also makes it impossible to handle complex objects like eaves.

Some applications handle vertical walls by slightly tilting them inwards at the top. The only reason for doing this is to be able to create a 2½D TIN by standard triangulation methods. As long as the offset is much smaller than the output pixelsize, a 2½D TIN is sufficient, but converting from 3D TIN to a valid 2½D TIN is not always a trivial task.

The undersides of the eaves are not necessary for creating true orthophotos, since an orthophoto wouldn't show what is beneath the eaves anyway. They still pose a problem for applications not able to handle 3D TINs, since many 3D city models will contain eaves. Reusing this data would require a pre-process that removes eaves by removing the undersides and moving the walls to the edge of the eaves.

One of the design goals in this project is to be able to handle 3D TINs. Investigations in [4] showed that there actually were no large obstacles in handling 3D TINs over 2½D, since the 2½D limitation only lay in the triangulation of the surface model. Triangulation is not a part of this project, but instead has its starting point at the

already triangulated DSM. Both 2½D and 3D TINs can still be created using existing software for the triangulation. The city model that is used in this project is an existing 3D TIN of Copenhagen. It generally doesn't contain eaves, but still have vertical objects, like building walls.

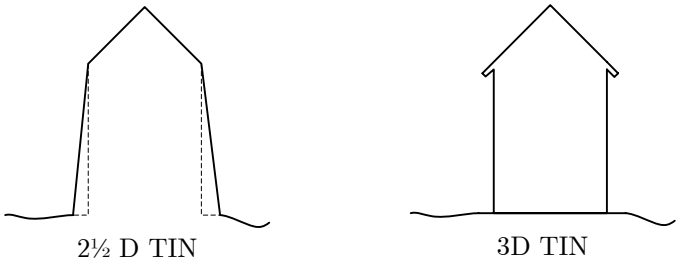


Figure 5.1 - 2½D vs. 3D TINs. The 3D TIN can contain vertical surfaces or several surfaces over the same point.

One of the most difficult tasks in creating true orthophotos is placing the seamlines during the mosaicking. One of the most promising applications was the Sanborn METRO True Orthophoto application ([8],[9]) that uses several methods for determining the best image to use in each pixel. It uses a score-method for each pixel, and the pixel with the highest score is assigned anywhere where a master image doesn't provide coverage. This is a method that is fairly simple to implement, yet giving many ways of calculating the score and thus affecting the final mosaic. The method has many similarities with maximum likelihood classifications, and several well-known methods from this field can possibly be applied to enhance the final mosaic. Having one master image and several slave images is a limitation, especially when creating large true orthophotos whose extents are far larger than that of the master image. Instead by just treating each image equally this limitation can be overcome.

The method used here requires that all input images are ortho rectified and visibility tests created for each image. Another approach would be to rectify only one "master" image, and afterwards only rectify pixels from other images where pixels were obscured in the master. This would decrease the amount of processing time, but also limit the options of creating a good mosaic.

5.2 Creating true orthophotos – Step by step

The overall true orthophoto generation process can be boiled down to the following crucial steps:

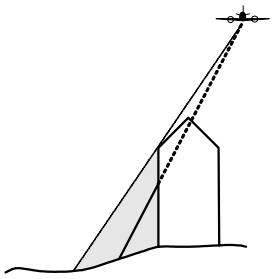
1. Rectify images to orthophotos.
2. Locate obscured pixels (visibility testing)
3. Color match orthophotos.
4. Create mosaic pattern and feathering of seamlines.

5.2.1 Rectification

The orthophoto rectification is a standard well-known method where each output pixel is traced back to the pixel in the input image. When tracing back to the input image, the ray rarely hits in the center of a pixel. This is normal when resampling an image. The are several methods for interpolating between pixels, and some of them are: Bicubic, bilinear and Nearest neighbor and can all be found in [1]. Nearest Neighbor were used in this project due to its simplicity, but any other could be selected for better results. Nearest neighbor selects the pixel value from the pixel whose center is closest to the incoming ray.

The rectification method is described in chapter 2. Knowledge of the camera and the mathematics needed are described in chapter 6. The actual raytracing implementation is described in chapter 7.

5.2.2 Locating obscured pixels

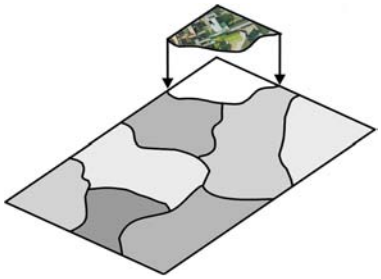


Locating the obscured pixels is crucial to a true orthophoto. We need to register any ray that gets "blocked" by other objects in the DSM on its path from a point on the surface to the camera. If it is blocked, the point on the DSM is not visible from the camera, and should be registered as obscured. Since we are already processing rays from the DSM to the camera during the rectification step, it would be appropriate to check for visibility at the same time. This means that the rectification and visibility test will run simultaneously. The visibility test is part of the raytracing and is described in chapter 7.

5.2.3 Color matching

When the images are mosaicked, any relatively large differences in color between each orthophoto will make the seamlines visible and the final result poor. This process could be done before the rectification, but by using the orthophotos and the knowledge of visibility, the color matching can be strengthened by only analyzing pixels that are spatially comparable amongst the orthophotos. The radiometric differences are investigated and methods to compensate for them are dealt with in chapter 8.

5.2.4 Mosacking and feathering



The mosaic process creates a pattern of classes for the final true orthophoto, where each class corresponds to the orthophoto from where pixels should be inserted at. As mentioned earlier, each pixel has a score for each class. The score can rely on several parameters like distance to nadir or

distance to any obscured pixels. The score functions are treated in chapter 9.

Feathering the orthophotos along the seamlines to conceal them is the last step before the actual orthophotos are merged to a final true orthophoto mosaic. This will ensure that any remaining differences from the color matching will be less visible. The feathering method is described in chapter 9.

The true orthophoto process is illustrated on the diagram below.

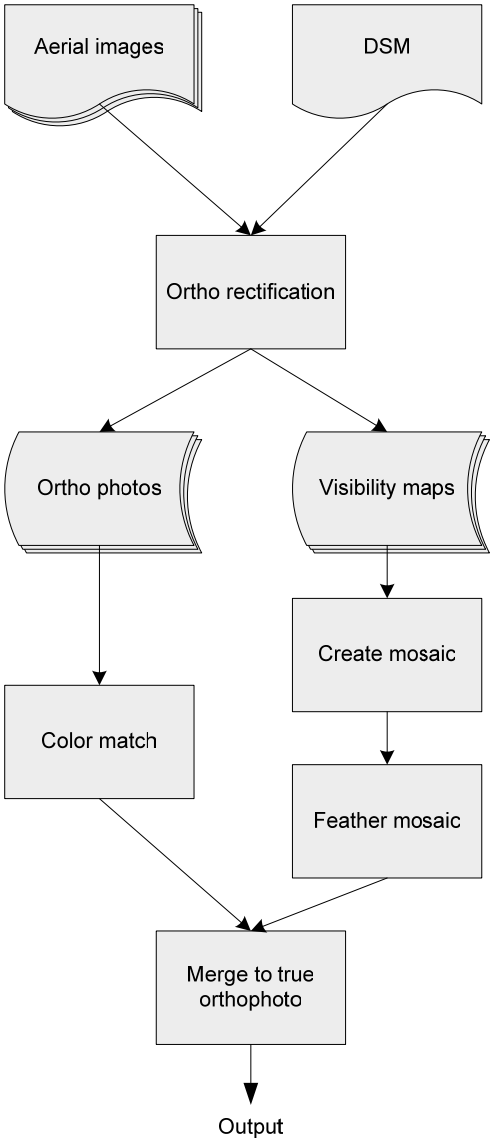


Figure 5.2 – Principle sketch of the true ortho rectification process.

5.3 Implementation

The true ortho application are implemented in two parts. The first part is basically an orthophoto application that also identifies the obscured pixels. It also included loading, indexing and preparing the TIN. This application is a Microsoft Windows application written in C#.NET. The second part of the implementation handles the steps from orthophotos to true orthophoto. This includes color matching, mosaicking and feathering. This part has been developed using MATLAB.

Initially the goal was to have one application written in C#.NET that could handle the whole true orthophoto process. Since MATLAB is a very strong tool for raster data processing, a portion of the implementation was instead moved to this platform. With MATLAB it was easy to test several methods and change the parameters during testing. The MATLAB code is easy to read and understand, and is therefore also good for illustrating the methods used.

It is possible to compile the MATLAB code at a later stage and incorporate it into the C#.NET environment. It is very hard though, and several time-consuming problems arose with this, and was therefore dropped from this project. The basics of how to implement MATLAB scripts in C#.NET can be found in [11], [12] and [13].

The orthophoto application can be found on the companion CD and a userguide in Appendix C. MATLAB code snippets is also available on the CD and in Appendix B.

Chapter 6 The Camera Model

In order to relate an image of the world to a world coordinate system, it is necessary to be able to model the light rays in order to determine where the light rays came from. To do this, knowledge of the position and orientation of the camera and the inner geometry of the camera is needed. This model is normally described in two sets of orientations: the interior and exterior orientations.

The simplest camera is the pinhole camera that lets light in through a small hole and projects it onto a surface at the back of the camera (figure 6.1). The image projected to the back side is an image of the world scaled down to f/H, where f is the distance from the pinhole to the backside, and H is the distance from the pinhole to the object imaged. f is also known as the *focal length* or the camera constant.

The smaller hole a pinhole camera has, the better is the resolution, but the exposure time can also increase to several hours, which makes it practically unsuitable for most types of photography. Instead of a pinhole, a large lens is used that lets in more light, but due to inaccuracies in the lens also makes the camera model much more complex. The parametric description of the more complex thick lens camera is described in the "inner orientation" section below.

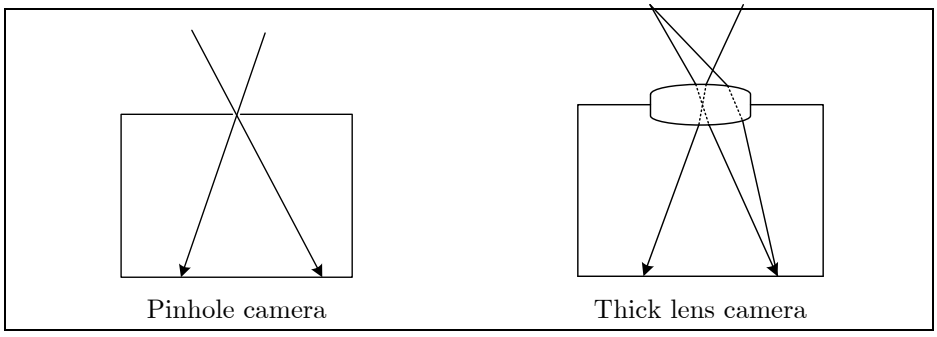


Figure 6.1 – The rays parsing through a pinhole camera is simple to model. The thick lens camera lets in more light because of the larger "hole" which decreases the exposure time significantly but also adds distortions due to inaccuracies in the lens and has a lower depth of field.

6.1 Exterior orientation

The exterior orientation of a camera specifies the position of the camera in the object space. This can be described by a position of the camera center (X_0, Y_0, Z_0), and three rotation parameters (Ω, Φ, κ) for the orientation as illustrated on figure 6.2. These parameters are found by a bundle adjustment or by fitting the image over a set of ground control points. The absolute orientation between the image coordinate system and the object coordinate system can be written as [2]:

$$\boldsymbol{Y}=\mu\boldsymbol{M}\boldsymbol{X}+\boldsymbol{T}$$

where $\boldsymbol{Y}$ is the vector of known world coordinates, μ is the scale factor, $\boldsymbol{M}$ is the rotation matrix from model coordinate to object coordinate system, $\boldsymbol{X}$ is the vector of model coordinates and $\boldsymbol{T}$ is the translation vector.

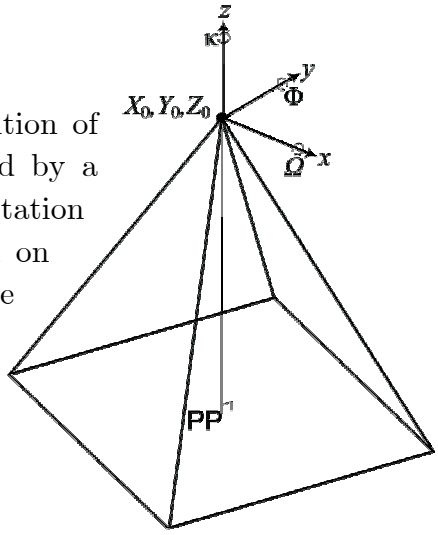


Figure 6.2 - The outer orientation parameters of a photograph. The principal point (PP) is located directly below the projection center.

The solution is nonlinear, so approximations are used as initial parameters. The initial X_0,Y_0,Z_0 parameters can be derived from a GPS receiver, and the rotations Ω, Φ (the tilt) can for a vertical photograph be approximated to zero. κ is approximated by the direction of the flight, which again can be found by relating the photograph to the position of preceding or subsequent photographs [2].

The relation between the camera space (x_c,y_c) and object space (X,Y,Z) consists of a scale, a translation and a rotation in three dimensions. These operations are expressed in the *colinearity equations* [1]:

$$x_c = -f\,\frac{r_{11}(X-X_0)+r_{21}(Y-Y_0)+r_{31}(Z-Z_0)}{r_{13}(X-X_0)+r_{23}(Y-Y_0)+r_{33}(Z-Z_0)}$$

$$y_c = -f\,\frac{r_{12}(X-X_0)+r_{22}(Y-Y_0)+r_{32}(Z-Z_0)}{r_{13}(X-X_0)+r_{23}(Y-Y_0)+r_{33}(Z-Z_0)}$$

The equation above uses a rotation matrix r, whose elements are:

$$
\begin{aligned}
r_{11} &= \cos(\Phi)\cos(\mathrm{K}) \\
r_{12} &= -\cos(\Phi)\sin(\mathrm{K}) \\
r_{13} &= \sin(\Phi) \\
r_{21} &= \cos(\Omega)\sin(\mathrm{K}) + \sin(\Omega)\sin(\Phi)\cos(\mathrm{K}) \\
r_{22} &= \cos(\Omega)\cos(\mathrm{K}) - \sin(\Omega)\sin(\Phi)\cos(\mathrm{K}) \\
r_{23} &= -\sin(\Omega)\cos(\Phi) \\
r_{31} &= \sin(\Omega)\sin(\mathrm{K}) - \cos(\Omega)\sin(\Phi)\sin(\mathrm{K}) \\
r_{32} &= \sin(\Omega)\cos(\mathrm{K}) + \cos(\Omega)\sin(\Phi)\sin(\mathrm{K}) \\
r_{33} &= \cos(\Omega)\cos(\Phi)
\end{aligned}
$$

The reverse calculation from camera to object coordinate system can be done if at least one coordinate is known in the object coordinate system. This is handy for approximating the *footprint* of a photograph. When calculating the footprint, the coordinates of the corners of the photograph are used as input to find the approximate corners of the footprint. The Z-coordinate is approximated to a mean level of the area, and the X and Y coordinates are then calculated by [2]:

$$
\begin{aligned}
X &= (Z - Z_0)\frac{r_{11}(x_c - x_0) + r_{12}(y_c - y_0) + r_{13}(-f)}{r_{31}(x_c - x_0) + r_{32}(y_c - y_0) + r_{33}(-f)} + X_0 \\
Y &= (Z - Z_0)\frac{r_{21}(x_c - x_0) + r_{22}(y_c - y_0) + r_{23}(-f)}{r_{31}(x_c - x_0) + r_{32}(y_c - y_0) + r_{33}(-f)} + Y_0
\end{aligned}
$$

6.2 Interior orientation

The lens in a camera doesn't have a single point where all rays parses through, but has a relatively large area to let in more light during the shot exposure time. The lens hasn't got a "perfect" surface, which means that the image gets somewhat distorted. Therefore the rays parsing from the object space to the image space, isn't a simple calculation. The interior orientation is trying to model the bundle of rays passing through the lens to the image plane. This includes the lens distortion, the focal length, and the distance between the principal point in the image plane and the image center. All these parameters are included in the *Camera Calibration Certificate* available for all metric cameras.

The center of the photograph is found by intersecting lines between opposite pairs of fiducial marks, also referred to as the fiducial center. The Principal Point is given with respect to the center of the photograph, as illustrated at figure 6.3. The Principal Point of Autocollimation (PPAC) is determined using a multicollimator, and is located very close to the principal point. PPAC serves as the origin of the image coordinate system [7].

When the image space rays aren't parallel to the incoming object space rays, it is caused by a distortion in the lens. The distortion consists of several components, where the radial distortion usually is the largest in a metric aerial camera. The radial distortion is modeled as an odd polynomial, describing the distortion with respect to the distance r to the Principal Point of Best Symmetry (PPBS). PPBS is the origin of the radial distortion and is determined during the calibrating process. In a metric aerial camera this point is located very close to the fiducial center and PPAC.

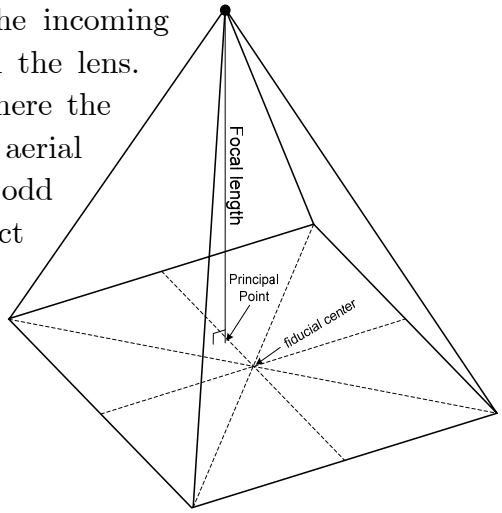


Figure 6.3 – The principal point isn't placed exactly over the image center, though it is greatly exaggerated in this illustration.

A sufficient model of the radial distortion is described by an odd polynomial [1]:

$$dr = a_1 r + a_3 r^3 + a_5 r^5 + a_7 r^7$$

The coordinate corrected for radial distortion can be found by:

$$x_r = (1+\Delta)x_c$$
$$y_r = (1+\Delta)y_c$$

where: $\Delta = a_1 + a_3 r^2 + a_5 r^4 + a_7 r^6$

When determining the radial distortion the distortion is measured in several points throughout the image during the calibration process. The result is a set of distortions with respect to the distance to PPBS. An example of a set of distortion data is given in table 6.1.

r (mm)	10	20	30	40	50	60	70	80	90	100	110	120	130	140	148
dr (µm)	0	-0.4	-0.3	0.3	0.7	1.3	1.7	1.8	1.6	1	0.1	-0.9	-1.7	-2.6	-0.1

Table 6.1 - Distortion measurements for a normal angle camera used in this project.

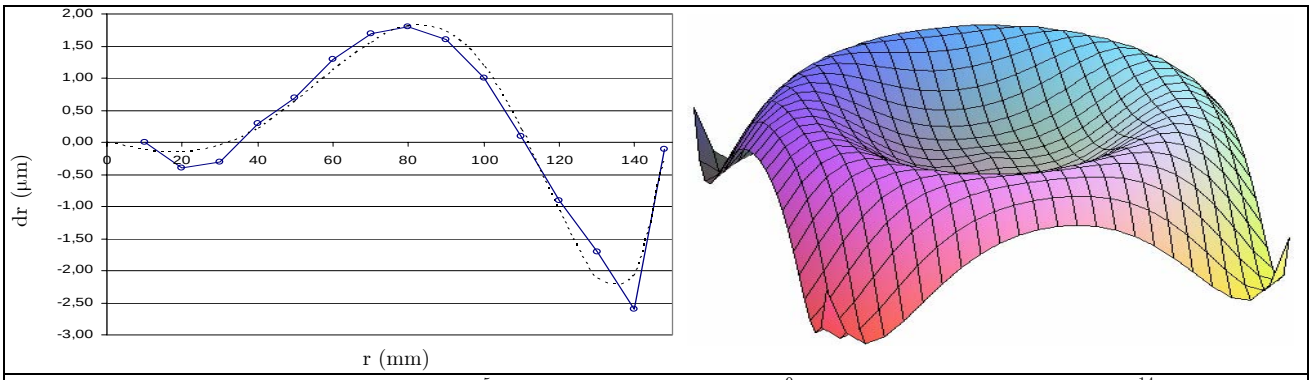
The a_n parameters can then be calculated by using a least squares regression. The regression can be described as four equations with four unknowns [4]:

$$\sum r_i^2 a_1 + \sum r_i^4 a_3 + \sum r_i^6 a_5 + \sum r_i^8 a_7 - \sum dr_i\, r_i = 0$$
$$\sum r_i^4 a_1 + \sum r_i^6 a_3 + \sum r_i^8 a_5 + \sum r_i^{10} a_7 - \sum dr_i\, r_i^3 = 0$$
$$\sum r_i^6 a_1 + \sum r_i^8 a_3 + \sum r_i^{10} a_5 + \sum r_i^{12} a_7 - \sum dr_i\, r_i^5 = 0$$
$$\sum r_i^8 a_1 + \sum r_i^{10} a_3 + \sum r_i^{12} a_5 + \sum r_i^{14} a_7 - \sum dr_i\, r_i^7 = 0$$

or on matrix form:

$$\begin{bmatrix} \sum r_i^2 & \sum r_i^4 & \sum r_i^6 & \sum r_i^8 \\ \sum r_i^4 & \sum r_i^6 & \sum r_i^8 & \sum r_i^{10} \\ \sum r_i^6 & \sum r_i^8 & \sum r_i^{10} & \sum r_i^{12} \\ \sum r_i^8 & \sum r_i^{10} & \sum r_i^{12} & \sum r_i^{14} \end{bmatrix} \begin{bmatrix} a_1 \\ a_3 \\ a_5 \\ a_7 \end{bmatrix} = \begin{bmatrix} \sum dr_i \cdot r_i \\ \sum dr_i \cdot r_i^3 \\ \sum dr_i \cdot r_i^5 \\ \sum dr_i \cdot r_i^7 \end{bmatrix}$$

The parameters given in table 6.1 have been applied to this equation and are illustrated at figure 6.4.



a1 : -0.012990936 a3 : $1.39116 \cdot 10^{-5}$ a5 : $-1.59 \cdot 10^{-9}$ a7 : $4.45 \cdot 10^{-14}$

Figure 6.4 – The curve for the measured distortions from table 6.1 shown as a line to the left. The calculated distortion curve is shown as a dotted line, and the estimated parameters are given below. To the right is an illustration of the corresponding distortion in 3D throughout the image area.

