Dirk Feldmann

Real-Time Rendering and Synthesis of Digital Surface Models Using Textures of Time-Varying Extension

Münster  ♦  2013
Real-Time Rendering and Synthesis of Digital Surface Models Using Textures of Time-Varying Extension
Abstract

*Digital surface models* represent surfaces by finite sets of sampling points based in some plane of reference together with associated sampled values such as elevation and color. These models can be stored as texture data and are used in many applications to create three-dimensional renderings of the underlying surface. Due to advances in data processing and data transmission, it has become possible to already render these surface models while the data acquisition is still in progress. However, the texture data required for rendering will not only vary in content, but also in extension in the course of time during data acquisition, and thus pose a challenge to real-time rendering. This dissertation addresses the problem of creating digital surface models by means of aerial photographs, as well as managing and rendering such data by using textures of time-varying extension, while the acquisition of the data about the underlying surface is still proceeding.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>V</td>
</tr>
<tr>
<td>Preface</td>
<td>IX</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Digital Surface Models</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Representing DSMs</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Images, Photographs and Texture Maps</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 Images</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2 Texture Maps</td>
<td>10</td>
</tr>
<tr>
<td>2.2.3 Representing Images and Textures</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Aerial Images</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1 Classification of Aerial Images</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2 Aerial Image Acquisition</td>
<td>13</td>
</tr>
<tr>
<td>2.3.3 The AVIGLE Project</td>
<td>13</td>
</tr>
<tr>
<td>2.4 Coordinate Systems</td>
<td>14</td>
</tr>
<tr>
<td>2.5 Basic Principles of Photogrammetry</td>
<td>15</td>
</tr>
<tr>
<td>2.5.1 Perspective Projection</td>
<td>16</td>
</tr>
<tr>
<td>2.5.2 Camera Parameters</td>
<td>22</td>
</tr>
<tr>
<td>2.5.3 Image Properties</td>
<td>23</td>
</tr>
<tr>
<td>3 The Flexible Clipmap</td>
<td>25</td>
</tr>
<tr>
<td>3.1 Related Work</td>
<td>26</td>
</tr>
<tr>
<td>3.2 The Flexible Clipmap</td>
<td>27</td>
</tr>
<tr>
<td>3.2.1 Requirements for Handling Spatially Time-Variant Textures</td>
<td>28</td>
</tr>
<tr>
<td>3.2.2 The Clipmap</td>
<td>29</td>
</tr>
<tr>
<td>3.2.3 Managing Aerial Images by Spatial Indexes</td>
<td>30</td>
</tr>
<tr>
<td>3.2.4 Layout Scheme for Tiles</td>
<td>33</td>
</tr>
<tr>
<td>3.2.5 Adding and Updating Tiles</td>
<td>35</td>
</tr>
<tr>
<td>3.3 Architecture and Implementation Details</td>
<td>39</td>
</tr>
<tr>
<td>3.3.1 Caching</td>
<td>39</td>
</tr>
<tr>
<td>3.3.2 Tile Arrays</td>
<td>40</td>
</tr>
<tr>
<td>3.3.3 Scheduling Tile Updates</td>
<td>42</td>
</tr>
<tr>
<td>3.3.4 Tile Map and LOD Calculation</td>
<td>43</td>
</tr>
<tr>
<td>3.3.5 Rendering</td>
<td>45</td>
</tr>
</tbody>
</table>
### Table of Contents

3.4 Performance Analysis ................................................. 47
  3.4.1 Evaluation Setup ............................................. 48
  3.4.2 Results ....................................................... 48
3.5 Discussion ............................................................ 49

4 Digital Surface Model Rendering ....................................... 51
  4.1 Related Work ....................................................... 52
  4.2 GPU based Single-pass Ray Casting Using Clipmaps ............... 53
    4.2.1 Clipmaps for DSM Storage ..................................... 53
    4.2.2 Rendering and Accelerated Ray Traversal ..................... 55
    4.2.3 LOD-determined Ray Termination ............................. 58
    4.2.4 Refinement of Block-sampled Heightfield Reconstruction ... 61
    4.2.5 Sampling Color Textures .................................... 65
  4.3 Performance Results .............................................. 65
    4.3.1 Evaluation Setup and Results ................................ 66
    4.3.2 Performance with Surface Refinement ........................ 68
  4.4 Discussion .......................................................... 68

5 GPU-based DSM Synthesis ............................................... 71
  5.1 Methods for Stereo Matching ...................................... 71
    5.1.1 Matching Cost Functions .................................... 73
    5.1.2 Cost Aggregation and Support Window Size ................. 75
  5.2 DEM Generation Using Space Sweep .............................. 75
    5.2.1 Input Image Selection ....................................... 77
  5.3 GPU-based Implementation ........................................ 79
    5.3.1 Program Setup ............................................... 79
    5.3.2 Program Execution .......................................... 80
  5.4 Color Texture Generation ......................................... 82
  5.5 Improvements of DEM Quality and Results ....................... 84
    5.5.1 Stereo Matching Errors ...................................... 84
    5.5.2 Smoother DEM Data .......................................... 85
    5.5.3 Cost Aggregation Over Support Windows ..................... 86
    5.5.4 Better Half Sequence ....................................... 87
  5.6 Performance Evaluation and Discussion .......................... 88
    5.6.1 Discussion .................................................. 88

6 Texturing Lateral Surfaces ............................................. 93
  6.1 Complete Color Textures for DSMs ................................ 93
  6.2 Aspects of Projective Texturing .................................. 94
    6.2.1 Aerial Image Selection ..................................... 95
    6.2.2 Occlusions Between Lateral Surfaces ....................... 96
    6.2.3 Multiple Projections ....................................... 97
  6.3 Implementation and Results ...................................... 98
    6.3.1 Results .................................................... 99

VI
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4 Discussion</td>
<td>99</td>
</tr>
<tr>
<td>7 Framework Design</td>
<td>103</td>
</tr>
<tr>
<td>7.1 Framework Overview</td>
<td>103</td>
</tr>
<tr>
<td>7.2 Flexible Clipmap Implementation</td>
<td>105</td>
</tr>
<tr>
<td>7.2.1 Multithreading</td>
<td>107</td>
</tr>
<tr>
<td>7.2.2 Communication in Response to the Insertion of Aerial Images</td>
<td>113</td>
</tr>
<tr>
<td>7.3 R*-tree Implementation</td>
<td>116</td>
</tr>
<tr>
<td>7.4 GPU Programs</td>
<td>117</td>
</tr>
<tr>
<td>7.4.1 DSM Rendering</td>
<td>118</td>
</tr>
<tr>
<td>7.4.2 DSM Synthesis</td>
<td>123</td>
</tr>
<tr>
<td>7.5 Further Implementation Details</td>
<td>126</td>
</tr>
<tr>
<td>8 Conclusions</td>
<td>127</td>
</tr>
<tr>
<td>Appendix A GPU Programs</td>
<td>129</td>
</tr>
<tr>
<td>A.1 Flexible Clipmap Shaders</td>
<td>129</td>
</tr>
<tr>
<td>A.2 DSM Synthesis Shaders</td>
<td>147</td>
</tr>
<tr>
<td>Appendix B Additional UML Diagrams</td>
<td>153</td>
</tr>
<tr>
<td>Bibliography</td>
<td>157</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>165</td>
</tr>
</tbody>
</table>
This dissertation is the result of the work and research which was conducted from February 2010 to December 2012 at the Department of Computer Science at the University of Münster. Many thanks go to Prof. Dr. Klaus Hinrichs for giving me the opportunity to conduct my research and for his kind support and guidance. I also thank Prof. Dr. Frank Steinicke for introducing me to the “AVIGLE” project and his support at the beginning of my work, as well as my colleagues at the Visualization and Computer Graphics Research Group for our pleasant on- and off-topic conversations. Furthermore, I would like to give thanks to Aerowest GmbH in Dortmund, Germany, for their kind provision of elevation data and orthophoto images of the city of Münster, and the numerous people who were involved in the acquisition and provision of the original data of the data sets “Blue Marble”, “ETOPO1” and “Mars” which are used throughout this dissertation.

This work was developed in the context of the project “AVIGLE” which is funded by the state of North Rhine-Westphalia, Germany, and the European Union, European Regional Development Fund, “Europe – Investing in your future”.

Münster, December 2012

Dirk Feldmann
1 Introduction

In many application scenarios it is often required to render surfaces which are given as a finite set of point-sampled data. The sampling points, which are assumed to be based in some plane of reference, together with the sampled values form a digital surface model (DSM) of the surface. An important property of a surface is the elevation which denotes the deviation from the plane of reference (ground plane). The sampling positions and the corresponding elevations form a digital elevation model (DEM) of the surface which can be considered as a part of the DSM. If the DSM contains no other information about the surface than elevation, the DEM and the DSM of a surface are identical. However, a DSM may contain any kind of information about a surface and does not necessarily have to provide surface elevation data. In the context of this thesis, we assume that a DSM contains at least elevation data and consider a DSM as a DEM which may provide in addition to elevation data further information about the surface. In other words, we consider a DEM as a minimal DSM so that we can use the term DSM in situations in which the term DEM is used, but not vice versa.

Digital surface models allow to produce three-dimensional (3D) renderings of the surfaces for many different purposes. For instance, 3D terrain models of rural and urban areas are useful for landscape planing, surveying, field mapping and navigation. Further applications can be found, for instance, in the field of bathymetry where the elevation of the ocean floor is used to create maps and depth profiles, or in material science, where surfaces of materials and components are examined at microscopic scales to find structures or defects.

The creation of a DSM requires to sample the surface and to register the resulting values with their corresponding sampling positions in the ground plane. Since it is convenient for many applications to organize point samples on rectangular regular grids, DSM values are often represented as 2D arrays. Such arrays can be visualized as shown in figure 1.1 and may be used directly as texture maps (textures) for rendering.

Depending on the underlying application, different technologies for surface scanning are available, e.g., radar based techniques, echo sounding, laser scanning devices or scanning tunnel microscopes. Another possibility to obtain surface elevation information is to employ photogrammetric methods which under certain conditions allow to recover 3D information from two-dimensional images. These methods are used very often in conjunction with orthographic aerial photographs acquired from airplanes or satellites. In particular, aerial photographs captured from airplanes can be used to create high resolution DSMs of large areas like entire cities. The data shown in figure 1.1 are an example of a DSM generated by means of photogrammetric methods from aerial photographs. The field of photogrammetry has benefited from developments in digital photography and from the increased performance.
of computers, and it has become possible to extract accurate elevation data automatically from aerial images.

A widely used method for surface rendering is to derive a polygonal mesh from the elevation data contained in a DSM and to use traditional rendering techniques based on triangle rasterization. A triangular mesh is an alternative representation of a surface and can be created by triangulating the original point-sampled data. If the DSM contains additional data about the surface, it is common practice to represent this information in textures and to map them onto the mesh. As the generation of high quality meshes and matching textures can be a time-consuming task, this is preferably done when the data acquisition has finished and no further data are added to the DSM. Otherwise the shape and the size of the corresponding mesh and additional textures may have to be modified each time the underlying data change. Furthermore, a DSM may become very large and may have sizes of several hundred gigabytes, which makes it necessary to partition the data in order to make the model manageable by graphics hardware with its limited physical memory. The task of finding an appropriate partitioning can be dealt best with after the final extensions of the domain covered by the DSM are known. Therefore the creation of meshes and textures is usually postponed until after the data acquisition has finished.

Rendering digital surface models during their creation and while the acquisition of the underlying surface data is still proceeding has the advantage that the creation process can be monitored and instant feedback can be given about the acquired data and the underlying surface. This requires that the source data are directly available for further processing. In the case of traditional aerial photography, the airplane usually has to land before the storage devices with the acquired images can be accessed for further processing. By exploiting advances in wireless communication, which enable the transfer of even large amounts of data over greater distances, the creation of digital surfaces models with matching textures and their rendering can already be started while the image acquisition is still in progress. The creation of DSMs which contain elevation data and matching color textures by means
of photogrammetric methods is not restricted to a certain image acquisition technique. It is sufficient if the images look like aerial images and have similar properties. For instance, appropriate photographs can be captured by suitable unmanned aerial vehicles (UAVs) instead of airplanes. Small UAVs offer an attractive alternative for this task, because they are less expensive and can capture images from a wider range of perspectives and at even higher resolutions due to their potentially lower operational altitudes. A drawback of such UAVs is their sensitivity to drift and that they may be less capable of following strict routes above the ground, which is important in traditional aerial photography. A rendering of a DSM which is directly derived from the acquired images can provide an operator with additional information for controlling the flight or for directing a UAV to an area of interest.

Another problem specific to the rendering of digital surface models in a virtual 3D environment arises when using orthophoto textures only. Orthographic images depict surfaces from an orthographic point of view and intentionally do not provide any information about (lateral) surfaces which are perpendicular or near-perpendicular to the plane against which the elevation is measured, e.g., frontages of buildings in urban areas or canyon walls. Figure 1.1(b) contains an example of such an orthophoto texture. Such images are amongst others very useful for creating schematic maps or for enhancing traditional maps with photographic details. In virtual 3D environments, however, a digital surface model is frequently not shown from an orthographic point of view, and the lack of color information on lateral surfaces becomes clearly visible to the viewer. The problem is mitigated, for example, in case of rendering certain kinds of digital terrain models, in which any elevation information resulting from vegetation, buildings or structures other than terrain has been removed. However, in virtual 3D views of cliffs or canyons, the missing color information is still apparent. In order to provide color information for lateral surfaces, oblique aerial photographs can be captured from airplanes by means of specially arranged camera systems or from suitable UAVs, and the images can be incorporated into 3D renderings.

This thesis addresses the problem of creating digital surface models and matching color texture maps from images similar to aerial photographs and managing and rendering such DSMs. But instead of waiting with the processing of a static set of images until after their acquisition has been completed, our goal is to process a dynamically changing set of images in order to enable the rendering of the corresponding surface already during the acquisition. The rendering poses a challenge since not only the sampled elevation and color data may change during acquisition, but also the domain covered by the digital surface model and thus the extension of the textures may vary over time. We assume vertical and oblique aerial photographs as source images which are captured by UAVs at low altitudes and immediately transferred by wireless networks to a ground station for processing. However, our methods are not confined to a certain image acquisition technique. The requirements for the source data and detailed background information about the underlying concepts of our work as well as definitions of important terms are given in chapter 2. Chapter 3 describes our work on managing growing texture data by means of a Flexible Clipmap (FCM). This data structure was developed in order to provide partitioning and level-of-detail concepts for very large
texture data sets that can vary in size during the course of time. By design, the FCM is also capable of accelerating our DSM rendering process which is based on ray casting and is explained in chapter 3. The method we use to obtain elevation data from appropriate source images by means of photogrammetric techniques is presented in chapter 5 and relies on a 3D space sweep. In chapter 6 we present a solution to the problem of missing color information on lateral surfaces in DSMs which utilizes projective textures and some features specific to ray casting. The framework of implemented data structures and algorithms with its main components and properties is described in chapter 7. Our work concludes with a discussion and summary in chapter 8.

The following papers which have been published previously contain parts of the contributions of this thesis: The FCM as described in chapter 3 has been presented at the International Conference on Computer Graphics Theory and Applications (GRAPP) [28]. The work on managing and rendering texture maps of time-varying sizes by means of the FCM is part of papers presented at the Joint Virtual Reality Conference (JVRC) [75] and at the International Symposium on VR innovation (ISVRI) [74]. Chapter 4 is a detailed presentation of the DSM rendering method that has been published in the digital proceedings of the Computer Graphics International (GGI) conference [27].
2 Background

This chapter serves as an introduction into various topics that are essential for the remainder of this dissertation. The terms *digital surface model*, *image* and *texture* are important and used throughout this work, but their meaning in the field of computer graphics is to a certain degree context-sensitive. Hence in sections 2.1 and 2.2 we discuss these terms and specify how we understand them. Since aerial images form the basic input to the methods presented within this thesis their most important properties and aspects of their acquisition are covered in section 2.3. In section 2.4 we present the different coordinate systems that are used throughout this thesis and how they are related to each other. Section 2.5 introduces the basic principles of photogrammetry and the underlying mathematical formulations.

2.1 Digital Surface Models

The term *digital surface model (DSM)* is not clearly defined, because it depends on the context of its usage within the different sciences and applications, as well as the underlying data. In geoscience and photogrammetry, several closely related terms are used, which are sometimes considered to be synonyms, but may also refer to distinct types of models in different disciplines. According to Li et al. [46, pp. 6–9], common terms are

- digital terrain model (DTM)
- digital elevation model (DEM)
- digital terrain elevation model (DTEM)
- digital height model (DHM)
- digital ground model (DGM)

The authors consider DTMs as more complex models that may have additional information about the terrain, such as land-use plans, road networks and natural features, and they view a DEM as a special case of a DTM where the only available information is the elevation. DTEM, DHM and DGM are taken as synonyms for DEM, but interestingly, Li et al. do not use the term *digital surface model* at all.

In contrast, Kraus [42, p. 354], who discusses the earth surface from a photogrammetric point of view, distinguishes DTMs, which contain only elevation information about the terrain, but not about other structures like vegetation or buildings, from DSMs, which contain elevation data for these additional structures.

For the mere purpose of rendering and managing, the actual semantics of the underlying surface data is less important and our methods are not limited to terrains or applications in
geoscience. Most of the methods presented in this work can be applied to data acquired from surfaces at microscopic scales, e.g., by means of a scanning tunnel microscope, or to other complex surfaces such as reliefs and friezes on buildings or works of art. Therefore we prefer the general term surface and formally define a DSM within the context of this thesis as follows:

**Definition:** Digital Surface Model (DSM)

A digital surface model (DSM) is a finite set of $n$-tuples

$$\text{DSM} = \{(t_1, t_2, t_3, \ldots, t_n) \in \mathbb{R}^n\}$$

for which $t_1 = x$ and $t_2 = y$ denote the position of a point $p = (x, y)$ in a section $D \subset \mathbb{R}^2$ of the zero plane, and $t_3 = h$ denotes a deviation at $p$ from the zero plane, i.e., an elevation or depression. Any two distinct $n$-tuples in the DSM have to correspond to different points in $D$, i.e.

$$\forall s = (s_1, \ldots, s_n), t = (t_1, \ldots, t_n) \in \text{DSM} : s_1 = t_1 \land s_2 = t_2 \Rightarrow s = t$$

This definition allows the tuples of a DSM to contain further scalar values besides the spatial position information $(x, y, h)$. In case of $n = 3$ the DSM is identical to a DEM, but if $n > 3$, other information about the surface may be part of the DSM. A DEM is formed by projecting the $n$-tuples of the DSM to their first three components. When indicated, we will use attributes like color textured or simply textured DSM, if the additional information is color, for example, in order to emphasize the presence of additional surface information and its meaning. Otherwise the terms DSM and DEM are used interchangeably and the respective distinction can be inferred from the context or is not relevant at that point. In addition, the zero plane of a DSM will also be called ground plane within this thesis.

Though our comprehension of a DSM is very similar to the one of a DTM by Li et al. in [46], the difference is that our definition does not force the kind of surface to be terrain. Furthermore, it requires at least one scalar value to be present in every DSM that has the semantics of elevation or depression at point $p$ relative to the ground plane in which $p$ is located (height value). The latter aspect is more important for the interpretation of the renderings generated by our methods than for the techniques themselves. It is also intended to make our definition somewhat distinct from the concept of fields as used in physics, although the difference may be rather subtle. For example, a finite set $T$ of triples $(x, y, h) \in \mathbb{R}^3$ may be interpreted as a DSM and treated in the same way. But if $h$ denoted, e.g., air pressure or some kind of density instead of a height value at location $(x, y)$, the interpretation of $T$ as a DSM and the resulting renderings might be less reasonable.

### 2.1.1 Representing DSMs

From a mathematical point of view, there is no reason to confine a DSM to be a finite set of $n$-tuples as in the definition given above. But most surfaces in real-world applications, especially in geoscience, are too complex for creating analytical models by means of mathematical methods, and therefore surfaces are usually sampled at discrete locations, which
2.1 Digital Surface Models

(a) triangular regular network (TRN)  
(b) triangular irregular network (TIN)

Figure 2.1: Illustration of two different types of triangular networks for representing DEMs having 16 samples on a regular grid (left) and with an irregular distribution (right).

results in a representation as finite sets of sampling points (cf. Li et al. [46]). The size of these sets is limited by the spatial resolution of the employed sampling technique and the time spent on sampling, and furthermore by the amount of available physical memory when the data are stored for processing by computers. Depending on the sampling method, the sampled values are distributed either irregularly, or arranged on a regular orthogonal grid.

According to our definition, the first three components of each sample correspond to a point in 3D Euclidean space. Sets of such points can be represented as triangular regular networks (TRNs) or triangular irregular networks (TINs). Depending on the sampled data and the application, one may prefer to create either TRNs or TINs, but both types of networks can be converted into each other by means of re-sampling and sample selection. An illustration of DEMs represented as triangular networks is given in figure 2.1. The generation of TRNs from DSM samples given on a regular grid is straightforward, but not necessarily unique, whereas TINs are frequently generated by a process called triangulation, like Delaunay triangulation, for example. The Delaunay triangulation for a set of points in Euclidean space is the dual graph of its Voronoi diagram, which is also known as Dirichlet tessellation or Thiessen polygons [46, pp. 75–85], and it is unique, if and only if all points are in general position [17, pp. 189–190]. More details on triangulation can be found in [17], for instance.

Triangular networks are the most basic polygonal meshes that can be used for rasterization based rendering, and many algorithms for their creation are available. Due to the requirements of graphics hardware, triangular meshes are very common in computer graphics and used exclusively in numerous applications. However, if these meshes are used for rasterization-based rendering in a virtual 3D environment, triangles which are small, e. g., because they are distant from the viewer, may become projected to less than one screen pixel and therefore will cause aliasing. This effect is usually avoided by using level-of-detail techniques in order to generate meshes in which the size of triangles increases with the distance from the viewer. Since the creation of appropriate triangular meshes requires extra processing of the DSM samples, this representation is less appropriate for our application in which the DSM may frequently change. We prefer to directly employ the DSM.
samples and refer to \cite{46} for further details on the representation of DSMs respectively DTM as triangular networks.

If the sampled DSM values are given on a regular rectangular grid, they are prevalently stored as arrays for processing by computers. The whole DSM may be stored as one *interleaved array* where each element of the array corresponds to one complete *n*-tuple. As an alternative, the DSM can be stored in multiple arrays all having the same number of elements where each element corresponds to one or more components of the DSM *n*-tuple. The individual arrays may be stored in sequence in a contiguous block of memory and can be interpreted as a two-dimensional (2D) array where the second index is limited by the number *k* of arrays which store the data. The two different possibilities for splitting one array of 4-tuples are depicted in figure 2.2. The array in figure 2.2(a) is split into *k* = 4 arrays of single values. In figure 2.2(b) the same array is divided into *k* = 2 arrays, where the first array contains single values and the second one contains triples. By splitting the DSM data with respect to their semantics into *k* different arrays, we can consider each of these arrays as a *layer* of the DSM, and each layer depicts a certain feature of the surface, e.g., the DEM layer which contains elevation data and is present in all of our DSMs.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.pdf}
\caption{Interleaved arrays which store \textit{n}-tuples may be split up into 1 < \textit{k} \leq \textit{n} arrays that can be treated as 2D arrays. Each of the arrays has the same number of elements, but the elements themselves may have different sizes.}
\end{figure}

If a DSM given on a regular rectangular grid in Cartesian coordinates is stored in an array, random access to individual DSM values for a given location in Euclidean space or a certain grid position \((i, j) \in \mathbb{N}^2\) is very efficient\footnote{Throughout this thesis, \(\mathbb{N}\) will be considered to include zero, i.e., \(\mathbb{N} = \mathbb{N}_0\).}. On a regular rectangular grid, which will be called grid for simplicity for the remainder of this dissertation, the spacings between each two samples are uniform and fixed, and each sample position \(p = (x, y) \in \mathbb{R}^2\) can be represented as

\[ p = (x, y) = (i \cdot g_x, j \cdot g_y) + p_0 \]

where \(g = (g_x, g_y) \in \mathbb{R}^2\) denotes the grid spacings in x- respectively y-direction and \(p_0 = (x_0, y_0)\) is the offset of the grid in the reference coordinate system of the DSM at grid position \((i = 0, j = 0)\). \((i, j)\) is the grid position within the grid of size \(W \times H\), and
2.2 Images, Photographs and Texture Maps

$W, H \in \mathbb{N}$ denote the grid size $^2$ in x-direction respectively in y-direction in terms of sample points so that $0 \leq i < W$ and $0 \leq j < H$.

Usually $p_0, W, H$ and $g$ are constant for DSMs and hence can be stored separately. This allows to drop the original sample positions $p = (x, y)$ from the DSM representation and to reduce the amount of memory that is required for storage, because at grid position $(i, j)$, $p$ can be restored by means of the grid spacings $g$ and a reference position $p_0$. Note that this reduction of data cannot be achieved for irregularly sampled data and the TIN representation, and $p$ is therefore made a part of a DSM in our definition. As the arrays in both layouts described above are contiguous sections of linear computer memory, grid samples are conceptually stored either row- or column-wise.

Within this thesis, we use the layered and reduced representation of DSMs as 2D arrays. The employed DSMs either consist of only a single DEM layer or have a second color texture layer which contains color information about the surface. We use row-wise storage and index the $n^{th}$ element of a DSM array by $n = j \cdot W + i$ where $i$ and $j$ denote the column and the row of the grid, respectively. The rows and columns of a grid are indexed starting with zero and grid position $(i = 0, j = 0)$ is conceptually located at the left lower corner.

2.2 Images, Photographs and Texture Maps

The terms image and texture map, or texture in short, have already been used several times and are important in the context of our work. In the area of computer graphics, these two terms can be well differentiated, but their synonymous usage is acceptable if a distinction is not required or only of minor importance. This is justified by the fact that a large number of textures which are used in computer generated renderings are images. Since we make use of textures that are not images in the proper sense in order to handle DSMs, we want to distinguish images from textures and assign different meanings to the two words.

2.2.1 Images

A digital 2D raster image can be defined as a mapping $I$ of spatial locations $(x, y)$ given on a raster $D \subset \mathbb{R}^2$ to a finite set $Q$ of values:

$$I : D \subset \mathbb{R}^2 \rightarrow Q, \ (x, y) \mapsto I(x, y)$$

$Q$ may be either a finite set of discrete scalar values or a finite set of tuples of discrete scalar values. In the former case, the images are frequently called monochrome, and in the latter case they are called color images (see section 2.2.3). The elements of a digital image are called pixels and the values from the co-domain $Q$ of $I$ are often referred to as intensities [34, pp. 1]. This definition allows to interpret any such function $I$ as an image.