The focal length f is determined during the calibration process, and is the length that produces a mean overall distribution of lens distortion [7]. The focal point it therefore located directly above PPBS at a distance corresponding to the focal length.

The photograph is scanned to a digital image consisting of pixels. Therefore a relation between the image coordinate system (x_r,y_r) and the pixel coordinate system (x_p,y_p) is needed for the ortho rectification process. When the image is scanned, it is highly likely that the process will add both rotation and affinity to the image. This makes it necessary to apply both scaling, rotation, translation and affine transformations in the conversion between the two coordinate systems. The relation between the two coordinate systems can be described by two vectors, where their origin corresponds to the principal point, the directions corresponds to the axis direction and the length to the scale along each axis. This relation is illustrated on figure 6.5, and can be written as:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \begin{bmatrix} V_{x1} & V_{x2} \\ V_{y1} & V_{y2} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \end{bmatrix} + \begin{bmatrix} X_{PP} \\ Y_{PP} \end{bmatrix}$$

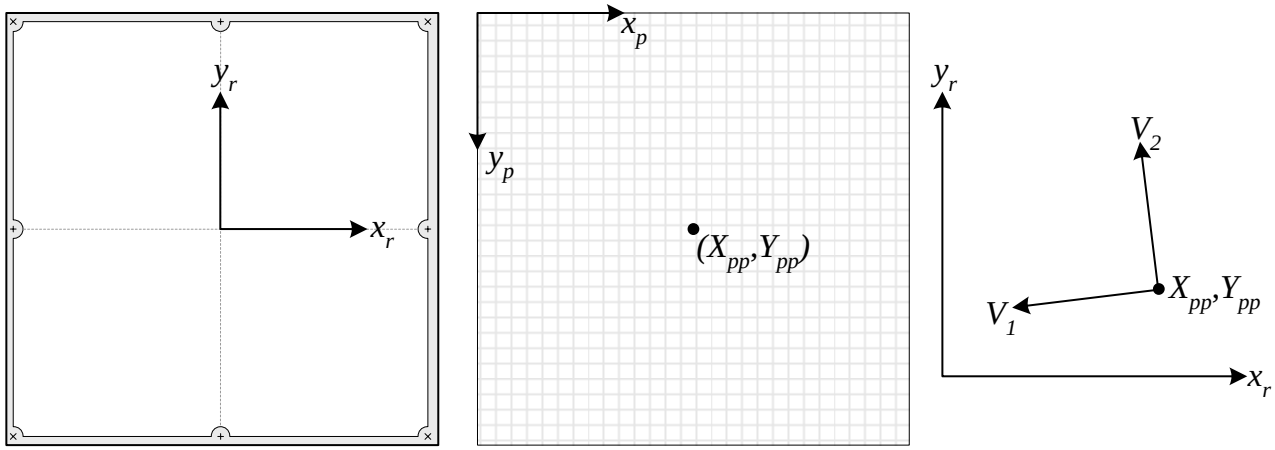


Figure 6.5 –The relation between the image coordinates (left) and the pixel coordinates (middle) is specified by two vectors V_1, V_2 and the principal point. It is worth noting that the pixel coordinate system is a left oriented coordinate system as opposed to most world coordinate systems.

An example of transformation parameters are given below:

$(X_{pp},\ Y_{pp})=$	(7889.625 , 7708.232)	pixels
$V_1 =$	(-66.6759 , -0.1604)	pixels / mm
$V_2 =$	(-0.1542 , 66.6788)	pixels / mm

These parameters specify that the principal point can be found at pixel (7890, 7708), and that each millimeter in the image covers approximately 67 pixels. It can also be seen that the angle between the two vectors isn't exactly 90°, which shows that a small amount of affinity is present. These transformation parameters are estimated by measuring the fiducial marks in the digital image. Since the locations of the fiducial marks are known in the image coordinate system, it is a simple process to locate the center, rotation and scale of the digital image.

The distortion data given in table 6.1 has its largest distortion around 140 mm from PPBS at 2.7µm. Since the transformation parameters given above is around $67^{\text{pixels}}/_{\text{mm}}$, the distortions for this particular digital image is at the subpixel level.

6.3 Summary

The theory of the camera model was introduced. The model of the camera was split up in two parts: the exterior and interior orientations. The two combined makes it possible to trace a ray from the object space to the camera, through the lens and onto the image plane with great accuracy, taking especially the distortion of the lens into account.

Chapter 7 Raytracing the surface model

When generating an orthophoto, the output pixels are traced from the world coordinates back through the camera lens and onto the image plane along two rays. The first ray finds the Z coordinate above the output pixel, and the second ray traces back from this point to the camera. When creating a true orthophoto, the second ray should also check whether the point is visible from the camera or not. These two steps are illustrated at figure 7.1.

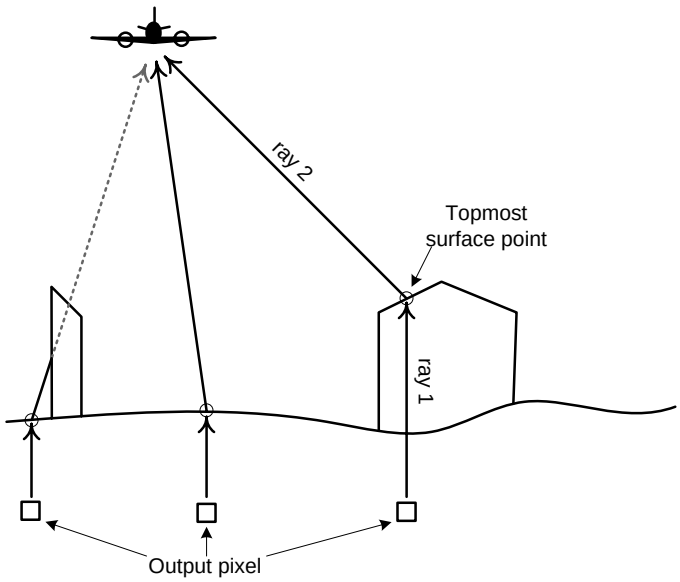


Figure 7.1 - When raytracing from a coordinate back to the image, the first ray finds the topmost surface point above a given coordinate. The second checks for visibility between the point and the camera. The leftmost ray is obscured by the tower.

A 15cm resolution true orthophoto of 1 km^2 would contain roughly 67 million pixels and twice the number of ray traces, so an efficient way of performing the ray tracing is needed. Some orthophoto applications speed up this process by doing a ray trace for every 2-3 pixels only and then interpolating between them. This can result in jagged lines along the roof edges where there are rapid changes of height in the surface model. The method is sufficient with DTMs where the surface is smoother and it increases the speed significantly.

The raytracing process has been implemented with two very different methods. -One using a database approach, and the other using a binary tree data structure.

7.1 Raytracing using a database

The first raytracer implementation was done using a Microsoft SQL Server database. A surface model may contain millions of triangles, so testing all the triangles for intersection one by one is very inefficient. Most databases has the option of indexing simple datatypes like numbers and strings, which makes searching very fast. Since all the triangles are stored as a set of three coordinates, each consisting of three numbers X, Y and Z, it is possible to limit the intersection tests to those triangles that are within a given extent. The use of indexes on these coordinates makes the process much faster and can eliminate most of the candidate triangles, before the much more processor demanding intersection test is done.

It is straightforward to do intersection tests with a vertical ray. Since the ray is parallel to the Z axis the problem can be seen as a 2D problem between a point and a flat triangle. Therefore checking only triangles whose vertices are within a given distance from the point in the X and Y directions is simple. If an intersection is found, the Z coordinate can afterwards be calculated. This approach isn't possible with rays that are not parallel to one of the axes. Therefore limiting the search extent by the triangles' vertices isn't possible, since the extent of possible intersectionpoints can be very large.

In order to do the visibility testing efficiently, the complex 3D triangle intersection test would have to be simplified into the simpler 2D triangle test. This is done by transforming the triangles to image coordinates. The transformation is a simple matter of updating the database once for each image. The Z coordinates are given a value equivalent to the euclidian distance to the camera. When this transformation has been done, the same 2D triangle intersection test can be repeated, and the actual X,Y,Z intersection can be calculated afterwards. The simplification process is illustrated in figure 7.2 and figure 7.3.

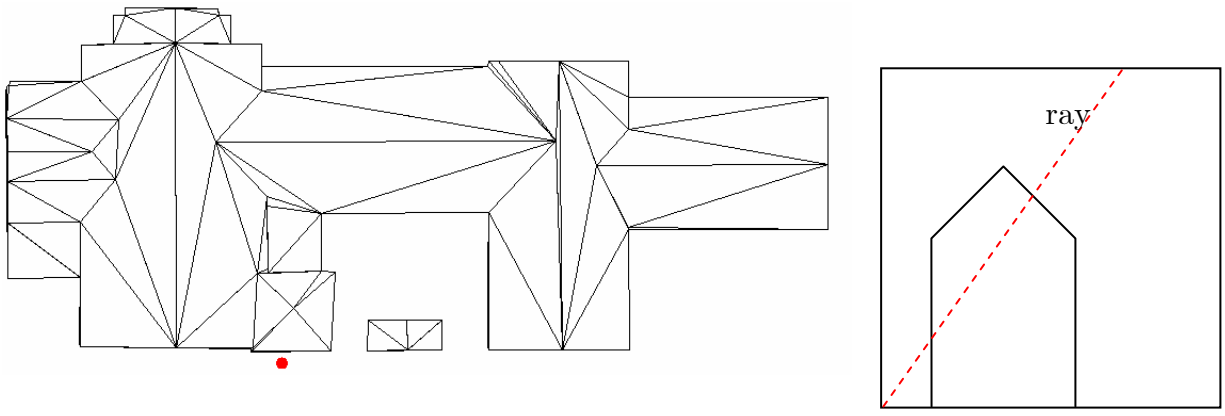


Figure 7.2 - Original surface model of building seen from above. The red dot is actually hidden from the camera as illustrated to the right and on the next illustration.

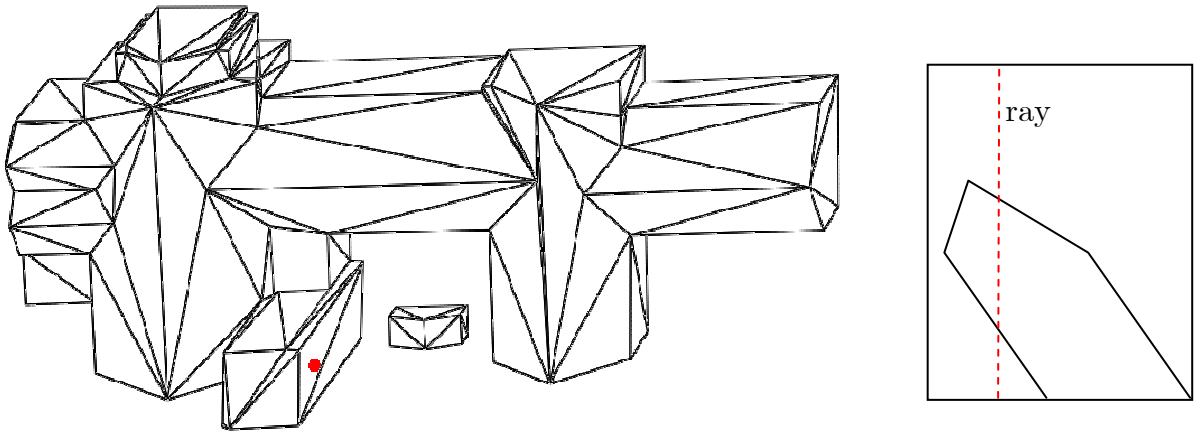


Figure 7.3 - Transformed surface model of the same building seen from above. This view corresponds to a perspective view, and is equivalent to how a camera "sees" the model. The red dot is the same as on the previous illustration, but here it is obvious that it is obscured by the building.

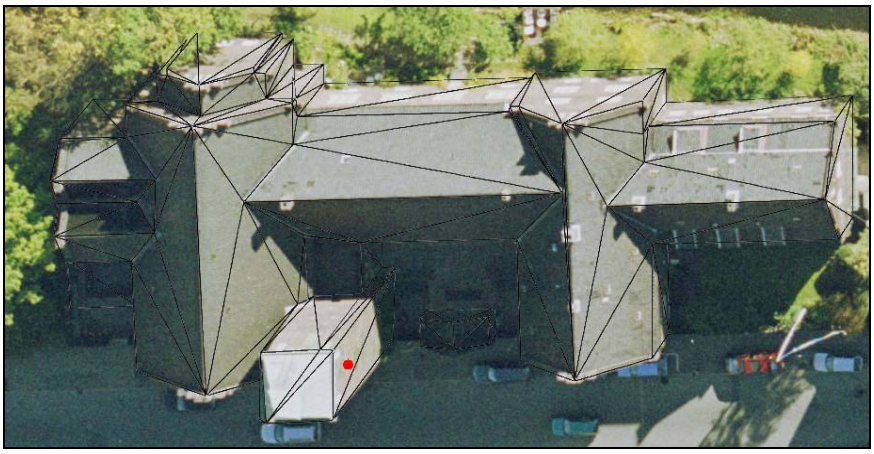


Figure 7.4 - Image of the same building as illustrated on the two previous figures. The transformed surface model has been superimposed.

The method has some limitations, since it only tests triangles where at least one vertex is within a given extent (a boundary box). The search extent has to be large enough to ensure that all the correct triangles are tested, but not so large that too many triangles has to be tested for intersections, causing a drop in performance (figure 7.5). The method is very easy to implement, and seemed very promising, but when put to the test it didn't perform very well. Even though the data searched are limited by a boundary box, a lot of triangles were still left for a time consuming intersection test. Each ray trace required around 1 second of processing time. The 1 km^2 orthophoto mentioned above would then require more than 150 days of processing, making this approach highly infeasible.

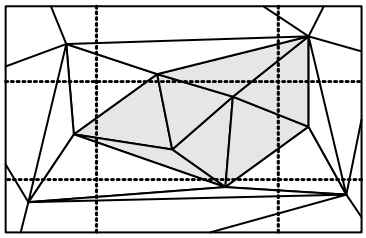


Figure 7.5 - Limiting the search extent in X and Y direction (limits illustrated as dashed lines) will not ensure that all triangles within the extents are checked for intersection. The gray triangles are the ones that have one or more vertices inside the extent.

7.2 Binary Trees

Instead of using a database, techniques used in 3D computer games were tested. Computer games are often highly optimized to provide the player with a vast amount of graphics and gameplay. Examples of ray tracing in computer games are collision detection used to prevent the player from walking through the walls or to calculate where a bullet would hit when fired. Computer Graphics Renderers used for creating photo realistic visualisations are also an area where raytracing are heavily used.

A common way of optimizing a raytracer in games is by using a binary tree structure which can be searched very quickly. The idea is to pre-compute the binary tree, before the ray tracing is needed. This only has to be done once for a 3D model.

A tree structure consists of a root node, several nodes below this, and at the bottom of the tree, the leaves. Every node can then have several sub nodes. A common tree structure is the binary tree that has at most two sub nodes at every node. In a tree designed to store geometric objects, each node and sub node stores a bounding box large enough to hold all the objects in the sub nodes. When the tree is being used for ray tracing, the bounding box is tested for intersection with the ray. If the ray intersects the bounding box, then the sub nodes are recursively tested for intersection. Testing a bounding box for intersection is very fast compared to testing a complex 3D object for intersection. The 3D object intersection calculations are done only at the leaves that are reached by the search. Some variations of the tree-structures not only store objects at the leaves, but also stores 3D objects in the nodes. Popular binary trees are the Binary Space Partion Tree (BSP tree) and the Axis Aligned Bounding Box tree (AABB tree). Only the AABB tree will be covered here.

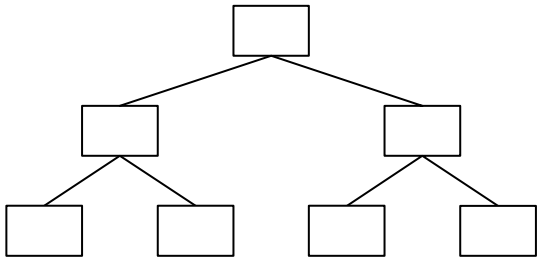


Figure 7.6 - A schematic view of a binary tree with one root node, four leaves and seven nodes in total.

7.3 Axis Aligned Bounding Box Tree

The Axis Aligned Bounding Box Tree is easy to implement, and the tree structure can be build from bottom up, which has some implementation benefits when initializing the size and structure of the tree. It is regarded to be almost as efficient as the BSP tree. For these reasons the AABB tree was implemented in this project [5].

The AABB tree contains all the triangles at the leaves of the tree, along with the minimum AABB. A minimum AABB is the minimum size bounding box which can contain an object, and whose sides are aligned to the three axes. By aligning the bounding box to the axes, only two points need to be stored in memory to represent the bounding box. Each leaf are then paired with another leaf, and their common AABB are then stored at a super node. The super nodes are then again paired in another super node, until the root of the tree is reached. Each node will then hold two points to represent the AABB and a left and right pointer that points to the two sub nodes, or if it is a leaf, a triangle.

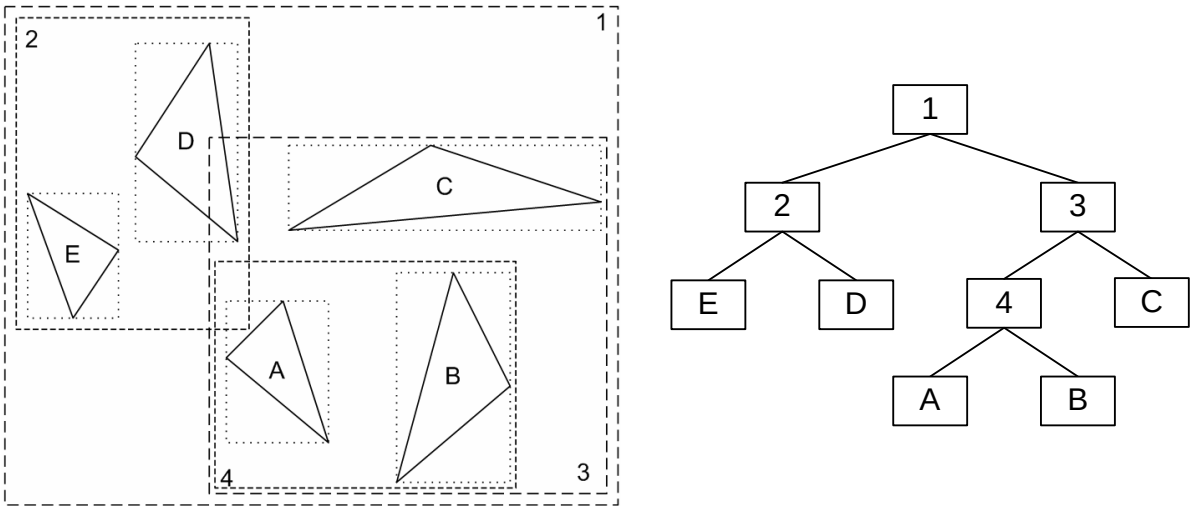


Figure 7.7 - A 2D case of an AABB tree. Five triangles have been joined in bounding boxes and the resulting tree is illustrated to the right.

7.3.1 Creating the tree

The AABB tree is very similar to an R-tree structure used to index 2D GIS data. The method described here are inspired by the R tree structure [10], but adapted to a bottom-up creation process.

To start with, the tree is initialized to a size that will fit the complete binary tree structure. Since the bottom nodes in the tree (the leaves) will contain all the triangles, the width of the tree will be n, where n is the number of triangles. Combining the nodes two and two, will require a tree with a total of $2n$-1 nodes. All the leaves are then filled with an AABB enclosing each triangle (figure 7.8).

Initialize tree:

```
Create a list A of length (2n-1)
For each triangle ‘i’:
   Calculate AABB_i for triangle i
   Insert AABB_i in list A at position (i+n-1), and mark it as non-paired
```

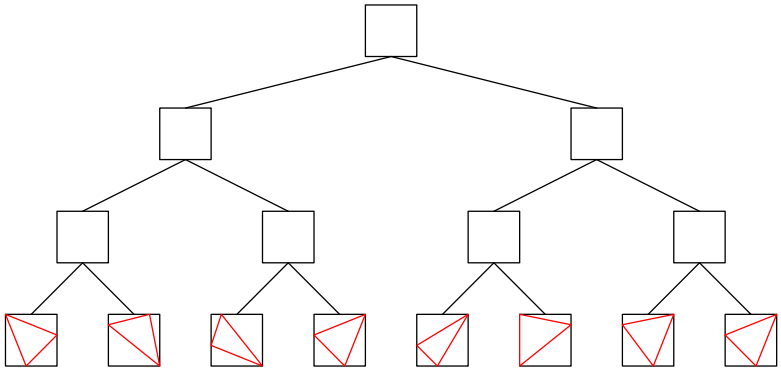


Figure 7.8 - The initialized tree. All triangles are enclosed in bounding boxes at the leaves and all the non-leaves nodes are still empty.

The algorithm that creates the rest of the tree is crucial for doing efficient searches. Since intersection tests are processor-consuming, we want to be able to search as few nodes as possible to find all the intersections along a ray. Therefore, combining the nodes must be done in such a way that the parent bounding boxes are as small as possible, thereby reducing the chance that a ray will intersect with it. There are several ways to do this, but two of the simplest are:

1. Combine the two bounding boxes whose centres are as close to each other as possible.

2. Combine the two bounding boxes that will result in the least increase in the combined bounding box surface area.

Calculating the centres and the distance between the two centres in method 1 turned out to be very processor demanding, and made the process of creating the tree three times slower than with method 2. This is only due to the fact that the distance calculatings are more complex than the area calculations. When searching for intersections at the leaves, method 2 usually had 1.3 – 1.5 more intersections in the tree than method 1, making a tree created with method 1 the most efficient, but also the slowest to create. Why this method generated the most efficient tree, can have something to do with the triangles in a DSM. Many triangles are vertical (walls) or horizontal (ground) and therefore the bounding boxes tend to be very flat, possibly making method 2 less appropriate to use.

The algorithm described below will process each level in turn, starting with the leaf-level. When all nodes have been paired at a given level, all their super-nodes will be paired. A special case is when there are an odd number of nodes in a level; the one non-paired node remaining at a level will be promoted to a higher level and paired with one of the objects in the next level.

Fill tree:

```
Set k = n
Set d = maximum depth of tree.
While k > 0:
   AABBj = Next_Non-paired_AABB.
   Select nearest non-paired AABBi located at depth d.
   If AABBj found:
      Calculate AABB of AABBj+AABBi and insert in list A at position k.
      Set k = k-1
      Mark AABBj and AABBi as paired.
   Else
      If the width of the current depth d is an odd number:
         Regard AABBj as being at depth d-1.
      Set d = d-1
```

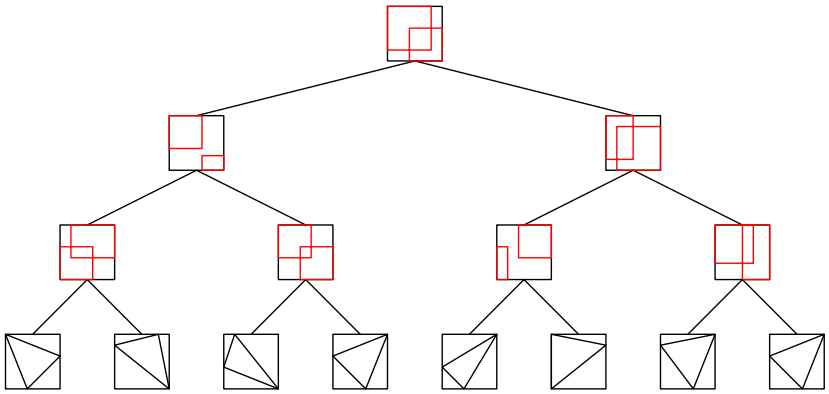


Figure 7.9 - The filled tree. Each node contains bounding boxes that encloses the bounding boxes in its subnodes.

After the process has completed, the root node will contain an AABB that describes the complete extents of the model, and each node will describe the extents of all the sub nodes.

7.3.2 Intersecting the tree

Finding intersections between a ray and the triangles is a matter of traversing the tree, and process all sub nodes where a node's AABB are intersecting the ray. When a leaf is reached, it is checked whether the object at the leaf actually intersects the ray. Because we're using a binary tree structure, the processing time required finding the intersections are dependent of the depth of the tree. This means that the intersections can be performed with a complexity[4] of only $O(\log n)$, compared to $O(n)$ if we where to check all objects for intersection. This makes it very robust to large terrain models, since the depth is only increased one level each time the number of

[4] The Big-O notation is used to describe the asymptotic behaviour of an algorithm. It specifies an upper bound complexity of the algorithm compared in terms of another algorithm: Ref: Wikipedia.

objects are doubled. The recursive method for finding all the intersections along a ray is described below:

IntersectTree (node, Ray):

```
if (AABB_node is a leaf):
          if Ray Intersects AABB_node->Triangle:
                    Add PointOfIntersection to IntersectionList;
else if Ray intersects AABB_node :
          IntersectTree(AABB_node->LeftNode, Ray)
          IntersectTree(AABB_node->RightNode, Ray)
```

Finding the intersections between a ray and an AABB is very fast. The slabs method is a simple ray/AABB intersection algorithm. A slab is two parallel planes described by the two sides that are opposite each other in the AABB. In the 3D object space, a bounding box will have three slabs. This is illustrated in the 2D case at figure 7.10.

By calculating the distance from the ray origin to the two intersections $t_i^{\min}$ and $t_i^{\max}$ the test can quickly reject if there are any AABB intersections. The trick is to calculate:

$$t^{\min} = \max(t_x^{\min}, t_y^{\min}, t_z^{\min})$$
$$t^{\max} = \min(t_x^{\max}, t_y^{\max}, t_z^{\max})$$

Testing whether $t_{\min} \leq t_{\max}$ is true determines if the ray intersects the AABB. This is also illustrated at figure 7.10. Since this test would be done on two axes in turn, rejection of intersection can often be done early in the loop causing the test to finish earlier [5].

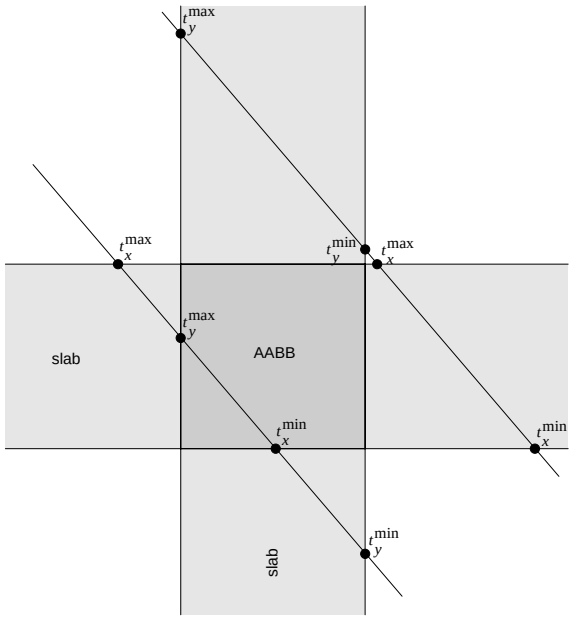


Figure 7.10 - Intersecting ray and slab planes. When the intersections of the two slabplanes are compared, t_{min} and t_{max} can be used to determine if the ray intersects the bounding box.

When a leaf in the AABB tree has been found, the next step is to test if the triangle stored at the leaf is intersected by the ray. This is a more processing-intensive calculation than the ray/AABB intersection test, but is fortunately only done at the relatively few leaves that are reached. Therefore the optimization of this algorithm isn't that dependent of fast calculation. Standard trigonometry can be used for finding the intersection point on the triangle. The actual implementation the that were used for the raytracer is based

on a simple intersection algorithm described by [16], and will not be covered here.

7.3.3 Performance evaluation

The tree was tested with a DSM containing the same 270.000 triangles that were used for testing the database approach. Each ray trace usually required between 60 and 80 ray-AABB intersections and at most ten ray/triangle intersections. Using the same reference computer as was used for testing the database approach, a raytrace used approximately 5ms of processing time. This method proved superior to the database approach, with a performance gain of around 20.000 times! As mentioned earlier the tree structure ensures that even though the number is doubled or even tripled, the processing time will not increase significantly. Worst case processing time is $O(\log\ n)$. An estimated time for a DSM twice the size can be found by:

$$\frac{\log(270{,}000)}{\log(540{,}000)} = \frac{12.506}{13.199} = \frac{5ms}{t} \Rightarrow t = 5.3ms$$

Other properties have to be considered though. Large memory consumptions causing disk swapping can significantly increase processing time.

The AABB tree has been implemented as a separate raytracer library using C# and is included on the companion CD-ROM. It can be useful for other projects that need to raytrace a TIN. An implementation description can be found in Appendix D.

7.4 Summary

Two methods of intersecting a surface model were introduced. The first one, using a database was quickly abandoned due to its poor performance. Using an advanced tree structure, the axis aligned bounding box tree, performance of the raytracing was increased with a factor of 20.000 compared to the database approach.

Chapter 8 Color matching

The images that the orthophotos are based on takes three basic steps to create; the exposure, the develop process and the digital scanning. All three steps can have influence on the quality of the final image, and can make the images look very different in color. With the advent of digital cameras, the two last steps can be disregarded. At the time of this project, only a few digital cameras useful for mapping have been sold, but it would be reasonable to think that in the near future these cameras will replace analog cameras.

Some of the imagery available in this project changes a lot between the flight lines and even between the images in the same flight line. Some looks yellowish and other very bluish (figure 8.1). A uniform color shade throughout the image like that in figure 8.1 is usually due to errors in the scanning. Color differences can also occur if the images are from two different rolls of film. If these images were to be mosaicked without any color-correction, the seamlines would be highly visible and the general overall orthophoto result would be very poor.

Local radiometric differences also occur in the image. Since the images rely on light from the sun, the relative angle to the sun may also have great influence as illustrated on figure 8.2. The different surfaces also reflect the light differently. A *lambertian* surface has special characteristics. The brightness radiance of a lambertian surface is invariant regardless of the angle from which it is viewed. Surfaces like soil are close to be a lambertian surface, whereas water is at the opposite end. It would be wrong to assume the ground to be close to lambertian, and therefore the luminance is often brighter in the southern part of the image if the sun is to the south during the exposure. If the sun was at zenith, the central part of the image would be the brightest.

Figure 8.1 – Two neighboring images. The image on the left has a very bluish look compared to the image on the right. Since the original analog images doesn't have this color difference, it must originate from the scanning.

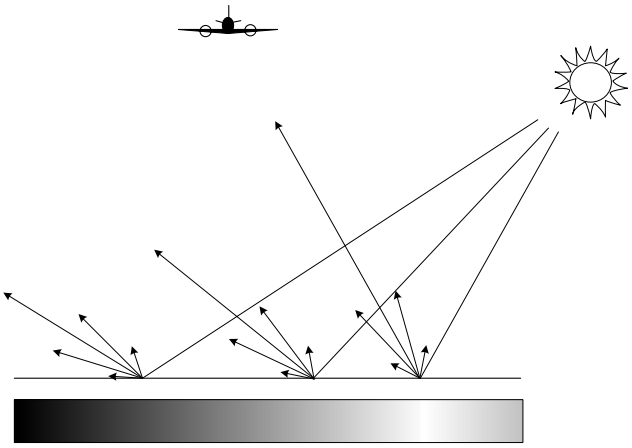


Figure 8.2 - The overall brightness of the image rely on the reflection of the surfaces. "Bumps" in the terrain and different reflection characteristics makes this pattern more complex. The gray-scale below shows the amount of reflected light received at the camera from the locations.

The sun only directly lights up one side of the buildings, and therefore these will look brighter. In a vertical photograph, half of the image (opposite the sun) will contain the lit walls and the other half the walls hidden in shadow. This will make the image look like there is a big difference in brightness in the two halves. Fortunately the walls aren't visible in a true orthophoto, so this effect should only be evident in ordinary orthophotos.

The image can also tend to look brighter in the centre of the image where the distance to the ground and objects is the shortest. Especially the blue colors are

affected. The worst part of this radiometric distortion is corrected with a filter on the lens that is darker in the centre of the image and thus reduces this effect.

Normal orthophoto imagery tries to match the colors of the neighboring images in the overlapping regions, and the seamlines are few and simple enough to manually place them where the radiometric differences are smaller. To hide the remaining small differences, feathering can be used along the seamlines to blur the edges. Feathering is covered in chapter 9.

For true ortho imagery the images will be replacing each other everywhere in the image, requiring the images to be much more uniform in color throughout the entire image. Furthermore automatic seamline-generation is required because of the large number of parts the mosaic consists off.

8.1 Color spaces

There are several ways of describing the color of a pixel. Usually colors are described in a three dimensional color space, and can be divided into three general color space coordinates [3]:

- Tristimulus coordinates
- Perceptual color spaces
- Chromaticity coordinates

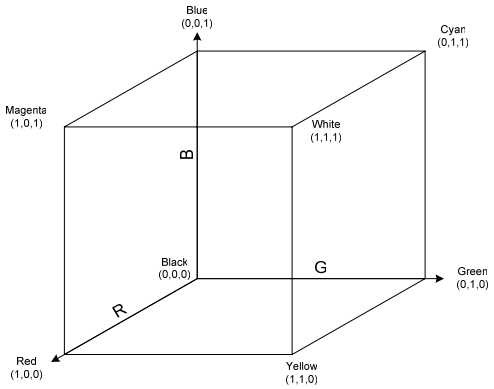


Figure 8.3 - The RGB color cube.

The tristimulus coordinates are a rectangular space and used for the most common color space: the Red/Green/Blue (RGB) color space, named after the three primary colors. A color coordinate describes the mixture of the three colors red, green and blue. For instance (1 , 0 , 0) is pure red, and (½ , ½ , ½) is gray. It is widely used in monitors, computers, TVs, digital cameras and most digital image formats. For digital images, the three primaries are usually divided into 256 levels each, which give around 16.7 million combinations within the color space.

The perceptual color space is based on variables like hue, purity, brilliance, brightness and saturation [3]. The most common is the Intensity/Hue/Saturation (IHS) color space. It is based on two vectors and a rotation, and has the form of two cones as illustrated on figure 8.4. It is useful for adjusting the brightness by increasing the intensity or adding more color by increasing the saturation.

If compared to the RGB color cube at figure 8.3, the intensity axis corresponds to the line between (0,0,0) and (1,1,1) in the RGB cube. The direct relation between the RGB and IHS color spaces is described below [1]:

$$\begin{bmatrix} I \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ -\frac{1}{2} & -\frac{1}{2} & 1 \\ \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} & 0 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}, \quad H = \tan^{-1}\left(\frac{v_2}{v_1}\right), \quad S = \sqrt{v_1^2 + v_2^2}$$

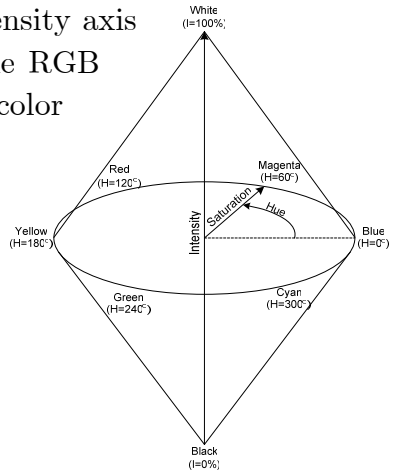


Figure 8.4 - The IHS color space.

The inverse color transformation is:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & -\frac{1}{3} & \frac{1}{\sqrt{3}} \\ 1 & -\frac{1}{3} & -\frac{1}{\sqrt{3}} \\ 1 & \frac{2}{3} & 0 \end{bmatrix} \begin{bmatrix} I \\ v_1 \\ v_2 \end{bmatrix} \quad \text{where:} \quad \begin{aligned} v_1 &= S\cos H \\ v_2 &= S\sin H \end{aligned}$$

The chromaticity coordinate system consists of a luminance and two chromatic coordinates. The color space covers a much larger space than the RGB and IHS color space, and can therefore describe many more colors, though most computer monitors can't display all of them. One of them is the standardized CIE[5] L*a*b color space has which three coordinates: Luminance (L), redness/greenness (a) and blueness/yellowness (b). It is rarely used for digital analysis since the coordinate system is harder to comprehend. It is mostly used for offsetting where special colors are needed. For instance the Danish bills contain colors outside the RGB space, which makes them much harder to copy using a standard scanner and PC.

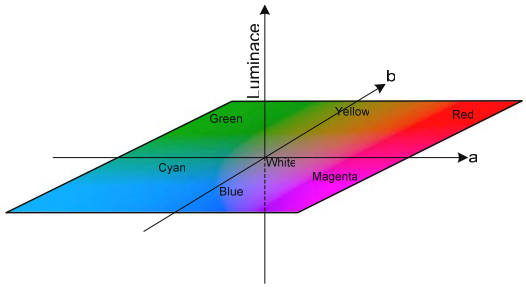


Figure 8.5 - The CIE L*a*b color space.

8.2 Histogram analysis

Since the three types of color spaces all have different principal components, they can be useful for analyzing and correcting different properties of an image. Histogram analysis is a simple method for analyzing the colors of an image by counting the principal components in each pixel and interpret them in a histogram.

[5] CIE : *Commission Internationale de l'Eclairage* (*International Commission on Illumination*)

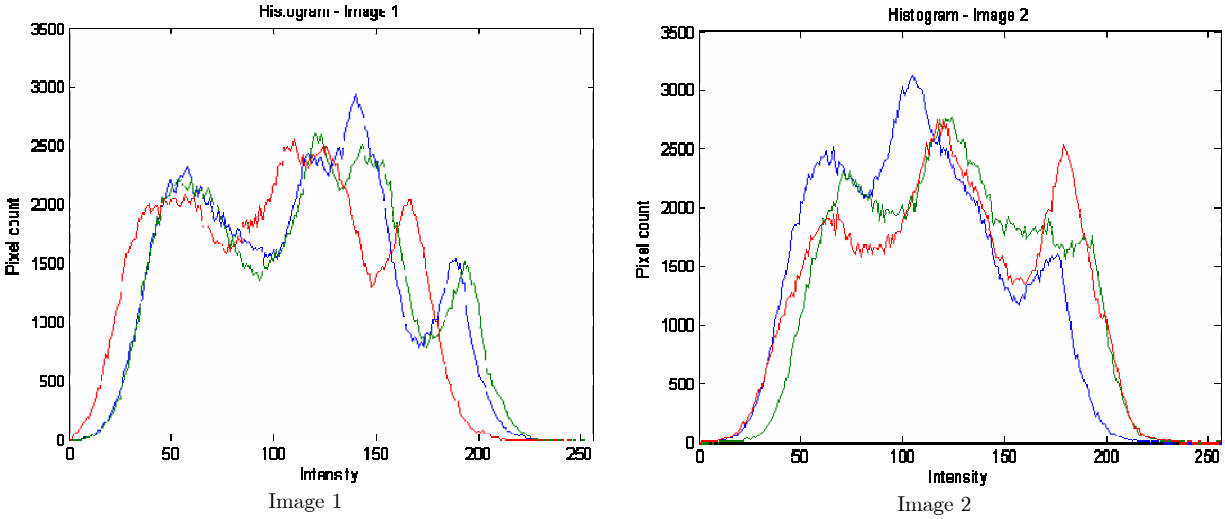


Figure 8.6 – Histogram of ortho images. Each curve corresponds to one of the red, green and blue primary colors. Any pixel obscured in any of the two images is excluded from both histograms.

Figure 8.6 shows a RGB histogram of an orthophoto made from the two source images shown on figure 8.1. Since they both are orthophotos of the same area, any pixel i,j in both images depict the same object. Nevertheless the histograms are very different. The histograms also shows that the full color spectrum isn't utilized, since there doesn't exist any pixel with an intensity above ≈ 230. By stretching / scaling the intensities, the full dynamics of the RGB color space can be used for increased contrast. This method is called a *histogram stretch*.

Image 2 has a high intensity peak and a green "plateau" which is probably why the image looks warmer. The blue peak around intensity 140 in image 1 is probably what makes it look bluer.

The RGB histogram does have some limitations, since is lack the spatial relations. For instance it doesn't tell you how many blue pixels there are, only how many pixels that have a certain amount of blue in it. If most of the pixels with a high blue intensity also have similar high red and green intensities, they are actually different shades of gray. By analyzing the images in another color space that has the hue as one of the principal components, this relation can be drawn out.

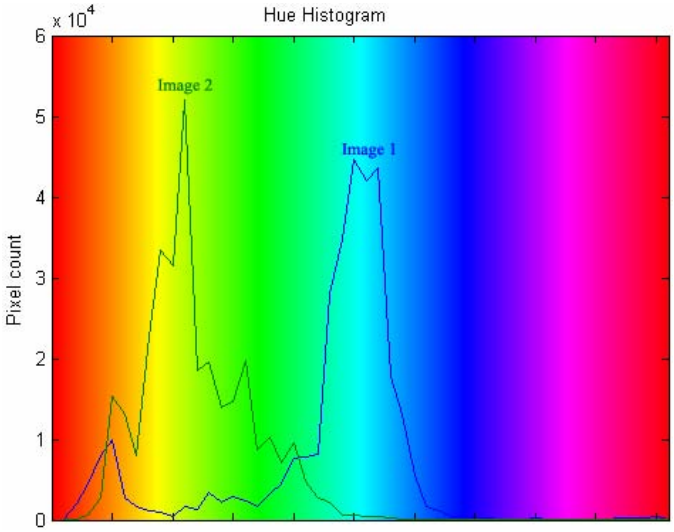


Figure 8.7 - The hue histogram of the two different images. Image 1's bluish shade is cleary visible in this histogram.