---

$^2$In this dissertation, we use the notation $x \times y$ with scalar quantities $x$ and $y$ to emphasize the 2D character of sizes of 2D structures such as grids, textures, matrices and 2D arrays. The notation $x \cdot y$ is used in situations where the actual result obtained by the multiplication of the two values $x$ and $y$ is meant.

$^3$Within this dissertation, the terms image and texture always refer to digital data in the field of computer graphics and not to any analog photographic images or physical objects.
But within this thesis, we use the term *image* to imply that the pixels are associated with real-world objects or scenes in the sense of a *photographic image*. In addition, we assume that the domain $D$ of $I$ is always a Cartesian grid. The actual image acquisition is less important than the content of an image and its property that the pixels are related to real-world objects. According to our interpretation, the term *image* therefore comprises photographs acquired by still image cameras, as well as, for instance, computer generated renderings of real-world objects.

2.2.2 Texture Maps

According to Shreiner [71, p. 390], we consider a *texture map*, or simply *texture*, as a rectangular array of *texels*. For a long time, these arrays have been used in computer graphics prevalently for adding properties such as color, surface normals, reflectance or roughness to surfaces in order to reproduce a certain texture in the proper sense (cf. [83, pp. 223]). Because of the evolution of graphics processing units (GPUs), and in particular due to their increasing employment for general purpose computing (GPGPU computing), the interpretation of these arrays as textures according to their original meaning has become imprecise in many cases, but the term *texture* is still prevailing [59].

By the term *texture* within the scope of this thesis, we only imply that these arrays can be handled by graphics hardware in the same way like traditional texture maps. The texels are not limited to a certain kind of semantics, in contrast to our interpretation of pixels in images. Texels normally need to have a certain format (see section 2.2.3) in order to be suited for processing by graphics hardware. In addition, the maximal extension of a texture which can be employed by graphics hardware, i.e., its lateral sizes in terms of texels along each direction, is confined by some manufacturer specific limits. Typically, textures are conceptually two-dimensional arrays (2D textures), but one-dimensional (1D textures) or three-dimensional (3D textures) arrays are also common. If nothing else is stated, we use the term *texture* as an abbreviation for 2D texture.

2.2.3 Representing Images and Textures

Depending on the co-domain $Q$ of the mapping $I$ which defines an image, a single pixel may be a single scalar value or a tuple of scalar values. A single texel from a texture may likewise be either a single scalar value or a tuple of scalar values. The number of bits required to store one pixel or one texel is called *depth*. The *pixel format* respectively *texel format* is defined by the depth and, if there is more than a single value, by the order of values, i.e., the order of the elements of the tuple. Pixel and texel values are in most cases related to (discretized) intensities from the visual electromagnetic spectrum [34, pp. 1], and we call them *grayscale*s if there is only one value per pixel respectively texel. If there are multiple values for different wavelengths, e.g., red, green and blue in the RGB color model, the values are called *colors*. The different pixel and texel formats, which are frequently used in the context of our work, and their respective labels, properties and short descriptions are given in table 2.1.
### 2.3 Aerial Images

In this thesis we focus on DSMs which are derived from aerial images by means of photogrammetric methods. The most common source of aerial images are digital photographs which are traditionally captured from airplanes. Miniature UAVs (mUAVs) have become a less expensive alternative to conventional airplanes [25] and are increasingly used for civil purposes, e. g., for surveying archaeological sites [8,21,22,68]. The actual image acquisi-

<table>
<thead>
<tr>
<th>label</th>
<th>depth [bits]</th>
<th>no. values</th>
<th>data type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I8</td>
<td>8</td>
<td>1</td>
<td>BYTE</td>
<td>unsigned 8-bit intensity values, primarily used for grayscale images and textures</td>
</tr>
<tr>
<td>F32</td>
<td>32</td>
<td>1</td>
<td>float</td>
<td>32-bit IEEE 754 floating point values, mainly used for storing elevation data of a DSM</td>
</tr>
<tr>
<td>R8G8B8</td>
<td>24</td>
<td>3</td>
<td>BYTE</td>
<td>triple of unsigned 8-bit values, one each for the red, green and blue components in the RGB color model</td>
</tr>
<tr>
<td>R8G8B8A8</td>
<td>32</td>
<td>4</td>
<td>BYTE</td>
<td>quadruple of unsigned 8-bit values, same as R8G8B8 but with an additional alpha channel for controlling the opacity</td>
</tr>
</tbody>
</table>

**Table 2.1:** Description of pixel and texel formats that are used within this thesis. The column data type denotes the interpretation of the bit pattern of a single value.

Conceptually textures and images are 2D arrays of size $T_u \times T_v$ where $T_u, T_v \in \mathbb{N}$ denote the number of elements in each row respectively column and $T_u, T_v > 0$. Their elements are stored in linear computer memory and are accessed in the same way as the elements of DSM layers (see section 2.1.1). In contrast to the prevalent convention in the field of image processing, we assume the first row of an image (resp. texture) to be located at the bottom of the image (resp. texture) rectangle instead of at its top, so that the first pixel (resp. texel) is located at the bottom left. In this way, we can interpret the data from any DSM which is given in the reduced layered array representation as described in section 2.1.1 as a set of textures and the DSM samples of each layer can be identified with texels. Layers which hold DEM data are interpreted as F32 textures (DEM textures), and layers with color information as R8G8B8 or R8G8B8A8 color textures. If necessary, the data types of DSM layers are converted in order to match the required texel format.

We can summarize the differences between images and textures according to our definitions as follows: if a pixel format is compatible with a texel format, or if it can be converted accordingly, we can take any image for a texture. A texture, however, cannot be taken for an image in general, because even if a texel format is a valid pixel format, texels may not be associated with real-world objects, in contrast to the pixels of an image. This is a common case in GPGPU computing, for instance, but it already appears somewhat odd to us if we interpret a DEM texture as a grayscale image, because the content of the image may not be comprehensible without any further explanation.
tion is less important for our methods than the fact that the images look like aerial images, but we intend to employ primarily aerial images captured by mUAVs.

This section gives a classification of aerial images and outlines the differences in image acquisition by means of mUAVs and airplanes. According to our interpretation of images, the terms *photograph* and *image* are used interchangeably in this section (cf. section [2.2]).

### 2.3.1 Classification of Aerial Images

According to Paine and Kiser [60, pp. 28], we use the attribute *aerial* in contrast to *terrestrial* to refer to all kinds of images that are captured from an arbitrary platform in the air, including scaffolds or cranes, for example, and classify them likewise into *vertical* and *oblique* images.

The ground plane is the virtual plane which is oriented in such a way that the normal \( \vec{n}_g \) of this plane has the opposite direction as the vector of gravity which points towards the Earth’s center. Let \( \vec{n}_I \) denote the normal of the image plane of an aerial image \( I \) in world space as illustrated in figure 2.3. \( \varphi = \angle (\vec{n}_I, -\vec{n}_g) \in \left[ -\frac{\pi}{2}, \frac{\pi}{2} \right] \) denotes the angle enclosed between \( \vec{n}_I \) and \( -\vec{n}_g \). According to [60], an aerial photograph is called *vertical* if \( 0 \leq |\varphi| < \varphi_{\text{max}} \), and *oblique* if \( \varphi_{\text{max}} \leq |\varphi| \leq \frac{\pi}{2} \) where \( \varphi_{\text{max}} = \frac{\pi}{60} = 3^\circ \). Paine and Kiser call a vertical aerial photograph *true vertical* if \( |\varphi| = 0 \), and *tilted* otherwise. Oblique photos are called *low oblique* if they do not depict the horizon, and *high oblique* if the horizon is visible.

![Figure 2.3: Aerial images are classified according to [60] as vertical aerial images (left) and oblique aerial images (right) by the angle \( \varphi \) enclosed between the normal of the image plane \( \vec{n}_I \) and the inverse normal of the ground plane \(-\vec{n}_g\).](image)

The two kinds of aerial images have different advantages over each other. It is easier, for instance, to measure horizontal distances and areas in vertical photographs, and they can be used as substitutes or enhancement for conventional schematic maps. Oblique images, on the other hand, provide a more natural perspective and depict structures perpendicular to the ground [60, pp. 29–30]. Because of their different properties, we use vertical aerial...
images as input data for generating DEM data and orthophoto textures, whereas oblique aerial images are exclusively used for texturing lateral surfaces in DSM renderings. These two aspects are the subjects of chapter 5 and chapter 6 respectively.

2.3.2 Aerial Image Acquisition

Aerial images are traditionally acquired as vertical photographs from airplanes. Airplanes are capable of keeping approximately constant heights above ground and following fixed paths. This way, a certain overlap between the vertical photographs can be guaranteed, which is important for reconstructing 3D information as described in section 2.5. The camera systems which are employed in traditional aerial photography are usually equipped with sophisticated optical lens systems and mountings in order to capture high resolution images. These systems weigh several tens of kilograms (e.g., [89]), but can be easily carried by airplanes. The acquired photographs are preferably stored in uncompressed data formats on hard disk drives or solid state disks and cannot be accessed until the plane has landed.

Vertical and oblique aerial photographs can also be captured by UAVs [39]. While the flight characteristics and capabilities of acquiring aerial photographs of larger UAVs are similar to those of conventional airplanes, mUAVs are more maneuverable, but may not be able to carry heavy camera systems as payload [82, 84]. Hence, the cameras mounted to mUAVs may provide images at lower resolutions, but this can be compensated by their potentially lower operational altitudes. With mUAVs it is also possible to capture several images from different perspectives, so that the same points of a surface are depicted in multiple images and stereo matching methods can produce better results, as explained in chapter 5. As smaller UAVs are more prone to drift and may additionally not be capable of following fixed paths at constant altitudes, the images will likely be captured in a less organized manner than in case of traditional aerial photography.

For the purpose of recovering 3D information from photographs, it is essential to have the camera’s accurate position and orientation in space. Position and orientation data of airplanes and UAVs are usually available by Global Positioning System (GPS) and inertial measurement units (IMUs) and can be linked to the images. Mapping camera systems borne by airplanes are in fact designed for using data from IMU and GPS devices [89], [42, pp. 150–161].

2.3.3 The AVIGLE Project

The Avionic Digital Service Platform (AVIGLE) project [64] is a cooperation of seven partners from the high tech industry and three universities and is funded by the ministry of Innovation, Science and Research of the State of North Rhine-Westphalia, Germany, and the European Union. The goal of the project is amongst others to develop a multi-functional mUAV for civil purposes with a total weight of approximately 10 kg. By design, the aircraft is capable of horizontal and vertical flight, as well as hovering in place, and should operate in heights of up to 300 meters above ground and for about 60 minutes. The payload of $\approx 1$ kg weight is intended to be either equipment for providing cellular network communications, or a calibrated digital still image camera for capturing high-resolution digital...
photographs. Aerial images together with orientation data from the GPS and IMU devices of the mUAV can be transmitted directly to the operators via wireless network technology. This allows to create DSMs for rendering in a virtual 3D environment while the image acquisition is still in progress. Thus, the operators can be immediately provided with a recent overview of the surveyed area in order to direct the aircraft to locations of special interest.

As the development of the aircraft is conducted in parallel to our methods for creating and rendering DSMs during an ongoing image acquisition, and has not finished at the time of writing this dissertation, efforts have been made to simulate the acquisition of aerial images by mUAVs [75]. The simulation of aerial image acquisition basically comprises the preferably photorealistic rendering of 3D models of cities or landscapes, and the provision of individual frames together with their metadata. These metadata are the position, orientation and some additional parameters of the virtual camera at the time of image acquisition. In case of the AVIGLE project, aerial images are accessed by means of a database management system which allows to separate the image acquisition process from our methods for rendering and generating DSMs.

### 2.4 Coordinate Systems

Within this thesis we only employ Cartesian coordinate systems in 2D and 3D, but with different orientations and labels for the axes.

Coordinates from real world space, e.g., locations where images were captured, are assumed to be precise and referenced to a 2D geographic coordinate system. The coordinates are assumed to be provided, for instance, by GPS with eventually applied corrections and accuracy enhancements. We prefer such coordinates in UTM format over WGS84 or similar, because UTM values are already given in a Cartesian coordinate system [60, p. 181]. Height values and altitudes are both assumed to be given relative to a plane of reference based in the xy-plane respectively east-north-plane at \( z = 0 \) of a 3D coordinate system as depicted in figure 2.4(a). The unit of measurement and scale of the three axes has to be the same, e.g., meter. For convenience, we express all real world coordinates in the virtual world coordinate system \( \Sigma \), which is shown in figure 2.4(b). The scale and the unit of measurement of \( \Sigma \) are the same as the ones of the real world coordinate system, and \( \Sigma \) is simply referred to as world coordinate system throughout this work.

For cameras which are used to capture real world images, as well as virtual cameras, we use the local 3D Cartesian coordinate system \( \Sigma' \). \( \Sigma' \) is shown in figure 2.4(c) the center of projection (CoP) of an image is located at its origin. The basis of \( \Sigma' \) is expressed in \( \Sigma \) by \( \vec{r} = (r_x, r_y, r_z) \) (right-vector), \( \vec{u} = (u_x, u_y, u_z) \) (up-vector) and \( \vec{v} = (v_x, v_y, v_z) \) (view-vector). In contrast to \( \Sigma \), \( \Sigma' \) has a negative orientation (left-handed coordinate system), i.e., the determinant

\[
\begin{vmatrix}
  r_x & u_x & v_x \\
  r_y & u_y & v_y \\
  r_z & u_z & v_z 
\end{vmatrix} = (\vec{r} \times \vec{u}) \cdot \vec{v} < 0
\]
is negative \[1\] p. 901. Due to its positive orientation, $\Sigma$ is called a \textit{right handed coordinate system}.

![Diagram of coordinate systems](image)

\textbf{Figure 2.4: Illustrations of the different coordinate systems that are used throughout this dissertation. Height values and elevation in the real world coordinate system in (a) are stored at the $z$-component of coordinates and the ground plane is based in the $xy$-plane. In the (virtual) world coordinate system in (b), height values and elevation are stored at the $y$-component and the ground plane is based in the $xz$-plane. Except for $\Sigma'$ in (c), these coordinate systems are right-handed, i.e., their orientation is positive.}

## 2.5 Basic Principles of Photogrammetry

Photogrammetry is the science of determining the shape, position and orientation of objects in 3D space from 2D images \[42\] p. 1. In an image, the objects in 3D space are mapped onto a 2D \textit{image plane} by geometric planar perspective projections (\textit{projections} in short). There are other kinds of projections, e.g., orthographic projections, but they are of less interest in the context of this thesis. The mathematical foundations and detailed information about these different types of projections can be found, for instance, in \[29\] pp. 229–283 and \[1\] pp. 89–97.

The depth information of an object is lost during projection and hence a single image is usually not sufficient to reconstruct the 3D positions of the depicted points of an object. But if the same points of an object are present in at least two different images, and if the parameters of the projections for each image are known, the depth information can be recovered. The principle works similar to human stereoscopic vision \[60\] pp. 44–67.

Under the assumption that optical effects related to lenses, e.g., fraction and distortions, are negligible in calibrated, high quality mapping cameras, these camera systems can be approximately modeled as pinhole cameras. The model of a pinhole camera may also be applied to consumer cameras, but the results are usually less accurate. Recent advances in image processing, however, allow to mitigate this problem by incorporating multiple images of the same objects and statistical methods. An example can be found, for instance, in \[72\] where images from many different types of cameras are employed for 3D reconstruction.

In section 2.5.1 a mathematical introduction into planar central perspective projections is presented and it is explained, how 3D information can be extracted from 2D images by
effectively reversing projections. The camera parameters which are required for this task are described in section 2.5.2 and section 2.5.3 summarizes the most important properties of images which are appropriate for recovering 3D information.

2.5.1 Perspective Projection

Central perspective projection emulates the effect of foreshortening so that objects that are more distant to the viewer appear smaller than those that are closer. The projection of a 3D object in a 2D image is obtained at the intersections of projection rays with the image plane. In the case of central perspective projection, the projection rays emerge from a single center of projection (CoP) and pass through each point of the depicted object as illustrated in figure 2.5 [29, p. 230].

Let the image plane be located at a distance \( f > 0 \) in front of the CoP. The CoP is located at the origin \( O' \) of the local coordinate system of (virtual) camera \( \Sigma' = (\vec{r}, \vec{u}, \vec{v}) \). \( \Sigma' \) is usually translated by \( \vec{t} = (t_x, t_y, t_z) \) and rotated relative to the world coordinate system \( \Sigma = (\vec{x}, \vec{y}, \vec{z}) \) as shown in figure 2.6(a). The orthonormal basis of \( \Sigma \) is given by

\[
\vec{x} = (1, 0, 0) \quad \vec{y} = (0, 1, 0) \quad \vec{z} = (0, 0, 1)
\]

and the orthonormal basis of \( \Sigma' \) is expressed in \( \Sigma \) by

\[
\vec{r} = (v_x, v_y, v_z) \quad \vec{u} = (u_x, u_y, u_z) \quad \vec{v} = (v_x, v_y, v_z)
\]

In neutral position, i.e., \( \vec{t} = (0, 0, 0) \) and with all three rotation angles equal to zero, \( \Sigma' \) is oriented in such a way, that \( \vec{r} = \vec{x}, \vec{u} = \vec{y} \) and \( \vec{v} = -\vec{z} \) as illustrated in figure 2.6(b).

In order to transform locations from \( \Sigma \) to \( \Sigma' \) and to project them onto image planes, 4D homogeneous coordinates can be used. Homogeneous coordinates allow to conveniently represent a sequence of geometric transformations, including translations and projections, as a single matrix [1, pp. 54, 905–906], [29, pp. 213–217]. The transformation of a point \( p = (x, y, z) \in \mathbb{R}^3 \) to \( \hat{p} \) in 4D homogeneous coordinates is achieved by

\[
(x, y, z) \mapsto (w \cdot x, w \cdot y, w \cdot z, w)
\]

where \( w \neq 0 \). The additional fourth component is usually set to 1. A point \( \hat{p} = (\hat{x}, \hat{y}, \hat{z}, \hat{w}) \), \( \hat{w} \neq 0 \) in 4D homogeneous coordinates is transformed to 3D Cartesian coordinates by
2.5 Basic Principles of Photogrammetry

Figure 2.6: Illustration of the orientation of the local camera coordinate system $\Sigma'$ within the world coordinate system $\Sigma$ in general position (left) and in neutral position (right).

dividing each component of $\hat{p}$ by $\hat{w}$ and by dropping the fourth component:

$$(\hat{x}, \hat{y}, \hat{z}) \mapsto \left( \frac{\hat{x}}{\hat{w}}, \frac{\hat{y}}{\hat{w}}, \frac{\hat{z}}{\hat{w}} \right)$$

The division of $\hat{p}$ by $\hat{w}$ is called homogenization.

A point $p = (x, y, z)$ can thus be transformed from $\Sigma$ to $q = (q_r, q_u, q_v)$ in $\Sigma'$ by multiplying its representation $\hat{p}$ in homogeneous coordinates with a regular $4 \times 4$ matrix $V$, followed by homogenization in order to obtain Cartesian coordinates, as expressed in equation (2.1).

$$\hat{q} = \begin{pmatrix} \hat{q}_r \\ \hat{q}_u \\ \hat{q}_v \\ \hat{q}_w \end{pmatrix} = V \cdot \hat{p} = V \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \Rightarrow \quad q = \begin{pmatrix} \hat{q}_r \\ \hat{q}_u \\ \hat{q}_v \\ \hat{q}_w \end{pmatrix}$$ (2.1)

Matrix $V$, which is given in equation (2.2), is composed of a translation $T$, a rotation $R$ which performs a rigid body transformation of the coordinate axes, and a mirroring operation $-Z$ along the z-axis in $\Sigma$.

$$V = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad , \quad \begin{pmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad , \quad \begin{pmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(2.2)
In order to obtain the projection $p'$ of $q$ in the image plane at $v = f > 0$ in $\Sigma'$, the projection matrix $P$ from equation (2.3) is applied to $\hat{q}$.

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{pmatrix} \quad (2.3)$$

$p'$ and $q$ must satisfy the relations in equations (2.4) which result from applying the intercept theorem to the two pairs of similar triangles $\triangle O',q,q_r$ and $\triangle O',p',p'_r$ respectively $\triangle O',q,u$ and $\triangle O',p',p'_u$ as illustrated in figure 2.7.

$$q_v = \frac{p'_v}{p'_r} \Rightarrow p'_r = \frac{f}{q_v} \cdot q_r$$

$$q_u = \frac{p'_u}{p'_r} \Rightarrow p'_u = \frac{f}{q_v} \cdot q_u$$

Equation (2.5) confirms that applying $P$ to $q$ in homogeneous coordinates and transforming back to Cartesian coordinates yields $p' = (p'_r, p'_u, p'_v)$ as desired:

$$\begin{pmatrix} \hat{p}'_r \\ \hat{p}'_u \\ \hat{p}'_v \\ \hat{p}'_w \end{pmatrix} = P \cdot \begin{pmatrix} q_r \\ q_u \\ q_v \\ 1 \end{pmatrix} = \begin{pmatrix} q_r \\ q_u \\ q_v \\ f \end{pmatrix}$$

$$\Rightarrow p' = \begin{pmatrix} \hat{p}'_r \\ \hat{p}'_u \\ \hat{p}'_v \\ \hat{p}'_w \end{pmatrix} = \begin{pmatrix} f q_r \\ f q_u \\ f q_v \end{pmatrix}$$

The transformation matrix $V$ and the projection matrix $P$ are usually combined into a single Matrix $M$, so that the projection of $p$ in $\Sigma$ to $p'$ in the image plane at $v = f$ in $\Sigma'$ can be
Since projection is not injective, $P$ where expressed as in equations (2.6).

*\[ \hat{p}' = M \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} r_x p_x + r_y p_y + r_z p_z - \tilde{e} \cdot \tilde{r} \\ u_x p_x + u_y p_y + u_z p_z - \tilde{t} \cdot \tilde{u} \\ -(v_x p_x + v_y p_y + v_z p_z) + \tilde{f} \cdot \tilde{v} \end{pmatrix} \]

*\[ \Rightarrow \hat{p}' = \begin{pmatrix} p'_x \\ p'_y \\ p'_z \end{pmatrix} = \begin{pmatrix} -r_x p_x + u_y p_y + u_z p_z - \tilde{e} \cdot \tilde{r} \\ v_x p_x + v_y p_y + v_z p_z - \tilde{t} \cdot \tilde{u} \\ -v_x p_x + v_y p_y + v_z p_z - \tilde{f} \cdot \tilde{v} \end{pmatrix} \]

*\[ = f \begin{pmatrix} r_x (p_x - t_x) + r_y (p_y - t_y) + r_z (p_z - t_z) \\ -v_x (p_x - t_x) + v_y (p_y - t_y) + v_z (p_z - t_z) \\ -u_x (p_x - t_x) + u_y (p_y - t_y) + u_z (p_z - t_z) \end{pmatrix} \]

where

*\[ M = P \cdot V = \begin{pmatrix} r_x & r_y & r_z & -\tilde{e} \cdot \tilde{r} \\ u_x & u_y & u_z & -\tilde{t} \cdot \tilde{u} \\ -v_x & -v_y & -v_z & \tilde{f} \cdot \tilde{v} \\ 0 & 0 & 0 & 1 \end{pmatrix} \]

Since projection is not injective, $P$ and hence $M$ are both non-invertible. However, the $x$- and $y$-coordinates of $p$ can be expressed in dependency on $z$ as shown in equations (2.7).

*\[ \hat{p} = V^{-1} \cdot \begin{pmatrix} q_x \\ q_y \\ q_z \\ 1 \end{pmatrix} = \begin{pmatrix} r_x & u_x & 0 & t_x \\ r_y & u_y & 0 & t_y \\ r_z & u_z & 0 & t_z \end{pmatrix} \cdot \begin{pmatrix} q'_x \\ q'_y \\ q'_z \\ 1 \end{pmatrix} \]

*\[ = V^{-1} \begin{pmatrix} \frac{r_x q_x}{q_x} + \frac{u_x q_x}{q_x} - v_x q_x + t_x \\ \frac{r_y q_y}{q_y} + \frac{u_y q_y}{q_y} - v_y q_y + t_y \\ \frac{r_z q_z}{q_z} + \frac{u_z q_z}{q_z} - v_z q_z + t_z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \]

Equations (2.7) can be written as

*\[ x - t_x = q_x \left( \frac{p'_x}{f} + \frac{p'_u}{f} - v_x \right) \]

*\[ y - t_y = q_y \left( \frac{p'_x}{f} + \frac{p'_u}{f} - v_y \right) \]

*\[ z - t_z = q_z \left( \frac{p'_x}{f} + \frac{p'_u}{f} - v_z \right) \]
and dividing the first two equations by the third one results in
\[
\begin{align*}
\frac{x - t_x}{z - t_z} &= \frac{r_x p'_r + u_x p'_u - v_x f}{r_z p'_r + u_z p'_u - v_z f} \\
y - t_y &= \frac{r_y p'_r + u_y p'_u - v_y f}{r_z p'_r + u_z p'_u - v_z f}
\end{align*}
\]
Therefore
\[
\begin{align*}
x &= t_x + (z - t_z) \frac{r_x p'_r + u_x p'_u - v_x f}{r_z p'_r + u_z p'_u - v_z f} \\
y &= t_y + (z - t_z) \frac{r_y p'_r + u_y p'_u - v_y f}{r_z p'_r + u_z p'_u - v_z f}
\end{align*}
\]
(2.8)

From equations (2.8) follows that at least two images are needed in order to reconstruct the 3D position of \( p \) from its projections \( p' \), since the equations depend on the unknown coordinate \( z \). Another possibility to determine \( z \) is to incorporate ground control points with known positions, but their registration is an elaborate task (cf. [42, p. 19]).