Figure 8.7 shows the hue of the same two images from figure 8.6. Here it is clearly visible that image 1 contains more pixels with blue shades than image 2 that obviously has a more greenish look to it. The hue histogram only gives a very narrow view on the images, since it doesn't say anything about the saturation or intensity, but it does display a tendency.

8.3 Histogram matching

The previous section pointed out the differences in two images on the basis of the histogram. It would be natural to try to make the two histograms match each other in order to make them have similar radiometric properties. Histogram matching is such a method.

Histogram matching tries to take an input histogram $h_0(v_{in})$ and transform it, so that it will resemble a reference histogram $h_1(v_{out})$, where v is a given value in the histograms.

The relation between the two histograms can be written as:

$$c_1(v_{out}) = c_0(v_{in}) \text{ , where } c \text{ is the cumulated histogram: } c_i(v_j) = \sum_0^{v_j} h_i(v_j)$$

The histogram matching equation can then be written as [3]:

$$v_{out} = c_1^{-1}(c_0(v_{in}))$$

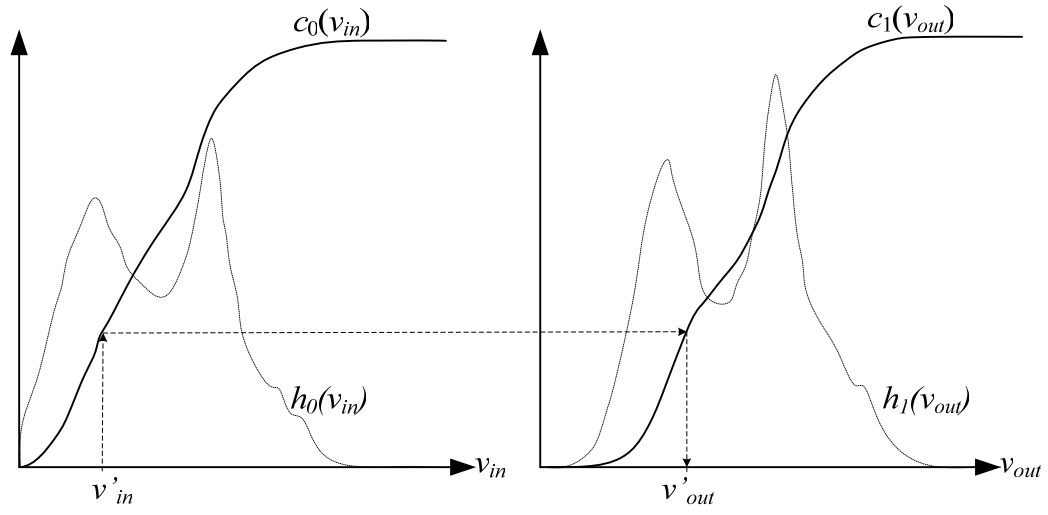


Figure 8.8 – The histogram matching method. The left histogram $h_0(v_{in})$ is to be mapped to $h_1(v_{out})$. Note that the scale of the histogram and cummulated histogram is different.

The reference histogram $h_1(v_{out})$ can have any appropriate shape. A gauss-shaped histogram will increase the contrast in shadows and highlights, while a constant level

will increase the total dynamic range. The latter is also referred to as a *Histogram Equalization*.

8.4 Histogram matching test on orthophotos

In this specific case we want to match images to each other, so using one image as reference histogram and match other images to it, is one approach. In this case two orthophotos based on the aerial photos from figure 8.1 are used. The two ortho photos are shown at figure 8.9. Both images could be matched to a common mean histogram, but since visual inspection shows that the colors in image 2 looks fine, it would be appropriate to match image 1 to image 2.

For each intensity value from 0 to 255 in each R, G and B color band, a Lookup table (LUT) is constructed that maps every input intensity to the output intensity by using the histogram matching equation. Afterwards the LUT is used to assign new intensity values to the input image. A MATLAB histogram match implementation can be found in Appendix B.

Input	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…	255
Output	0	2	2	4	5	5	6	7	7	7	8	9	11	14	14	15	18	19	20	…	240

Table 8.1 - Example of a LUT (Lookup Table)

Applying the histogram match to the images shown in figure 8.9 gives a very good result. The color and intensity matches very good as illustrated on figure 8.11. The cumulated histogram matches very closely, and the differences are due to the intensities doesn't have floating precision but is integer. An example of this can be seen on figure 8.12, where the cumulated histogram for blue input, output and reference is illustrated. The histogram matching could also have been applied to other color spaces, for instance IHS. This doesn't seem to be necessary, when looking at the IHS histograms for the RGB-histogram matched image. The fit is quite good as shown on figure 8.13, even though the fit wasn't done in the IHS color space. Note that the scale on the histogram is much smaller than on the *cumulated* histogram, and therefore the deviations looks worse.

To test how the histogram matching would perform in an image mosaic, this method was applied to all images that were to be incorporated in the mosaic, and the with- and without-match results are compared. Figure 8.14 shows the true orthophoto based on the unmatched images. The different colors are very obvious, and even if feathering were applied to smear the seamlines, it wouldn't make the image look much better. Figure 8.15 shows the same mosaic, but based on color matched images. The seamlines are almost invisible after applying feathering and are generally only detectable if you know where to look for the seamlines.

The histogram matching was also tested on some images with very different properties. The reference image on figure 8.16 was taken in October 2003 where leaves on the trees were still present and with a normal angle lens from 1500m. The colors and contrast in the image are very high. As input, the same image 1 from figure 8.9 was used. The input image was taken in May same year before the trees had leaves using a wide angle lens from a lower altitude. The shadows are also shorter in this image, and the higher sun angle slightly decreases the contrast difference between shadows and sunlit areas.

The result of this test can be seen on figure 8.17 and compared to the reference image on figure 8.16. The biggest differences between the two images are the vegetation in the reference image which makes the colors looks very vibrant. Looking at more consistent objects like rooftops and ground, the colors match fairly well. Especially in the reference image some rooftops that are positioned oblige to the sunrays seem very bright. This means that even if the images are taken under very different circumstances, the same objects can come to look more or less the same, but the fairly large differences in vegetation and shadows will probably cause problems along seamlines anyway

8.5 Summary

This chapter introduced three basic methods for describing colors in a computer system, and how to analyze the radiometric properties of an image using a histogram. Sources of radiometric errors in aerial photographs were described and images was analyzed and compared by their histograms. A concept of histogram matching was introduced and the method was tested on imagery with different radiometric properties. The method proved very successful for matching the colors of one image to another.

Image 1 (input) Image 2 (reference)

Figure 8.9 –Two orthophotos of the same area, based on different images.

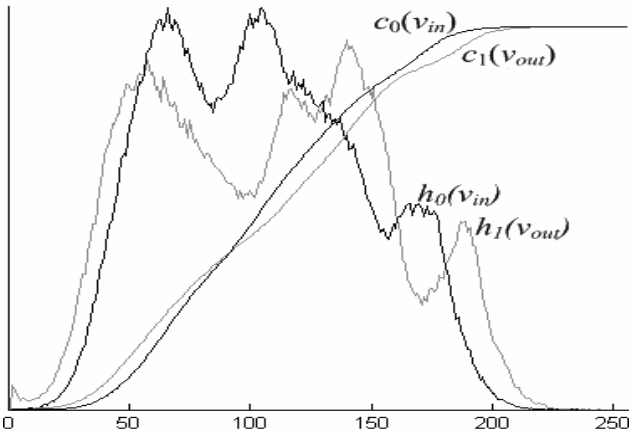


Figure 8.10 – Input and reference histogram for the blue band.

Figure 8.11 - Image 1 after histogram-match

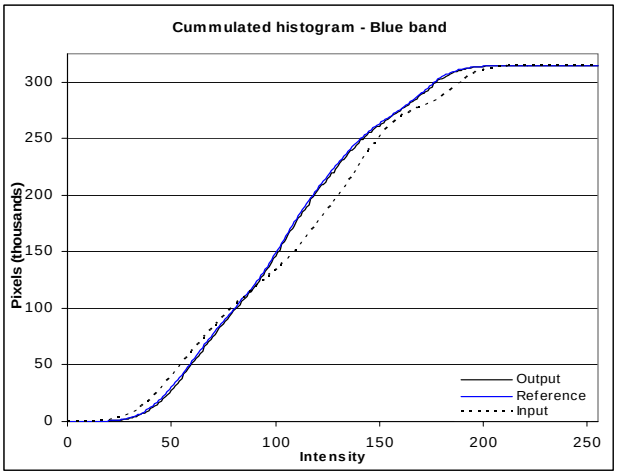


Figure 8.12 – Cumulated histogram of the input, output and reference images. The output curve is 'choppy' because of the integer mapping.

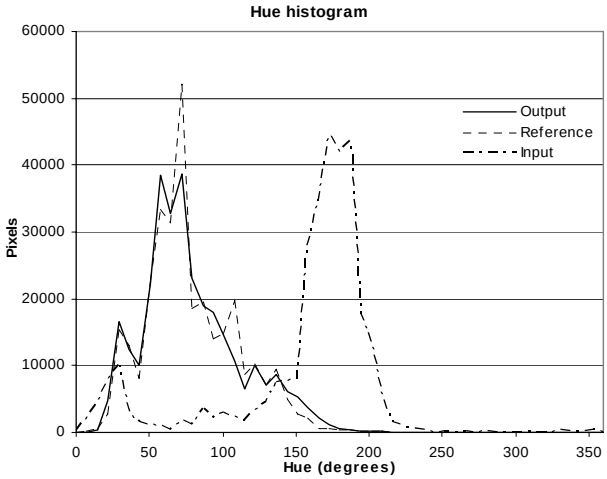


Figure 8.13 – Even if the histogram matching was done on the RGB color space, the hue histogram is affected and gives a very good fit.

Figure 8.14 –True orthophoto mosaic. The original imagery are very different in color and intensity which causes a poor mosaic with very visible seamlines. Because of the big differences in color, feathering the seamlines isn't enough to hide the differences.

Figure 8.15 - Mosaic of histogram matched orthophotos. With a little feathering to smooth the edges, the seamlines are almost invicible.

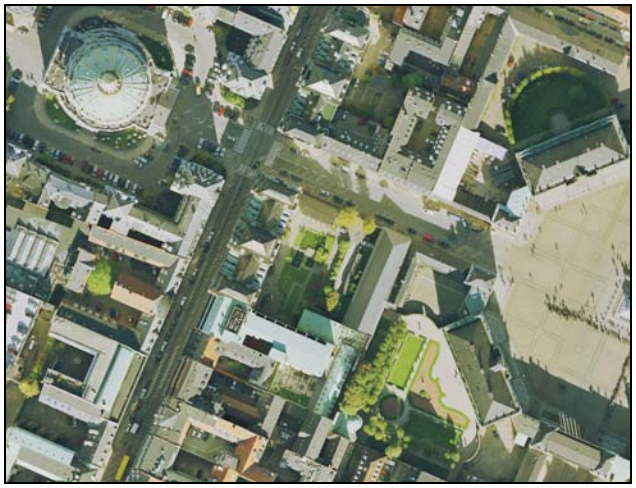


Figure 8.16 – Reference image taken at fall with normal angle lens from 1500m altitude.

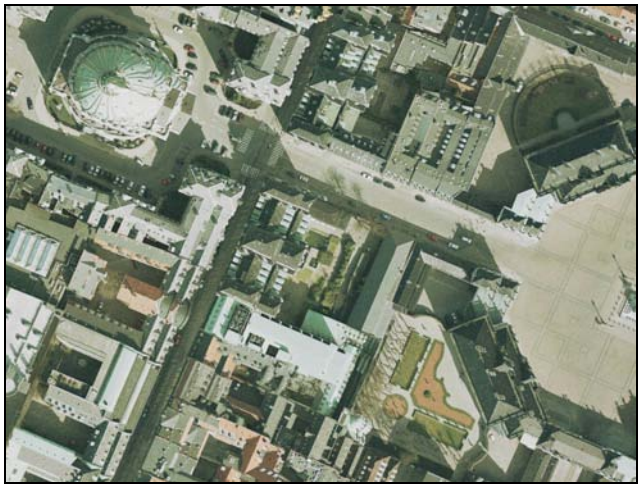


Figure 8.17 – Image 1 matched to the reference image. Image is taken in spring with a wide angle lens from a 750m altitude. The biggest difference to the reference image is the vegetation and less contrast between sunlit and shadowed areas.

Chapter 9 Mosaicking

When generating large scale ortho imagery, multiple images have to be merged to form a mosaic of images. Neighboring images are usually assembled along seamlines that run roughly along the centre of the overlapping areas. The exact path of the seamlines can be generated both automatically and manually. Placing the seamlines along the roads often gives the best results, compared to cutting over buildings that have different relief displacements. This is often only visible in the final result, where buildings will have relief displacements in different directions.

With true orthophotos, one advantage is that the relief displacements are non-existing for all the objects that are modeled by the DSM, so forcing the seamlines to run along the roads isn't necessary. Usually the most visible relief displacements left will be those originating from objects like vegetation that isn't included in the DSM. Even though, the problem is more complex, because the images doesn't only have to be mosaicked with neighboring images, but also any place where the obscured pixels in one image has to be replaced by pixels from other images.

At figure 9.1 an analysis of the visibility from four normal-angle images illustrates that everywhere along the buildings; one or more images have obscured data at the ground. Making the final true orthophoto look as seamless as possible requires a lot of image processing due to all the seamlines needed.

9.1 Mosaicking methods

The mosaicking methods presented in this section all rely on a pixel-by-pixel score method, inspired by the methods presented in [8]. Each method is tested and evaluated using the same test area, which is a part of the central Copenhagen. It contains both narrow backyards and tall buildings which are expected to cause problems during the mosaicking.

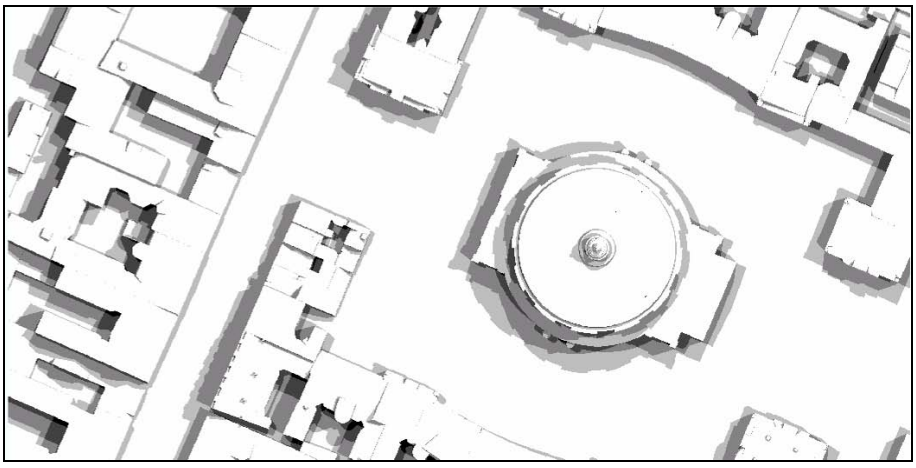


Figure 9.1 - Visibility analysis for four images. White is visible in all images, and black means obscured in all images. The three levels of gray specify that it is obscured in respectively 1, 2 or 3 images.

9.1.1 Mosaicking by nearest-to-nadir

The simplest way of mosaicking the images, are by assigning a pixels its color from the image whose nadir point is closest. If the pixel is obscured in that image, the second closest is selected and so forth. If the pixel is close to the nadir point, the relief displacements are smaller, and therefore also less likely to hit a blindspot. Using this approach for the same case shown at figure 9.1 is illustrated at figure 9.2.

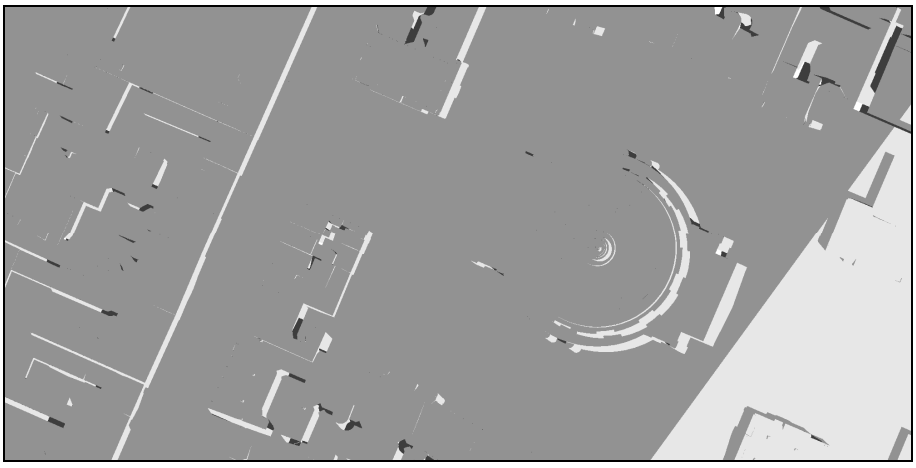


Figure 9.2 - Mosaic pattern generated by selecting nearest-to-nadir non-obscured pixel. Each gray-level corresponds to a different source image.

The nearest-to-nadir approach is fast and simple to implement. One of the biggest disadvantages to this approach is that inaccuracies in the surface model are very noticeable in the final mosaic. Since the surface model used in this project doesn't contain the eaves, this causes a major problem in the final result. During the rectification, they are projected onto the terrain and left there on the ground (cf. figure 9.3).

Figure 9.3 – The surface model doesn't contain the eaves, and therefore leaves them visible at the ground.

An example of implementation in MATLAB can be found in Appendix B: *Mosaic_ nadir.m.*

9.1.2 Mosaicking by distance to blindspots

To reduce the problem with an inaccurate surface model, another method should be used. The approach used here, is to use the pixels from a source image that are the farthest from data obscured in the image. This would mean that the seamlines would be placed between the blindspots in the images. Usually this would roughly be at the roof ridges and between the buildings.

The method with which to calculate the distance from a pixel to the nearest blindspot can be done by using Distance Transformations (DT) as described in [1]. It's a method that maps binary images into a gray-level image, where the distance to the nearest object corresponds to the gray-level. In this case, the visibility map is the binary image, where the objects in the image are the blindspots.

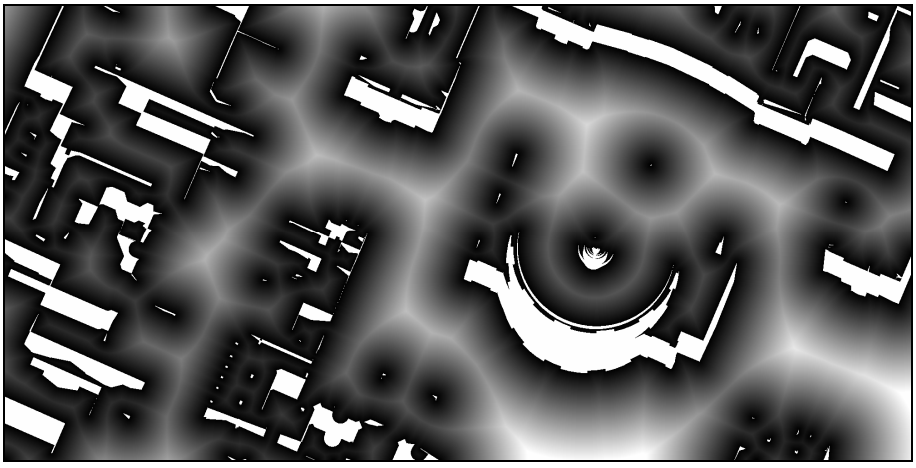


Figure 9.4 – Euclidean distance transformation of blindspot image with the blindspot image superimposed as white. The brighter gray the longer is the distance to a blindspot.

The Euclidian distance is very time consuming, so another simpler distance transformations can be used. The hexagonal distance doesn't calculate the exact distance to the nearest objects, but the approximation is very good, and much faster to calculate. The difference between the two transformations is illustrated at figure 9.5 and figure 9.6.

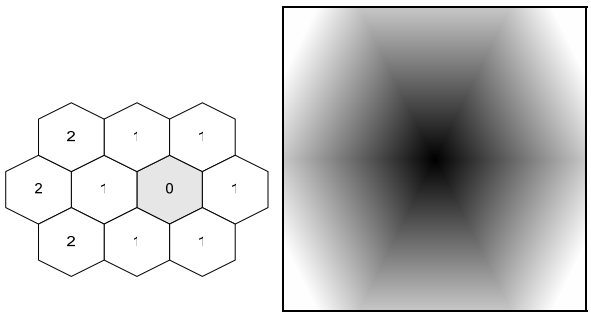


Figure 9.5 – Hexagonal distance transformation.

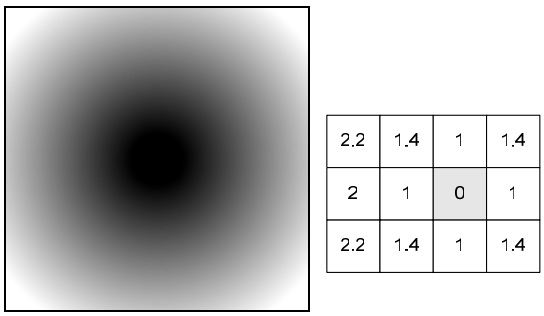


Figure 9.6 – Euclidian distance transformation.

Basically hexagonal transformations are implemented by shifting every other row a half pixel to the left or right and apply convolution of the masks shown at figure 9.7. Another approach is to shift the masks at every other row, as illustrated at figure 9.8.