An image which contains the projection \( p' \) of \( p \) is usually a rectangular section in a plane. By knowing its size, i.e., the width \( w \) and the height \( h \) of the image in \( \Sigma' \), the perspective transform matrix \( P_{\text{image}} \) from equation (2.9) can be used instead of \( P \) from equation (2.3).

\[
P_{\text{image}} = \begin{pmatrix}
\frac{2f}{w} & 0 & 0 & 0 \\
0 & \frac{2f}{h} & 0 & 0 \\
0 & 0 & 1 & -f \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
(2.9)

This matrix is frequently used in computer graphics for transformations of a symmetric, perspective view frustum into a cube with minimum corner \((-1, -1, 0)\) and maximum corner \((1, 1, 1)\), where the far plane \( f_{\text{far}} \) of the frustum is set to infinity [1, pp. 92–97, 345]. By using the identities from equations (2.10), which are illustrated in figure 2.8, \( P_{\text{image}} \) can be written as

\[
P_{\text{image}} = \begin{pmatrix}
\frac{1}{a \tan \left( \frac{\phi_{\text{hor}}}{2} \right)} & 0 & 0 & 0 \\
0 & \frac{1}{\tan \left( \frac{\phi_{\text{hor}}}{2} \right)} & 0 & 0 \\
0 & 0 & 1 & -f \\
0 & 0 & 0 & 1
\end{pmatrix} = \begin{pmatrix}
\frac{1}{\tan \left( \frac{\phi_{\text{vert}}}{2} \right)} & 0 & 0 & 0 \\
0 & \frac{a}{\tan \left( \frac{\phi_{\text{vert}}}{2} \right)} & 0 & 0 \\
0 & 0 & 1 & -f \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
a = \frac{w}{h}, \quad h = 2f \tan \left( \frac{\phi_{\text{vert}}}{2} \right), \quad w = 2f \tan \left( \frac{\phi_{\text{hor}}}{2} \right)
\]
(2.10)

\( a \) denotes the aspect ratio of the image, \( f > 0 \) is the distance from the CoP to the center of the image located at \( \left( \frac{w}{2}, \frac{h}{2} \right) > 0 \) and \( \phi_{\text{hor}} \) and \( \phi_{\text{vert}} \) are the horizontal respectively vertical angles of view.
The advantage of $P_{\text{image}}$ over $P$ is that $P_{\text{image}}$ is invertible, and its inverse $P_{\text{image}}^{-1}$ is given in equation (2.11).

\[
P_{\text{image}}^{-1} = \begin{bmatrix}
\frac{w}{2f} & 0 & 0 & 0 \\
0 & \frac{h}{2f} & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & -\frac{1}{f} & \frac{1}{f}
\end{bmatrix} = \begin{bmatrix}
0 & \tan\left(\frac{\phi_{\text{vert}}}{2}\right) & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & -\frac{1}{f} & \frac{1}{f}
\end{bmatrix}
\]

(2.11)

We can therefore model the imaging process by using $M_{\text{image}} = P_{\text{image}} \cdot V$ and directly obtain the $x$- and $y$-coordinates of a pixel in world space from $p' = (p'_r, p'_u, p'_v)$ by equation (2.12) in a more convenient way.

\[
M_{\text{image}}^{-1} \cdot \hat{p}' = V^{-1} \cdot P_{\text{image}}^{-1} \cdot \hat{p}'
\]

(2.12)

Note that $p'_v$ is in general unknown and not present in a single image, so that $x$ and $y$ will still depend on $z$ as shown in equations (2.13), which are obtained analog to equations (2.8).

\[
\begin{align*}
x &= t_x + (z - t_z) \frac{w_r p'_r + h_u p'_u - 2v_z f}{w_r p'_r + h_u p'_u - 2v_z f} \\
y &= t_y + (z - t_z) \frac{w_r p'_r + h_u p'_u - 2v_z f}{w_r p'_r + h_u p'_u - 2v_z f}
\end{align*}
\]

(2.13)
2.5.2 Camera Parameters

In section 2.5.1 the CoP was assumed to be located at the origin $O'$ of the camera’s local coordinate system $\Sigma'$. Usually the CoP has an offset $(\alpha'_r, \alpha'_u)$ relative to $O'$ and hence equations (2.8) have to be modified as shown in equations (2.14).

\[
\begin{align*}
    x &= t_x + (z - t_z) \frac{r_x(p'_r - \alpha'_r) + u_x(p'_u - \alpha'_u) - v_x f}{r_z(p'_r - \alpha'_r) + u_z(p'_u - \alpha'_u) - v_z f} \\
    y &= t_y + (z - t_z) \frac{r_y(p'_r - \alpha'_r) + u_y(p'_u - \alpha'_u) - v_y f}{r_z(p'_r - \alpha'_r) + u_z(p'_u - \alpha'_u) - v_z f}
\end{align*}
\] (2.14)

The modification of equations (2.13) for incorporating $O'$ is analog. Manufacturers of high quality camera systems strive for the coincidence of $(\alpha'_r, \alpha'_u)$ with $O'$, i.e., $\alpha'_r = \alpha'_u = 0$ [42, p. 9]. In total, nine image-independent parameters are required for recovering 3D information from 2D images. The three intrinsic parameters $f, \alpha'_r$ and $\alpha'_u$ are usually known in advance from the calibration of the camera system or can be determined, e.g., if ground control points with known positions in world space are depicted in the images.

The remaining six parameters are the extrinsic parameters and comprise the position and orientation of the camera’s local coordinate system $\Sigma'$ relative to $\Sigma$, i.e., $\hat{t} = (t_x, t_y, t_z)$ and the three rotation angles. These parameters are frequently determined by GPS and IMU devices, but other methods can also be used, if the provided data are accurate. The precision of GPS, however, may be insufficient for obtaining precise positions, but in this case, methods like bundle adjustment, for instance, can be used to improve the results [42, pp. 306–307].

Additional parameters of interest for photography-like images are the focal length $\varphi$, the horizontal and vertical angle of view, which is also called field of view, and the resolution of the image in terms of pixels per unit in world coordinates. These parameters determine the size of an image and assist in locating positions in world space (see section 2.5.1). The focal length $\varphi$ is frequently stored in the image file created by a digital still image camera, e.g., if the image is stored in the Exif file format [11]. In case of calibrated cameras, the focal length is fixed and is hence known in advance.

Provided that the distance of the depicted objects is large compared to the focal length, the distance from the CoP to the image plane equals $\varphi$, if the image is sharp. Given the effective sensor size, i.e., the width $w_{\text{sensor}}$ and height $h_{\text{sensor}}$ of the CCD-sensor, $\varphi_{\text{hor}}$ and $\varphi_{\text{vert}}$ can then be determined by means of equations (2.15).

\[
\begin{align*}
    \varphi_{\text{hor}} &= 2 \arctan \left( \frac{w_{\text{sensor}}}{2 \varphi} \right) \\
    \varphi_{\text{vert}} &= 2 \arctan \left( \frac{h_{\text{sensor}}}{2 \varphi} \right)
\end{align*}
\] (2.15)

The underlying geometry is the same as in figure 2.8 where $f = \varphi, w = w_{\text{sensor}}$ and $h = h_{\text{sensor}}$, and these quantities are usually given in millimeters.

---

4With analog images, the sensor size corresponds to the size of the film.
In the context of this work, it is assumed that the employed images are sharp and that
the sensor sizes are known. \( \alpha'_r \) and \( \alpha'_u \) are assumed to be negligible, i.e., \( (\alpha'_r, \alpha'_u) \approx 0' = (0, 0) \), so that the projection onto the image plane can be considered symmetric around its
geometric center. Furthermore, the distances to the depicted objects are assumed to be much
larger than \( \varrho \) and hence \( f = \varrho \) in the corresponding equations in section 2.5.1.

### 2.5.3 Image Properties

The images which are used within this thesis for generating DSMs are vertical aerial im-
ages. These images are captured in traditional aerial photography on a fixed flight path such
that they are placed on parallel stripes as sketched in figure 2.9. In order to recover 3D
information, the images need to overlap each other along the stripes in direction of flight by
at least 50%. Since deviations from the flight path may occur and need to be compensated,
overlaps of at least 60% along the direction of flight and 25% – 30% between two adjacent
stripes are recommended [42, pp. 145–148]. According to [44], overlaps of aerial images
in cities and woodland areas should even be 80% – 90% along the direction of flight and
60% – 70% between the stripes in order to reduce occlusions which would complicate the
recovery of 3D information. However, such fixed flight paths may be difficult to follow for
small UAVs and the images are likely to have a less organized layout.

![Figure 2.9](image_url)

**Figure 2.9:** In traditional aerial photography, the images must overlap and their centers are located on
stripes along the direction of flight.

In contrast to real photographs, computer generated images do not suffer from the prob-
lem of inaccuracies like lens effects or imprecise position or orientation data. These data
are well-suited for developing and testing algorithms, because many sources of potential
errors, e.g., image noise or color defects, are eliminated. On the other hand, the data are
less realistic and thus can influence the algorithms as well, e.g., like with finding matching
feature points in images, if the underlying 3D models of cities or buildings employ repeti-
tive textures. Frequently, only a relatively small amount of artificial textures is used many
times on different 3D models in order to reduce the required amount of memory. While
these repetitions may not be noticed by the viewer, they can cause algorithms to detect false
positive matches of points in different images, although they belong to different objects.

For the remainder of this thesis the additional metadata in table 2.2 of any aerial image
are assumed to be available. As the accurate measuring of position and orientation of real
photographs is beyond the scope of this dissertation, it is furthermore assumed that these data have already been corrected by appropriate methods, like bundle adjustment, known ground control points or other existing information, if necessary, and that they are precise.

<table>
<thead>
<tr>
<th>metadata</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>a unique identifier for each image</td>
</tr>
<tr>
<td>position</td>
<td>position of center of projection given in world coordinates at the moment the image was captured</td>
</tr>
<tr>
<td>orientation</td>
<td>three rotation angles (yaw, pitch and roll) of the local camera coordinate system relative to the world coordinate system at the moment the image was captured</td>
</tr>
<tr>
<td>focal length</td>
<td>focal length of the camera in millimeters</td>
</tr>
<tr>
<td>sensor size</td>
<td>effective width and height of the camera’s CCD sensor in millimeters</td>
</tr>
<tr>
<td>resolution</td>
<td>width and height of the image in terms of pixels</td>
</tr>
<tr>
<td>timestamp</td>
<td>timestamp of the moment when the image was captured</td>
</tr>
</tbody>
</table>

Table 2.2: The methods presented in this thesis assume that these metadata are available for each (aerial) image, and that the values are precise.
In many applications that involve DSM rendering, texture data may have sizes in the gigabyte range and thus often exceed the available physical video memory. For this reason, different techniques for handling very large textures have been developed. However, in many of these applications textures are frequently treated as resources of fixed extension and static or rarely changing content.

Our application relies on texture data of frequently varying content and of time-varying extension, i.e., the size of the textures may change in the course of time. New aerial images are directly processed for generating a single virtually contiguous DEM and matching color textures which can be rendered in a virtual 3D environment. Although the covered areas may be vast and the images may have high resolution, it is desirable to update the content of such a very large virtual texture in real time. Furthermore, any newly acquired image may not only update existing content, but also extend the area covered so far as illustrated in figure 3.1. Related techniques for handling large textures, like common clipmap implementations, are not able to handle such growing textures.

Another issue that arises with textures during rendering is the problem of aliasing. By using perspective projection, a single screen pixel may cover more than one texel of a texture if it is applied to objects distant from the viewer, especially when the object is viewed from an oblique angle (see figure 3.2). The prevalent method for mitigating aliasing is to use pre-filtered versions of the textures, like mipmaps [85]. Special texture sampling and interpolation techniques like trilinear or anisotropic filtering are available in order to conceal transitions between the different levels of detail (LODs). Usually these anti-aliasing techniques are supported by graphics hardware and should be utilized by methods for handling large virtual textures for efficiency reasons.

![Figure 3.1: Not only the content, but also the extension is altered in the course of time with textures of time-varying extension.](image)
3 The Flexible Clipmap

Figure 3.2: Aliasing becomes apparent on distant textured objects when one screen pixel covers more than one texel. Different anti-aliasing techniques (center and right) are available for reducing this effect.

In this chapter we present the Flexible Clipmap (FCM) to incrementally generate large, virtual textures of time-varying extension and dynamically changing content from multiple input aerial images on-the-fly. Parts of the work on this topic have been previously published in [28]. The resulting virtual textures may contain arbitrary data, but for our purposes we confine their contents to DEM and color texture data. The FCM makes use of a tile-based clipmap approach, a spatial indexing data structure and commodity graphics hardware. Its application does not depend on the underlying geometry or a certain kind of polygonal meshes.

3.1 Related Work

If the size of textures to be displayed exceeds the hardware limits, a common and obvious solution is to divide them into smaller, manageable texture tiles. For instance, Cline and Egbert proposed a simple division of the texture data, but their approach was limited by a strict dependency on texture coordinates and underlying geometry [13].

A clipmap, as introduced in [79], is based on mipmaps [85] and keeps only portions of the entire texture mipmap in video memory. Like mipmaps, the clipmap provides a level of detail (LOD) concept and thus avoids texture aliasing. The lowest level \( l = 0 \) denotes the level of highest resolution and largest extension of the texture. The clipmap uses a roaming window in video memory to copy only those texels which are visible to the viewer by centering the window around the current view point (clip center). As the view point moves, the window is updated by copying new texels using toroidal addressing [79]. This is done for each clipmap level corresponding to a texture of a size greater than a certain clip size. Since the extensions of textures of mipmaps and clipmaps become smaller towards higher levels, these parts of a clipmap are small enough to fit entirely into video memory and hence do not need to be clipped. Therefore, the texture data at higher clipmap levels are treated as an ordinary mipmap. However, the original clipmap implementation in [79]...
3.2 The Flexible Clipmap

The Flexible Clipmap (FCM) relies on special graphics hardware for loading the texels into video memory. Details about the clipmap are explained in section 3.2.2.

Modern GPU features have eliminated the need for special hardware, and the clipmap concept has been implemented on the GPU in vertex and fragment programs (shaders) on commodity hardware making it even more attractive for handling large textures. The virtual texture in [23] makes use of texture tiles and can employ shaders for texture mapping. It supports multi-texturing, but does not use toroidal addressing for loading tiles, the texture coordinates are still coupled to the geometry and it requires multiple texturing units, even if only a single texture is mapped.

In [70] a roaming tile cache for each level is used, which is updated using toroidal addressing and texture stacks. Geometry and texture data are kept independent from each other by computing the texture coordinates within a shader. The approaches in [23] and [70], as well as the solution presented in [45] generate complete mipmaps for each tile at each LOD.

Crawfis et al. [16] also employ roaming tile caches for each level, toroidal addressing, texture compression and fragment programs, but they do not generate mipmaps for the tiles. They investigate different methods for clipping and level determination by utilizing fragment programs and arrays of textures to hold the relevant portions of the logical mipmap. Furthermore, they make use of a tile map to indicate to the shader the highest available texture resolution, i.e., the lowest LOD, for selecting the optimal clip level. The tile map is implemented by a separate texture and requires additional resources. In addition, the authors propose the usage of more efficient texture arrays, which became available in DirectX 10 hardware [49].

More related work on using large virtual textures on modern GPUs can be found in [5] and [50].

Although the previously mentioned research on texture clipmaps deals with large textures of fixed extension, the proposed approaches do not seem to be capable to update the texture content at a frequent rate. In [78] a method for handling large fixed-size textures of frequently changing content is presented. However, to our knowledge none of the current techniques is able to handle textures of time-varying extension.

3.2 The Flexible Clipmap

The FCM allows to derive from a growing set of images captured at different locations a single virtual texture which can be rendered immediately on any geometric model. It has been developed primarily for being used with aerial images that become directly available after their acquisition, e.g., by transmission over wireless networks as described in chapter 1. For this kind of images, the metadata as listed in table 2.2 must be provided. Though the FCM is not limited to a certain type of input images or to a certain kind of surface onto which the resulting virtual texture can be applied, we will assume for the remainder of this chapter that images are aerial images and that the accompanying metadata are exact. Precise position data for input images can be obtained in a preprocessing step which comprises the adjustment of positions, georeferencing and registration, as well as coordi-
nate transformations from GPS to UTM coordinates for the reasons given in section 2.4. The resulting virtual texture can be viewed as a rectangular section located in a plane like a common texture. Its texels are addressed via normalized texture coordinates in the range $[0, 1] \times [0, 1] \subset \mathbb{R}^2$, independent of the current extension of the FCM.

In addition, corrections of perspective distortion have to be performed, e.g., in order to obtain true orthographic images. Such images depict surfaces from an orthographic point of view and intentionally do not provide any information about (lateral) surfaces which are perpendicular or near-perpendicular to the plane against which the elevation is measured, e.g., frontages of buildings in urban areas or canyon walls. Orthographic images, or orthoimages in short, can only be generated if information about the elevation of the underlying surface is available, i.e., if a DEM is given. The latter is generated from the images themselves and hence DEM and orthoimage generation are combined into one logical step which is subject of chapter 5. As a result of the image transformation and warping, the area covered by an input image on the ground plane does not need to be rectangular anymore, but is an arbitrary convex quadrilateral. A 2D minimal enclosing axis-aligned bounding box (AABB) of the covered area can be computed efficiently.

### 3.2.1 Requirements for Handling Spatially Time-Variant Textures

In order to handle textures of time-varying extension, the FCM cannot be confined in advance to a fixed area. Also it may easily become too large to fit entirely into video memory or even main memory, and therefore it has to be partitioned, stored in secondary memory and provided with a LOD concept to avoid texture aliasing. Furthermore, for a frequently changing set of images overlapping the same area, there are different possibilities to decide which of the images (or parts of the images) are the “best” to contribute to the resulting virtual texture. A rating can be based, for instance, on image properties such as timestamp, contrast, brightness, signal-to-noise ratio, degree of perspective distortion, or even content. In any case, whenever “better” images are available, the affected regions within the virtual texture have to be updated. To summarize, the FCM must satisfy the following requirements to address the challenges stated above:

1. Partitioning must be supported, since the virtual texture may have a physical extension which exceeds hardware limits.
2. An LOD concept is required in order to avoid texture aliasing.
3. Efficient updates of the virtual texture must be possible, because updates of the underlying set of images will occur frequently.
4. The virtual texture must have time-varying extension, i.e., it cannot be limited to a fixed extension.

As the FCM may have to be updated during rendering, special attention has to be paid to ensure that the processes for rendering and updating do not excessively interfere with each other as explained in section 7.2.1.
3.2.2 The Clipmap

The first two requirements given in section 3.2.1 can be achieved with texture clipmaps, since they allow to control memory consumption and provide LOD concepts. The structure of a clipmap as presented by Tanner et al. [79] is similar to the one of a mipmap and is illustrated in figure 3.3. At the lowest clip level \( l = 0 \) the clipmap contains texture data at the highest available resolution (cf. [79]). Like with mipmaps, going from clip level \( l \) to clip level \( l + 1 \) the resolution of the texture decreases by a factor of two along each dimension so that each \( 2 \times 2 \) neighboring texels are aggregated into one texel. The aggregation scheme depends on the texture data, but a frequently used method for color textures is to calculate the averaged intensities per color component for the resulting texel. At the highest clip level \( L - 1 \), the clipmap contains a down-sampled version of the entire texture. During texture mapping the minification of a texel in screen space determines the clip level to be used. But in contrast to a mipmap, for each level \( l < L - 1 \) of a clipmap only a section of the same size of the entire logical texture is kept in video memory. This section is called active area, and it is centered around the clip center. The clip area is a somewhat larger section centered around the clip center which contains the active area and is kept in main memory for updating the active area in video memory as the clip center is moved. Any remaining texture data are stored in secondary memory, e.g., on hard disk, and are loaded when the clip area needs to be updated. Texture data at levels \( l \geq L - 1 \) can be treated as a standard mipmap, because they are small enough to fit entirely into video memory.

![Figure 3.3: Structure of a clipmap with \( L = 4 \) clip levels and additional three mipmap levels. The active area (dark gray) and the clip area (light gray) are centered around the clip center (yellow) and are updated as the view point moves.](image)

The position of the clip center has to be chosen carefully and may be set to the intersection of a ray cast from the position of the virtual camera with the textured surface. This method works well in situations where the angle enclosed between the opposite viewing direction and the normal vector of the textured surface is small. But if this angle is large, the clip center is too far away from the viewer and other heuristics may be more appropriate.

The original clipmap by Tanner et al. relies on special graphics hardware for copying individual texels from the clip area to the border region of the active area [79]. Our FCM implementation is based on a variation of the clipmap similar to the one by Crawfis et al. [16].
which makes use of fragment programs and texture tiles without requiring any special hardware. With this approach, entire texture tiles, or tiles, in short, are swapped between the active areas and the clip areas instead of replacing individual texels on the border of the roaming window. The principle of a tile-based clipmap is illustrated in figure 3.4. We also adopted the idea by Crawfis et al. of employing a tile map and making use of explicit index-based clipping [16], but in contrast, we use a special scheme for indexing and generating the required texture tiles. Due to the tiling approach, updates of the virtual texture may be confined to few tiles only, which is helpful for achieving the third requirement stated in section 3.2.1.

![Figure 3.4: Illustration of tile-based clipmap. The active area consists of at most 3 × 3 texture tiles (dark gray) and the clip area of 5 × 5 tiles of the same size.](image)

### 3.2.3 Managing Aerial Images by Spatial Indexes

To satisfy the third requirement from section 3.2.1 we need to be able to efficiently retrieve the aerial images and associated metadata contributing to a certain tile, i.e., the images whose projections cover the area in world space which corresponds to the tile. This can be accomplished by using a spatial indexing data structure. If all aerial images project onto the plane in which the FCM tiles are based, e.g., if only vertical aerial images are employed for creating a virtual texture based in the ground plane, the images can be organized in a 2D spatial index based on their projections. Since our application makes use of aerial images of all kinds of orientations, e.g., in order to use them as projective textures as described in chapter 6, we store the images and their associated metadata in a 3D spatial index. Different types of aerial images can thus be handled in the same way. For an image we use the world coordinates of the 3D axis-aligned bounding box (AABB)\(^5\) of its view pyramid as key. To simplify matters, assume that the FCM is used to create a virtual texture of the ground plane, so that all vertical images project onto that plane.

The view pyramids of the images are determined by the parameters of the associated cameras as follows: In case of vertical aerial images, the top of the pyramid is located at the CoP and the base is formed by the projection of the image onto the ground plane (see figure 3.5a). Oblique aerial images, and in particular high oblique aerial images, do

\(^5\)Throughout this dissertation the term *axis-aligned bounding box*, or AABB in short, refers to the 3D *minimal enclosing* AABB of an object.
not project onto the ground plane if at least one of the four rays emanating from the CoP of the associated camera through one of the corners of the image does not intersect the ground plane. Even if all four rays intersect the ground plane, the resulting projection for this kind of aerial images can be rather distorted. To obtain an appropriate view pyramid for such images, we place a plane oriented perpendicular to the camera’s view direction at a predefined distance. The base of the pyramid is then formed by the intersections of the four rays emanating from the CoP with this plane. Figure 3.5 illustrates the view pyramids and AABBs as described above. The AABBs of view pyramids are represented by their minimum corner \( c_{\text{min}} = (x_{\text{min}}, y_{\text{min}}, z_{\text{min}}) \) and maximum corner \( c_{\text{max}} = (x_{\text{max}}, y_{\text{max}}, z_{\text{max}}) \), having coordinates given relative to an arbitrary but fixed origin \( O = (O_x, O_y, O_z) \). \( O_z \) is usually set to \( H_{\text{min}} = 0 \), and we assume that the height above ground of the center of projection for each image is part of the provided metadata.

Figure 3.5: In order to manage the source images that are used to generate the tiles of the FCM, the axis-aligned bounding boxes (gray) of the view pyramids of the associated cameras of the images (red) are stored in a spatial index.

Since the boxes serve as keys for storing aerial images, and since we have to determine the overlap with a 3D query box as described in section 3.2.5, the spatial index has to support intersection queries. Given a 3D box (query window), an intersection query on a spatial index returns all items whose keys (i.e., boxes) intersect the query window or are entirely contained inside. When using boxes as keys for associated data, this kind of key can be stored in a spatial index designed for point data, such as the kd-tree by Bentley [9], if the boxes are represented as points in a higher dimensional space. Because multiple representations for boxes in 3D exist, the choice depends on the data, and whether the embedding space is organized (trie-based methods [67, pp. 2–3]) or the data points are organized [37, pp. 24–26]. For example, a 3D AABB can be represented as a point in 6-dimensional space by the six coordinates of its minimum and maximum corner \( c_{\text{min}} \) and \( c_{\text{max}} \). Another possibility is to represent the box by its minimum corner and its extension in each direction \((x_{\text{min}}, y_{\text{min}}, z_{\text{min}}, s_x, s_y, s_z)\). In order to perform intersection queries with a given box \( Q \), \( Q \) has to be transformed according to the chosen representation, and the affected objects are located within a certain subspace of the higher dimensional space.
Other types of spatial indexes can directly employ spatially extended objects, like boxes, e. g., the R-tree by Guttman [35] and its variations. R-trees are height-balanced trees related to B-trees, and the data are stored only at the leave nodes (leaves). Except for the root node, each node has at least \( m \), but at most \( M > m \) entries where \( 1 < m < M \), \( m, M \in \mathbb{N} \). The root may contain less than \( m \), but also at most \( M \) entries. An AABB which encloses all children of the assigned subtree is stored at each node. In particular, the root holds an AABB of the entire domain which is covered by the data of the tree. This feature is used by the FCM for handling expansions of the virtual texture and for generating tiles as described in section 3.2.5. The structure of a two-dimensional R-tree is illustrated in figure 3.6. The entries of intermediate nodes are either pointers to other intermediate nodes at the next level or to leaf nodes. At leaf nodes, the entries point to the actual data and their AABBs. Like the leaves of B-trees, the leaf nodes of an R-Tree are split by certain criteria into two nodes if they have to store more than \( M \) entries (overflow). An entry for the resulting second node is inserted into the same parent node and may result in an overflow of the parent node as well which is handled in the same way. This process ends if either no further split of a node is necessary, or if the root has to be split. In the latter case, the root is replaced by a new one, and the former root together with its new sibling become the only children. Likewise, if nodes contain less than \( m \) children due to deletion of data, they are merged and deleted accordingly. After insert or delete operations, the bounding rectangles of the affected node and its ancestors usually need to be adjusted [35].

Compared to other spatial indexing structures, R-trees are appropriate for incremental construction by successive insertions [67, pp. 282–286] and some variations of the R-tree, e. g., the R* -tree by Beckmann et. al., do not require periodic global reorganization [6]. The latter aspect is especially desirable, because that data structure does not need to be completely rebuilt after successive insertion of a large number of items, which can degrade the performance of subsequent find operations. This can happen with a standard point kd-tree, for instance, where successive insertions may degenerate the tree structure and decrease its performance [67, p. 58].
3.2 The Flexible Clipmap

For our purposes, we store 3D AABBs in an $R^*$-tree by Beckmann et al. [6], which is a variation of the R-tree. The $R^*$-tree differs from the R-tree in the method for determining node splits, and it utilizes forced reinsertions of existing nodes. Reinserting existing nodes can avoid unnecessary node splits and improves the weight balance of the tree, i.e., it can increase the capacity utilization of the nodes [6]. Further information on the R-tree and the $R^*$-tree can be found in [35] and [6], respectively, and a revised version of the $R^*$-tree is presented in [7]. The wide-ranging topic of spatial indexes is covered in detail in the comprehensive book by Samet [67].

3.2.4 Layout Scheme for Tiles

Since the $R^*$-tree stores at each node an AABB of all elements assigned to the corresponding subtree, the 3D AABB containing the view pyramids of all aerial images is located and maintained at the root node. If only vertical aerial images were inserted into the spatial index, the 2D projection of this AABB onto the ground plane coincides with the rectangular area $A$ currently covered by the FCM. Due to the possible presence of oblique aerial images in the spatial index, which do not contribute to the content of FCM tiles, we keep track of the 2D area covered only by vertical or near-vertical aerial images. Therefore, we store and update an additional 2D bounding rectangle at each $R^*$-tree node. Whenever a vertical aerial image is inserted into the 3D spatial index, we use the 2D projection of its view pyramid’s AABB onto the xy-plane to update 2D AABBs at the $R^*$-tree nodes. Hence, the 2D AABB at the root node with minimum and maximum corners $R_{\min} = (R_{x_{\min}}, R_{y_{\min}})$ and $R_{\max} = (R_{x_{\max}}, R_{y_{\max}})$ coincides with the area $A$ covered by the FCM. The total number of clip levels $L$ depends on the extension of $A$, a previously chosen tile size of $T_x \times T_y$ texels, and the desired resolution $(\text{res}_x, \text{res}_y)$ in terms of texels per unit of the virtual texture. The tile resolution preferably matches the resolution of the aerial images so that the quality of the resulting virtual texture is not diminished by further re-sampling. In order to satisfy the fourth requirement from section 3.2.1, the key is to provide the FCM with additional tiles whenever the underlying virtual texture becomes larger. This is accomplished by the following layout scheme for the tiles.

Like with a common mipmap, the $L$ clip levels of the FCM are numbered starting with the most detailed level $l = 0$ to the least detailed level $l = L - 1$. The top-most level $L - 1$ contains a downsampled version of the entire texture and is the first level of the remaining mipmap section (see figure 3.3). Conceptually, we consider the virtual texture as the entire $\mathbb{R}^2$, and the region $\mathbb{R}^2 \setminus A$ is considered as empty, i.e., no texture data are available in this area.

Let $O_{xy} = (O_x, O_y)$ denote the projection of the origin of the world coordinate system onto the xy-plane at $H_{\min}$. $O_{xy}$ is used by the spatial index to reference the coordinates of the AABBs in world coordinates (see section 3.2.3). We span a Cartesian grid centered around $O_{xy}$ with spacing $(g_x, g_y) = (T_x / \text{res}_x, T_y / \text{res}_y)$, where $T_d$ denotes the size of a tile in texels, and $\text{res}_d$ its resolution in texels per unit along direction $d \in \{x, y\}$. Starting at the origin, in each quadrant, a cell of size $2^{l+1} \cdot g_x \times 2^{l+1} \cdot g_y$ at level $l + 1$ is formed.
by the underlying $2 \times 2$ neighboring grid cells from level $l$. We then proceed with the next level $l + 1$ in the same way in order to build a grid hierarchy, similar to the bottom-up construction of a quadtree. But in contrast to a quadtree, the grid hierarchy does not possess a root, because its domain is unbounded. Therefore, this process is repeated only until $A$ is covered in x- and y-direction by at most two grid cells each, but by at least one in one of the directions, i.e., until $A$ is covered by $1 \times 2$, $2 \times 1$ or $2 \times 2$ grid cells. Each grid cell at level $l$ is identified with exactly one tile at clip level $l$ and indexed as shown in figure 3.7. The at most four remaining cells covering $A$ are also grouped to a single cell at the top-most level $l = L - 1$, but this cell differs from the other ones in that it may contain the origin $O_{xy}$ and provides some sort of root for the grid hierarchy. In this way, the grid hierarchy limited to the area $A$ of the FCM is bounded, which allows to directly address grid cells respectively tiles at a given clip level $l$ via their indices. Direct addressing of tiles is essential for efficiently caching them and updating the active areas and clip areas of the FCM for the purpose of rendering. Furthermore, the FCM fragment program accesses individual tiles by their indices for sampling texture data.

![Figure 3.7](image)

**Figure 3.7:** The tiles are indexed at each level starting from the origin $(O_x, O_y)$. Each $2 \times 2$ tiles from level $l$ are grouped into one tile at level $l + 1$. At the top-most level (green), only one tile is left that covers the entire domain $A$.

The top-most grid cell respectively tile at level $L - 1$ requires some special attention, because as a constraint for the FCM, this level must cover $A$ entirely while consisting of exactly $2 \times 2$ tiles at the next lower level (children). In cases of having only $1 \times 2$ or $2 \times 1$ children, the missing ones are replaced by the nearest, possibly empty, neighbors in the corresponding direction, i.e., grid cells not covering $A$ may be used. Besides, the top-level tile is indexed $(0, 0)$, because it may be formed of tiles which are not considered as neighboring according to the scheme described above, e.g., if it was formed by the four tiles $(1, -1), (2, -1), (1, 0)$ and $(2, 0)$ at level $L - 2$. In this example, the corresponding parent tiles would be assigned the indices $(0, -1), (1, -1), (0, 0)$ and $(1, 0)$.
Given a certain level $0 \leq l < L$, we can determine the index $n(l, r) = (n_x, n_y)$ of the tile (tile index) containing a location $r = (r_x, r_y)$ given in world coordinates by equation (3.1).

$$n(l, r)_d = \begin{cases} \left\lfloor \frac{r_d}{2^l \cdot g_d} \right\rfloor & 0 \leq l < (L - 1) \\ 0 & l = (L - 1) \end{cases}, \quad d \in \{x, y\} \tag{3.1}$$

Since $L$ depends on the position of $R_{\text{min}}$ and $R_{\text{max}}$ and hence indirectly on the choice of the origin $O_{xy}$, the total number of clip levels $L$ cannot be determined directly by the quantity $\hat{L}$ from equation (3.2).

$$\hat{L} = \max_{d \in \{x, y\}} \left( \lceil \log_2(R_{\text{max}} - R_{\text{min}}) \rceil - \lceil \log_2(g_d) \rceil \right) + 1 \tag{3.2}$$

If $A$ would be covered by a common mipmap, $\hat{L} - 1$ would indicate the number of mipmap levels at which the corresponding textures would be too small to entirely cover $A$. Mipmap level $\hat{L} - 1$ would then be the first level that could completely cover $A$ by a texture of size $g_x \cdot \text{res}_x \times g_y \cdot \text{res}_y$. But according to the description above, $L - 2$ is the level $l'$ where the maximum of the differences of the two components of the tile indices $n(l', R_{\text{max}})$ and $n(l', R_{\text{min}})$ equals one as expressed in equation (3.3).

$$\max_{d \in \{x, y\}} (n(l', R_{\text{max}})_d - n(l', R_{\text{min}})_d) = \max_{d \in \{x, y\}} \left( \left\lfloor \frac{R_{\text{max}}}{2^{l'} \cdot g_d} \right\rfloor - \left\lfloor \frac{R_{\text{min}}}{2^{l'} \cdot g_d} \right\rfloor \right) \equiv 1 \tag{3.3}$$

Therefore the required number of clip levels $L$ is actually determined by checking if $l' = \hat{L}$ from equation (3.2) satisfies equation (3.3). If it does not, $l'$ is incremented and checked again, until a suitable $l'$ is found.

Most important about this indexing scheme is that the tile index $n(l, r)$ does not directly depend on the area $A$, i.e., the current area of the FCM. This facilitates adding new tiles to the FCM, because each tile at each level has a unique index and conceptually already exists, though it may not contain image information.

### 3.2.5 Adding and Updating Tiles

Assume that the FCM is used to create a virtual texture for the ground plane. In this case, only vertical or near-vertical aerial images affect the area $A$ covered by the FCM according to the 2D bounding boxes of their projections onto the ground plane, and we only consider this type of aerial images in the following.

Before any aerial images are inserted, $A$ is empty. The insertion of aerial images into the spatial index whose projections onto the ground plane are not completely covered by $A$ triggers an expansion of $A$ and thus of the FCM. This leads to the addition of at least one new clip level if $A$ exceeds the boundaries of the current top-most tile at $L - 1$ as illustrated in figure 3.8. The total number of new clip levels depends on the extension of $A$. After an expansion of $A$, the former tile at level $L - 1$ may no longer satisfy the constraint of covering at most $2 \times 2$ tiles from the next lower level, and the maximum clip level has to
be increased by the number $k$ of new levels, so we denote the former number of clip levels by $L'$ and set $L = L' + k$. However, it is actually not necessary to determine $k$ since $L$ can be recalculated from the new positions of $R_{\text{min}}$ and $R_{\text{max}}$ which are obtained from the expanded 2D AABB of all (near-)vertical aerial images at the root node of the $R^*$-tree as described in section 3.2.4. Except for the former top-level tile $T$ at $L' - 1$, the indices of all other tiles remain unchanged. Since $T$ is no longer the top-level tile, but a child of another tile, its former index $(0, 0)$ is according to equation (3.1) not necessarily correct any longer, and $T$ is therefore discarded. The former $2 \times 2$ children of $T$ may furthermore become assigned to different parent tiles at level $L' - 1$ during the addition of one or more clip levels as illustrated in figure 3.9.

Based on the clip level, two cases for the update and creation process of the content a tile have to be distinguished:
1. creating and updating tiles at the base clip level \( l = 0 \) directly from the aerial images

2. creating and updating tiles at clip levels \( l > 0 \) from their \( 2 \times 2 \) children

In the first case, a 3D intersection query on the spatial index is used to determine all aerial images whose projections overlap the area of a tile. We create a 3D query box, because 3D boxes are easier to handle by our 3D R*-tree implementation. The intersection query is triggered each time a certain number of aerial images has been inserted into the spatial index. The affected tiles are determined by computing the indices \( n_1, n_2, n_3 \) and \( n_4 \) at the four corners of the projection of an aerial image onto the ground plane in world space according to equation (3.1). Let

\[
\begin{align*}
    n_{\text{min}}^d &= \min_{i \in \{1,2,3,4\}} (n_i^d), & n_{\text{max}}^d &= \max_{i \in \{1,2,3,4\}} (n_i^d), & d \in \{x,y\}
\end{align*}
\]

Since an aerial image may project onto multiple tiles, all tiles whose indices are in the range \([n_{\text{min}}, n_{\text{max}}] \times [n_{\text{max}}, n_{\text{max}}]\) can potentially contain parts of the projected image and must be considered for updating. This is illustrated in figure 3.10. For each considered tile \( t \), the query box \( Q \) is constructed from the 2D bounding rectangle of \( t \) in the \( xy \)-plane, and the \( z \)-values of the box are set to \( H_{\text{min}} \) and \( H_{\text{max}} \). \( H_{\text{min}} \) denotes the minimum and \( H_{\text{max}} \) the maximum \( z \)-value of all bounding boxes of the aerial images currently stored in the R*-tree. Given the index \( n = (n_x, n_y) \) of \( t \) and the grid spacing \( g = (g_x, g_y) \) of the tile grid, the 2D bounding rectangle of \( t \) is computed by using equations (3.4).

\[
\begin{align*}
    q_{\text{min}}^d &= n_d \cdot g_d, & q_{\text{max}}^d &= (n_d + 1) \cdot g_d, & d \in \{x,y\}
\end{align*}
\]

By using the query box \( Q = (q_{\text{min}}, q_{\text{max}}, H_{\text{min}}, q_{\text{max}}, q_{\text{max}}, H_{\text{max}}) \), the result set can also contain non-vertical aerial images. These images can be identified during further processing by the viewing direction of the associated camera, for instance.

![Figure 3.10: An aerial image may project onto multiple tiles. The indices \( n_1, n_2, n_3 \) and \( n_4 \) of the tiles containing the four corners of the image’s projection (red) determine the range of tiles that potentially contain image data (gray shaded). For each of these tiles, a 3D query box (blue) is created in order to determine all images having projections that overlap the area of the tile.](image)

A very simple way to generate texture data for each tile \( t \) is to project the returned aerial images according to their orientation and camera parameters onto \( t \) and to replace existing
texels with values from the images. The problem with this method is that the pixels become projected onto the same plane in world space, although they probably originate from non-planar objects and from images captured from different perspectives. Therefore, the borders of overlapping images will not match each other and the results look similar as depicted in figure 3.11. The solution to this problem is to incorporate information about the surface elevation and to project the images according to the depicted objects in world space as described in section 5.4.

Figure 3.11: If the aerial images are all projected onto the same plane, their borders may not match each other as can be seen in the regions highlighted by red rectangles.

As the number of contributing images and hence the costs for updating one tile may increase over time, different selection criteria can be applied to discard images which are not supposed to contribute to the final texture, e.g., if the contrast of the image is below a certain threshold or if it is outdated. Depending on the desired quality of the resulting virtual texture and the application, it may be sufficient to update existing tiles incrementally by processing only the latest images and blending them with the previously generated tiles. The tiles need to be updated in principle as soon as they are covered by a new aerial image, but it may be sufficient to start updating affected tiles only if a certain threshold for the number of new images has been reached or after a certain period of time.

In the second case, tiles at levels \( l > 0 \) are recursively updated in increasing level order from their children as follows. For each tile \( t \) at level \( l \) with a given index \( (n(l)_x, n(l)_y) \), equation (3.5) denotes the index \( n(l+1) \) of the tile’s parent at level \( l + 1 \).

\[
n(l+1) = \begin{cases} \left(\left\lfloor \frac{n(l)_x}{2} \right\rfloor, \left\lfloor \frac{n(l)_y}{2} \right\rfloor \right) & l < L - 2 \\ (0, 0) & l = L - 2 \end{cases}
\] (3.5)

Whenever \( t \) is updated, the respective quadrant in its parent tile is replaced by a version of \( t \) which is downscaled by a factor of two. Afterwards, the updated parent takes the role of its children, and the process is repeated until the top-most tile has been updated. Note that only quadrants corresponding to updated children are replaced.

The number of updates required at level \( l = 0 \) depends on the area and locations of new images, the chosen tile size and the resolution of tiles in terms of texels per unit. At levels
3.3 Architecture and Implementation Details

In this section we discuss the architecture and some technical details of the FCM. The FCM is implemented in C++ and GLSL 1.50 and is an important part of our framework for DSM rendering which is covered in chapter 7.

3.3.1 Caching

One characteristic of clipmaps is the employment of caching and secondary storage. In the FCM, we keep the $c_x(l) \times c_y(l)$ texture tiles from all $L$ clip areas in a cache in main memory. As described in section 3.2.2, the set is chosen based on the current clip center with the $c_x(l) \times c_y(l)$ neighboring tiles centered around it. This set corresponds to the clip stack in [79]. Its content is updated as the virtual viewpoint and thus the clip center moves by a distance greater than some threshold. The quantities $c_d, d \in \{x, y\}$, depend on the clip level $l$ and are limited by user-defined constants $C_d$ and hence $C_d(l) = 1$. At the top-most clip level $l = L - 1$, only a single tile exists and hence $c_d(L - 1) = 1$. Likewise, at $l = L - 2$, there are at most $2 \times 2$ tiles and hence $c_d(L - 2) = 2$, etc.

The smaller subset of $k_x(l) \times k_y(l)$ tiles with $k_d(l) \leq c_d(l)$, which are also centered around the clip center and visible to the viewer, i.e., the active area, is copied to the tile texture array in video memory, which is realized as a 1D texture array, accessible by GPU fragment programs. Similar to $c_d(l)$, the quantities $k_d(l)$ also depend on the clip level $l$ and are limited by user-defined constants $K_d \leq C_d$. $K_d$ and hence $C_d$ should be chosen with respect to the tile size $T_x \times T_y$ in texels so that $T_d \cdot K_d$ are at least as large as the width respectively height of the application’s viewport in pixels. Otherwise, the borders between different clip levels may become visible to the viewer during rendering, if the scene is shown from a perpendicular point of view.

The cache of an FCM of $L$ clip levels consists of a tile level cache and a tile buffer. These two data structures identify individual tiles by means of their clip level $l$ and tile index $(n_x, n_y)$. The tile level cache consists of an array of $L$ 2D tile arrays where the 2D tile array at index $l$ corresponds to the clip area at clip level $l$. Each of the 2D tile arrays can store at most $C_x \times C_y$ tiles. The tile arrays for the two top-most levels hence need to contain only one respectively four tiles. If clip levels are added to the FCM, the tile level cache must be adjusted as well in order to keep it consistent with the clip areas. This requires not only to update the number of 2D tile arrays contained in the tile level cache, but also the sizes of the tile arrays representing the top-most clip areas, because after resizing, they need to store more tiles than they did before. For instance, if the FCM is resized from $L' = 5$ to $L = 6$ levels, the tile array representing the clip area at level $l = L' - 1 = 4$ must be resized from a $1 \times 1$ array to a $2 \times 2$ array, since $l = L - 1 = 5$ becomes the new top-most clip level and may only need to store a single tile. The transfer of tiles between the cache of the FCM and
secondary memory is performed asynchronously in a separate thread in order not to stall the rendering process as explained in section 7.2.1.

To reduce the transfer of tiles from secondary memory, we additionally keep a certain amount of tiles that have already been loaded during updating, but which are not yet cached, in the tile buffer which resides in main memory. The tile buffer is an unsorted, associative array and is accessed via a textual string which is constructed from the requested clip level \( l \) and the tile index \((n_x, n_y)\) according to the pattern \( l x n_x y n_y \). For instance, the tile having the index \( n = (−7, 4) \) at clip level \( l = 3 \) is accessed via the textual string \( 13 x -7 y 4 \). This pattern was designed to be short, but human readable, e.g., in order to facilitate debugging during development. Besides, the string is used as the name of the file storing the tile in secondary memory. If the tile buffer overflows, tiles that are currently not marked as being about to be updated and which are not in use otherwise are written back to secondary memory. Figure 3.12 contains a sketch of the components and entities of the FCM.

Figure 3.12: An overview of the architecture of the FCM.

### 3.3.2 Tile Arrays

The 2D tile arrays of the tile level cache of the FCM for storing the clip areas are implemented by 1D arrays. In order to access a tile at a certain clip level \( l \) within the corresponding 1D array by means of its 2D tile index \( n = (n_x, n_y) \), we need to transform \( n \) into a 1D array index. Each tile array holds \( c_x(l) \times c_y(l) \) tiles from a contiguous rectangular section of the tile grid (cf. section 3.3.1), so that the indices of the tiles in the clip area are in the range \([n_x, n_x + 1, \ldots, n_x + c_x(l) - 1] \times [n_y, n_y + 1, \ldots, n_y + c_y(l) - 1]\). The tile indices in this range can be mapped to \([0, 1, \ldots, c_x(l) - 1] \times [0, 1, \ldots, c_y(l) - 1]\) by using the function \( j(n) = (j_x(n_x), j_y(n_y)) \) with \( j_d(k), d \in \{x, y\} \), as given in equation (3.6).

\[
j_d(k) = k \mod c_d(l) \tag{3.6}
\]

In this way, we can map the range of tile indices in a 2D tile array to a 1D range of size \( c_x(l) \cdot c_y(l) \) by using equation (3.7).

\[
f(n) = f(n_x, n_y) = j_y(n_y) \cdot c_x(l) + j_x(n_x) \tag{3.7}
\]
In some programming languages, like C/C++, or specific implementations, the result of the modulo operator depends on the signs of the operands and may become negative. We therefore ensure that the components of a tile index $n_d$ are always mapped to positive values by adding an integral multiple $\mu$ of the (positive) divisor $c_d(l)$ to $n_d$, where $\mu > n_d$. The modified versions of equation (3.6) and equation (3.7) as used in our FCM implementation are given in equations (3.8).

$$\vec{j}_d(k) = \left(k + \left(\left\lfloor \frac{|k|}{c_d(l)} \right\rfloor + 1\right) \cdot c_d(l)\right) \mod c_d(l)$$

$$\vec{j}(n) = \vec{j}(n_x, n_y) = \vec{j}_g(n_y) \cdot c_x(l) + \vec{j}_x(n_x)$$

The elements of a 2D array can thus be stored in a 1D array as illustrated in figure 3.13. If the virtual viewpoint and hence the clip center moves, this kind of toroidal addressing allows to directly access and update the entries in the 1D array which correspond to elements from the rows or columns at the border of the 2D array, without relocating the remaining ones.

We use the same technique to linearize the subset of tiles from the active areas in the tile arrays of the FCM’s tile level cache in order to pack the texture data from all clip levels into a single 1D tile texture array in video memory for rendering. Texture arrays consist of textures of the same size and texel format as opposed to arrays of textures which are not restricted in this way. Arrays of textures are furthermore bound to one texture unit per contained texture, whereas one texture array is bound to only one texture unit. Since this eliminates
the need for switching textures and multiple draw calls during rendering, if there are more textures than texture units, texture arrays can yield better rendering performance [54]. At each clip level \( l \), the active area consists of \( \nu(l) = k_x(l) \cdot k_y(l) \) tiles. The total number of elements for storing the tiles in the active areas from all clip levels in a 1D tile texture array thus equals \( \sum_{l=0}^{L-1} \nu(l) \). The 2D tile indices are linearized by means of the formulas in equations (3.6) and (3.7) where \( c_d(l) \) is replaced by \( k_d(l) \). The first of the \( \nu(l) \) tiles at level \( l \) is stored in the tile texture array at offset \( \Omega(l) = \sum_{i=0}^{l-1} \nu(i) \). Like the clip areas, the active areas at the top-most clip levels are smaller than those at lower levels, because at level \( L - 1 \) there is only one tile, at \( L - 2 \) are at most four tiles, etc., so that the ranges and offsets within the tile texture array are not constant. As explained in section 3.3.5, the FCM’s fragment program for rendering uses the clip level and a 2D tile index \( n = (n_x, n_y) \) to access texture data of a tile within the tile texture array. In order to compute the tile’s index within the tile texture array in the fragment program, the offsets \( \Omega(l) \) are required. Therefore, the offsets are stored in a separate array indexed by \( l \) and provided to the fragment program via a uniform variable. Figure 3.14 illustrates the tile texture array as stored in video memory and the array containing the offsets \( \Omega(l) \).

![Figure 3.14: Illustration of the 1D tile texture array and the array for storing the offsets \( \Omega(l) \) of the first tile from each clip level \( l \). In this example, the FCM has \( L = 5 \) clip levels, the clip areas (light gray) have sizes of at most \( 5 \times 5 \) tiles and the active areas (dark gray) consist of at most \( 3 \times 3 \) tiles. At levels 3 and 4, the sizes of both types of areas are only \( 2 \times 2 \) and \( 1 \times 1 \), respectively. The two 1D arrays are located in video memory for being accessed by the fragment program for rendering.](image)

### 3.3.3 Scheduling Tile Updates

Updating and creating tiles as described in section 3.2.5 is also performed in a separate thread. Tiles at level \( l = 0 \) are created by a GPU program for DSM synthesis which is subject of chapter 5.

When images are inserted into the spatial index, the tile update process calculates the indices for the tiles at \( l = 0 \) that are covered by the new images. For each tile that is about to be updated, all images whose projections overlap the tile’s area are retrieved from the spatial index as described above. The texture data are then generated by projecting the returned aerial images onto the tile and assigning texel values according to one of the
3.3 Architecture and Implementation Details

After updating a tile at level \( l = 0 \), equation (3.5) is used to determine its parent tile and a tile update job at level \( l = 1 \) is inserted into a schedule. A tile update job consists of a sequential number (ID), the clip level of the parent and the indices of the parent and its child. The schedule is implemented as a priority queue, and the jobs are assigned priorities according to the parent level and the ID of a job. Given two distinct tile update jobs \( j_1 \) and \( j_2 \), the priority of \( j_1 \) is considered to be higher than the one of \( j_2 \), if

\[
(l_1 < l_2) \lor ((l_1 = l_2) \land (ID_1 < ID_2))
\]

where \( l_i \) and \( ID_i \), \( i = 1, 2 \) are the level and ID of tile update job \( j_i \), respectively.

As soon as no tiles at level \( l = 0 \) are left for being updated from images, the thread responsible for updating tiles continues to process the jobs from its schedule. A parent tile and its respective child are retrieved from the cache of the FCM and are both locked during updating, i.e., they cannot be accessed by any other process. If a tile is requested for updating, but already locked for other reasons, e.g., because it is about to be copied into video memory by the rendering process, the job is re-scheduled with a greater ID. Updated tiles are unlocked again, marked as updated and stored in the tile buffer of the cache of the FCM, since they are likely to be used for updating their own parents or for being copied to video memory by the rendering process. Updating tiles at \( l > 0 \) will cause further jobs to be scheduled in the same way, until the top-most tile has been updated and the schedule runs empty.

3.3.4 Tile Map and LOD Calculation

In order to avoid aliasing during rendering, the texture data that determine the final color of a screen pixel have to be selected from tiles at clip level \( l_{opt} \). A single texel of a tile at \( l_{opt} \) aggregates a number of texels from levels \( < l_{opt} \) and thus provides a pre-filtered version of the texture data. Therefore we have to calculate the level of detail (LOD) \( l_{opt} \) which best avoids texture aliasing, and we have to locate the corresponding tile that contains the texture data within the tile texture array.

Following the idea by Crawfis et al. [16], we use an extra texture, the tile map. Each texel in the tile map corresponds to the area of a tile at level 0 in world space. The value of the texel indicates the lowest clip level \( l_{low} \) for which a tile is present in the tile texture array in video memory. A value of zero, for instance, means that a tile is available at level 0. Only a rectangular region of \( k_x \times k_y \) texels, i.e., the size of the active area, around the location of the clip center in the tile map can have a value of zero. A tile at level \( l > 0 \) corresponds to \( 2^l \times 2^l \) texels within the tile map. Note that the example of a tile map shown in figure 3.15 only contains the lowest clip levels available in video memory, but the corresponding FCM has two more clip levels, so that \( L = 5 \) due to the total size of the area \( A \) covered by the FCM. Tiles at higher levels are implicitly present in video memory as well, because they partially contain the more detailed areas, and the top-most tile must always be available. The tile map is recomputed and uploaded into video memory each time the clip center is relocated due to movements of the virtual camera, and its lateral extension \((tm_x, tm_y)\) in
texels must satisfy equation (3.9).

$$\log_2 (\min (tm_x, tm_y)) \geq (L - 1) \quad (3.9)$$

Figure 3.15: Illustration of a tile map: each grid cell represents a texel and corresponds to a tile at level 0. The color indicates the value of the minimum clip level of a tile that covers the corresponding area of the FCM and which is present in the tile texture array in video memory.

Aliasing caused by texture sampling is best avoided by the texture data from a tile at clip level \(l_{opt}\). This particular level is computed in the same way as a corresponding mip-map level. The computation is based on the amount of texels that are covered by a single screen space pixel (minification) where pixels and texels are assumed to be representable as squares, although they are actually point samples \([26]\). The minification \(j\) corresponds to the number of texels at the base clip level 0 that need to be aggregated into one screen pixel. In this way, the texture is spatially filtered and aliasing is avoided.