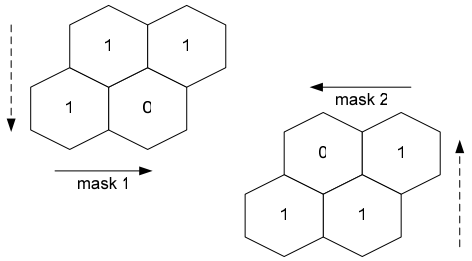


Figure 9.7 - Calculating the hexagonal distance transformation can be done by applying the masks above, after shifting every other row a half pixel [1].

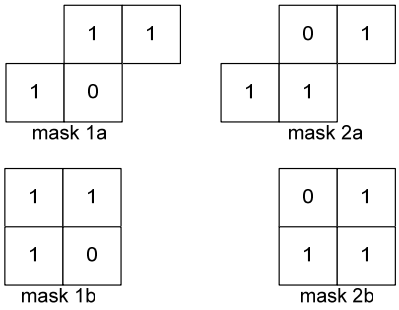


Figure 9.8 – Instead of shifting every other row a half pixel, the masks can be shifted instead. For instance apply all odd rows with mask 1a and 2a, and 1b and 2b on all even rows.

For each source image, a visibility map is created along with a corresponding distance map. The distance map can then be used to determine which source image data should be taken from. This is determined by a score function for each pixel, where the score is corresponding to the brightness (distance) of the pixel in the distance map. The principle is illustrated in figure 9.9. An example of a mosaic can be seen in figure 9.10.

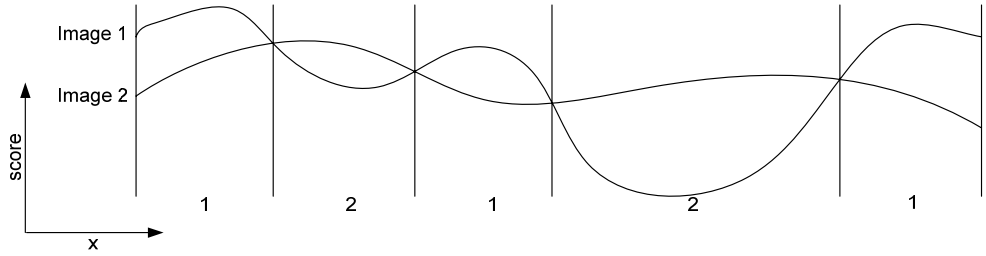


Figure 9.9 - Score functions for two images along a pixel-row. The score determines which image data should be used in the mosaicked image, as indicated by the numbers below the score.

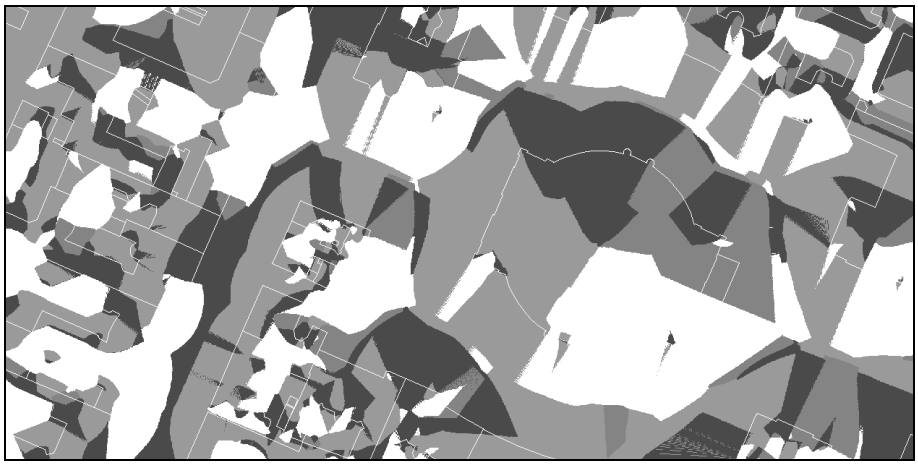


Figure 9.10 – Image mosaic created from four euclidian distance maps (building outlines are superimposed).

It turned out that the mosaics generated with the two types of DTs where almost identical. The main differences are that seamlines tends to be more "straight" with the hexagonal distance map.

An example of implementation in MATLAB can be found in Appendix B: *Mosaic_ blindspots.m*

The mosaics are also much more fragmented than the simple nearest-to-nadir mosaic at figure 9.2. An advantage is that the eaves are now completely gone from the ground, as illustrated on figure 9.11. Unfortunately seamlines are sometimes running through moving objects like cars, and half cars often occur in the image. This is a tradeoff, since having a lot of seamlines

Figure 9.11 – Close-up of true orthophoto generated with the mosaic at figure 9.10. Compare this with the nearest-to-nadir result illustrated at figure 9.3.

increases the chance of cutting through unrectified objects or objects that have moved between the shooting of images. This problem is treated in section 9.3.2.

9.1.3 Mosaicking by distance to nadir and distance to blindspots

The two previous methods described both have some nice properties. Close proximity of the nadir point makes the blindspots small and minimizes distortion of objects that are not modeled, such as cars and vegetation. Keeping distance to the blindspots decreases the chance of leaving the eaves on the ground, and also leaves room for a overlap necessary for feathering.

It would be natural to try to combine these two methods. By placing a buffer around the blindspots, the nearest image will be used everywhere where there is a certain distance to the blindspots. This basically works as thresholding the distance. Within the bufferzone the score is lowered the closer it is to the blindspot. This method should be able to deal with the missing-eaves problem illustrated at figure 9.3. Furthermore by increasing the distance to the blindspot, feathering can be applied in the overlapping regions (see section 9.2) without risking to feather into the obscured areas.

The implementation of this function can be done by multiplying the distance-to-nadir score (dN) with a distance to the blindspots (dB). The distance to blindspots should be between 0 and 1, where 0 is at the blindspot and 1 is at the preferred minimum distance to the blindspots. Between these two values, a fraction is linearly interpolated. The score function is illustrated on figure 9.12. Figure 9.13 is an example of the score for one of the source orthophotos. It is here illustrated how the score decreases when moving away from the principal point or when getting close to blindspots.

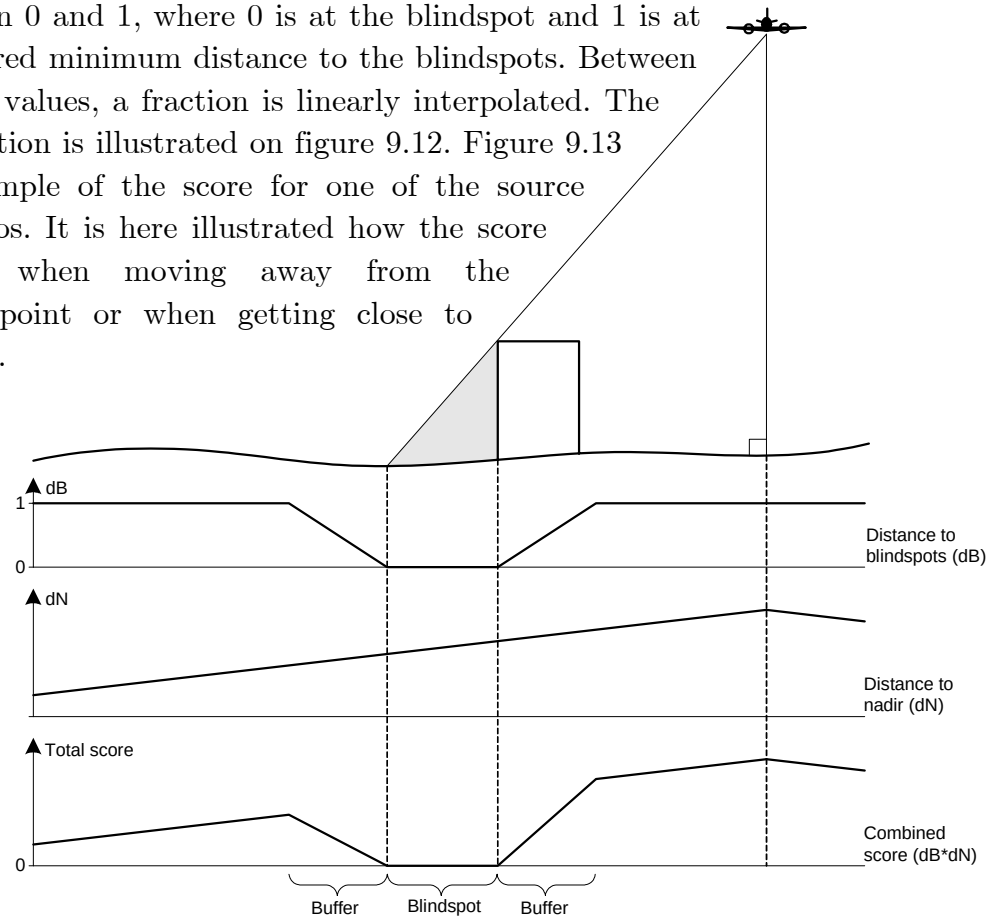


Figure 9.12 . Principle of the combined score method.

Figure 9.13 - Score function for one of the source images. Bright represents a high score. The nadir point is located to the right of the image.

The size of the buffer should not be so large that they will cover the entire mosaic. If it is too large, dN will have almost no effect. Still it has to be large enough to provide a good distance to the blindspots, leaving room for feathering and the missing eaves. Tests showed that a buffer distance of 5 meters generally gives good results. Figure 9.14 shows an example of the mosaic pattern and figure 9.15 the final true orthophoto.

The mosaic generated with this method is by far the best of the three methods presented so far. It is the less fragmented while able to deal with some of the inaccuracies in the DSM. The mosaic at figure 9.14 mostly use the two images that are the closest ones, and is only supplemented by the remaining source orthoimages at the few spots that are obscured in the two "main" images. A pattern this homogeneous will also be less likely to cause cut-throughs like the half car from figure 9.11.

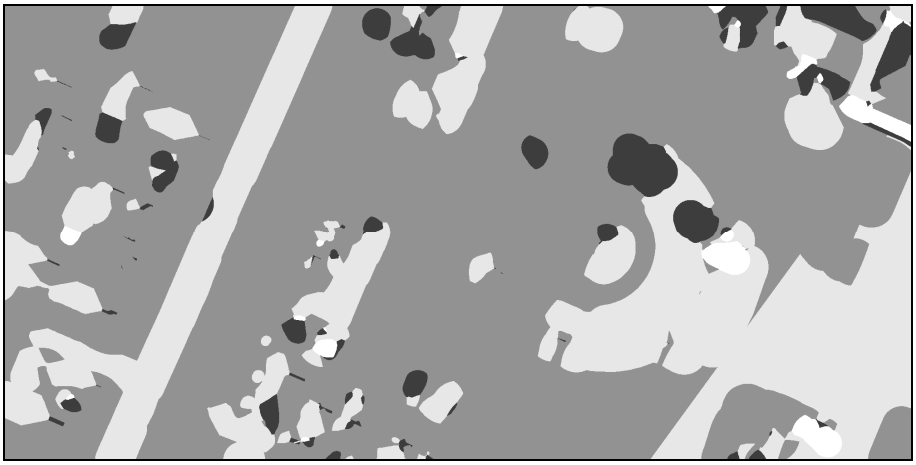


Figure 9.14 - Mosaic pattern based on the combined nearest-to-nadir and blindspot-buffer score function. Blindspot buffer is 5 meters. Compare this pattern with the simple mosaic at figure 9.2.

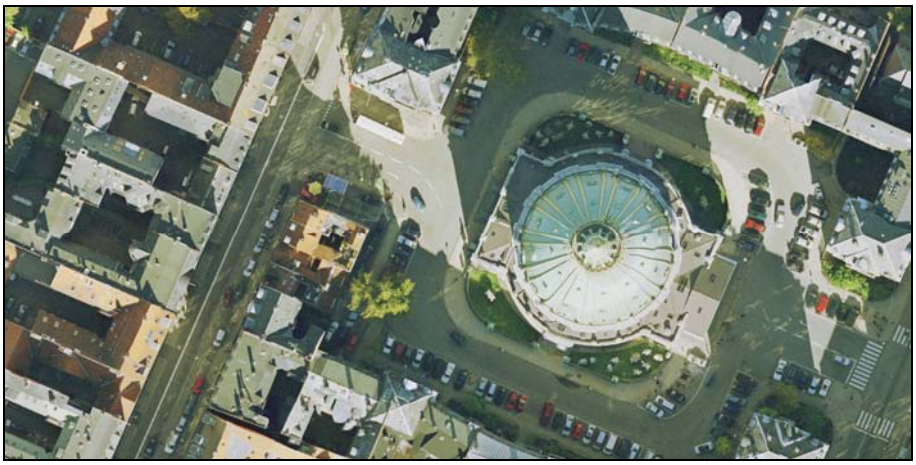


Figure 9.15 - True orthophoto based on the combined nearest-to-nadir and farthest-from-blindspots score function.

9.2 Feathering

Seamlines can be further hidden by the use of *feathering*. Feathering smears the seamline edges by gradually shifting from one image to another. One method for feathering a mosaic is by applying a mean-filter to the mosaic. The mean filter is normally used to smooth digital images by averaging the pixels. It eliminates pixels that are unrepresentative of their surroundings, and is therefore often used for removing noise in an image. It belongs to the family of convolution filters, which is able to multiply arrays of different sizes but of the same dimensionality.

When used on the mosaic pattern, the mean filter will even out the edges gradually between the images. The mean filter is an n x n filter where the pixel in the center kernel is assigned the mean value of the filter values.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

3 x 3 mean filter

1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25

5 x 5 mean filter

There is two ways of increasing the size of the feathering; -either by using a larger filter or by applying the same mean filter several times. The final results are very similar. Performance tests in MATLAB showed a small gain in speed by applying the small filter n times, instead of applying one large filter.

If a 3 x 3 mean filter is applied once, the feathering will be 2 pixels wide. When applied n times, the feathering will be $2n$ pixels wide. This is illustrated on figure 9.16.

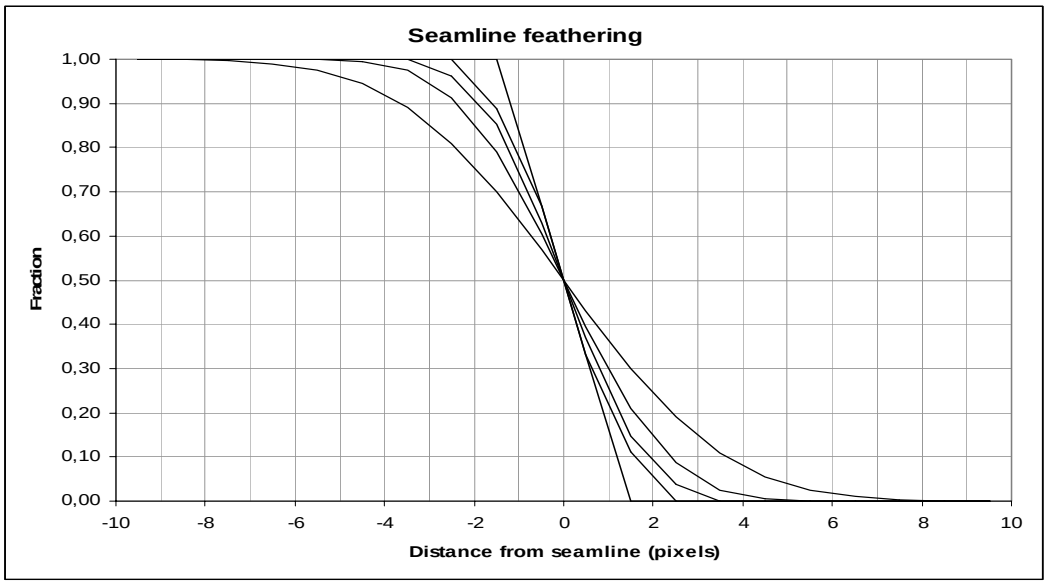


Figure 9.16 – The effect of a mean filter on the seamline when applied to the mosaic. The curves are a 3x3 mean filter applied 1, 2, 3, 5 and 12 times. The curve shows the fraction of the pixel value to include from one image. A similar mirrored curve will exist for the image on the opposite side of the seamline, and combined they will sum to 1.

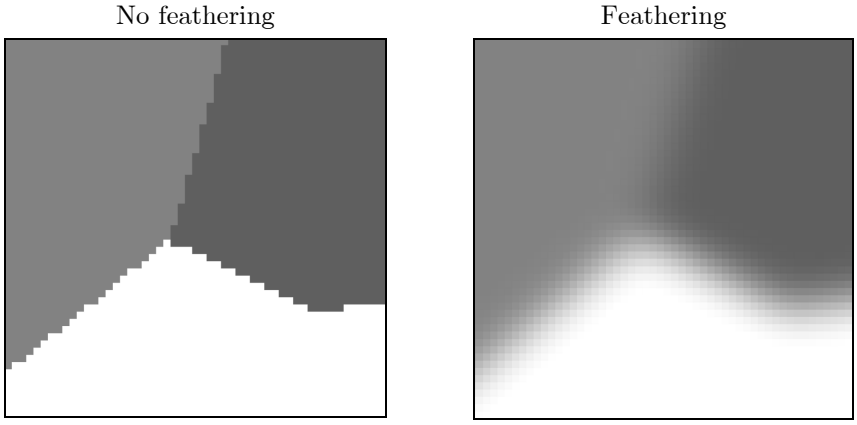


Figure 9.17 – The effect of feathering the mosaic. The fraction of a class that is present in the feathered mosaic is the fraction to include from the corresponding orthophoto.

The mosaic is split up in n separate binary images/layers, one for each image, where a value of 1 means *include pixels from this image* and 0 means *exclude*. The filter is then applied to each of the n images. The fraction in pixel i,j from layer m in the mosaic specifies the amount to include from pixel i,j from the orthophoto m. If the fraction in a pixel is 0.5 for two adjacent images, the resulting pixel value will be the sum of half of the intensities in each band in the two images. This can be written as:

$$trueortho(i,j) = \sum_n mosaic_n(i,j) \cdot orthoimage_n(i,j) \text{ , where } \sum_n mosaic_n(i,j) = 1$$

The feathering is implemented in the MATLAB mosaic code in Appendix B (*Merge.m*). An effect of the mosaic can be seen on figure 9.18.

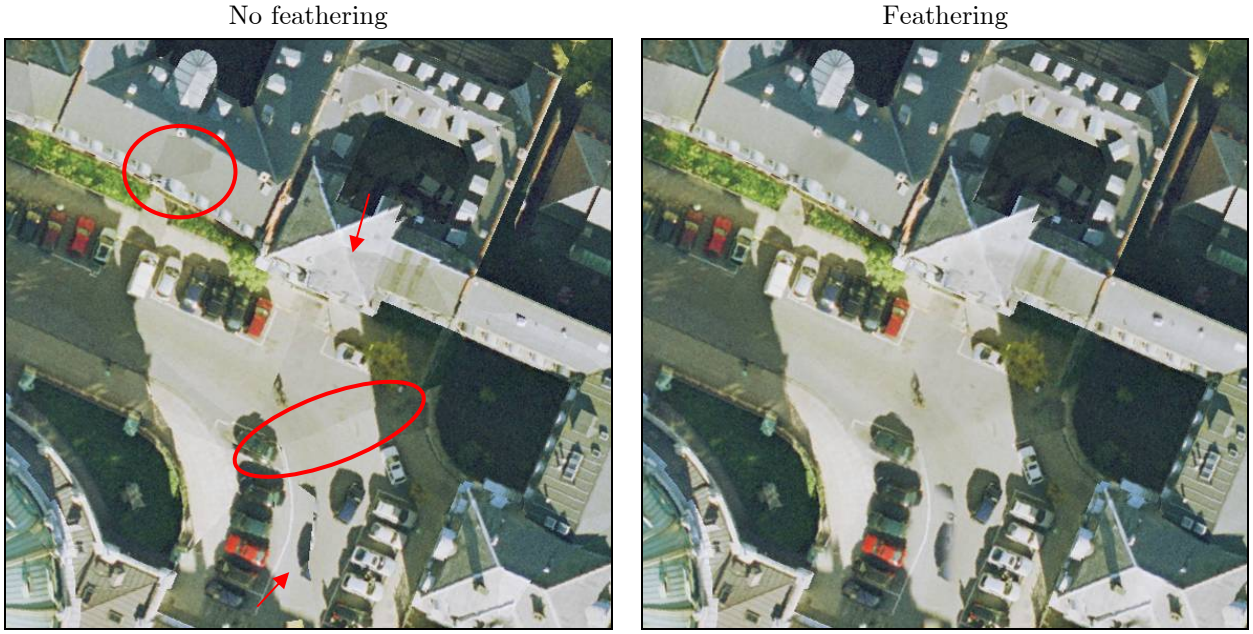


Figure 9.18 – Left: No feathering. Some of the visible edges are marked with red. Right: Feathering applied. Also notice the feathering on the half car at the center bottom of the images.

9.3 Enhancing the mosaic

The mosaic is crucial for a good final product, and the combined score method presented in section 9.1.3 has shown some good results. This section presents some methods and ideas for further development of the automatic mosaicking.

9.3.1 Changing the scores

The scoring method used in this project is easy to change or extend with other properties. As of now, the score only consist of the distance to the nadir point and the distance to blindspots.

A property that could be nice to include, is by assigning higher scores to images where the surfaces are more perpendicular to the camera. If the surface is perpendicular, the resolution in the image is higher than a surface with a narrow angle (cf. figure 9.22). This is a method that is used with the Sanborn METRO software [8].