If the partial derivatives of the texture coordinates \((u, v)\) with respect to the screen coordinates \(x\) and \(y\) are assumed to be constant across the area of a single pixel in screen space, its shape in texture space can be approximated by a parallelogram \([26]\). \(j\) can then be determined, for instance, by hypotenuse comparison as presented in \([26]\), i.e., by the length of the greater of the two sides \(s_a\) and \(s_b\) of the pixel’s parallelogram in texture space. As illustrated in figure 3.16, \(s_a\) and \(s_b\) are given by

$$s_a^2 = \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 \quad \text{and} \quad s_b^2 = \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2$$

where \(\frac{\partial u}{\partial d}, \frac{\partial v}{\partial d}, d \in \{x, y\}\) denote the changes of the texture coordinates \((u, v)\) along the screen space directions \(x\) and \(y\). At clip level \(l\), \(2^l\) texels from the base level 0 are aggregated along each dimension, and we compute \(l = l_{opt}\) according to \([26]\) by means of equations (3.10).

$$j = \max (s_a, s_b) = 2^l$$

$$\Rightarrow l = \log_2 \left(\sqrt{\max (s_a^2, s_b^2)}\right)$$

$$= \frac{1}{2} \log_2 \left(\max \left(\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2, \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2\right)\right) \quad (3.10)$$
3.3 Architecture and Implementation Details

Figure 3.16: The projected shape of a screen pixel onto a texture (blue) is approximated by a parallelogram in texture space. Its sides are related to the partial derivatives of the texture coordinates \((u, v)\) with respect to the directions \(x\) and \(y\) in screen space. Illustration following [26].

The four partial derivatives can be calculated in GLSL fragment programs by applying the built-in functions \(dFdx\) and \(dFdy\) to the normalized texture coordinates scaled by the size of the virtual texture in texels.

Though it is computationally more expensive, anisotropic texture filtering, which is supported by current graphics hardware [1, p. 168], yields better visual results, since the resulting images look less blurry as shown in figure 3.2(b) and figure 3.2(c). To take into account the maximum number of samples \(N_{\text{aniso}}\) along the line of anisotropy [1, pp. 168–170] in order to select an appropriate clip level for texture sampling, equation (3.10) needs to be modified. The modified formula is given in equation (3.11).

\[
\begin{align*}
\sigma_{\min} &= \min (s_a, s_b), \\
\sigma_{\max} &= \max (s_a, s_b) = j, \\
r_{\text{aspect}} &= \min \left( \frac{\sigma_{\max}}{\sigma_{\min}}, N_{\text{aniso}} \right) \\
l &= \log_2 \left( \frac{\sigma_{\max}}{r_{\text{aspect}}} \right) = \log_2(j) - \log_2(r_{\text{aspect}})
\end{align*}
\]  

(3.11)

The same formula is also used in [16] and [53] for selecting appropriate levels of detail with respect to anisotropic texture filtering.

3.3.5 Rendering

Before a frame is rendered, the FCM checks if the point of view of the virtual camera has changed, and updates its cache if necessary. All tiles from the active area must be uploaded into video memory, i.e., they must be copied into the tile texture array. Since the active areas are subsets of the clip areas, the tiles within the active areas are contained in the tile arrays of the tile level cache. These tiles are retrieved from the cache of the FCM and sent to video memory in descending clip level order. In this way, if some tiles cannot be uploaded on time, at least the tile at top-most level \(L - 1\) is likely to be present in video memory.
Furthermore, the FCM checks whether tiles are marked as updated or whether previously any tile could not be transferred to video memory because it was locked for updating or not yet loaded into main memory. Tiles that are not available for the aforementioned reasons are marked as delayed, and a counter for the number of unsuccessful accesses of the corresponding tile will be increased. Subsequent attempts to access a delayed tile for rendering in a later frame are made, if the counter is below some threshold, but not until a predefined timeout has expired. If the threshold for the maximum number of delays is exceeded, the tile is marked as absent and will not be used for rendering again, because it is then considered missing. This avoids further unsuccessful and hence unnecessary attempts to retrieve an absent tile from the cache. Absent tiles are usually tiles which have a valid tile index according to the area $A$ covered by the FCM, but which do not contain any texture data. In this case, the rendering must rely on texture data from the parent tile at the next coarser LOD, if such a level exists.

Texture mapping using the FCM is implemented by a GLSL fragment program. The tile texture array and the tile map have to be bound to one texturing unit each and are accessed by the shader. The texture coordinates $(u, v)$ of the surface to be textured are assumed to be in $[0, 1] \times [0, 1]$ and correspond to the area $A$ covered by the FCM. In this way, whenever the FCM is resized, the texture coordinates can remain unchanged, although the underlying geometry of the textured surface, e.g., a planar quadrilateral mesh, is preferably also scaled in order to reflect the extension of the depicted area.

The actual texture mapping of a surface performed by the fragment program of our FCM implementation comprises the steps listed below.

1. determine the clip level $l_f$ for sampling texture data
2. determine world coordinates $p$ at current fragment
3. compute the 2D tile index $n = (n_x, n_y)$ from $p$ and $l$
4. transform $n$ into a 1D index to access the tile texture array by using the clip level’s offset $\Omega(l)$
5. compute the sampling position $(s_u, s_v)$ sampling the tile’s texture data

In the following, we describe each of these steps in detail.

At each fragment, the fragment program performs a look-up in the tile map to determine the tile with the lowest clip level $l_{\text{low}}$, i.e., the highest LOD, that is available in the tile texture array, and it furthermore calculates $l_{\text{opt}}$ as described in section 3.3.4. The final clip level and LOD $l_f$, at which the corresponding tile has to be accessed, is computed by $l_f = \max (l_{\text{low}}, l_{\text{opt}})$. Since $l_f$ is usually not an integral level, trilinear interpolation between $\lfloor l_f \rfloor$ and $\lceil l_f \rceil$ by means of the fractional part of $l_f$ can be used to obtain smooth transitions of texture data between each two LODs.
The corners $R_{\text{min}}$ and $R_{\text{max}}$ in world coordinates of $A$ are passed to the fragment program via uniform variables. Let $(u, v)$ denote the texture coordinate at the current fragment. The corresponding location $r = (r_x, r_y)$ in world coordinates is computed by

$$
\begin{align*}
    r_x &= u \cdot (R_{\text{max}} - R_{\text{min}}) + R_{\text{min}} \\
    r_y &= v \cdot (R_{\text{max}} - R_{\text{min}}) + R_{\text{min}}
\end{align*}
$$

The grid spacing of the tile grid $(g_x, g_y)$ is passed as a uniform variable to the fragment program as well. The 2D tile index $n = n(l, r)$ of the tile containing the previously computed position $r$ at clip level $l = l_f$ is then computed by equation (3.1).

Using the tile index $n$, the number of tiles $k(l) = (k_x(l), k_y(l))$ in the 2D tile array from the active area and the array of offsets $\Omega(l)$ where the first tile from the active area is located within the tile texture array, the tile’s index within the tile texture array $\xi$ is computed by means of equations 3.6 and 3.7 where $c_d(l)$ is replaced by $k_d(l), d \in \{x, y\}$. The arrays holding $k(l)$ and $\Omega(l)$ are indexed by $l$ and passed via uniform variables to the shader.

The position $(s_u, s_v)$ where the tile at index $\xi$ in the tile texture array has to be sampled is calculated by the shader by means of the texture coordinates $(u, v)$ of a fragment according to equations (3.12).

$$
\begin{align*}
    (G_u, G_v) &= (R_{\text{min}} \cdot \text{res}_x, R_{\text{min}} \cdot \text{res}_y) \\
    (N_u, N_v) &= ((R_{\text{max}} - R_{\text{min}}) \cdot \text{res}_x, (R_{\text{max}} - R_{\text{min}}) \cdot \text{res}_y) \\
    (F_u, F_v) &= \begin{cases} \\
        \left(\left\lfloor \frac{G_u}{2^l} \right\rfloor, \left\lfloor \frac{G_v}{2^l} \right\rfloor \right), & l < L - 1 \\
        \left(\left\lfloor \frac{G_u}{2^l} \right\rfloor - \frac{T_x}{2^l}, \left\lfloor \frac{G_v}{2^l} \right\rfloor - \frac{T_y}{2^l} \right), & l \geq L - 1
    \end{cases} \\
    (t_u, t_v) &= \left(\frac{u \cdot N_u}{2^l} + F_u, \frac{v \cdot N_v}{2^l} + F_v \right) \\
    (s_u, s_v) &= \left(\frac{t_u \mod T_x}{T_x}, \frac{t_v \mod T_y}{T_y} \right)
\end{align*}
$$

$(\text{res}_x, \text{res}_y)$ denotes the tile resolution in terms of texels per unit in world coordinates and $(T_x, T_y)$ is the size of the tile’s texture in texels. Both quantities are also available in the fragment program. This formula is implemented by the GLSL functions `getTileCoordUniform()` and `getTileCoord()` which can be found at the beginning of the complete source code of our FCM fragment program in listing A.2 in appendix A. The latter function is used for texel-precise texture sampling and hence computes sampling positions $(s_u', s_v')$ in the range $[0, T_x - 1] \times [0, T_y - 1]$ instead of $[0, 1] \times [0, 1]$ by omitting the divisions by $T_x$ and $T_y$.

### 3.4 Performance Analysis

We analyzed the performance of the FCM in terms of the number of updated tiles per second during continuous addition of aerial images. Aerial images data were created using the
The Flexible Clipmap simulation framework in [75] which allows to generate virtual vertical aerial images at locations, orientations, resolutions and frequencies specified by the user from renderings of digital 3D models. The images with added metadata were sent by the simulator over network via TCP to a client application where they were processed by the employed FCM.

3.4.1 Evaluation Setup

The run-time performance of the rendering and the visual quality of the resulting texture map in the FCM depend on the following quantities: the resolution of the incoming aerial images in texels, the number of aerial images per second (image rate), the number of tile updates per second (update rate), the resolution of the final texture in texels per unit \((\text{res}_x, \text{res}_y)\), the tile size in texels \((T_x, T_y)\), the sizes of the clip areas \((c_x(l), c_y(l))\) and the active areas \((k_x(l), k_y(l))\) in number of tiles at each level \(l\), and capacity of the tile buffer \(B\).

In our evaluation setup, we always used a viewport/screen size of \((V_x, V_y) = (1024, 768)\) pixels, \(\text{res}_d = 20\), \(K_d = \lceil V_d / T_d \rceil\), \(C_d = 2 \cdot K_d\) with \(d \in \{x, y\}\) and \(B = C_x \cdot C_y\). For each tested configuration of tile sizes, image sizes and average images rates, we simulated a single camera with a vertical field of view of \(60^\circ\) and following exactly the same path at a constant altitude of 50 m above ground, and we performed 10 runs each to obtain means. Means are necessary, because the images received by the client are located at random positions along the path of image acquisition during different runs, as there was no synchronization between the client application and the simulator. In addition, the simulator cannot guarantee to provide a certain update rate, which also strongly depends on the resolution of the aerial images and the network. Tile updates were executed before every frame whenever at least one new aerial image had been received. The simulator and the client application were executed in parallel on a desktop computer with an Intel i7 860 CPU at 2.8 GHz, a NVidia GeForce 470 GTX with 1280 MB dedicated video memory, 6 GB RAM and 64-bit Windows 7 operating system.

3.4.2 Results

The performance of the FCM in terms of updated tiles per second from the setup described in section 3.4.1 for tile sizes of \(T_x = T_y = 256\) and \(T_x = T_y = 512\) texels and for different image sizes at different averaged image rates is shown in figure 3.17. Error bars indicate the standard deviations. During the evaluation the FCM had at most \(L = 8\) clip levels in case of \(T_x = T_y = 256\) and \(L = 7\) in case of \(T_x = T_y = 512\) after the area of \(A\) reached its maximum extension. The amount of texture data managed by the FCM thus corresponds to a single 32-bit R8G8B8A8 texture of \(2^{15} \times 2^{15}\) texels with a memory size of 4 GB. The most demanding configuration \((4000 \times 3000\) texels@0.26\) corresponds to an average data rate of 11.9 MB per second but still achieved tile update rates of 427.59 tiles per second for tile size \(256 \times 256\) and 150.57 tiles per second for tile size \(512 \times 512\). The total number of tiles in the tile texture array was only 85 respectively 25. This implies that the average update rates are sufficient to update the entire tile texture array multiple times per second and still permit interactive rendering frame rates.
3.5 Discussion

Our evaluation has shown that the most important limiting factors in our setup were the simulation of image sources providing high resolution images at high frequency, and the transmission over the network. Therefore the simulator and client were executed in parallel on the same machine using the network loopback device in order to eliminate transmission errors. This prevented us from performing further measurements and determining the limits of the FCM in the given scenario.

A problem arises if the projections of the aerial images cover the entire area of the FCM or even more, e.g., when the images are taken from high altitudes: the updates then would not be confined to few tiles and would affect the entire FCM. However, this issue can be avoided by limiting the number of tiles updated in each frame, which is advisable anyway, because performing many tile updates between two frames would stall the rendering and reduce interactivity.
Another severe issue is caused by the 32-bit floating point precision of the texture coordinates \((u, v)\), because they need to cover the entire \(\mathbb{R}^2\) (see section 3.3.5). This also arises in other GPU based clipmap implementations (cf. [23], [78]). By deriving the geometry from the FCM as well, we could drop that constraint and map the texture coordinate range of \([0, 1] \times [0, 1]\) only to the visible portion of the geometry.

Furthermore, the size of the tile map can exceed the hardware limits as the number of clip levels increases (cf. section 5.3). This can be countered by virtualizing the clip stack, as already introduced in [79], by processing only a subset \([L_{\text{min}}, L_{\text{max}}] \subset [0, L - 1]\) of clip levels which are relevant for the viewer at the current viewpoint. Though the tile map requires an additional texture unit, it provides a very easy LOD selection and avoids invalidating entire detail levels, if a single tile is missing (cf. [16]).
4 Digital Surface Model Rendering

Since textured polygonal meshes can be processed and rendered by GPUs at high speed, a widely used rendering technique stores a digital elevation model (DEM) in (grayscale) texture maps (so called heightmaps or heightfields) and uses them to displace the vertices of a sufficiently tessellated planar polygonal mesh. This technique is known as displacement mapping [1, p. 198] and is considered an extension of bump mapping [15]. Bump mapping employs textures to perturb the surface normals to modify the perceived smoothness of a surface during lighting computations, without modifying the underlying geometry [1, pp. 183–190].

However, most renderers accept only triangle meshes which can become rather complex and may easily consist of millions of triangles. During mesh generation, particular attention has to be paid to different issues, e.g., to not produce any cracks, to choose appropriate tessellations and to avoid aliasing caused by small or distant triangles.

Furthermore, when a large mesh becomes rasterized, many triangles result in at most a few pixels whose corresponding fragments succeed in passing all of the numerous tests encountered on their way through the rendering pipeline. Therefore it appears to be attractive to bypass the entire process of converting a heightfield into a mesh. Techniques like relief mapping [63] or parallax occlusion mapping [80] can make use of pixel shaders on modern GPUs to perform real-time ray casting on heightfields. This allows to conveniently calculate the displaced sample positions in corresponding color textures for determining the final fragment color. During this ray casting fine-structured details can be added to surfaces without further tessellating the underlying polygonal mesh. In many cases this even allows to reduce the polygonal mesh to a planar quadrilateral which may be tessellated into only two triangles.

In order to speed up the ray casting and to achieve real-time frame rates, many GPU-based heightfield rendering techniques employ maximum mipmap to access the DEM. At each level, a maximum mipmap aggregates $2 \times 2$ texels into one texel at the next higher mipmap level by using the maximum of the four texel values instead of their averaged value. As the size of texture maps that can be handled by GPUs is currently limited by manufacturer specific restrictions and ultimately by the amount of available video memory, large DEMs cannot be stored in a single heightfield texture for direct access during GPU-based ray casting.

In this chapter, we present a GPU-based heightfield ray casting technique for single-pass rendering of heightfields of almost arbitrary sizes in real time. The technique employs the Flexible Clipmap (FCM) discussed in chapter and current graphics hardware to speed up the ray casting while alleviating the aforementioned video memory limitations.
is achieved by taking into account the desired image quality on a per-pixel basis and early ray termination based on level of detail selection: At each intersection of a ray with the surface to be rendered, we check if the corresponding element of the surface becomes projected to less than one screen pixel. In this case, we can stop ray casting, because it would be unnecessary to depict more details about the surface at the current pixel. Additionally, two different refinement methods for improving the appearance of the reconstructed surfaces in our renderings can be employed.

## 4.1 Related Work

Much research has been done on CPU-based ray casting of heightfields as well as on terrain rendering based on polygonal meshes. Since summarizing these two areas would exceed the scope of this thesis, we confine ourselves to an overview of recent GPU-based heightfield ray casting methods related to our work.

Qu et al. [65] presented one of the first GPU-based ray casting schemes for heightfields which primarily aims at accurate surface reconstruction of heightfields but does not use any sophisticated structures for acceleration.

Relief mapping [57] and parallax (occlusion) mapping [40] are techniques for adding structural details to polygonal surfaces, which have their origin in CPU-based rendering and improve upon the disadvantages of bump mapping [10]. Both techniques have been implemented for GPUs (e.g., [63, 80]) and benefit from programmable graphics pipelines. But as most of these implementations resemble the strategies used in CPU-based ray casting, like iterative and/or binary search to detect heightfield intersections, they are prone to the same kind of rendering artifacts caused by missed intersections in highly spatially variant data sets. An introduction to these closely related techniques can be found for instance in [1, pp. 183–199], and more details are given in the comprehensive state-of-the-art report by Szirmay-Kalos and Umenhoffer [77] which focuses on GPU-based implementations.

Oh et al. [55] accelerate ray casting and achieve real-time frame rates by creating a bounding volume hierarchy (BVH) of the heightfield, which is stored in a maximum mipmap and allows to safely advance along the ray over long distances (see section 4.2.2). They also present a method based on bilinear interpolation of heightfield values to improve the quality of the reconstructed surface obtained from point-sampled data.

The method presented by Tevs et al. [81] also relies on BVHs stored in maximum mipmap, but uses a different sampling strategy. Their method advances along the ray from one intersection of the projected ray with a texel boundary to the next such intersection, whereas Oh et al. use a constant step size to advance along the ray. In addition, Tevs et al. store in each heightfield texel the height values at the four corners of a quadrilateral encoded as an RGBA value instead of point samples, which allows surface reconstruction on parametric descriptions.

Compared to other techniques which also rely on preprocessed information about the heightfield and acceleration data structures, like for instance relaxed cone step mapping [20, 48, 62], maximum mipmap creation is much faster and can be performed on the GPU [81].
All these methods have in common that they operate on single heightfields of relatively small extents which are intended to add details to surfaces at meso- or microscales instead of representing vast surfaces themselves. Recently Dick et al. [18] have presented a method for ray casting terrains of several square kilometers extent at real-time frame rates. Their method also employs maximum mipmaps to accelerate the ray casting process and a tiling approach to render data sets of several hundred GB size. They also presented a faster hybrid method which uses ray casting or rasterization-based rendering, but requires knowledge of the employed GPU respectively a training phase to decide whether to use rasterization or ray casting [19]. Another hybrid technique is presented in [3].

Our method aims at rendering very large heightfields only by means of GPU ray casting. It has been inspired in large parts by the works of Dick et al. [18] and Tevs et al. [81] as we also employ a tile-based approach and their cell-precise ray traversal scheme. But in contrast to the technique by Dick et al., which creates a complete mipmap for each tile and requires additional rendering passes to determine the visibility of the tiles, our method further accelerates the ray casting process and requires only a single rendering pass by using our tile-based FCM implementation (see chapter 3).

Another concept worth mentioning in the context of DSM rendering and clipmaps is the geometry clipmap as introduced by Losasso et al. [47]. Geometry clipmaps and derived GPU-based variations [4, 12] store DEM data in vertex buffers at different resolutions, but according to our knowledge they are only used in the context of mesh-based rendering and not for accelerating ray casting.

4.2 GPU based Single-pass Ray Casting Using Clipmaps

Our FCM implementation can be used to store different DSM data, and in particular elevation information, i.e., a DEM, in order to produce 3D renderings of the underlying surface by means of ray casting. In this section we explain our storage scheme for DSM data and describe the employed ray traversal method, which is basically the same as the one presented in [18]. We discuss how to accelerate ray casting and how to avoid aliasing by using the FCM. Furthermore, we present two refinement methods that can be used to improve the appearance of the reconstructed surfaces.

4.2.1 Clipmaps for DSM Storage

In order to store DSM data, the tiles in our FCM implementation are extended to consist of several textures (tile layers) of the same extension for the different types of data of a DSM. For our purposes, two layers are required, one for storing elevation data (DEM layer) and one for color information (color layer). The layers correspond to textures, and their texel formats are based on the underlying data according to table 2.1 in section 2.2.3. For each layer, the GPU uses one separate texture tile array as described in chapter 3.3.2 to access the texture data of the tiles in the active area.
To use a digital elevation model (DEM) for rendering, in our approach the height values are stored in the DEM layer of FCM tiles at the highest resolution (lowest) level \( l = 0 \). A texel at level \( l > 0 \) obtains as value the maximum height value of the corresponding \( 2 \times 2 \) subordinate texels at level \( l - 1 \). This is the same construction scheme as used with maximum mipmaps \([18, 55, 81]\). If we identify each texel with a bounding box defined by its height value and its grid cell, i.e., the boundaries of the associated square texel in the texture, we obtain a bounding volume hierarchy (BVH) of the underlying DEM as illustrated in figure 4.1. In the method presented by Dick et al. \([18]\), the heightfield is split into tiles as

![Figure 4.1: The BVH is derived from the DEM data on a regular grid. Gray boxes correspond to elevation samples at level 0. Bounding boxes on higher levels and their maximum value are highlighted by the same color.](image)

well, but a separate maximum mipmap is created for each tile. To render vast DSMs, this approach may require either lots of tiles and thus mipmaps to be present in video memory or additional rendering passes, especially if the heightfield is shallow and there is little occlusion between tiles. Furthermore, the tiles located far away from the viewer may contain fine spatial details, e.g., steep summits of distant mountains, which are not only not perceivable from far away but may also expose spatial aliasing artifacts due to minification caused by perspective projection. The latter aspect is the same which motivated the development of mipmaps for texture mapping and also applies to mesh-based rendering techniques which therefore strive to determine an appropriate level of detail (LOD) in order to avoid rasterizing triangles that would become projected to less than one pixel in screen space.

The important difference between the usage of a clipmap and multiple mipmaps is that in case of a clipmap the BVH spans the entire domain at the topmost level. A proper placement of the clip center results in the selection of only those tiles of highest resolution at level \( l = 0 \) which are closest to the virtual camera and thus potentially have to be rendered in full detail. Compared to level \( l \), at level \( l + 1 \) the area of the heightfield covered by a tile is four times larger, and the spatial resolution is divided in half along each direction of the grid. Thus the entire domain is spatially pre-filtered and the level of detail of the heightfield decreases with increasing distance towards the viewer. Because higher clipmap levels also correspond to larger bounding boxes, we can exploit this fact to accelerate GPU ray casting in distant parts of the scene as described in section 4.2.2.
4.2 GPU based Single-pass Ray Casting Using Clipmaps

4.2.2 Rendering and Accelerated Ray Traversal

The ray casting of DSMs respectively DEMs using the FCM is done on the GPU by means of a vertex and a fragment program. For the mere purpose of ray casting, only the DEM layer of the tiles in the active area is required. The textures containing elevation data are accessed by the GPU via a separate texture tile array in the same way as the texture tile array of an FCM, that is used for texturing a (planar) surface (see section 3.3.5). Given a DSM stored in an FCM of $L$ clip levels, we set the clip center simply by projecting the center of the viewport into the scene. Using this method, the location of the clip center depends on the angle $\varphi_{\text{view}}$ enclosed between the viewer’s viewing direction and the opposite surface normal of the plane containing the rendered elevation data. If $\varphi_{\text{view}} = 0$, i.e., the viewer is looking perpendicularly onto the ground plane, the clip center is located in the center of the viewport as desired. If $\varphi_{\text{view}}$ increases, the clip center becomes located in more distant parts of the scene near the horizon. This is undesirable, since the viewer probably wants to see elevated surfaces in parts of the scene closer to her. We therefore control the maximum distance of the clip center from the viewer by confining the maximum angle for computing the distance to the clip center to $70^\circ$, which is an empirical value. In this way, the clip center will remain located within the viewport between its lower border and the horizon of the scene, even if the viewing direction is parallel to the ground plane or directed upwards. This method for placing the clip center is illustrated in figure 4.2. We also ensure that all tiles in the active areas of all clip levels or at least the highest ones can be stored in video memory by choosing appropriate sizes for the tiles and the active area.

The axis-aligned bounding box (AABB) of the entire DEM, which is associated with the topmost tile, is based in the xy-plane of the world coordinate system. While the extension of this AABB in x- and y-direction can be obtained directly from the box stored at the root node of the FCM’s R*-tree, the size in z-direction is determined by the maximum elevation value of that tile. The AABB is used to construct a polygonal box mesh consisting of only

![Figure 4.2](image)

**Figure 4.2:** The location of the clip center (yellow dot) in the scene is computed by projecting the geometric center of the viewport into the scene. It depends on the angle enclosed between the opposite surface normal $n$ of the ground plane containing the elevation data of the rendered surface and the viewing direction $v$. If $v$ is (near-)parallel to the ground plane (right image), we ensure that the clip center remains located between the horizon of the scene (dashed line) and the bottom of the viewport.
12 triangles which serves as proxy geometry for the ray casting process. A vertex program obtains the dimensions of the box in world space and calculates normalized 3D texture coordinates from the vertex coordinates of the box corners. The clipmap is positioned at the bottom of the box corresponding to the minimum height value \( z = H_{\text{min}} \) of the DSM. \( H_{\text{min}} \) and the maximum height value \( H_{\text{max}} \) are both determined during loading of the topmost clipmap tile into main memory. The texture coordinates on the box range from \((0, 0, 0)\) on its minimum corner to \((1, 1, 1)\) on its maximum corner and are used to setup the rays as described in [43]. By rendering the back faces of the proxy geometry we obtain each ray’s exit point \( e \), and we pass the camera position and the geometry of the bounding box in world coordinates to the fragment shader which calculates each ray’s normalized direction \( \vec{d} = (d_x, d_y, d_z) \) and entry point \( s \) to the proxy geometry in normalized 3D texture space\(^6\).

If the camera is located inside the bounding box the entry point \( s \) becomes the position of the virtual camera [48]. In order to avoid that faces of a possibly very large proxy geometry are clipped against the far plane of the view frustum of the virtual camera and hence exit points are missing, the box is clipped in advance to fit into the view volume when the virtual camera is moved.

The actual ray traversal is performed by projecting the ray onto a clip level dependent 2D grid with each grid cell corresponding to a texel. For a given clip level \( 0 \leq l < L \) the extensions of this grid \((G_u, G_v)\) are determined by equation (4.1) with \((W, H)\) being the extensions of the entire DSM in sample points, i.e., texels.

\[
(G_u(l), G_v(l)) = \left( \frac{W}{2^l}, \frac{H}{2^l} \right)
\]

Hence, the grid at level \( l \) has the same size a single texture containing the entire DSM at mipmap level \( l \) would have. The current height \( p_z \) of a location \( p = (p_x, p_y, p_z) = s + k \cdot \vec{d} \) is given in grid coordinates. In order to test for intersections with the heightfield, the sampled height values have to be converted from world coordinates to grid coordinates via the height resolution \( \text{res}_z \) of the DEM. As an alternative, \( p_z \) can be transformed into world space for comparing height values and finding intersections of the ray with the surface. During ray traversal we move from one intersection of the projection of the ray onto the xy-plane \( \vec{d}_p = (d_x, d_y) \) with a texel boundary to the next such intersection, i.e., from the projected ray’s entry point \( \text{en}_p \) into a grid cell directly to its exit point \( \text{exp}_p \) as shown in figure [4.3]. The only exception is at the first entry point which is the projection of the starting point of a ray \( s_p \).

Ray casting is started at the highest clip level \( L - 1 \) with the coarsest resolution of the BVH at which the entire DEM is given in a single tile and each pixel corresponds to the maximum value and thus to the bounding box of \( 2^{L-1} \times 2^{L-1} \) texels at the base level 0. To determine whether a ray hits a bounding box at level \( l \), the clipmap tile containing the grid cell which belongs to the current \( \text{en}_p \) and \( \text{exp}_p \) has to be sampled for the associated height value \( h \). The tile is determined by the uniform texture coordinate \((u, v)\) as described

\(^6\)The direction \( \vec{d} \) of a ray is noted with an arrow to emphasize its vectorial character and to distinguish it from the previously used variable \( d \).
in section 3.3.5. Since the direction of the ray is needed to determine this grid cell and the borders where the projected ray enters respectively exits, the sign bits of the components of $\vec{d}$ are stored in the lower three bits of an integer. This bit mask is created once for each ray using bit-wise operations in the fragment shader, and it is evaluated as needed by switch-statements to determine the direction of a ray. Details on the implementation are given in section 7.4.1.

When moving along the ray from point $en$ to point $ex$ we hit the box surface in the following two situations:

1. the ray is directed downwards and $ex$ lies below the top of the box at height $h$
2. the ray is directed upwards and $en$ lies below the top of the box at height $h$

If a ray hits a bounding box $B$ at the current level $l$, it may also hit a bounding box contained in $B$ at a lower level of the BVH. Therefore the ray casting process is repeated at the next lower level $l' = l - 1$ from the current position $en$ of the ray, but only if it is possible and reasonable to proceed as described in section 4.2.3. Otherwise the lowest possible level $l = l_{\text{min}}$ has been reached, and the exact intersection $i$ on the bounding box surface is calculated according to equation (4.2).

$$i = \begin{cases} en + \vec{d} \cdot \max \left( \frac{h - en_z}{dz}, 0 \right) & d_z < 0, ex_z < h \\ en & d_z \geq 0, en_z \leq h \end{cases}$$  (4.2)

If a ray does not intersect a bounding box $B$ at level $l$, it cannot intersect any of the bounding boxes contained in $B$ at any lower level either, and we therefore advance along the ray to $ex$ which becomes the entry point $en$ of the next cell. Compared to a ray traversal performed just on level 0, only one instead of $2^l \times 2^l$ samples have to be tested for intersection, which results in a significant speed up of the process [18, 81].

Three different cases for the intersection of a ray with a bounding box are illustrated in figure 4.4. If a ray hits a bounding box $B$ at some level $l > 0$ it does not necessarily have
to hit any bounding boxes contained in $B$ at level $l - 1$, but this cannot be determined without descending to the lower level. In order to avoid using the smaller step size over longer distances, we move up again to level $l$ if we detect that the ray does not hit any bounding box at level $l - 1$ (cf. [18, 81]). The level is increased after four steps without intersecting another box, because after four steps the containing box that caused the decrease of the level is safely passed and the ray traversal cannot get stuck by switching alternately between two levels.

The ray casting process is terminated if either a valid intersection point $i$ on a bounding box has been found or if the ray leaves the domain of the DSM. In the latter case, the fragment from which the ray originates is discarded by the shader.

The complete GLSL source code of the vertex and fragment programs is given in appendix A.1.

4.2.3 LOD-determined Ray Termination

Ray casting can be terminated at the current level $l$, if $l = \max(l_{\text{low}}, l_{\text{opt}})$. The termination criterion is designed to speed up ray casting and to avoid aliasing that may be caused if projections of distant or small boxes are smaller than one screen pixel.

At each intersection of the ray with a bounding box we determine the highest resolution available, i.e., the lowest clip level $l_{\text{low}}$ of a tile which covers the corresponding area of the DSM and is present in video memory. This is done by transforming the hit point $i$ on the bounding box surface to normalized texture coordinates, followed by a single texel-precise texture lookup in the tile map of the FCM as described in section 3.3.4 and section 3.3.5.

The clip level $l_{\text{opt}}$ at the current hit point $i$ is chosen in such a way that aliasing resulting from boxes which cover less than one pixel in screen space is avoided. But instead of using the projection of a single screen pixel into texture space like in section 3.3.4 the size of the corresponding heightfield box in screen space is considered, because a single texel from the DEM layer represents only the top of the box on which $i$ is located. Furthermore, the top may not be visible, e.g., if the viewing direction is nearly parallel to the ground plane of the rendered surface and thus to the top of the box. In this situation, the height of the box has
to be considered as well in order to determine an appropriate LOD based on the number of covered screen pixels.

For this purpose, \( i \) is transformed from grid coordinates into world space to obtain \( i_{\text{world}} = (i_x, i_y, h) \). Let \((\Delta_x, \Delta_y)\) denote the spacing of the DEM grid in terms of units per sample in world coordinates and let \( M \) denote the combined model, view and projection matrix of the virtual camera. The four corners of the box on which \( i \) is located are transformed from world space into normalized screen space by means of \( M \). The resulting 2D positions \( \kappa, \lambda, \mu, \nu \) are used to compute the three sides \( \vec{a}, \vec{b}, \vec{c} \) of the box in window coordinates as shown in equations (4.3) where \( V_x, V_y \) denote the size of the view port in pixels.

\[
\vec{a} = (a_1, a_2) = ((\lambda_x' - \kappa_x') \cdot V_x, (\lambda_y' - \kappa_y') \cdot V_y)
\]
\[
\vec{b} = (b_1, b_2) = ((\mu_x' - \kappa_x') \cdot V_x, (\mu_y' - \kappa_y') \cdot V_y)
\]
\[
\vec{c} = (c_1, c_2) = ((\nu_x' - \kappa_x') \cdot V_x, (\nu_y' - \kappa_y') \cdot V_y)
\]

\( \vec{a}, \vec{b}, \vec{c} \) correspond to three of the sides of the bounding box of the DEM texel in window coordinates and can be used to form the matrix \( T_{\text{tex}} \) in equation (4.4).

\[
T_{\text{tex}} = \begin{pmatrix}
a_1 & a_2 \\
b_1 & b_2 \\
c_1 & c_2
\end{pmatrix}
\]

\( T_{\text{tex}} \) allows to transform coordinates from 2D screen space onto a plane in 3D texture space \( \Sigma_{\text{tex}} \). If \( h \) was zero, the box would be degenerate and identical to its corresponding texel, since \( \vec{c} \) would be 0. In this case, the last row in equation (4.4) can be dropped, and the texture coordinate system would be two-dimensional.

Assuming that the screen pixels are unit squares, they can be transformed to \( \Sigma_{\text{tex}} \) using equations (4.5).

\[
\vec{r} = T_{\text{tex}} \cdot \begin{pmatrix}1 \\ 0\end{pmatrix} = \begin{pmatrix}a_1 \\ b_1 \\ c_1\end{pmatrix}, \quad \vec{s} = T_{\text{tex}} \cdot \begin{pmatrix}0 \\ 1\end{pmatrix} = \begin{pmatrix}a_2 \\ b_2 \\ c_2\end{pmatrix}
\]

By means of \( \vec{r} \) and \( \vec{s} \), we can compute the fraction of one screen pixel that is covered by the box defining \( \Sigma_{\text{tex}} \). This quantity corresponds to the reciprocal of the amount of texels that are covered by one screen pixel in texture space, i.e., the minification \( j \) as described in section 3.3.4. The relation becomes directly apparent, if \( h = 0 \) and the box can be identified with its corresponding texel. In contrast to the texture space in section 3.3.4, \( \Sigma_{\text{tex}} \)
corresponds to the projection of a single box instead of an entire texture. Hence, if $|\vec{r}| < 1$ or $|\vec{s}| < 1$, the projection of the box covers less than one pixel and would cause aliasing. The larger of the reciprocals of $|\vec{r}|$ and $|\vec{s}|$ therefore indicates the minification $j$, which is calculated according to equations (4.6).

$$
|\vec{r}| = \sqrt{a_1^2 + b_1^2 + c_1^2}, \quad |\vec{s}| = \sqrt{a_2^2 + b_2^2 + c_2^2}
$$

$$
j = \max \left( \frac{1}{|\vec{r}|}, \frac{1}{|\vec{s}|} \right) = \frac{1}{\min (|\vec{r}|, |\vec{s}|)} \quad (4.6)
$$

Analog to the LOD computation given in equations (3.10), $j = 2^{l_{\text{opt}}}$, and we compute $l_{\text{opt}}$ according to equation (4.7).

$$
2^{l_{\text{opt}}} = j = \frac{1}{\min (|\vec{r}|, |\vec{s}|)}
$$

$$
\Rightarrow l_{\text{opt}} = -\log_2 (\min (|\vec{r}|, |\vec{s}|)) \quad (4.7)
$$

Clip level $l_{\text{opt}}$ is calculated in screen space at each intersection $i$ of a ray with the heightfield. The different spaces for computing $l_{\text{opt}}$ are illustrated in figure 4.5.

![Figure 4.5: Illustration of the three different spaces used for computing $l_{\text{opt}}$ by means of a bounding box which corresponds to a DEM texel.](image)

Instead of descending to the full resolution clip level which may cause aliasing, we can terminate ray casting at level $l_{\min} = \max (l_{\text{low}}, l_{\text{opt}})$. The two different LODs $l_{\text{low}}$ and $l_{\text{opt}}$ are visualized in figure 4.6 where each level is encoded by a different color.

Note that this calculation of $l_{\text{opt}}$ is different from the one we presented in [27] which is based on the projection of the box surface area containing $i$ into screen space. Both methods are suitable for LOD determination since there are several ways [1 pp. 687–691], but we prefer the approach in equation (4.7), because it is more similar to the LOD selection presented in equation (3.10).

A problem that was left open in [27] is the transition between two LODs which becomes apparent as the virtual camera moves during user interaction or animations. We solved this issue by linearly interpolating between the height values as follows:

Let $h$ denote the height of the grid cell’s bounding box for which an intersection $i$ at the clip level $l_{\min}$ has been detected. The height value $h_p$ of the parent box is sampled and used
4.2 GPU based Single-pass Ray Casting Using Clipmaps

Figure 4.6: The two different LODs $l_{\text{low}}$ and $l_{\text{opt}}$ are used to terminate the ray traversal and to avoid aliasing. Different colors encode different levels. Their order becomes visible in the right image in the almost planar region on the left. Note that regions of higher elevation are assigned lower levels of detail (right) since they cover more screen pixels.

Together with $h$ and the fractional part $\tau$ of $l_{\min}$ to compute $\bar{h}$ according to equation (4.8).

$$\bar{h} = h + \tau \cdot (h_p - h), \quad \tau = l_{\min} - \lfloor l_{\min} \rfloor$$

(4.8)

The final intersection $i$ is then calculated in equation (4.2) with $h$ being replaced by $\bar{h}$. As desired, the interpolation has no visual effect, if the box has the same height as its parent. Transitions between different LODs are only perceivable during animations and user interaction, especially in areas where the difference $h_p - h$ is large. The interpolation scheme described above makes these transitions less abrupt and is illustrated in figure 4.7.

Figure 4.7: The heights of the cell’s bounding box and its parent are linearly interpolated by using the fractional part $\tau$ of $l_{\min}$ to determine the height value of the final intersection. Actually, the value $\tau = 1$ does not occur, but is included for illustration.

4.2.4 Refinement of Block-sampled Heightfield Reconstruction

As pointed out by Oh et al. in [55], the point sampled DSMs and their treatment as boxes results in blocky images which from a closeup view remind of models built of bricks (see figure 4.8(a)). Because this effect may be unwanted in most applications, we also implemented two refinement methods to obtain smooth surfaces. Both refinement methods are applied after the intersection $i$ on the bounding box surface has been determined as described in section 4.2.2.

The first method is the one presented by Oh et al. [55] and relies on linear interpolation of two samples obtained from the linearly interpolated heightfield, which are taken at a
4 Digital Surface Model Rendering

Figure 4.8: Demonstration of the improvement in surface quality achieved by different refinement methods.

distance of one half cell from $i$ in forward respectively backward direction along the ray. This method works quite well and has hardly any effect on the overall performance on modern GPUs, but in our implementation, some defects – presumably caused by numerical inaccuracies – on surfaces with steep slopes remain, as shown in figure 4.8(b). Despite these small defects, which are barely noticeable during animations or from farther viewing distances, the surfaces look much smoother.

Our second method uses Hermite bicubic surfaces to improve the reconstruction of the heightfield. Let $(i_u, i_v)$ denote the projection of the point $i$ onto the grid of the heightfield at which ray casting has been terminated. Furthermore let $C$ denote the height value of the cell containing $(i_u, i_v)$ and $SW, S, SE, E, NE, N, NW, W$ the height values of the neighboring cells, starting at the left lower cell and enumerating them in counterclockwise order (see figure 4.9). We interpret the four points given in equations 4.9 as the corners of

Figure 4.9: Construction scheme for a Hermite bicubic patch from $3 \times 3$ heightfield samples surrounding the projection of intersection point $i$ on the bounding box (left) and the resulting patch (right).

62
a Hermite bicubic surface patch.

\[
\alpha = (\alpha_s, \alpha_t, \alpha_h) = ([i_u], [i_v], \min (SW, S, C, W))
\]

\[
\beta = (\beta_s, \beta_t, \beta_h) = ([i_u] + 1, [i_v], \min (S, SE, E, C))
\]

\[
\gamma = (\gamma_s, \gamma_t, \gamma_h) = ([i_u], [i_v] + 1, \min (W, C, N, NW))
\]

\[
\delta = (\delta_s, \delta_t, \delta_h) = ([i_u] + 1, [i_v] + 1, \min (C, E, NE, N))
\]

Each patch is parametrized along the grid axes by \( s, t \in [0, 1] \subset \mathbb{R} \), and the height \( h(s, t) \) on the surface patch is given in equation (4.10).

\[
h(s, t) = (s^3 \ s^2 \ s \ 1) \cdot H \cdot G \cdot H^T \cdot (t^3 \ t^2 \ t \ 1)^T
\]

\[
H = \begin{pmatrix}
2 & -2 & 1 & 1 \\
-3 & 3 & -2 & -1 \\
1 & 0 & 1 & 0 \\
\end{pmatrix}, \quad G = \begin{pmatrix}
\alpha_h & \gamma_h & \frac{\partial \alpha_h}{\partial u} & \frac{\partial \gamma_h}{\partial v} \\
\beta_h & \delta_h & \frac{\partial \beta_h}{\partial u} & \frac{\partial \delta_h}{\partial v} \\
\frac{\partial \alpha_h}{\partial u} & \frac{\partial \gamma_h}{\partial u} & \frac{\partial^2 \alpha_h}{\partial u^2} & \frac{\partial^2 \gamma_h}{\partial u \partial v} \\
\frac{\partial \beta_h}{\partial u} & \frac{\partial \delta_h}{\partial u} & \frac{\partial^2 \beta_h}{\partial u^2} & \frac{\partial^2 \delta_h}{\partial u \partial v}
\end{pmatrix}
\]

Superscript \( T \) denotes transposed matrices and vectors, and \( H \) and \( G \) are the Hermite basis matrix and Hermite geometry matrix respectively. Their derivations can be found, for instance, in [29, pp. 483–488, 516–520]. In [27], we approximated the partial derivatives which define the tangential planes on the patch by using forward respectively backward differences and by making the simplifications in equations (4.11) for the first order derivatives.

\[
\frac{\partial \alpha_h}{\partial u} = \frac{\partial \gamma_h}{\partial u} \approx C - W, \quad \frac{\partial \beta_h}{\partial u} = \frac{\partial \delta_h}{\partial u} \approx E - C,
\]

\[
\frac{\partial \alpha_h}{\partial v} = \frac{\partial \beta_h}{\partial v} \approx C - S, \quad \frac{\partial \gamma_h}{\partial v} = \frac{\partial \delta_h}{\partial v} \approx N - C
\]

(4.11)

This causes discontinuities at the patch boundaries, although they are hardly perceivable during rendering. However, these discontinuities become apparent in closeup views and for certain viewing angles. To achieve true \( C^1 \) continuity between the patches, we compute the first order partial derivatives \( \frac{\partial}{\partial u} \) as given in equations (4.12).

\[
\frac{\partial \alpha_h}{\partial u} \approx \frac{C + S - (W + SW)}{2}, \quad \frac{\partial \beta_h}{\partial u} \approx \frac{E + SE - (C + S)}{2}
\]

\[
\frac{\partial \gamma_h}{\partial u} \approx \frac{N + C - (NW + W)}{2}, \quad \frac{\partial \delta_h}{\partial u} \approx \frac{NE + E - (N + C)}{2}
\]

(4.12)

The computations of the derivatives \( \frac{\partial}{\partial v} \) are given in equation (4.13).

\[
\frac{\partial \alpha_h}{\partial v} \approx \frac{C + w - (S + SW)}{2}, \quad \frac{\partial \beta_h}{\partial v} \approx \frac{E + C - (SE + S)}{2}
\]

\[
\frac{\partial \gamma_h}{\partial v} \approx \frac{N + NW - (C + W)}{2}, \quad \frac{\partial \delta_h}{\partial v} \approx \frac{NE + N - (E + C)}{2}
\]

(4.13)
The mixed second order partial derivatives are approximated by equations (4.14).

\[
\begin{align*}
\frac{\partial^2 \alpha_h}{\partial u \partial v} & \approx C - W - S + SW, \\
\frac{\partial^2 \beta_h}{\partial u \partial v} & \approx E - C - SE + S, \\
\frac{\partial^2 \gamma_h}{\partial u \partial v} & \approx N - NW - C + W, \\
\frac{\partial^2 \delta_h}{\partial u \partial v} & \approx NE - N - E + C.
\end{align*}
\]

The visual difference between the simplified method for computing the first order partial derivatives in equation (4.11) and the method given above is shown in figure 4.10. Although the matrix \( G \) at each grid cell respectively texel of the clipmap storing the heightfield is constant, we calculate it directly in the fragment shader as needed.

The pair of parameters \((s,t)\), which corresponds to an intersection of the ray with the bicubic patch instead of the bounding box, is determined by a second ray casting. Starting at \( i \) on the bounding box surface, the ray \( p = i + k \cdot \hat{d} \) is advanced to its exit point on the box at a fixed step width until it either hits the bicubic patch, i.e., \( p_z \leq h(s,t) \), or it leaves the domain of the box without intersection. In the latter case, we treat \( i \) as an entry point on the proxy geometry and proceed with the accelerated ray casting process described in section 4.2.2 from the current level. We found a subdivision into 16 steps for traversing the bounding box of a cell to be completely sufficient, independent of the clip level \( l \). Fewer subdivision steps expose defects by missed intersections, whereas increasing the number of subdivision steps only reduces frame rates without further improving the reconstruction of the surface.

Besides their simplicity and the possibility to calculate all the relevant information in the fragment shader, we decided to use Hermite bicubic patches because we wanted to ensure that the surface remains inside the bounding boxes of the BVH. By constructing the patches as described above, we can ensure that they stay completely inside the bounding boxes as we control the defining tangential planes. The direct usage of forward and backward differences according to equations (4.12) and (4.13) avoids any scaling of the tangents and therefore leads to desired \( C^1 \) continuity between neighboring patches, because their...
tangents have the same direction and magnitude (cf. \cite{29} pp. 517–520). The most severe drawback of this method is its high computational cost, although we still may achieve interactive frame rates (see section \ref{4.3.2}). Furthermore, as this method ensures that the height of each patch is less or equal than the height of its bounding box, and the tangents are not scaled, isolated peaks in the heightfield become clearly flattened as can be seen in figure \ref{4.8}(c).

However, both refinement methods presented in this section rely on interpolation of point sampled data on a regular grid, and only serve in making the resulting renderings visually more appealing. Besides, even if it might appear to be sufficient to apply refinement only in cases when the viewer is close to a highly detailed area where the block sampled nature of the data becomes apparent, we refine the surface at all discrete LODs, because the transition between large distant boxes and smooth surfaces is rather disturbing during animations. In addition, the lighting conditions on smooth surfaces and blocks are different due to distinct surface normals.

\subsection*{4.2.5 Sampling Color Textures}

The mapping of color information onto the surface obtained by ray casting as described above works on the assumption that the DSM provides a color layer with an orthophoto texture or a similar texture, e.g., a schematic map. As long as the color layer and DEM layer cover the same area in world space and have the same resolution, one heightfield sample corresponds to one color sample. In this case, only texture data on the tops of grid cell bounding boxes of corresponding DEM layer texels are available, but the color value is also used for the lateral faces and hence for the entire box. Therefore the final clip level $l_{\text{min}}$ where ray casting has terminated can also be used to perform a texel precise texture lookup in the color layer of the tile that contains the intersection with the DEM.

If the color and DEM layers are of different resolutions, the determination of color values by the FCM can be the same as described in section \ref{3.3.4}. Calculating the clip level $l_{\text{opt}}$ which best avoids texture aliasing can be integrated into the ray casting procedure or can be performed separately at the final hit point $i$ in the heightfield.

The most severe drawback of using the color from a single texel for the entire box of a heightfield sample is that the lateral faces of elevations do not depict information as expected. By viewing surfaces from oblique angles, a viewer might expect to see details on these faces, e.g., frontages of buildings in urban areas or different layers of rock on canyon walls, which are not available due to the nature of the employed color textures. This problem is illustrated in figure \ref{4.11} and addressed in chapter \ref{6}.

\subsection*{4.3 Performance Results}

The performance of our technique is demonstrated for renderings of four different data sets of fixed size listed in table \ref{4.1}. The data set City 2 was acquired by means of photogram-
Figure 4.11: Orthophoto textures do not contain color information about lateral surfaces. This becomes apparent, for instance, in urban areas if the virtual camera is lowered towards street level (right).

<table>
<thead>
<tr>
<th>name</th>
<th>extent [km]</th>
<th>W × H</th>
<th>L</th>
<th>scale</th>
<th>size DEM</th>
<th>size color texture</th>
<th>time [min]</th>
</tr>
</thead>
<tbody>
<tr>
<td>City 1</td>
<td>1.4 × 1.0</td>
<td>5600 × 4000</td>
<td>5</td>
<td>1.0</td>
<td>133 MB</td>
<td>99 MB</td>
<td>0:54</td>
</tr>
<tr>
<td>City 2</td>
<td>20.9 × 26.3</td>
<td>83600 × 105200</td>
<td>9</td>
<td>1.0</td>
<td>31.6 GB</td>
<td>–</td>
<td>3:34</td>
</tr>
<tr>
<td>ETOPO1</td>
<td>≈ 40075.0 × 19070.0</td>
<td>21600 × 10800</td>
<td>7</td>
<td>10.0</td>
<td>1.3 GB</td>
<td>–</td>
<td>9:53</td>
</tr>
<tr>
<td>Blue Marble</td>
<td>≈ 40075.0 × 19070.0</td>
<td>86400 × 43200</td>
<td>9</td>
<td>10.0</td>
<td>19.2 GB</td>
<td>14.4 GB</td>
<td>13:10</td>
</tr>
</tbody>
</table>

Table 4.1: Properties of the different data sets used to evaluate performance. L denotes the total number of clip levels which have been created, W × H is the grid size at level 0 respectively the size a single texture would have. Column time contains the durations of the virtual camera flights for our evaluation in minutes.

metric methods from aerial images. City 1 depicts a small area in City 2 in which we have a color texture available that has been derived from orthographic aerial images. The data sets ETOPO1 [2] and Blue Marble [51] depict the entire Earth and are both derived in large parts from SRTM data [52], but ETOPO1 also contains bathymetric data, whereas Blue Marble possesses a color texture derived from satellite images. In addition, we created a fifth data set which contains elevation data of almost the entire surface of Mars and has an extent of 46080 × 22528 samples. The data from Mars were acquired from the Mars Orbiter Laser Altimeter experiment in the context of the Mars Global Surveyor Mission by NASA [61]. We did not use the Mars data set in our evaluation, because its properties are very similar to those of the ETOPO1 data set. Example renderings of the data sets are shown in figure 4.12.

When being sampled in the fragment shader, the height values are scaled by factors given in column scale in order to avoid flattened surfaces. Shallow surfaces do not challenge our ray caster because less mutual occlusions lead to fewer level changes in the BVH during ray traversal.

4.3.1 Evaluation Setup and Results

We used tile sizes of 512 × 512 texels, active area sizes of 5 × 5 tiles and clip area sizes of 7×7 for all data sets in our tests. The near and far plane of the virtual camera were set to 1.0 and 2000.0 units, respectively. Heightfield layers consist of single channel 32-bit floating point textures, and color texture layers consist of 24-bit RGB textures. The results were
Figure 4.12: Example renderings of the four data sets which we used in our performance evaluations and the Mars data set. Color textures are only available for City 1 and Blue Marble. ETOPO1 was rendered using a pseudo topographic color map and the rendering of City 2 depicts shadows which were computed by means of secondary ray casting. The colors in the rendering of Olympus Mons on Mars were chosen according to artistic considerations.
recorded during virtual camera flights along fixed paths over the heightfields on a desktop computer with an Intel i7 860 CPU at 2.8 GHz, 6 GB RAM, NVIDIA GeForce GTX 470 graphics adapter with 1280 MB dedicated video memory and Windows 7 OS (system A). To make our results comparable to the results reported in [18], we additionally ran the same tests on a second desktop computer (system B) with a hardware configuration more similar to theirs (Intel Q6600 CPU at 2.4 GHz, 4 GB RAM, NVIDIA GeForce GTX 285 with 1024 MB dedicated video memory and Windows 7 OS). Table 4.2 shows the results for different screen resolutions on system A and system B in terms of frames per second (fps). The frame rates take into account the delays caused by updating the tile caches in main memory and video memory as described in section 3.3. The times for rendering the given number of frames are the same on both systems and are denoted by column time in table 4.1.