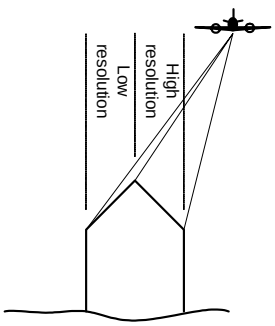


Figure 9.19 - Different resolution due to intersection angle.

The true orthophoto application developed in this project is already capable of creating an "angle-map" that can be used for this method. Figure 9.20 shows an example of an angle-map.

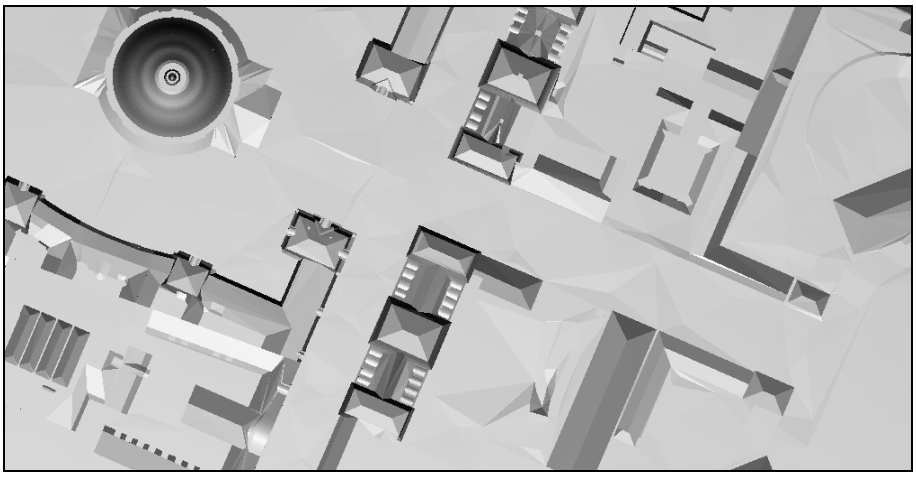


Figure 9.20 - Angle map of the surface model. Bright surfaces are more perpendicular to the camera than darker surfaces.

The seamlines are preferable placed where the source images look most alike. Placing the seamlines along similar "routes" in the image decreases the change of cutting through vegetation that isn't included in the DSM, or through cars that is only present in one of the source images. The seamlines can be placed by using a weighted-graph search algorithm which finds where the images are the most similar [8]. Figure 9.21 shows a similarity map, which is created by subtracting one image from the other.

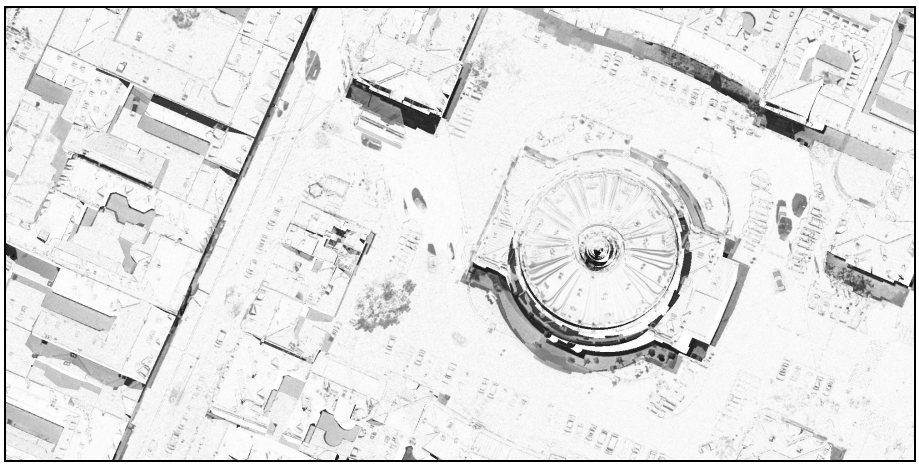


Figure 9.21 – Similarity map between two source images. Similar pixels have white shades.

9.3.2 Reducing mosaic fragmentation

The results generated with the method described in section 9.1.2 with feathering applied are all in all very good, but the very fragmented mosaic increases that chance of having seamlines running through cars, trees and anything else that is not included

in the DSM. These types of cut-throughs are very visible in the image. Although this is only a smaller problem with the combined method presented in section 9.1.3, there also exist small fragments in this mosaic.

A method for decreasing the fragmentation and thereby further decreasing the chance of creating these unfortunate cuts could be a sufficient solution. When looking at figure 9.9, it is obvious that even though the score function says that image 2 is the best to select, it could just as well still select image 1, if this is not obscured. A neighborhood analysis could help determine whether it wouldn't be better to select one image over the other. The general idea is that if the neighboring pixels refers to image A, the pixel-in-question should also be A.

Image classifications that use maximum likelihood estimators usually give a certain amount of noise. There exist several methods for removing noise in an image classification, and since the method described in section 9.1.2 basically is a maximum likelihood method, some of these methods can be applied.

A very simple filter for removing noise in a classification is the mode filter. The filter looks for the most common class around a pixel, and assigns this class to the center pixel [3]. For instance if a pixel refers to image A, but is mostly surrounded by pixels referring to image B, the center pixel will be assigned to B. The filter ignores the scores for the classes, which doesn't really matter in this context. The only thing that the filter should be aware of is not to assign any obscured pixels to the mosaic.

The size of the filter also determines how large clusters that can be removed. The centre pixel in the mosaic at figure 9.22 will be classified as class 5 in a 3x3 search window, but as class 2 in a 5x5 window. All class 5 pixels would also disappear in a 5x5 window. If class 2 is obscured in the centre pixel, the class would instead be changed to class 3 which is the next-best class. Unfortunately the objects in the mosaic can be rather large, which will demand fairly large search windows, thus increasing the number of computations needed. There exist several other advanced methods for strengthening an image classification which will not be covered here.

1	1	2	4	4
1	3	5	5	3
1	3	5	5	3
2	2	2	2	3
2	2	2	3	3

Figure 9.22 - Part of a mosaic pattern. The numbers functions as a reference to the source image.

9.4 Summary

In this chapter, a process for mosaicking images together to form a seamless true orthophoto was sought after. Three pixel-score methods were devised that selected source images by a score in each pixel. The best method found, was based on a score scheme combined by the distance to a nadir point and the distance to obscured areas. A method based on a mean convolution filter was used for feathering the seamlines to hide small differences at the seamlines.

Chapter 10 Test results

The methods devised in the previous chapters, are here put to the test on some different areas of Copenhagen. Problems are drawn out as examples and commented. The mosaic method used in all the examples are the combined score function described in section 9.1.3. Most are generated with normal angle imagery and the Copenhagen 3D City model. A few is also generated on basis on a simpler DSM, and with both wide and normal angle images for comparison.

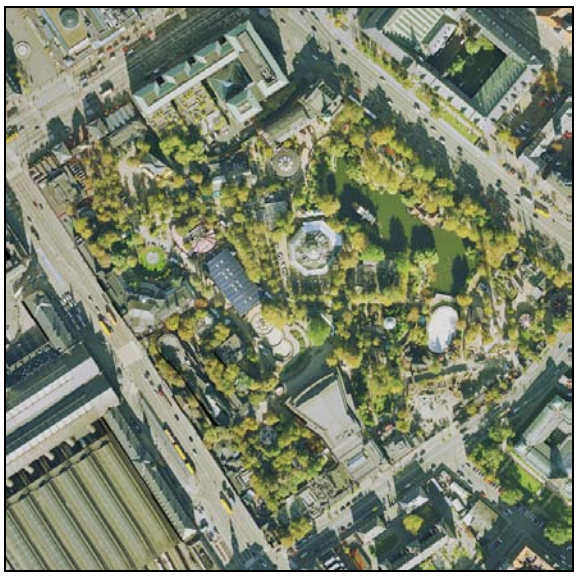
Due to the size and resolution of the true orthophotos, it is not possible to show the full images here. Instead detailed close-ups will be used. The images can be found on the companion CD-ROM in full resolution and extent. They are supplemented by images of the mosaic patterns and ESRI world files for spatial reference in GIS applications.

Below are shown an overview of the generated true orthophotos evaluated.

Central Copenhagen

0.5 km^2. Pixelsize: 0.15 m

Tivoli



0.195 km^2. Pixelsize: 0.15 m

Christiansborg

0.214 km^2. Pixelsize: 0.15 m

Marmorkirken

0.02 km^2. Pixelsize: 0.10 m

Wide/normal angle and simple/complex DSM

0.08 km^2. Pixelsize: 0.25 m

10.1 Performance

As mentioned in chapter 7, the speed of the orthorectification was somewhat crucial to make it feasible to orthorectify larger areas. When the true orthophotos was created, the processing time was registered. The testing was done on what is comparable to a standard home computer at the time of writing this report. The overall specifications were:

Processor:	Pentium 4, 2.8Ghz, 800Mhz system bus.
Memory:	768 MB DDR RAM
Operating System:	Windows 2000 SP4

The largest true orthophoto was done on an area of size 730m x 700 m which is approximately a half square kilometer. The pixel resolution was 0.15 m creating roughly 22.7 million pixels. 10 input images were used, which all needed to be rectified over the same area. The processing time is given below:

Orthorectification and locating blindspot:	10 hours ($\approx$ 1 hour per image)
Mosaicking and feathering:	20 minutes.

To speed up the process, the rectification can be distributed among several computers, letting each computer rectify one or more of the source images. Using this approach to let available computers rectify over night, would enable the option of creating much larger true orthophotos. The rectifier application that was created for this project does currently have a small bug, which means that it sometimes runs out of memory after rectifying several images. It hasn't been possible to locate the error, since it is not persistent, but needs to be resolved before relying on long unsupervised rectifications.

10.2 Pros...

The overall results of the true orthophoto rectification are very good. Most of the narrow backyards are fully visible, all rooftops are moved to their correct position and no walls are visible in the final result. Figure 10.1 and figure 10.2 shows a few examples.

Tall objects with enormous relief displacements are still rectified perfectly, and the large areas they obscure are filled with data from other images. The very tall chimney below is rectified so that it is actually possible to look down the chimney.

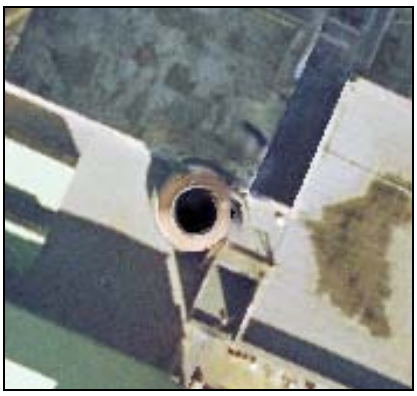


Figure 10.1 - Rectified chimney. (67 meters tall)

Figure 10.2 - Tower at the Danish Westminster. (100 meters tall)

The narrow backyards of central Copenhagen are clearly visible. In normal orthophotos visible backyards was only available very close to the nadir point.

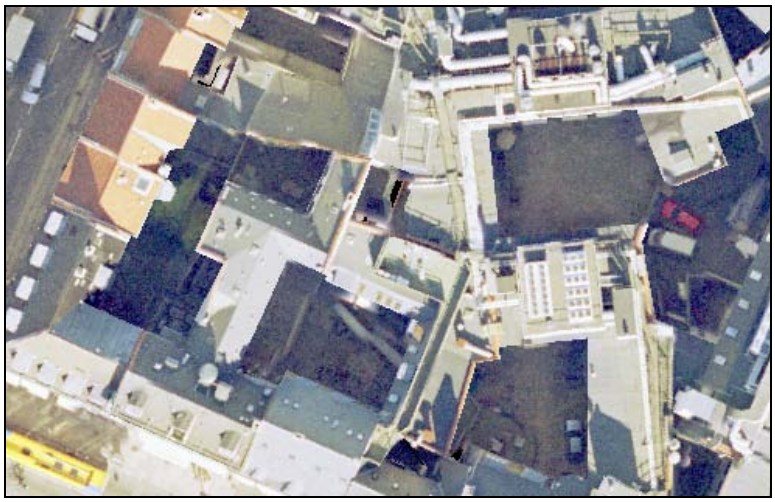


Figure 10.3 - Narrow backyards visible in a true orthophoto.

10.3 ...and cons

Some minor problems still occur in the true orthophoto generation. Some of them were expected based on the theoretic conclusions, and are visualized below, along with some new ones. It should be emphasized that most of these remaining errors are small and usually only noticeable if looking hard for them.

A problem treated earlier is the chance of cutting through objects not included in the DSM or has moved between the shooting of the source images. This still occurs, but they are rare, and feathering often makes them less visible. Compared to the number of cars in the image, only a relatively few cases of cut-through cars were found. Figure 10.4 shows an example of a bus placed around the seamline. On of the more curious incidents is the red and white car at figure 10.15. When looking closer, it is revealed that it is actually two cars mosaicked together.

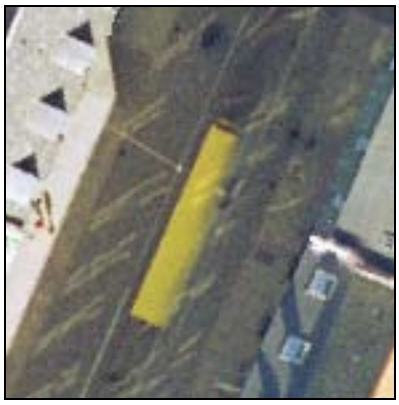


Figure 10.4 - Seamline running through bus.

Figure 10.5 - Two cars mosaicked together.

The same unfortunate cut-throughs can also happen to vegetation. Merging images with different relief displacements can cause a tree to look weird if a seamline runs

through it. Often the structure of a tree combined with the feathering hides the seamlines. Furthermore the trees are usually placed a few meters from the buildings outside the blindspot buffers, thus the image used should be the one that causes the least relief displacements. Below is the two cases found. The errors aren't that noticeable after all.

Figure 10.6 - Mosaicking of non-rectified trees is weird-looking, though hardly noticable.

Deviations in the DSM can cause a poor fit along the seamlines. The dome on the church below isn't measured that exact. This results in a similar poor correction of the relief displacements. The gold-decoration on the dome should look like straight lines in the true orthophoto, but they bend and leap, especially at the edge of the dome.

Figure 10.7 - Errors in DSM causes inaccurate rectification.

Because the surface model doesn't contain eaves, they are cut off rather hard. During the rectification, the roof is rectified back to its correct position, but the eaves are not moved with them, and therefore the edge of the roof sometimes gets completely removed. This results in a pixilated (aliased) cut between roof and the terrain. This is illustrated on the image below. Compare this with roof-edge at the right half of the image where the pixilation doesn't occur.

Figure 10.8 - Hard cut-off of roof eaves (pixilation exagerated).

Minor differences in color can cause some visible seamlines. This is especially the case with surfaces that reflects the light very differently depending on the angle of the surface to the camera and the sun. An illustration of this problem is shown below. Notice that there is no visible color difference on the ground though the seamlines continues here, so the difference must come from the roof surface's reflectance.

Figure 10.9 - Color match not perfect caused by light reflected differently in the source images.

Some areas are still completely obscured; especially where there isn't a sufficient overlap available. This also reveals another problem with the left-out eaves in the DSM. The blindspots are detected correctly but they don't extend enough. The blindspots are too small to cover the "real" blindspot. Below edges of the roof can be seen just outside the detected blindspot (marked with red). Normally this doesn't pose a problem for the mosaic, since the score methods tries to use images located far from the blindspots.

Figure 10.10 - Incorrectly detected blindspots caused by eaves missing in the DSM.

An insufficient overlap pose a large problem as illustrated below. The rollercoaster in Tivoli has many obscured pixels, because only one image provides coverage here. Furthermore it is illustrated how the vegetation is completely ignored when locating blindspots.

Figure 10.11 - Insufficient overlap causes many blindspots.

10.4 Using simpler DSMs and wide angle imagery

The method devised in this project has also been tested on other sets of source data. Images created with a wide angle lens taken from a lower altitude should create images with larger relief displacements. Errors in the DSM should therefore be more noticeable than in the true orthophotos based on normal angle imagery. An example of this is the church where the dome isn't correctly modeled. The problem illustrated at figure 10.7 is much more evident on the image below:

Figure 10.12 - Inaccuracies in the DSM cause inaccurate rectification and are even more evident with wide angle imagery.

The wide angle imagery also generates many more and larger blindspots where none of the images provides coverage, especially in narrow backyards. This was expected from the coverage analysis in chapter 4, where almost 5% would be obscured.

A simpler DSM was generated using the national Danish topographic map *TOP10DK*. In TOP10DK, the buildings are registered as outlines along the roof edges. Only changes larger than 1 meter in both the horizontal and vertical directions are registered. Even though these changes can happen sudden, the changes are registered with less detail, often only at the corners of the buildings, as illustrated on figure 10.13.

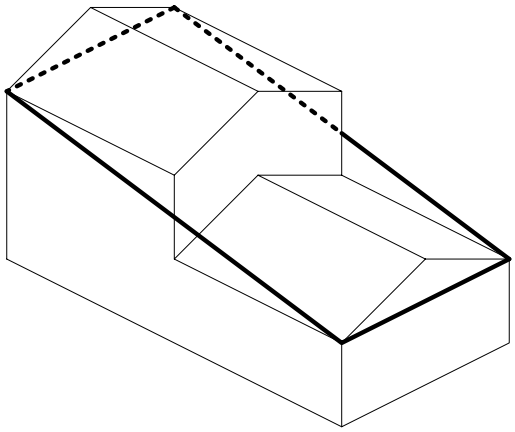


Figure 10.13 – Simplification in TOP10DK (thick lines)

A copy of the building outlines are pulled down to the terrain level, and using the two sets of outlines as breaklines combined with the terrain model gives a much simpler DSM, where the roof construction is missing. This is a cheap and fast way of creating a DSM based on existing data.

As shown earlier, an inaccurate DSM will generate larger relief displacements especially when wide-angle imagery is used. Especially the roof tops not modeled by the TOP10DK DSM will be much more distorted, as illustrated below.

Figure 10.14 - True orthophoto based on TOP10DK DSM with wide angle imagery.

Figure 10.15 - True orthophoto based on TOP10DK DSM with normal angle imagery.

The edges of the roof are moved to the correct position, but the "center" of the roof is not placed at the actual center of the building. This is illustrated at figure 10.14 to figure 10.16.

Figure 10.16 - Roof ridge not rectified to correct position.

Generally the errors are most evident with the wide angle imagery. Below is illustrated some of the worst parts of the true orthophoto. The wide angle-based true orthophoto (figure 10.17) are much more distorted and the mosaic is extremely poor. The normal angle-based true orthophoto (figure 10.18) doesn't give as poor a mosaic, because of the much smaller relief displacements. The normal angle images it thus less sensitive to the level of detail and accuracy of the DSM. It should also be noted that the nadir point of some of the normal angle images are located closer to the area in question. The wide angle images available for comparison only had the same nadir points in half of the source images. The reason for this is that the wide-angle images are only shot with 20 % sidelap, causing the second flight line to be placed farther away.

Figure 10.17 – Mosaic based on TOP10DK DSM and wide angle imagery.

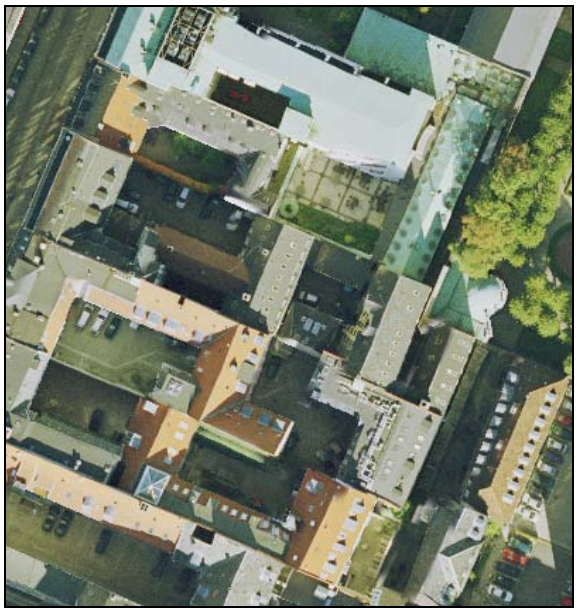


Figure 10.18 - Mosaic based on TOP10DK DSM and normal angle imagery.

10.5 Creating large-scale true orthophotos

It was interesting to see what the mosaic pattern would look like on a larger scale. Because an image is given a higher score close to its nadir point, the mosaic pattern should reflect this. Figure 10.19 shows that this is actually true. Even though the mosaic is generated from ten images, six images are generally selected and are roughly covering each of their square-like area. The remaining four images are only filled in at a few places towards the top-right and bottom-left corners.

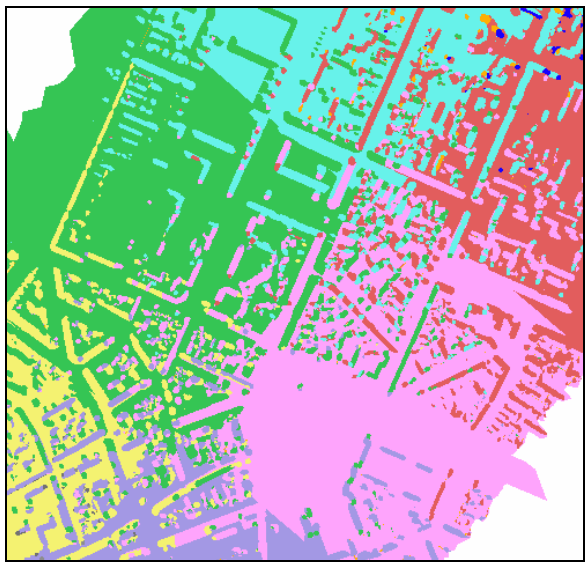


Figure 10.19 - Large true orthophoto mosaic pattern based on 10 source images.