<table>
<thead>
<tr>
<th>data set</th>
<th>resolution [pixel]</th>
<th>frames</th>
<th>min. [fps]</th>
<th>avg. [fps]</th>
</tr>
</thead>
<tbody>
<tr>
<td>City 1</td>
<td>1024 x 768</td>
<td>5450</td>
<td>6.2</td>
<td>100.9</td>
</tr>
<tr>
<td></td>
<td>1280 x 1024</td>
<td>3464</td>
<td>5.8</td>
<td>64.1</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
<td>2217</td>
<td>5.1</td>
<td>41.0</td>
</tr>
<tr>
<td>City 2</td>
<td>1024 x 768</td>
<td>25624</td>
<td>5.6</td>
<td>119.7</td>
</tr>
<tr>
<td></td>
<td>1280 x 1024</td>
<td>16700</td>
<td>5.6</td>
<td>78.0</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
<td>11189</td>
<td>5.1</td>
<td>52.3</td>
</tr>
<tr>
<td>ETOPO1</td>
<td>1024 x 768</td>
<td>105747</td>
<td>6.3</td>
<td>178.5</td>
</tr>
<tr>
<td></td>
<td>1280 x 1024</td>
<td>66869</td>
<td>5.9</td>
<td>112.9</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
<td>42907</td>
<td>4.9</td>
<td>72.4</td>
</tr>
<tr>
<td>Blue Marble</td>
<td>1024 x 768</td>
<td>121844</td>
<td>3.3</td>
<td>154.2</td>
</tr>
<tr>
<td></td>
<td>1280 x 1024</td>
<td>75721</td>
<td>4.0</td>
<td>95.8</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
<td>50028</td>
<td>1.8</td>
<td>63.3</td>
</tr>
</tbody>
</table>

(a) System A

<table>
<thead>
<tr>
<th>data set</th>
<th>resolution [pixel]</th>
<th>frames</th>
<th>min. [fps]</th>
<th>avg. [fps]</th>
</tr>
</thead>
<tbody>
<tr>
<td>City 1</td>
<td>1024 x 768</td>
<td>3848</td>
<td>5.1</td>
<td>71.2</td>
</tr>
<tr>
<td></td>
<td>1280 x 1024</td>
<td>2450</td>
<td>4.5</td>
<td>45.3</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
<td>1574</td>
<td>5.1</td>
<td>29.1</td>
</tr>
<tr>
<td>City 2</td>
<td>1024 x 768</td>
<td>18174</td>
<td>3.6</td>
<td>52.3</td>
</tr>
<tr>
<td></td>
<td>1280 x 1024</td>
<td>12327</td>
<td>3.6</td>
<td>57.6</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
<td>8290</td>
<td>3.3</td>
<td>38.7</td>
</tr>
<tr>
<td>ETOPO1</td>
<td>1024 x 768</td>
<td>75790</td>
<td>5.2</td>
<td>127.9</td>
</tr>
<tr>
<td></td>
<td>1280 x 1024</td>
<td>48027</td>
<td>5.2</td>
<td>81.1</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
<td>31134</td>
<td>4.7</td>
<td>52.5</td>
</tr>
<tr>
<td>Blue Marble</td>
<td>1024 x 768</td>
<td>95285</td>
<td>2.8</td>
<td>120.6</td>
</tr>
<tr>
<td></td>
<td>1280 x 1024</td>
<td>63409</td>
<td>2.5</td>
<td>80.3</td>
</tr>
<tr>
<td></td>
<td>1920 x 1080</td>
<td>43314</td>
<td>1.0</td>
<td>54.8</td>
</tr>
</tbody>
</table>

(b) System B

Table 4.2: Performance results of our rendering technique on two computers with different hardware configuration.

4.3.2 Performance with Surface Refinement

All values given in table 4.2 were obtained without any of the surface refinement methods described in section 4.2.4. The impact on the rendering speed and the relative loss in performance when using surface refinement in our implementation is shown in table 4.3. These data were acquired from another evaluation of the same camera flight through the City 2 data set on system A and system B, because this data set has high spatial frequencies in the rendered regions and is the most challenging for our ray caster.

4.4 Discussion

The results in table 4.2 show that – in accordance with the results of Dick et al. [18] – very large DSMs can be rendered in real time by using only ray casting and acceleration data structures. Due to its close relation to mipmaps, the FCM proved to be well suited for this
4.4 Discussion

Table 4.3: Impact of surface refinement methods on the performance in terms of average frames per second for City 2 data set and the loss compared to unrefined rendering.

<table>
<thead>
<tr>
<th>Method</th>
<th>1024 × 768</th>
<th>1280 × 1024</th>
<th>1920 × 1080</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>96.2 (-19.6%)</td>
<td>62.8 (-19.5%)</td>
<td>42.0 (-19.7%)</td>
</tr>
<tr>
<td>Bicubic</td>
<td>36.8 (-69.3%)</td>
<td>24.3 (-68.8%)</td>
<td>16.6 (-68.3%)</td>
</tr>
</tbody>
</table>

(a) System A

<table>
<thead>
<tr>
<th>Method</th>
<th>1024 × 768</th>
<th>1280 × 1024</th>
<th>1920 × 1080</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>75.4 (-12.4%)</td>
<td>49.8 (-13.5%)</td>
<td>33.7 (-12.9%)</td>
</tr>
<tr>
<td>Bicubic</td>
<td>36.4 (-57.1%)</td>
<td>17.4 (-69.8%)</td>
<td>11.9 (-69.3%)</td>
</tr>
</tbody>
</table>

(b) System B

task, because DEM data and color textures can be handled in the same way and the implementation of DEM layers as maximum mipmaps is straightforward. Although the hybrid approach of Dick et al. [19] performs faster rendering, it appears to be less flexible, because it requires to select representative tiles from the data set and views of the scene during its training phase.

As expected, table 4.3 shows that when using bicubic surface refinement the loss in performance is much larger than with the linear method, but even at the highest resolution we still achieve interactive frame rates. The linear method may expose some defects, but offers a good compromise between quality and speed at higher screen resolutions. As expected, when using bicubic surface refinement, the loss in performance is much larger than with the linear method, but even at the highest resolution we still achieve interactive frame rates. Besides, the refinement of the reconstructed surface only pays for coarse resolution DSMs respectively low grid densities where the block structure becomes apparent. The differences in the frame rates between the two city data sets and the two earth data sets result from different grid densities.

As the performance of GPU programs can still be significantly degraded by many conditional code executions, e.g., caused by if-statements [36], it is advisable to eliminate branching in dependency on the direction of a ray within the main ray casting loop. An alternative to using switch-statements within the ray casting loop as described in section 4.2.2 would be to duplicate the shader code for each of the overall eight possible branches that result from the direction of the ray. This is done, for instance, in [18], and we implemented this technique for test purposes, but we found that the two implementations yield the same performance on the employed graphics hardware.
5 GPU-based DSM Synthesis

In the previous chapter, methods for managing and rendering large DSMs have been presented, but the employed data were previously acquired and static. Our approach is designed to handle DEM and texture data of time-varying extension in order to allow instant updates and rendering, and we can generate these data automatically from aerial photographs by using photogrammetric methods. For this purpose, the source images should be available for processing directly after their acquisition and the DSM creation needs to be as fast as possible.

In this chapter we present a simple GPU-based method for generating DEM data and matching orthophoto textures from multiple vertical aerial photographs. The employed method yields credential results within few seconds and is similar to the approaches presented in [86] and [87]. It relies on a space sweep as presented in [14], and performs the essential steps in the DSM tile generation process in our FCM implementation (see section 3.2.5).

5.1 Methods for Stereo Matching

A dense DEM on a regular grid can be created from a set of aerial images by applying stereo matching techniques known from the field of computer vision. According to [76, p. 535], aerial image photogrammetry is the origin of this field of research, and stereo matching is subject of numerous books and publications. A comprehensive summary of the different methods would exceed the scope of this thesis. Hence we confine ourselves to a brief overview of the methods presented in [69] and [76] and describe their basic principles.

Stereo matching usually comprises the processing steps illustrated in figure 5.1 and requires to find regions in the input images, i.e., individual pixels or groups of adjacent pixels, which depict the same points of an object (features). In conjunction with the intrinsic and extrinsic camera parameters, the locations within the images allow to calculate the location of the corresponding features in world space (see section 2.5). The distances of these features from the center of projection (CoP) of the containing image are called depth values and may be stored in a depth map. The depth values of the objects depicted in the input images can be considered as elevations above a plane of reference, and a corresponding depth map can be identified with a digital elevation model (DEM).

As an alternative, the inverse depth of a feature can be computed, because this is frequently a more well-conditioned parameter for cameras [76, p. 49]. The inverse depth of a point in world space is related to its disparity, which is the distance between two positions.
of that point in a pair of images [76] pp. 539–540]. Analog to depth maps, disparities can be stored in a disparity map and are frequently measured in pixels. Depending on the application, depth values and disparities may be referenced to one of the input images used for stereo matching or to an intermediate image of a virtual camera.

![Figure 5.1: Illustration of the stereo matching process according to [69].](image)

One method for matching images employs epipolar geometry. Given two images which depict the same object, this method exploits the fact that the same feature depicted in one of the two images is located on the epipolar line in the other image (see figure 5.2). This limits the search space where correspondences can be found and thus allows to efficiently find matching features, especially if the images are warped in such a way that epipolar lines coincide with image scan lines (rectification). To determine the correspondence for a certain pixel $P$ from image $I_1$ in image $I_2$, a matching cost function or error function $C$ is evaluated at each pixel along the corresponding epipolar line in $I_2$. $C$ gives a measure of the similarity of pixels, and thus indicates whether the pixels may be related to the same feature. Hence, the pixel with the lowest associated matching costs corresponds best to $P$ in $I_1$, and the pixels are therefore likely to be related to the same feature. By means of the two locations in the two different images, the feature’s position in world space can then be computed by using equation (2.13) given in section 2.5.1. More details about epipolar geometry and rectification can be found in [76] pp. 537–540, for instance.

![Figure 5.2: The projection $p_2$ of point $p$ is located in image 2 on the epipolar line (blue, dashed), which starts at $e_2$ and contains all points on the ray emanating from the center of projection of image 1 (CoP$_1$) through $p$, and its projection $p_1$ in image 1. $e_1$ and $e_2$ are the so called epipoles, and are given by the intersections of the ray from CoP$_1$ to CoP$_2$ with the image planes of image 1 and image 2, respectively.](image)

Another approach to obtain depth values from a set of two or more images is to choose a plane of reference in world space for measuring elevation, e. g., a plane parallel to one of the image planes or a ground plane, and to sweep the space over a predefined distance by a sweeping plane, which is oriented parallel to that plane of reference. Assume that the plane of reference is the xy-plane, and that the sweeping plane is moving along the z-axis.
Each location \( r = (r_x, r_y, r_z) \) from a section of interest in the sweeping plane is projected back into the input images by using their associated camera matrices (see section 2.5.1). The similarity between the resulting pixel values for the projections of \( r \) into the images is determined by evaluating a matching cost function \( C \) as described above. The obtained matching costs at \( r \) are stored for the plane’s current sweep position. At location \((r_x, r_y)\) in the plane of reference, the z-position \( z' \) of the sweeping plane with lowest matching costs then indicates a location in world space for which the related pixels from input images have similar values. Hence, \((r_x, r_y, z')\) is likely the location of the same feature, which is depicted in most of, or even all of the input images. The space sweep\(^7\) approach requires no previous rectification of the input images, and allows to match more than two images \([14, 76, \text{p. 559}]\) (see section 5.2).

Finding the minimal matching costs can be achieved in both cases by means of local or global optimization methods. Local optimization methods usually rely on the aggregation of the matching costs over sections (support windows) in the disparity image and on choosing the disparity with lowest associated costs. Global optimizations try to optimize an energy function with respect to the costs and some smoothness constraints between the disparities \([76, \text{pp. 548–558}]\). While the latter methods usually yield better results \([76, \text{p. 554}]\), the former ones appear to be more appropriate for real-time applications \([87]\). In image regions with low densities of features, e. g., monochrome or structureless surfaces, and in areas of disparity discontinuities caused by occlusions of the depicted objects, matches are more difficult to find by local methods based on support windows of fixed size and constant shape \([41]\).

Furthermore, if only two images are used for matching, correspondences cannot be detected if a feature is missing or invisible in one of the images. By using \( n > 2 \) images which depict the same object from slightly different perspectives, better results can be achieved, because more information is available (cf. \([76, \text{pp. 558–562}]\)). The results of the matching process may additionally be refined by using robust statistics in order to remove outliers from the resulting set of reconstructed world positions \([14]\). Another possibility is to cross-check the results of one matching with the results from another matching of the same images, which uses a different image as reference for computing depths respectively disparities \([69, 87], 76, \text{p. 550}]\).

5.1.1 Matching Cost Functions

Matching cost functions are a measure of the similarity of two or more values and indicate a likelihood of correspondence \([76, \text{p. 546}]\). The costs can be calculated based on grayscale intensities, intensities for each color component of a pixel, or other photometric quantities, e. g., luminance \([76, \text{p. 384}]\). In order to yield reasonable results from matching cost functions, corresponding pixels in different images must have the same or at least very similar

\(^7\)The space sweep is called plane sweep in some works from the domain of computer vision, e. g., \([76, 86, 87]\). This may not be confused with the term plane sweep as used for 2D space sweeps which employ lines in order to sweep planes. This latter kind of plane sweep is used, e. g., in the algorithm for computing intersections of 2D line segments as presented in \([17, \text{pp. 19–29}]\).
values (*photo consistency*). Usually this requires the depicted object surfaces to be approximately Lambertian, i.e., they need to be ideally diffuse surfaces on which the reflected radiance, and hence the apparent brightness, depends only on the cosine of the angle of irradiance of light. This criterion is not met, for instance, by surfaces showing specular reflections since the reflected radiance depends on the angle of view [1 pp. 110–111].

Let $I_k(u, v)$ denote the pixel value at a discrete location $(u, v)$ in image $I_k$ with $u, v, k \in \mathbb{N}$ and let $(u, v)$ furthermore be valid, i.e., the coordinates are not located outside the image. Given two different images $I_0$ and $I_1$, a widespread method is to calculate the *sum of squared differences (SSD)* given in equation (5.1) as a measure of similarity.

$$\text{SSD} = \sum_j \sum_i (I_1(i, j) - I_0(i, j))^2 \quad (5.1)$$

The range of the sums for adding up costs depends on the application, but is usually a rectangular region of interest in the images (*support region* or *support window*), or even the entire image. If the two images $I_0, I_1$ do not have the same size, or if the support windows are located at different positions within the two images, an offset $(o_u, o_v)$ needs to be added to one of the sample positions.

The *sum of absolute differences (SAD)* in equation (5.2) is another popular measure, because of its low computational costs and as it is more robust against outliers [76, p. 384].

$$\text{SAD} = \sum_j \sum_i |I_1(i, j) - I_0(i, j)| \quad (5.2)$$

*Normalized cross-correlation (NCC)* as given in equation (5.3) is frequently employed for template matching [34, pp. 869–872] and in the area of signal processing [56, pp. 115–131] for finding similarities between two signals.

$$\text{NCC} = \frac{\sum_j \sum_i (I_1(i, j) - \bar{I}_1)(I_0(i, j) - \bar{I}_0)}{\sqrt{\sum_j \sum_i (I_1(i, j) - \bar{I}_1)^2} \sqrt{\sum_j \sum_i (I_0(i, j) - \bar{I}_0)^2}} \quad (5.3)$$

$$\bar{I}_k = \frac{1}{W_k \cdot H_k} \sum_j \sum_i I_k(i, j), \quad k \in \{0, 1\}$$

$W_k$ and $H_k$ denote the width and height of image $I_k$ in pixels, respectively. The values are in $[-1, 1] \subset \mathbb{R}$ where 1 indicates maximal correlation, i.e., highest similarity, and -1 corresponds to total dissimilarity between the two input signals or images. As this measure involves convolution, it is computationally expensive but can be used as well for stereo matching.

More information about matching cost functions can be found, for instance, in [76 pp. 546–548, 384–387].
5.1.2 Cost Aggregation and Support Window Size

Local optimization methods usually aggregate the matching costs over a support window of size $W_x \times W_y$ where $W_x, W_y \in \mathbb{N}$ and $W_x, W_y \geq 1$ [76, p. 548]. A small window size is more likely to produce matching errors, whereas the usage of a large window is less precise, especially if it is placed over areas in an image which contain occluding object boundaries [86]. However, larger windows are accompanied by higher computational costs. As shown in [86], square support windows of power of two sizes for cost aggregation can be efficiently realized by using graphics hardware. By automatically generating mipmap for each input image, each texel at mipmap level $\lambda \geq 0$ corresponds to a window of size $2^\lambda \times 2^\lambda$. The matching costs can be calculated at different levels and summed up to take advantage of the precision obtained by smaller support windows, and the robustness of larger ones. According to [86–88], large support windows are mainly useful in two-image stereo matching approaches and may not be necessary if multiple images are used. In [33] a comparison of different aggregation schemes is presented, and further methods can be found in [76, pp. 548–551].

5.2 DEM Generation Using Space Sweep

In our application, we obtain DEM values on a regular grid for individual FCM tiles at the base clip level 0 and store them in the DEM layer (see section 4.2.1). For this purpose, we employ a space sweep approach, because it was originally developed in the context of DEM creation from aerial images and appears to be well-suited for this task [14]. The technique is neither restricted to a certain type of input images nor to a particular application of stereo matching, like DEM data generation from aerial images, for instance. Multiple images can be matched without previous rectification and efficient GPU-based implementations can be realized on commodity graphics hardware [87]. It is furthermore possible to sweep the space in multiple directions and to combine the results [30]. A space sweep is also used, for instance, in [32] to obtain an initial depth map in a more complex system for two-image stereo matching.

The space sweep as used in our application is performed as follows: Given a finite set $\{I_0, \ldots, I_{n-1}\}, n \geq 2$ of vertical aerial images which depict the same scene from different points of view, each point of the scene is located at different locations in images $I_k, 0 \leq k < n$. Let $M_k = P_k \cdot V_k$ denote the $4 \times 4$ matrix composed of the view matrix $V_k$ and the projection matrix $P_k$ of the camera which captured image $I_k$ (see section 2.5.1).

A plane parallel to the xy-plane is moved along the z-axis, where $l \in \mathbb{N}$ and $0 \leq l \leq L$. $L$ is determined for some predefined height resolution $\Delta z$ by equation (5.4). $z_{\text{min}}$ and $z_{\text{max}}$ with $z_{\text{min}} \leq z \leq z_{\text{max}}$ may be chosen according to the underlying surface, e.g., if the minimum and maximum elevations are known.

$$L = \left\lceil \frac{z_{\text{max}} - z_{\text{min}}}{\Delta z} \right\rceil \in \mathbb{N} \quad (5.4)$$
In this way, we yield a series of \( L + 1 \) planes \( \Pi_l \) at discrete locations \( s_z = l \cdot \Delta z + z_{\text{min}} \) along the z-axis. Let \((s_x, s_y)\) denote a position on a rectangular regular grid with spacing \((g_x, g_y)\) based in the xy-plane. Each discrete sample position \( s = (s_x, s_y, s_z) \) is projected into the image plane of each image \( I_k \) according to equation (2.6) in section 2.5.1 by using \( M_k \) in order to obtain its 2D image coordinates \((u_k, v_k)\). The corresponding pixel value is denoted by \( p_k = I_k(u_k, v_k) \) and is probably the result of some sort of interpolation, e.g., bilinear interpolation or nearest neighbor sampling, because \( s \) will normally not be mapped to integral pixel positions.

For simplicity, assume that \( s \) is not projected outside the boundaries of any image \( I_k \), i.e., all \( p_k \) are valid, and that \( p_k \) is furthermore a single scalar value like grayscale intensity or luminance. The matching costs \( C_k(s) \) for each \( p_k \) are calculated according to equation (5.5) as the SAD of \( p_k \) and all other values \( p_i \).

\[
C_k(s) = \sum_{i=0}^{n-1} |p_i - p_k| \tag{5.5}
\]

Since in case of \( i = k \) the difference \( (p_i - p_k) \) is zero, it does not contribute to the costs and hence does not require any special treatment. The aggregated costs \( AC(s) \) in equation (5.6) at sample position \( s \) on the current plane \( \Pi_l \) are obtained as the sum of the matching costs \( C_k(s) \) for all \( n \) images, i.e., the sum of sum of absolute differences (SSAD) for all \( p_k \).

\[
AC(s) = \sum_{j=0}^{n-1} C_j(s) = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} |p_i - p_j| \\
= \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} |I_i(u_i, v_i) - I_j(u_j, v_j)| \tag{5.6}
\]

In equation (5.6), the costs for matching two pixel values \( p_k \) and \( p_i \) are summed twice, because \( |p_i - p_k| = |p_k - p_i| \). But since we only need to compare the relative costs between image pixels, and since the costs are duplicated for all of them, this duplication does not have any negative effect on the matching process itself\(^8\).

If the corresponding surface is approximately Lambertian, the projection of the same feature from world space into the different images \( I_k \) should result in very similar pixel values \( p_k \) and the residual error \( (p_i - p_j) \) should be almost zero. The plane \( \Pi_{l_{\text{best}}} \) containing the best correspondence between the pixels from all images at position \((s_x, s_y)\) in the xy-plane is then given by

\[
l_{\text{best}} = \arg \min_{0 \leq l \leq L} AC(s_x, s_y, l \cdot \Delta z + z_{\text{min}})
\]

and the resulting DEM height value is \( z_{\text{DEM}} = l_{\text{best}} \cdot \Delta z + z_{\text{min}} \).

---

\(^8\)Equation (5.6) is in fact convenient for our GLSL implementation, because the corresponding for-loops are of fixed lengths and may hence be unrolled by a GLSL compiler, if \( n \) is constant.
5.2 DEM Generation Using Space Sweep

5.2.1 Input Image Selection

The images employed for DSM creation by stereo matching are returned by the 3D spatial index of the FCM by means of an intersection query (see chapter 3.2.5). If the images are all vertical images, their projections according to the associated camera matrices \( M_k \) will be located on the sweeping plane. But oblique aerial images may also be stored in the spatial index and will be returned by a intersection query as well. These images can be easily removed from the set of input images by means of the viewing direction of their associated cameras, so that they are not taken into account for DEM layer generation. However, further changes to the set of input images may be necessary in the following situations:

1. the projection of an image onto the sweeping plane does not overlap the projection of any of the remaining images
2. there are more input images than texture units or video memory available

We call an image which suffers from the first problem a non-overlapping image in contrast to an overlapping image whose projection onto the sweeping plane has non-empty intersection with at least one other image. As the outlines of these projections are all convex quadrilateral polygons, an image can be identified as non-overlapping if its corresponding polygon does not intersect the polygon of any other image from the set. The intersection test is best performed with image projections on the sweeping plane at \( z_{\text{min}} \), i.e., near the ground plane, because the projected areas become smaller as the sweeping plane moves towards the CoPs of the images. Non-overlapping images are removed from a set of input images since they are not suited for stereo matching.

The second situation arises if \( n \) images are available, but only \( N < n \) textures can be processed at a time by the fragment program. This problem can be solved by splitting the set of input images into multiple sets of \( \leq N \) images and by performing the space sweep multiple times per FCM tile at clip level 0, each time with a different subset of images, but without clearing the render targets between the passes.

However, the images need to be redistributed with respect to the mutual overlap of their projections onto the sweeping plane. Otherwise, the resulting subsets might suffer from large numbers of non-overlapping images. These subsets do not need to be disjoint, since an image may be suited for stereo matching with other images that are non-overlapping. This situation is illustrated in figure 5.3 where \( n = 3 \) images are given, but only \( N = 2 \) images may be matched at a time. Such layouts of images are the normal case in traditional aerial photography [42, pp. 144–145] and these situations should hence be manageable in our approach. Therefore, we unfortunately cannot simply create a partition of a set of \( n > N \) images into \( \left\lceil \frac{n}{N} \right\rceil \) disjoint sets.

In order to distribute \( n \) images among a number of sets of \( N < n \) images, we propose the following proceeding:

For each image \( I_k \), the area of intersection \( A_{k,i} \) of its projection with that of another image \( I_i \) is computed and stored in a look-up table. The intersection can be obtained by
clipping the respective bounding polygons against each other, for instance, by using the Sutherland-Hodgman algorithm \[29\] pp. 124–126. If \(A_{k,i}\) is not empty, the polygon which results from clipping is convex and in particular simple, because it is the intersection of two convex polygons (cf. \[17\] p. 46, pp. 66–70). The area \(A_{\text{poly}}\) of a simple polygon can be calculated according to equation (5.7) where \(\chi \geq 3\) is the number of vertices, \((s_i, t_i)\) the respective Cartesian vertex coordinate and \(s_\chi = s_0, t_\chi = t_0\).

\[
A_{\text{poly}} = \frac{1}{2} \sum_{i=0}^{\chi-1} |s_i t_{i+1} - s_{i+1} t_i|
\]

A derivation of the formula in equation (5.7) and a proof of its correctness can be found in \[58\] pp. 20–22, for instance. A series of not necessarily disjoint sets \(S_j\) of images is created where each \(S_j\) may contain at most \(N\) elements. First, all images \(I_k\) are marked as UNUSED, e.g., by using an array of boolean values indexed by \(k\). Starting with \(j = 0\), each \(S_j\) is then constructed as follows: An arbitrary image \(I_{k'}\), which has not been marked as USED, because it has not yet been inserted into any previously created set \(S_i, i < j\), is added to the empty set \(S_j\) and becomes marked as USED afterwards. From the look-up table, successively those images with the largest area of intersection with image \(I_{k'}\) are determined in decreasing order and are added to \(S_j\) until either \(|S_j| = N\), or no more images overlapping \(I_{k'}\) which are marked as UNUSED are left. The overlapping images are added to \(S_j\) regardless of their markers, but are marked as USED afterwards. This process is repeated until all images are marked as USED. Sets with \(|S_j| \leq 1\) are discarded and will not be used as input for DEM generation, since they only contain a single image and thus cannot be used for stereo matching. As the area of intersection of an image \(I_k\) with another image can be determined from the look-up table, it is also possible to ignore images whose amount of overlap with \(I_{k'}\) is below a certain threshold.