The large area and the large number of source images gave some memory problems for MATLAB when trying to mosaic the images. Therefore the images were split up in smaller images of 1000x1000 pixels and processed separately. To make sure that the mosaic pattern would fit together, an overlap between the subimages were added. Only the buffers around the blindspots are influenced by neighboring pixels, so an overlap of the buffer size is sufficient. This proved to work very well, and the mosaic patterns fitted perfectly when they were merged back together. This entire split/merge process can easily be done fully automatic in MATLAB, where splitting and merging of matrices are standard functions.

10.6 Summary

The methods described in the previous chapters were tested on different areas of Copenhagen. The overall result was quite good, but small remaining errors was pointed out and described. These errors were usually caused by limitations or inaccuracies of the DSM used. The model was furthermore tested on a less detailed DSM, which mostly caused problems with unrectified relief displacements. The errors caused by a poor DSM were found to be much more evident when using low-altitude wide-angle imagery, where these relief displacements are much larger than corresponding high-altitude normal-angle imagery.

Chapter 11 Conclusion

The overall aim of this project was to devise a method for generating true orthophoto imagery. The method was divided in four steps: Rectification, color matching, mosaicking and feathering. The rectification was implemented in an easy to use stand-alone application, and the color matching, mosaicking and feathering in simple MATLAB scripts.

11.1 Evaluation

The rectification was done by raytracing from the surface model back to the camera, taking lens distortions into account and registering rays that were blocked by objects in the surface model. To improve the speed of which the rectification was done, a binary search tree was used. Using the search tree structure, tracing from a surface model back to a camera, and checking for blindspots could be done several thousand times per second.

The histogram matching algorithm proved useful for matching the colors of one image to a reference image. Using the rectified orthophotos and the knowledge of obscured areas, pixels representing the same objects in the images were used for finding the look-up table needed for color correction.

Three methods for mosaicking were devised, which all were based on a pixel-score method. One method showed some very good properties. It primarily increased the score for the source images that had the closest nadir point measured from the pixel-in-question. Within a buffer distance around obscured areas, the score was lowered linearly towards these spots. A buffer distance of 5 meters was found to be sufficient. Compared to the two first methods, this approach created a very homogeneous mosaic pattern, and only few visible errors remained. The score method was able to assign images, so inaccuracies of, or missing objects in the surface model were less likely to create problems in the final result.

Feathering was based on a mean convolution filter, and was able to hide most remaining differences in the mosaicked images, as well as making visible mosaic errors less noticeable.

It was assessed that the methods devised is able to be applied on larger scale true orthophoto mosaics.

A thorough investigation of the coverage that different setups of aerial photography provide was conducted. The test results showed that with imagery created with a wide-angle lens from an altitude of 750 meters, and with 60 % forward lap and 20 % sidelap, the coverage of the central parts of Copenhagen was approximately 95 %, and a provincial town like Ribe, the coverage was above 99 %. By primarily increasing the sidelap and secondarily increasing the flight altitude and focal length proved effective to increase the coverage. Increasing the sidelap to 60 % and doubling both altitude and focal length provides coverage of almost 99.9 % in central Copenhagen.

11.2 Outlook

Improvements to the true orthophoto process can still be made. As pointed out in the test results, it is not fortunate to have seamlines run through objects like cars and vegetation. Using difference maps, it should be possible to direct the seamlines around differences to prevent cut-throughs.

In all the test results, the completely obscured areas were painted black. Finding a method that is able to make these areas less striking could provide a visually better product. One approach to hide the blindspots could be to interpolate colors from the edges of the obscured areas.

Especially in the October images, the contrasts between areas in sun and shadow are very high. Since the exact position and time of the photograph is known, the surface model makes it possible to calculate where shadows from the buildings are thrown. It should be possible to reduce the effect of the shadows and brighten these shadowed areas. Histogram matching could be a method to improve these areas, either by equalizing the histogram or by matching to the histogram of the sunlit areas.

The rectification process is by far the slowest part of the true orthophoto generation. A recent article in the Danish “PC World” discusses using the new processors on the 3D graphic cards for fast calculations as an alternative to using super computers. These processors are highly optimized for a few set of calculations. Since raytracing is an essential part of a 3D graphic processor, it may be possible to speed up the process significantly by this hardware-based approach.

Today an alternative to the true orthophoto is normal orthophotos created on imagery with a large overlap, for instance 80 %. Using only the central part of the images, the worst part of the relief displacements can be removed. This method is expensive in flight hours and image preprocessing, but most likely cheaper than creating a detailed DSM first. In the near future the true orthophoto will probably be seen as a cheap add-on product for the digital city models, or the city model as a nice bi-product of the true orthophoto.

References

[1] Carstensen, J.M., et al., 2001: *Image Analysis, Vision and Computer Graphics*, 1st edition, Informatics and Mathematical Modelling, Technical University of Denmark

[2] Mikhail, E.M., et al.,2001: *Introduction to Modern Photogrammetry*, Wiley and Sons Inc.

[3] Niblack, W., 1985: *An Introduction to Digital Image Processing*, Strandberg.

[4] Nielsen, M., 2004: *Generering af True Ortofotos, en forundersøgelse*, Preparatory thesis, Informatics and Mathematical Modelling, Technical University of Denmark.

[5] Akenine-Möller, T., 2002: *Real-time rendering*, 2nd edition, Peters, A K Limited.

[6] Overmars, M., et al, 2000: *Computational Geometry*, 2nd edition, Springer-Verlag.

[7] Dewitt, B., Wolf, R., 2000: *Elements of Photogrammetry with Applications in GIS*, McGraw-Hill.

[8] Schickler, W., 1998: *Operational Procedure for Automatic True Orthophoto Generation.* International Archives of Photogrammetry and Remote Sensing, 32(4): p.527-532.

[9] Thorpe, A., 1998: *Digital Orthophotography in New York City.* URISA 1998 Annual Conference Proceedings, 722-730.

[10] Shekhar, W., Chawla S,, 2002: *Spatial Databases: A tour*, Prentice Hall

[11] Riazi, A., 2003: *MATLAB Shared Library*, The Code Project (www.codeproject.com)

[12] Frederik, E., 2002: *Compiled .m-file wrapper*, Mathworks MATLAB Central (www.mathworks.com).

[13] Ruffaldi, E., 2003: *1..2..3 ways of integrating MATLAB with the .NET*, The Code Project (www.codeproject.com)

[14] Nielsen, M., Hansen, S., 2002: *Arealklassification baseret på laserskanningsdata*, Informatics and Mathematical Modelling, Technical University of Denmark, B.Sc. thesis.

[15] Amhar, F., et al, 1998: *The Generation of True Orthophotos Using a 3D Building Model in Conjunction With a Conventional DTM.* IAPRS, Vol.32, p.16-22.

[16] Möller, T., Trumbore, B., 1997: *Fast, Minimum Storage Ray-Triangle Intersection*, Journal of graphics tools, 2(1):21-28

[17] Amhar, F., Ecker, R., 1995: *Accurate Mapping of Buildings in Digital Orthophotos.* Proceedings of 17th International Conference of Cartography, Barcelona,

[18] Zhou, G., W. Schickler, et al., 2002: *Technical Problems of Deploying Standard Initiative of National City True Orthophoto Mapping*. Sanborn.

[19] Rau, J. Y., Chen, N. Y., and Chen, L. C., 2000: *Hidden Compensation and Shadow Enhancement for True Orthophoto Generation*. Proceedings of the 21st Asian Conference on Remote Sensing, Vol. 1. p. 112-118.

[20] Triglav, J., 2000: *Coming in: Better Orthophotos*, GeoInformatics, Septemper 2000.

[21] Wynn, I., 2001: *ISTAR's State-of-the-Art Ortho Images*, GeoInformatics, November 2001.

[22] Haskell, L., O'Donnell, R.,: *Stand Straight Up: A True Ortho Perspective on Downtown Denver*, ArcUser, oktober-december 2001

[23] Rau, J.Y., N.Y. Chen, and L.C. Chen, 2002: *True Orthophoto Generation of Built-Up Areas Using Multi-View Images*. Photogrammetric engineering and Remote Sensing, 68(6): 581-588

[24] Orava, E., 1994: *Digitaaliset ortokuvat*, Technical University of Helsinki, Masters Thesis.

[25] Kuzmin, Y., at al, 2004: *Polygon-based True Orthophoto Generation*, Proceedings of the 20th ISPRS congress: 405.

Appendix

Appendix A Contents of companion CD-ROM 103

Appendix B MATLAB scripts 105

Appendix C True orthophoto generator - Users guide 113

Appendix D Raytracer library 121

Appendix A Contents of companion CD-ROM

During this project several applications and scripts have been developed to test the methods described. As a result of the tests, several true orthophotos have been created, and many of them are used throughout the report. All these items are included on a CD-ROM for reference.

Below is a listing of the contents:

Folder	**Description**
\MATLAB	MATLAB scripts used for color matching and mosaicking. These are also available in Appendix B.
\Raytracer	A raytracer class library used for raytracing a surface model. A class library reference is found in Appendix D.
\Rectify_setup	Setup files for installing the orthophoto rectifier application. Appendix C contains a user guide.
\Source	Source code for the rectifier application
\True_Orthophotos	Several true orthophotos created using the rectifier application and MATLAB scripts. Format: Tiff. Coordinate reference system: System 34 Sjælland.

Appendix B MATLAB scripts

This appendix contains the full MATLAB code scripts used for color matching and mosaicking.

Chapter 8 covers color matching, which is implemented using the following scripts:

HistogramMatch.m: Matches the histogram of one image to another image.

GetIndex.m Sub-function used by HistogramMatch.m

Chapter 9 covers mosaicking and feathering, which are implemented using the following scripts:

Mosaic_nadir.m Mosaicking script that assigns score by distance to nadir and visibility.

Mosaic_blindspot.m Mosaicking script that assigns score by distance to obscured areas.

Mosaic_combined.m Mosaicking script that assigns score by a combination of the two previous methods.

Merge.m Merges images on basis of a mosaic pattern.

All the scripts are also available on the companion CD-ROM in the folder

$$\backslash MATLAB \backslash$$

HistogramMatch.m

The histogram matching algorithm matches an input image to the histogram of a reference image. Masks are added to both the input and output image, so that only pixels unmasked in both images are compared.

The algorithm is described in section 8.3.

```
Iin = imread('input.tif');           %Load input image
Iref = imread('reference.tif');      %Load output image
Vin = imread('input_mask.tif');      %Load input mask
Vref = imread('reference_mask.tif'); %Load reference mask

Iout = Iin;             %Initialize output image
Levels = 256;           %Number of intensity levels
bins = [0:Levels-1];    %Create histogram bins 0->Levels-1)
LUT = zeros(Levels);    %Initialize LUT

%Create list of pixels that isn't black and isn't masked:
ValidPixels = find(Vin>0 & Vref>0 & sum(Iin,3)>0 & sum(Iref,3)>0);

%Process color bands. Band 1=red, 2=green, 3=blue
for Band=1:size(Iin,3)
    B = Iref(:,:,Band);    %Extract color band
    B = B(ValidPixels);    %Exclude black/masked pixels
    PixCountRef = size(B); %Pixels remaining in reference image
    histRef = hist(double(B(:)), bins); %Creates reference histogram

    B = Iin(:,:,Band);     %Extract color band
    B = B(ValidPixels);    %Exclude black/masked pixels
    PixCountIn = size(B);  %Pixels remaining in reference image
    histIn = hist(double(B(:)), bins); %Creates histogram

    %Build cummulated histograms
    c1 = cumsum(histRef)';
    c0 = cumsum(histIn)';

    %Create Look-Up Table
    for i=1:Levels
        LUT(i) = GetIndex(c0(i),c1); %see GetIndex.m
    end

    %Apply LUT
    for i=1:size(Iin,1)
        for j=1:size(Iin,2)
            Iout(i,j,Band) = LUT(double(Iout(i,j,Band))+1);
        end
    end
end

imwrite(Iout, 'output.tif'],'tiff'); %Save output image
```

GetIndex.m

This function is used by the histogram matching routine to create a Lookup-table. It returns the corresponding reference value based on an input value in the cumulated histograms.

```
function [GetIndex] = GetIndex(value,Histogram)
tmp = Histogram(1);
out=0;
for i=2:size(Histogram,1)
    if(Histogram(i)>value)
        if abs(tmp-value)>abs(Histogram(i)-value)
            out = i;
            break;
        else
            out = i-1;
            break;
        end
    end
    tmp = Histogram(i);
end
GetIndex = out;
```

Mosaic_nadir.m

Creates an image mosaic based on distance to nadir.

```
%Nadir of source images:
centers = [-70650.334 141936.475;
           -70910.641 141636.566;
           -70548.996 141371.670;
           -70288.263 141677.705];
%Upper left corner of output image
X0 = -70875;
Y0 = 141600;

%Visibility maps:
masks = ['5420V.tif';'5421V.tif';'5432V.tif';'5433V.tif'];
%Orthophoto images:
images = ['5420O.tif';'5421O.tif';'5432O.tif';'5433O.tif'];

Width=2000; %Width of images (pixels)
Height=1000; %Height of images (pixels)
GSD = 0.10; %Ground Sample Distance (PixelSize)

%Maximum score at nadir point. Should be the largest
%possible distance from principal point to a corner (WCS units)
ImgExtent = 850;
MaxDist = 5; %Buffer distance from blindspots(WCS units)

NoOfImages = size(centers,1);
img = zeros(Height,Width,NoOfImages);

for image=1:NoOfImages
    x = centers(image,1);
    y = centers(image,2);

    %Create distance matrix for image
    row = [X0-x : GSD : X0+GSD*(Width -1)-x+GSD]; %Horisontal distance
    col = [Y0-y :-GSD : Y0-GSD*(Height-1)-y-GSD]; %Vertical distance
    row = row.*row;
    col = col.*col;
    for i=1:Height
        img(i,:,image) = col(i)+row;
    end
    img(:,:,image) = abs(ImgExtent-sqrt(img(:,:,image)));

    mask = imread(masks(image,:)); %Load mask
    img(:,:,image) = img(:,:,image) .* mask; %Set score=0 when masked
end

%Create mosaic by maximum score
[dummy, mosaic] = max(img,[],3);

%Merge images
TO = Merge(images,mosaic,10); %see Merge.m
```

Mosaic_blindspot.m

Creates an image mosaic based on the distance to any blindspots.

```
%Orthophoto images:
images = ['5420O.tif';'5421O.tif';'5432O.tif';'5433O.tif'];
%Visibility maps:
masks = ['5420V.tif';'5421V.tif';'5432V.tif';'5433V.tif'];

FeatherSize=10; %width of feathering in pixels

NoOfImages = size(masks,1);

width=2000; %Width of images
height=1000; %Height of images

%Read visibility maps
I = zeros(width,height,NoOfImages);
for i=1:NoOfImages
    I(:,:,i) = ~imread(masks(i,:));
end

%Distance transform mask images
for i=1:NoOfImages
    I(:,:,i) = bwdist(I(:,:,i));
end

%Create mosaic by maximum likelihood
[dummy, mosaic] = max(I,[],3);

%Merge images (see Merge.m)
output = Merge(images, mosaic, FeatherSize); %See Merge.m
```

Mosaic_combined.m

Creates an image mosaic based on distance to nadir and distance to blindspots.

```
%Nadir of source images:
centers = [-70650.334 141936.475;
           -70910.641 141636.566;
           -70548.996 141371.670;
           -70288.263 141677.705];
%Upper left corner of output image:
X0 = -70875;
Y0 = 141600;

%Visibility maps:
masks = ['5420V.tif';'5421V.tif';'5432V.tif';'5433V.tif'];
%Orthophoto images:
images = ['5420O.tif';'5421O.tif';'5432O.tif';'5433O.tif'];

Width=2000;  %Width of images (pixels)
Height=1000; %Height of images (pixels)
GSD = 0.10;  %Ground Sample Distance (PixelSize)

%Maximum score at nadir point. Should be the largest
%possible distance from principal point to a corner (WCS units)
ImgExtent = 850;
MaxDist = 5; %Buffer distance from blindspots (WCS units)

NoOfImages = size(centers,1);
img = zeros(Height,Width,NoOfImages);

for image=1:NoOfImages
    x = centers(image,1);
    y = centers(image,2);

    %Create distance-to-nadir score (dN)
    row = [X0-x : GSD : X0+GSD*(Width -1)-x+GSD]; %Horisontal distance
    col = [Y0-y :-GSD : Y0-GSD*(Height-1)-y-GSD]; %Vertical distance
    row = row.*row;
    col = col.*col;
    for i=1:Height
        img(i,:,image) = col(i)+row;
    end
    img(:,:,image) = abs(ImgExtent-sqrt(img(:,:,image)));

    %Create buffer and calculate blindspot score (dB)
    mask = ~imread(masks(image,:));     %Load visibility map
    mask = bwdist(mask);                %Distance transform
    mask = mask*GSD;                    %Normalize distance
    mask(find(mask>MaxDist)) = MaxDist; %Threshold score above MaxDist
    mask=mask/MaxDist;                  %Normalize scores to 0..1
    img(:,:,image) = img(:,:,image) .* mask; %Multiply dN by dB
end

%Create mosaic by maximum score
[dummy, mosaic] = max(img,[],3);

%Merge images
TO = Merge(images, mosaic, 10); %see Merge.m
```

Merge.m

Based on a mosaic pattern, this function can merge several images. Feathering is applied along the seamlines. The function is used by the mosaic scripts.

Parameters:

images — List of filenames of the source images. This is a matrix with each row containing a filename.

mosaic — Integer matrix. Each cell refers to the index of a source image in "images". The mosaic is generated with some of the previous functions.

FeatherSize — The width of feathering in pixels.

Example:

```
images= ['5420O.tif';'5421O.tif';'5432O.tif';'5433O.tif'];

img = Merge(images, mosaic, 10);
```

```
function [output] = Merge(images, mosaic, FeatherSize)
 NoOfImages = size(images,1);
 width = size(mosaic,1);
 height = size(mosaic,2);

 %Seperate mosaic in a binary image for each source image
 I = zeros(width,height,NoOfImages);
 for i=1:width
    for j=1:height
        I(i,j,mosaic(i,j)) = 1;
    end
 end

 %Feather mosaic pattern
 if FeatherSize>0
    F = ones(3,3)/9;
    for i=1:FeatherSize
        for j=1:size(I,3)
            I(:,:,j) = imfilter(I(:,:,j),F,'replicate');
        end
    end
 end

 %Create output image
 output = zeros(width,height,3);
 for j=1:NoOfImages
    img = imread(images(j,:));
    for i=1:3
        output(:,:,i) = output(:,:,i) + im2double(img(:,:,i)) .* I(:,:,j);
    end
 end
 %Cut off values above 1 (rounding can cause values of 1+eps):
 output(find(output>1)) = 1;
```

Appendix C True orthophoto generator - Users guide

The true orthophoto rectification application developed during this project is capable of ortho rectifying aerial images and locate blindspots. The output of the application can later be used in the included MATLAB scripts to create a true orthophoto. The rectification is done on basis of a TIN. Tools for converting and preparing the TIN is also included with the application.

The application has been written using Visual C#.NET. Full source code is provided on the CD-ROM.

NB: The application uses a lot of memory. 1 GB of memory is recommended for rectification of large source images.

Installation

The companion CD-ROM contains a setup file used for installation in the folder:

$$\backslash Rectify\_Setup\backslash setup.exe$$

When the setup file is executed, a welcome screen is presented. Click the *Next* button and follow the instructions on the screen.

After installation, the tools can be started from:

$$Start \rightarrow Programs \rightarrow True\ orthophoto\ generator$$

Three tools have been installed:

DSFL2Triangles converter	Converts the 3D City models from DSFL format to a triangle-file usable by the true orthophoto creator and DSM indexer.

DSM indexer Builds a tree structure of the surface model for fast raytracing.

True Orthophoto creator Ortho rectifies images and locates blindspots.

Uninstallation

The application can be completely uninstalled using *Add/Remove Programs* from the control panel.

Preparing the surface model

The rectifier needs a surface model consisting of triangles and an index file

The triangles should be supplied in a text file of the following format:

$$X_1, Y_1, Z_1, X_2, Y_2, Z_2, X_3, Y_3, Z_3 \ <new\ line>$$

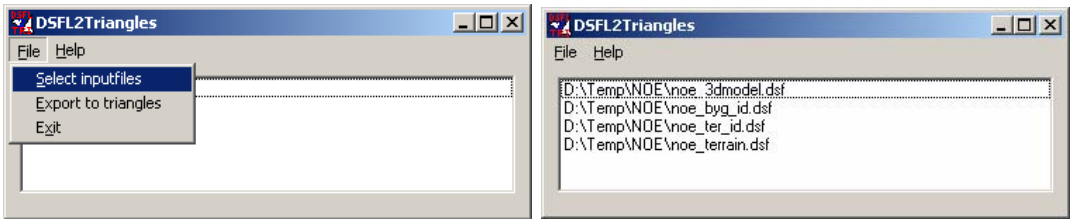
Where X_i, Y_i, Z_i is the vertices of the triangles. An example of the file format is given below, showing five triangles:

```
-72072.4,139931,24.5,-72060.9,139913.8,24.1,-72072.7,139930.7,24.1
-72043.8,139890.2,24.3,-72060.9,139913.8,24.1,-72072.6,139931.4,24.5
-72072.7,139929.3,20.6,-72073.0,139929.2,20.6,-72072.9,139930.6,24.0
-72060.9,139913.8,24.1,-72072.7,139929.3,20.6,-72072.7,139930.7,24.1
-72072.9,139930.6,24.0,-72073.0,139929.2,20.6,-72080.7,139940.5,20.6
```

The triangle file should be saved with the extension '*.tri*'.

The DSFL2Triangles converter is capable of converting a DSFL format surface model into a .tri file. The converter looks for 3D shapes defined by the %F1KR tag and converts the shapes to triangles. If the shapes have more than three vertices, they are split into separate triangles.

Converting DSFL files is done by first selecting the DSFL files that needs to be converted:



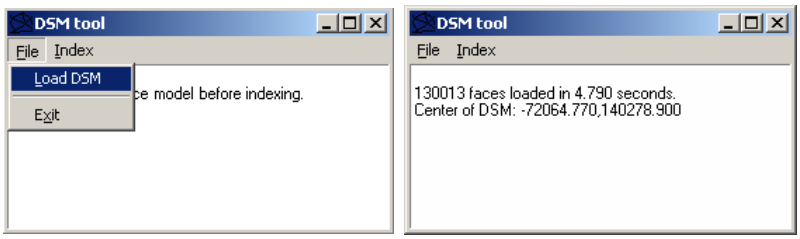
Afterwards, select *File* → *Export to triangles*. The files are converted to .tri files and saved as *<input-filename>.tri*. If you need to merge several surface models together,

this can afterwards be done using a text editor. Just copy/paste the contents of the .tri-files into one large file.

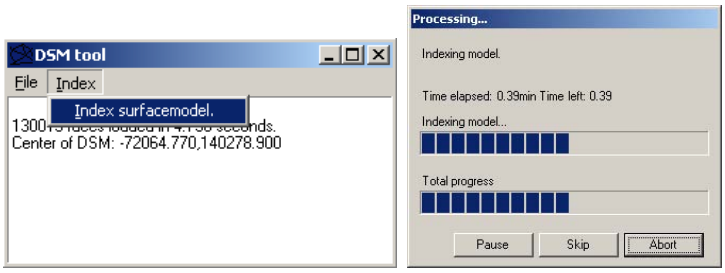
Indexing surface model

The triangle file needs a corresponding index file for the true orthophoto generator. The index is a binary search tree that significantly speeds up the rectification process. The indexing is only needed to be done once for each surface model. In order to do so, start the DSM indexer from the start menu.

Start by loading the .tri file from the menu *File* → *Load DSM*.



When the surface model has been loaded, select *Index* → *Index surfacemodel*.



NB: The index file is saves as *<input-file>.tree* in the same directory as the .tri file. These two files need to be located in the same directory when the True Orthophoto generator loads the surface model.

Prerequisites

In order to successfully create an orthorectification, the following prerequisites are needed:

- Surface model (.tri format)
- Surface model index tree (.tree format)
- Camera parameters
- Photo parameters

The surface model files are described in the previous sections. Camera and photo parameters should be located in two files named '\camera' and '\photo'. The format

of these two files corresponds to the ImageStation project files, so selecting a project directory containing these files is sufficient. If the ImageStation project files aren't available they can be constructed manually.

The '*camera*' file specifies parameters from the camera. These parameters should all be available in the camera calibration certificate. The format is as follows:

```
begin camera_parameters [Camera name]
 focal_length:       [Focal length, millimetres]
 ppac:               [Principal Point of Autocollimation, millimetres]
 ppbs:               [Principal Point of Best Symmetry, millimetres]
 film_format:        [Size of image, millimetres]
 distortion_spacing: [Distance of radial distortions, millimetres]
 distortions:        [Corresponding radial distortions, microns]
end camera_parameters
```

Each parameter can be delimited by either tabs or spaces. The *camera* file can contain several cameras. All cameras that are referenced in the *photo* file should be present in the *camera* file. An example of data is provided below:

```
begin camera_parameters C1713
 focal_length:       303.192
 ppac:               0.004   0.007
 ppbs:               0.02   0.004
 film_format:        230   230
 distortion_spacing: 10 20 30 40 50 60 70 80 90 100 110 120 130 140 148
 distortion_deltas:  0 .1 .2 .3 .7 1.3 1.7 1.8 1.6 1 0.1 -.9 -1.7 -2.6 .1
 distortions:        0 .1 .2 .3 .7 1.3 1.7 1.8 1.6 1 0.1 -.9 -1.7 -2.6 .1
end camera_parameters
```

The photo parameter file contains the interior and exterior orientation parameters for each photograph. The format is as follows:

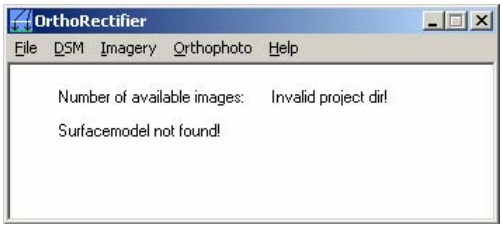
```
begin photo_parameters [Image name]
 camera_name:        [Camera name]
 image_id:           [Filename]
 IO_parameters:      [Inner orientation: Xh Yh VX1 VY1 VX2 VY2]
 EO_parameters:      [Exterior Orientation (angles in gon): X0 Y0 Z0 Ω Φ K]
 image_size:         [Image size in pixels: Width Height]
end photo_parameters
```

Each parameter can be delimited by either tabs or spaces. The *photo* file should contain all photos that should be rectified. All cameras that are referenced in the *photo* file must be defined in the *camera* file. An example of data is provided below:

```
begin photo_parameters 5420
 camera_name:        C1713
 image_id:           C:\images\2003-79-01-5420.jpg
 IO_parameters:      7889.6 7708.2 -66.6 -0.16 -0.15  66.678
 EO_parameters:      -70650.331 141936.475 1529.998 1.2640 2.1007 50.4569
 image_size:         15577 15435
end photo_parameters
```

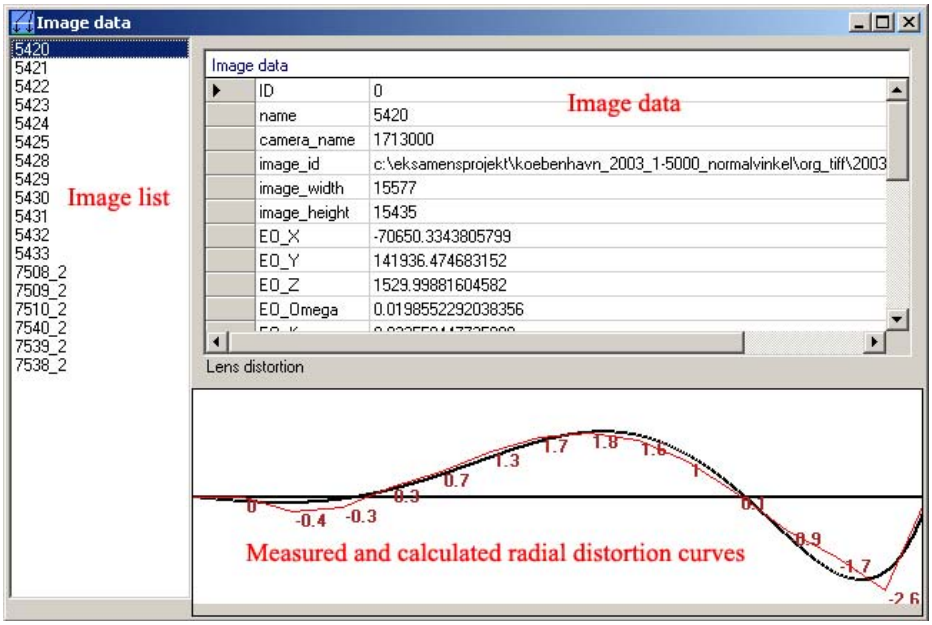
Using the True Orthophoto Creator

When the surface models have been prepared and the camera and photo description files are ready, the *True Orthophoto Creator* can be started from the start menu.



The first time the application is started it needs to be told where to look for the camera/photo files and the surface model. To load the surface model, select *DSM* → *Select surface model*. Make sure that the .tri file you select has a corresponding and valid .tree file. To load the photo and camera files, select *Imagery* → *Select project directory* and locate the directory where the '*photo*' and '*camera*' files are located. The main window will show the number of images found in the camera file and the surface that will be used for rectification:

To review the image data, you can select Imagery → Imagery info... from the menu. From here all specified and derived parameters are provided. A distortion curve of the camera is located at the bottom. The estimated footprints of the photos are calculated for an altitude of 0 meters.



Rectifying images

To start rectifying images, select *Orthophoto* → *Generate...* This brings up the Ortho dialog, where the area for processing is selected along with the images that should be rectified.

To add images to the rectification, double click them on the list to the left. When an image in the list is selected, it is highlighted with red dots in the overview map at the right. Images added to the rectification are shown in blue.

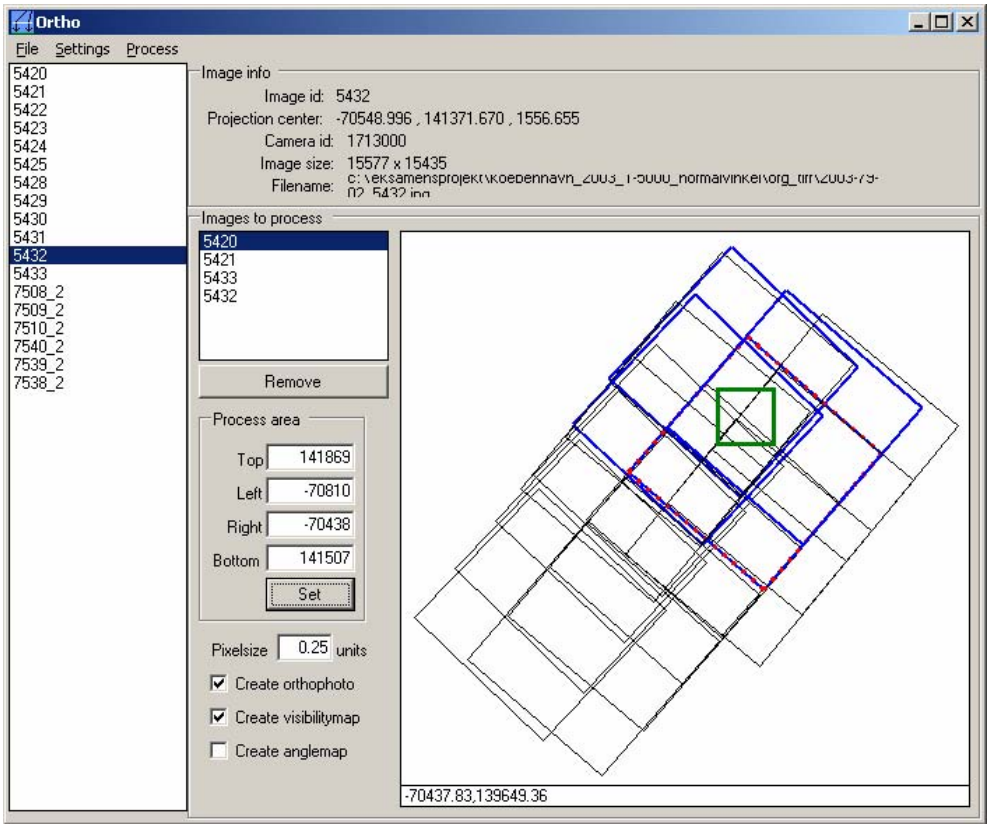
NB: The overview map shows the estimated footprint of the images. The footprint is calculated as the coverage of the photograph on a flat surface at height = 0 metres.

Selecting the output area is done by left-clicking the overview map and dragging the mouse. The output area can also be assigned by specifying the extents at the field "*Process area*" and clicking "*Set*".

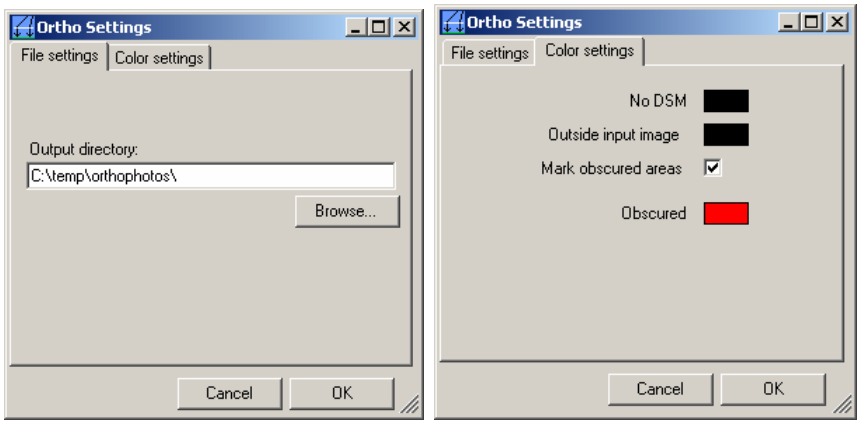
The application is capable of creating orthophotos, visibility maps and angle maps:

- Selecting orthophoto will orthorectify the selected input images and output a 24bit color tiff image for each selected source image The filename will be *<ImageName>O.tif*
- Selecting visibility maps will output binary tiff images, where any black pixels correspond to a blindspot. The filename will be *<ImageName>V.tif*
- Selecting angle map will create greyscale tiff images, where the brightness of a pixel corresponds to the angle between the surface and the ray to the camera. White corresponds to "oblique" and black to parallel. The filename will be *<ImageName>A.tif*

The settings of this dialog can be saved and loaded from the *File* menu.



Before rectification, color and output settings needs to be specified. This is done by selecting *Setttings* → *Options*. From here the output directory can be set. It is also possible to colorize the orthophoto if there is no coverage of either the DSM or the source image, or if the pixels are obscured. The recommended settings are to set all colors to black, and don't mark obscured areas (blindspots).



To start the ortho rectification, select *Process* → *Process orthos* from the menu.

When the rectification is done, the output directory will contain the selected types of output images, an Esri world-file for the orthophotos, and an XML file containing metadata about the images.

The metadata contains the name of the source image, output filenames, projection center of the source image, pixelsize and coordinate of the upper left corner of the orthophoto. An example of the metadata created is shown below:

```
<?xml version="1.0" standalone="yes" ?>
 <OrthoData>
   <OrthoImage>
      <ImageName>5420</ImageName>
      <Ortho>C:\temp\orthophotos\5420O.tif</Ortho>
      <VisibilityMap>C:\temp\orthophotos\5420V.tif</VisibilityMap>
      <AngleMap>C:\temp\orthophotos\5420A.tif</AngleMap>
      <X>-70650.334380579938</X>
      <Y>141936.47468315181</Y>
      <Z>1529.9988160458161</Z>
      <PixelSizeX>0.5</PixelSizeX>
      <PixelSizeY>0.5</PixelSizeY>
      <MinX>-70744.837</MinX>
      <MaxY>141469.916</MaxY>
   </OrthoImage>
</OrthoData>
```

Source code overview

The source code for the applications is available on the CD-ROM in the folder *\source*. A project file for Microsoft Visual Studio.NET 2003 is provided in *\source\TrueOrthoRectifier.sln*.

The source code has been divided into six separate sub-projects:

- **DSFL2Triangles** : The application that converts from DSFL to triangles.
- **DSMtool** : Creates an index tree.
- **Geometry** : Class library defining properties and methods of points, lines and triangles.
- **Raytracer** : Class library that loads and raytraces a surface model.
- **Rectifier** : The ortho rectification application.
- **SetupTO** : Setup project that creates the install utility

The raytracer class library can easily be incorporated into any application written in Microsoft .NET Framework compatible languages. Ideas of uses could be shadow studies of the city model, or predictions of GPS satellite visibility. The raytracer is therefore fully documented in Appendix D.

Appendix D Raytracer library

The raytracer class library is useful for calculating intersections between a ray and a surface model. The surface model should consist of triangles, and indexed using the *DSM tool* described in Appendix C. The raytracer class contains functions for loading and raytracing the surface model and index tree. It has been tested with over 250.000 triangles and was able to find all intersections in approximately 5ms on a Pentium 4 2.8Ghz.

The class library is written in C#, but can be referenced into any application written in Microsoft .NET Framework compatible languages, including C#, Visual Basic.NET and J#. Code examples have been provided in both C# and VB.NET.

The raytracer library is located at the CD-ROM in folder:

$$\backslash raytracer \backslash raytracer.dll$$

In the same directory a Windows help-file "*documentation.chm*" contains a browsable and searchable class library reference.

The source code for the raytracer is supplied in the folder.

$$\backslash source \backslash raytracer \backslash$$

Initializing the tree

To use the raytracer class, the DLL library '*Raytracer.dll*' should be added as a reference. The following command imports the library, and should be located at the top of the code which uses the raytracer:

C#:
```
using Raytracer;
```

VB.NET:
```
Imports Raytracer
```

Declaring a Raytracer object:

C#:
```
private Raytrace RT = new Raytrace();
```
VB.NET:
```
Dim RT As Raytrace = New Raytrace()
```

When the raytrace object has been declared, the surface model and tree can be loaded. This is done with the following command:

C#:
```
RT.LoadModel(filename_tri, filename_tree);
```
VB.NET:
```
RT.LoadModel(filename_tri, filename_tree)
```

filename_tri is the path and filename of the triangle file and *filename_tree* is the path and filename of the tree index.

Raytracing

The intersection ray is defined by two points; an origin and a direction vector.

C#:
```
Raytracer.Point3d or = new Raytracer.Point3d(); //origin vector
Raytracer.Point3d dir = new Raytracer.Point3d(); //direction vector
or.x = 10450.301;
or.y = 340520.33;
or.z = 42.324;
dir.x = 12.341;
dir.y = 4.235;
dir.z = 2.321;
```
VB.NET:
```
Dim or As Raytracer.Point3d = New Raytracer.Point3d() ‘origin vector
Dim dir As Raytracer.Point3d = New Raytracer.Point3d() ‘direction vector
or.x = 10450.301
or.y = 340520.33
or.z = 42.324
dir.x = 12.341
dir.y = 4.235
dir.z = 2.321
```

The raytracing can now be completed. Use the following to search for intersections:

C#:
```
ArrayList list = RT.FindIntersection(or,dir);
```
VB.NET:
```
Dim list As ArrayList = RT.FindIntersection(or,dir)
```

An array list is returned that contain a list of types *Raytracer.IntersectList*.

Below is an example of how to run through the results:

```
C#:
if(list.Count>0) //Intersection found?
{
  foreach(object o in list)
  {
    Raytracer.IntersectList isect = (Raytracer.IntersectList)o;
    Raytracer.Point3d IntersectionPoint = isect.point; //Point of intersection
    int FaceID = isect.FaceID; //ID of triangle intersected
    float DistanceToIntersection = isect.Distance; //Distance between origin and intersection
  }
}
VB.NET:
If list.Count > 0 Then 'Intersection found?
  For Each o As System.Object In list
    Dim isect As Raytracer.IntersectList = CType(o, Raytracer.IntersectList)
    Dim IntersectionPoint As Raytracer.Point3d = isect.point 'Point of intersection
    Dim FaceID As Integer = isect.FaceID 'ID of triangle intersected
    Dim DistanceToIntersection As Decimal = isect.Distance
  Next
End If
```

It is also possible to get the minimum angle between the ray and a triangle. This is useful for rendering, where the angle determines the amount of light it receives and reflects:

```
C#:
float angle = RT.GetIntersectionAngle(dir, FaceID);
VB.NET:
Dim angle As Decimal = RT.GetIntersectionAngle(dir, FaceID)
```

Cleaning up

When all the raytracing has been completed, it is a good practice to clean up the memory. This is done by disposing the raytracer object:

```
C#:
RT.Dispose();
```

```
VB.NET:
RT.Dispose()
```

Index

Axis Aligned Bounding Box Tree 50
backward projection 6
Binary Tree 50
blindspots 10
breaklines 21
C# 39
Camera Calibration Certificate 43
camera constant ..... *See* focal length
Camera Model 41
Chromaticity coordinates 59
colinearity equations 42
Color spaces 59
DBM ..... *See* Digital Building Model
Delaunay triangulation 21
Digital Building Model 19
Digital Surface Model 20
Digital Terrain Model 19
Distance Transformation 71
draping 14
DSM ..... *See* Digital Surface Model
DTM ..... *See* Digital Terrain Model
Exterior orientation 42
Feathering 76
fiducial center 43
focal length 41
footprint 43
forward overlap 27
forward projection 6
ghost image 9
Grid 22
Ground Sample Distance 6
GSD ..... *See* Ground Sample Distance
Hexagonal distance transformation ... 72
Histogram analysis 60, 61

Histogram Equalization 63

Histogram matching 62

histogram stretch 61

IHS 60

Intensity/Hue/Saturation ..... *See* IHS

Interior orientation 43

L*a*b colorspace 60

lambertian 57

Lookup table 63

LUT ..... *See* Lookup table

Mean-filter 76

Minimum AABB 51

Mosaicking 7, 69

ortho rectification 5

orthophoto 5

Perceptual color spaces 59

pinhole camera 41

pixel to world transformation 7

PPAC ..... *See* Principal Point of Autocollimation

PPBS ..... *See* Principal Point of Best Symmetry

Principal Point of Autocollimation 43

Principal Point of Best Symmetry 44

radial distortion 44

radiometric distortion 59

Raytracing 47

Red/Green/Blue ..... *See* RGB

Relief displacements 8

Reprojection 6

RGB 59

seamlines 7

sidelap 27

slab 54

Surface models 19

TIN ..... *See* Triangulated Irregular Network

Triangulated Irregular Network 21

Tristimulus coordinates 59

True orthophoto 9