The process described above is guaranteed to terminate after at most \(n\) iterations, because during each iteration at least one image becomes marked as USED. If we set \(N = n\), the algorithm can be used furthermore to remove not sufficiently overlapping images from a set of input images for stereo matching. Although this redistribution of images does no longer allow to compute the matching costs between all \(n\) input images, it will create sets of images in which the images have large overlaps and are thus likely to depict the same objects.
5.3 GPU-based Implementation

Our implementation of the space sweep method described in the previous section is similar to the one presented in [87] and relies on OpenGL and GLSL, but employs more modern graphics hardware features. In particular, OpenGL pixel buffers (P-buffers) have been superseded by so called frame buffer objects (FBOs) which essentially perform the same task of redirecting rendering output into textures in video memory (render targets). Since we use multiple input images for matching, we also adopted the idea of using only a certain number of images with lowest matching costs for aggregation, as it is presented in [41]. The resulting DEM and color texture data are stored in the corresponding layers of a single FCM tile at the lowest clip level as described in section 4.2.1 and can be used directly for rendering.

5.3.1 Program Setup

In order to use the GLSL program as described in the next section, it is necessary to set up a virtual orthographic camera which is oriented perpendicular to a rectangular quadrilateral mesh (quad). The quad represents the sweeping plane based in the xy-plane in world space and its only purpose is to generate input fragments for the fragment shader at known z-coordinates. Its size is set to \((S_x, S_y) = (T_{resx}, T_{resy})\) where \((T_x, T_y)\) is the size of a single FCM tile in texels and \((res_{sx}, res_{sy})\) denotes the desired resolution of the resulting DEM layer in texels per unit along the respective axis.

The range \([z_{min}, z_{max}]\) for the space sweep along the z-direction is obtained by setting \(z_{min} = H_{min}\) from the box that is associated with the root node of the R*-tree of the FCM (see section 3.2.5). To ensure that \(z_{max}\) is not located above any of the CoPs of the employed images, it is set to a value which is somewhat smaller than the smallest of all z-coordinates of the CoPs. If the maximum height \(H_{max}\) of the underlying surface is known, \(z_{max}\) may also be set to \(z_{max} = H_{max}\). The number of required rendering passes is equal to the number \(L + 1\) of discrete plane positions on the z-axis and is calculated according to equation (5.4) for a desired height resolution \(\Delta z\).

The z-position of the quad varies from \(z_{min}\) to \(z_{max}\) and must be located inside the frustum of the virtual orthographic camera. This camera is placed at \(\left(\frac{S_x}{2}, \frac{S_y}{2}, (z_{max} - z_{min}) + 1\right)\) and looks along the negative z-axis. It is oriented such that its up-vector \(u\) and right-vector \(r\) are aligned with the y- respectively x-axis as illustrated in figure 5.4; its orthographic projection matrix \(P_{ortho}\) is given in equation (5.8).

\[
P_{ortho} = \begin{pmatrix}
\frac{2}{S_x} & 0 & 0 & -1 \\
0 & \frac{2}{S_y} & 0 & -1 \\
0 & 0 & \frac{2}{z_{max} - z_{min}} & -1 \\
0 & 0 & 0 & 1
\end{pmatrix}
\] (5.8)

The input images \(I_k\) that contribute to the DEM layer of the FCM tile with index \(n_{x}, n_{y}\) and their associated \(4 \times 4\) camera matrices \(M_k\) are retrieved from the spatial index of the
Figure 5.4: In order to perform a GPU-based space sweep, a virtual orthographic camera is oriented as shown and looks perpendicular onto the center of a quad in order to generate the input fragments for a GLSL shader which calculates DEM values from multiple images.

FCM as described in section 3.2.5. Images are accessed by the fragment program as textures and may be stored in a texture array of \( n \) elements, if they are of the same lateral sizes. The matrices \( M_k \) are passed to the shader as the elements of a uniform array of \( n \) elements of GLSL data type \texttt{mat4}. Both kinds of input data for the shader remain associated with each other by their common index \( k \). Since arrays in fragment programs need to be of fixed size, a \texttt{#define} preprocessor directive with the total number \( N \) of input images which can be processed at the same time is inserted before the GLSL shader source code. In this way, \( N \) is made a constant value within the fragment program, which permits the static allocation of arrays of size \( N \). The number \( n \leq N \) of actually employed images is provided as uniform variable.

The camera positions in \( M_k \) are referenced to absolute world coordinates. Therefore, the minimum corner \((n_x, g_x, n_y, g_y, z_{\text{min}})\) of the box \( Q \) in equation (3.4), which is used to query the spatial index of the FCM, is added to the position of the camera and to the vertices of the quad.

5.3.2 Program Execution

All rendering output is redirected into a frame buffer object (FBO) that matches the pre-defined FCM tile size in texels. The FBO contains at least one color texture in R8G8B8 or R8G8B8A8 texel format and one depth texture in F32 texel format for storing Z-Buffer values.

In each of the \( L + 1 \) rendering passes for the different plane positions at \( z = l \cdot \Delta z + z_{\text{min}} \) where \( 0 \leq l \leq L \), the fragment program must be provided with the texture coordinates \((u_k, v_k)\) for sampling image \( I_k \) at each fragment that results from the rasterization of the quad. For efficiency reasons, a vertex program generates homogeneous coordinates \((\hat{u}_k, \hat{v}_k, \hat{z}_k, \hat{w}_k)\) directly from the vertex coordinates \((x, y, z)\) of the quad according to equation (5.9).
5.3 GPU-based Implementation

\[
\begin{pmatrix}
\hat{u}_k \\
\hat{v}_k \\
\hat{z}_k \\
\hat{w}_k
\end{pmatrix}
= \begin{pmatrix}
\frac{1}{2} & 0 & 0 & \frac{1}{2} \\
0 & \frac{1}{2} & 0 & \frac{1}{2} \\
0 & 0 & \frac{1}{2} & \frac{1}{2} \\
0 & 0 & 0 & 1
\end{pmatrix}
\cdot M_k 
\cdot 
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}
\tag{5.9}
\]

These \( n \) values are stored in an array indexed by \( k \) and are automatically interpolated across the surface of the quad as the array is transferred from the vertex to the fragment program. The normalized texture coordinates \((u_k, v_k)\) for sampling input image \( I_k \) are obtained by component-wise division of \((\hat{u}_k, \hat{v}_k)\) by \(\hat{w}_k\) in the fragment program:

\[
(u_k, v_k) = \left( \frac{\hat{u}_k}{\hat{w}_k}, \frac{\hat{v}_k}{\hat{w}_k} \right)
\]

Since the sample positions \((u_k, v_k)\) have to be accessed multiple times within the fragment program, they are computed only once at the beginning of the program and stored in a global array indexed by \( k \). The values sampled from image \( I_k \) are usually RGB triples \( p_k' = (R_k, G_k, B_k) \) and are converted into grayscale luminance values \( p_k \) in CIE XYZ color space\(^9\) according to equation (5.10).

\[
p_k = 0.212671 \cdot R_k + 0.715160 \cdot G_k + 0.072169 \cdot B_k
\tag{5.10}
\]

The numerical constants in equation (5.10) depend to a certain degree on the physical properties for rendering color spectra of the video output device, i.e., the monitor or projector, and are given in [1, p. 215]. The corresponding matching costs \( C_k \) are computed for each of the resulting \( p_k \) according to equation (5.5) and are stored at index \( k \) in an array of size \( n \). Instead of computing the aggregated matching costs according to equation (5.6) by summing up all \( n \) values, we may choose to take only the best \( \tilde{n} = \left\lfloor \frac{n}{2} \right\rfloor \) of the images into account for matching. This modification has been proposed in [41] in the context of selecting images from a temporal sequence as better half sequence and is intended to disregard images in which a feature is occluded. Therefore, the array of matching costs can be sorted in the fragment program, e.g., by using selection sort or insertion sort, and only the \( \tilde{n} \) lowest matching costs may be summed up.

In order to find the minimum matching costs and to select the corresponding DEM values, we employ the method presented in [87] which utilizes the Z-Buffer of the graphics hardware. Since we use FBOs, the Z-Buffer values are redirected into the attached depth texture. The current position \( z \) of the plane along the \( z \)-axis in world coordinates is emitted by the shader as the fragment color, whereas the aggregated matching costs are explicitly assigned to the Z-Buffer value of the fragment. \( z \) is available at the vertex coordinates of the quad in the vertex program and is passed on to the fragment program for this purpose. The employed Z-Buffer test is set up to select the fragment with the lowest Z-Buffer value at the current sweeping plane position. In this way, in each rendering pass, i.e., at each different location \( z \) of the sweeping plane in space, depth and color values in the FBO are successively replaced by values of new fragments if the new Z-Buffer values emitted by the

\(^9\)CIE is the abbreviation for "Commission Internationale d’Eclairage".
shader are smaller than the existing ones, and the matching costs at the current fragment are thus lower. After \( L + 1 \) rendering pases, the color texture attached to the FBO contains the elevation values for the area covered by the FCM tile and the data are copied from video memory into the DEM layer of the tile.

A problem arises in parts of a FCM tile which are covered by only a single image so that no matching costs can be computed. We therefore determine at each fragment the number of covering images by checking whether each coordinate pair \((u_k, v_k)\) is within the valid range \( [0, 1] \times [0, 1] \). If the coverage is below a certain threshold \( TR \in \mathbb{N}, TR \geq 2 \), the aggregated matching costs \( AC \) for this fragment are set to \( \infty \). Before emitting color and depth values, the shader checks \( AC \) and discards fragments if \( AC = \infty \). In this case, neither Z-Buffer nor color values will be generated and any existing data in the FBO will remain unaffected.

### 5.4 Color Texture Generation

In orthophotos distortions caused by perspective effects are corrected and lateral surfaces, like frontages of buildings, are intentionally not depicted. Orthophotos can be used, for instance, to create or enhance a variety of maps for different applications [42, pp. 410–411]. Moreover, they are employed as color textures in our DSM rendering approach as presented in chapter 4. The generation of an additional orthophoto texture of the same resolution in texels as the DEM layer of a certain tile can be integrated into the previously described DEM creation process. The resulting proceeding is essentially the same as the one outlined in [42, pp. 425–426].

The extension for generating an additional color texture merely requires the fragment shader to output a second color value at each fragment into an additional color texture. This texture should have R8G8B8 or R8G8B8A8 texel format and must be attached to the FBO. Instead of storing only the matching costs \( C_k \) in an array, a pair \((C_k, k)\) is stored in order to keep track of the associated image index \( k \). After the array has been sorted in ascending order of \( C_k \), the index of the image with the lowest matching costs \( k_{\text{min}} \) is stored at the first pair in the array. The RGB value \( p'_{k_{\text{min}}} \) from the corresponding image \( I_{k_{\text{min}}} \) at sample position \((u_{k_{\text{min}}}, v_{k_{\text{min}}})\) is assigned to the additional fragment color output. Afterwards, these data are copied from video memory to the tile, and the color layer is created in the same way as the DEM layer. As an alternative, the \( n' \leq n \) images with lowest matching costs may be sampled, and the component-wise mean of the obtained RGB values may be used to determine the color layer value. Figure 4.5 shows a comparison of the results of these two methods for generating orthophoto textures.

If a region of the tile is covered by only a single aerial image, no elevation data can be derived, and it is in general impossible to determine objects in the input images that are affected by perspective, e.g., skyscrapers, tall trees or mountains. In this case, it may be desirable to rely on using the only available image and to create a color texture layer which is not orthographic, instead of providing no color information at all. Such a texture can be created in a single rendering pass by executing the process for orthophoto texture
5.4 Color Texture Generation

![Images showing minimal matching costs, averaged color values, and a difference image.](image)

Figure 5.5: A comparison of an orthophoto texture generated by using pixels from the input images with lowest matching costs (left), and by component-wise averaging the color values of the input images (middle). Locations where no DEM data are available remain black. Due to the input data, the differences of the color values are very small and are illustrated in the contrast enhanced difference image (right).

generation described above only once and by projecting all images onto the quad located at $z = z_{\text{min}}$. In regions where only a single image is available, the sampled color values determine the final color at the corresponding texel in the color texture. If the projections of multiple images overlap, it is possible to use the pixels from one or more images with lowest matching costs. Other possibilities to handle the overlap of multiple pixels at one texel of the color texture are, for instance, blending, averaging or overwriting. Since the perspective distortion of objects depicted in an image increases with their distances from the geometric center of the image, ambiguities at a single texel may also be resolved by using the pixel with smallest geometric distance from the center of its image. These five different possibilities are illustrated in figure 5.6 and have different properties.

Averaging multiple pixels results in severe ghosting and creates fuzzy results. Since the fuzziness tends to increase with the elevation of the underlying surface, this effect might even be useful for visualizing the uncertainty about the elevation in the DSM in regions where no height information could be derived from the input images. Blending multiple pixels tends to blur the results and does not solve the problem of mismatching borders between different images. Considering the minimal matching costs in the plane at $z = z_{\text{min}}$ causes the resulting color texture to look speckled and unclean. Allowing different pixels to overwrite each other is the simplest solution, but also results in mismatching borders and depends on the order of the images (cf. section 3.2.5). Choosing the pixel with smallest distance from the respective center of its image does not eliminate this problem, but has an interesting effect: lateral faces, e. g., like frontages of buildings, tend to vanish in the resulting color texture, and the resulting texture looks almost orthographic in some regions. Since the latter method creates clear images and reduces the amount of lateral surfaces depicted in the resulting texture, we prefer it over the other methods if we have to create non-orthographic color textures.

It is also possible to initialize a color texture layer by means of one of these non-orthographic creation processes and to successively replace values with those obtained during DEM creation. In this way, we can create a color texture layer which provides color information where available, but which is furthermore truly orthographic in regions where DEM data are available.
Figure 5.6: Different methods for resolving ambiguities caused by projecting multiple pixels from different images to one texel in order to create a planar color texture, if no elevation information is taken into account. Properties specific to the methods are highlighted by red and blue rectangles.

5.5 Improvements of DEM Quality and Results

With the methods for creating DEM data and matching color textures as described above, we can rapidly create all layers of single FCM tiles. Although accuracy is important in our application, we strive for the generation of credential DSM data at high speed and therefore do not additionally refine the resulting DEM data by means of statistical analysis for removing outliers, for instance. In the following, we discuss the influence of different improvements for enhancing the quality of the resulting DEM data.

5.5.1 Stereo Matching Errors

As shown in figure 5.7(b), some isolated DEM layer texels exhibit faulty dents or spikes during rendering while their neighboring texels are correct. These outliers may emerge from errors during stereo matching or from ambiguous matching costs and the cost selection by using Z-Buffer tests. In principle, there should be no difference if the space sweep described in section 5.2 starts at \( z_{\text{min}} \) and the sweeping plane is moved up to \( z_{\text{max}} \), or if the sweeping order is reversed and goes from \( z_{\text{max}} \) down to \( z_{\text{min}} \). But due to the Z-Buffer test, there can be visible differences in the resulting DEM if the matching costs at a fixed image position \((u_k, v_k)\) are equal for two or more different sweeping plane positions. As the Z-Buffer test is set in such a way that only fragments with smaller Z-Buffer values are allowed to replace texels in the render target, the first of a number of different DEM values with equal matching costs will determine the final texel value. If the sweeping plane is moved along the
positive z-axis, faults in the DEM caused by ambiguous matches may show up in renderings as dents respectively spikes, if the sweeping is performed along negative z-direction. An alteration of the Z-Buffer test to let fragments with less or equal matching costs pass would correspond to a reversal of the space sweep order, because the final texel value would then be determined by the last of a number of DEM values with equal matching costs. These stereo matching related errors can be resolved by taking more different images into account in order to obtain unique and better matching costs for each plane position at each single fragment.

![DEM layer](a)

![resulting rendering](b)

**Figure 5.7:** Incorrect results from stereo matching or ambiguous matching costs result in faulty DEM layer texels which become apparent as spikes or dents in renderings as highlighted by the red rectangles. Contrast enhancement was applied to the depicted DEM layer (left) in order to visualize faulty texels. Brighter colors correspond to higher elevations.

The presence of ambiguous matches may be additionally favored by the kind of employed aerial images. Since we do not yet have a reliable source for providing aerial images as required in our application (cf. section 2.3.3), we create synthetic data from 3D renderings of a virtual city model which has been generated by the software “CityEngine” \cite{24}. The objects depicted in the images are quite artificial, because the underlying 3D models employ only a very limited amount of textures, especially on roofs and frontages, so that lots of repetitive patterns can be found in the renderings. These repetitions can easily lead to equal matching costs for single DEM texels on the sweeping plane at different z-coordinates.

### 5.5.2 Smoothening DEM Data

Since stereo matching errors and ambiguities may also arise in real-world data, e.g., in image regions depicting large, monochrome and structureless surfaces, we experimented with smoothening the DEM data before they are copied into the corresponding tile layer. For this purpose we use a rather simple technique which utilizes automatic mipmap generation capabilities of graphics hardware and bilinear interpolation.

By enabling automatic mipmap generation for the textures attached to the FBO, the texture data at mipmap level $\lambda$ correspond to the result of a convolution of the original texture with a box-shaped filter kernel of size $2^\lambda \times 2^\lambda$. If the texture at $\lambda > 0$ is copied back from video memory, we obtain a down-scaled version of the required DEM layer. This smaller version is scaled afterwards by using re-sampling and bilinear filtering in order to get a texture of the original size. As a result, the texture becomes blurred and this technique
5 GPU-based DSM Synthesis

is therefore also used in GPU image processing for creating certain effects [1, p. 471]. In this way, we achieve a replacement of DEM texels by averaged values that incorporate neighboring texels and dents and spikes are less prominent as shown in figure 5.8. However, potentially correct edges at steep elevations will be smoothened as well and turn into ramps. This effect is illustrated in figure 5.9 by means of a DEM of an isolated cube obtained from synthetic images.

![Figure 5.8: Smoothening DEM textures levels incorrect spikes and dents and makes the resulting renderings visually more appealing. The visual effect of defects in the DEM is enhanced by applied shading and lighting.](image)

With the data employed in our application, we found the usage of smoothening by copying DEM textures at mipmap level \( \lambda = 1 \) to be a reasonable compromise. Using more sophisticated and adaptive methods for compensating defects would probably further improve the resulting DEM data.

![Figure 5.9: Smoothening DEM textures by means of simple GPU-based blurring results in an unwanted transformation of edges at steep elevations into ramps.](image)

5.5.3 Cost Aggregation Over Support Windows

We also implemented the possibility of aggregating matching costs over support windows having power of two size by using mipmaps which are automatically generated by graphics hardware for each input image [86, 87]. For this purpose we compute the matching costs at sample position \( s = (s_x, s_y, s_z) \) by means of \( C_k'(s) \) from equation (5.11) for a given range
of mipmap levels \([\lambda_{\text{min}}, \lambda_{\text{max}}]\) instead of using \(C_k(s)\) from equation \(5.5\).

\[
C_k'(s) = \sum_{\lambda=\lambda_{\text{min}}}^{\lambda_{\text{max}}} \sum_{i=0}^{n-1} |p_i(\lambda) - p_k(\lambda)|
\]  
(5.11)

In equation \(5.11\), \(p_j(\lambda) = I_j(u_j, v_j, \lambda)\) denotes the luminance value which results from sampling image \(I_j\) at mipmap level \(\lambda\) at position \((u_j, v_j)\). The aggregated matching costs \(AC(s)\) according to equation \(5.6\) are then calculated by using \(C_k'(s)\) instead of \(C_k(s)\) in equation \(5.5\).

With the data in our application, we found that matching cost aggregation over support windows significantly improves the visual quality of renderings of the resulting DEMs, and comparative results are given in section \(5.6\). However, if the size of the support windows becomes too large, features are missing as illustrated in the resulting DEM renderings in figure \(5.10\). We found the cost aggregation over support windows up to sizes of \(8 \times 8\) completely sufficient for our kind of input images.

![Figure 5.10: Using support windows larger than \(8 \times 8\) for cost aggregation results in DEMs where features are missing. In the right image, for instance, the sizes of the roofs are reduced.](image)

5.5.4 Better Half Sequence

Our implementation also allows to consider only the \(\left\lfloor \frac{n}{2} \right\rfloor\) input images with lowest matching costs for aggregation (better half sequence) \([41]\). For this purpose, we sort the matching costs of the input images within the fragment program in increasing order by means of a simple selection sort implementation. Figure \(5.11(a)\) shows a rendering of a DEM generated by using the better half sequence method, whereas figure \(5.11(b)\) shows a rendering of a DEM generated by using all images for matching cost aggregation. Both DEMs were created from the same eight input images and costs were aggregated over support windows up to size \(8 \times 8\). The visual differences are small, but with the employed input images, the results obtained by using all input images for matching cost aggregation depict fewer defects in some regions, e. g., on the side of the main building which is closest to the viewer.
5.6 Performance Evaluation and Discussion

The performance and the quality of the output data depend in particular on the number of employed images and the chosen height resolution $\Delta z$ of the space sweep. Table 5.1 contains exemplary performance results for creating the DEM and color layers of $2 \times 2$ neighboring FCM tiles at clip level 0, which were all covered by the same employed images. The results were obtained on a desktop computer with an Intel i7 860 CPU at 2.8 GHz, 6 GB RAM, NVIDIA GeForce GTX 470 graphics adapter with 1280 MB dedicated video memory and Windows 7 OS. The input images had a size of $1920 \times 1080$ pixels, and we obtained results for different numbers of images and support window sizes. Column \textit{max. support window size} indicates the maximum size of the support window for cost aggregation according to equation (5.11). The values in table 5.1 are given in seconds and include the time for transferring the images from main memory to video memory. The tile size was set to $512 \times 512$ texels and the sweeping plane was moved from $z_{\text{min}} = 0$ to $z_{\text{max}} = 25$ at a step width of $\Delta z = 0.125$ units. This corresponds to $L = 200$ and thus 201 different plane positions and rendering passes. We furthermore measured the runtime using matching cost aggregation of all available images, and the runtime when using better half sequences, i.e., only the $\left\lfloor \frac{n}{2} \right\rfloor$ images with lowest matching costs contribute to cost aggregation. The tile creation was executed in parallel to a process for directly rendering the resulting DEMs. Renderings from the run in which all images were taken into account during cost aggregation are shown in figure 5.12.

5.6.1 Discussion

Table 5.1 shows that the runtime increases with the number of employed images, but the quality increases as well as can be seen in figure 5.12. The additional costs for the aggregation over several support window sizes are moderate, but in case of these particular input images, using support windows significantly increases the quality of the resulting DEMs, even if the number of images increases. As expected, the runtime is higher in case of using...
5.6 Performance Evaluation and Discussion

Figur 5.12: Renderings of the DEMs which were created during a performance evaluation for different numbers $n$ of input images and support window sizes.

The depicted DEMs were created by using all images for cost aggregation. The resulting DEM data were not smoothed and shading was applied to enhance the perception of defects, such as spikes and dents.
5 GPU-based DSM Synthesis

Table 5.1: Performance results of an exemplary DSM synthesis from up to eight different images at different numbers of input images, maximum support window sizes and matching costs aggregations. The values are given in seconds.

<table>
<thead>
<tr>
<th>max. support window size</th>
<th>all images</th>
<th>better half</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = 2</td>
<td>n = 4</td>
</tr>
<tr>
<td>1 × 1</td>
<td>2.32</td>
<td>3.03</td>
</tr>
<tr>
<td>2 × 2</td>
<td>2.20</td>
<td>3.14</td>
</tr>
<tr>
<td>4 × 4</td>
<td>2.38</td>
<td>3.18</td>
</tr>
<tr>
<td>8 × 8</td>
<td>2.26</td>
<td>3.43</td>
</tr>
</tbody>
</table>

better half sequences for cost aggregation, because it requires the GPU program to sort the aggregated matching costs, and we used a non-optimized selection sort implementation for this task. However, the runtime of the configurations given in table 5.1 was always dominated by the costs for transferring the input images into video memory, which took more than 50% of the total time. Therefore, much more time was required for creating the first of the 2 × 2 tiles than for creating the remaining three tiles, since they were generated from the same input images. After the input images were transferred to video memory, the creation of one of the remaining tiles never took more than 0.025 seconds. Figure 5.12 furthermore shows that using only two input images is insufficient in our approach. Reasonable results require at least \( n = 6 \) images and preferably the cost aggregation over support windows of sizes up to \( 8 \times 8 \).

The quality of the DEMs which we can generate by means of the method presented in this chapter also depends strongly on the chosen height resolution \( \Delta z \) as shown in figure 5.13. Both DEMs were created from the same eight images and matching costs were aggregated over support windows up to size \( 8 \times 8 \), but the DEM in figure 5.13(a) exposes less defects.

![Figure 5.13: The chosen height resolution \( \Delta z \) of the space sweep strongly influences the quality of DEMs which can be generated by means of the method presented in this chapter. Both DEMs were created from the same eight input images and use the same support window size of \( 8 \times 8 \) without smoothening.](image)

The DSM creation was executed in parallel to the DSM rendering process on the same computer with a single graphics adapter, and the hardware resources were hence shared between the two processes. We can therefore expect an increase of the performance of the mere DSM creation, if it is executed standalone with exclusive access to the graphics.
hardware resources. It is also important to note that all of our results presented in this section were obtained by employing synthetic aerial images. Real images have different properties and would suffer from noise, different illumination conditions, lens distortions and other optical phenomena. The eight input images which were used for creating DEMs of the urban area as presented in this thesis were captured at altitudes from $\approx 78.0$ to $95.32$ units and are shown in figure 5.14.
Figure 5.14: These eight synthetic aerial images at an original resolution of 1920 × 1080 pixels were used to create all DEMs of the urban area as presented throughout this chapter. The depicted city model was created by means of the software “CityEngine” [24].
6 Texturing Lateral Surfaces

In the previous chapters, we dealt with DSMs that do not contain any color information about lateral surfaces, i.e., the faces of elevations which are not parallel to the ground plane. We only employed orthophoto textures derived from vertical aerial photographs, but these images provide no color information about lateral surfaces. Such surfaces, like walls of canyons or frontages of buildings may become visible in renderings of DSMs in virtual 3D environments, and the viewer might expect to see structural details, e.g., different rock layers or windows and doors. Therefore, it is common practice to employ oblique aerial images to overcome this lack of information.

In this chapter, we extend our rendering technique as presented in chapter 4 in order to add color information to lateral surfaces of a DSM. This extension directly employs vertical and oblique aerial images, and benefits from rendering by means of ray casting.

6.1 Complete Color Textures for DSMs

In computer graphics, color information about surfaces is frequently stored in rectangular 2D color textures. Surfaces and texture space are usually parametrized by normalized texture coordinates \((u,v) \in [0,1] \times [0,1] \subset \mathbb{R}^2\). Normalized texture coordinates allow to conveniently transform locations on the surface into texture space and vice versa, independent of their respective physical extensions. Some geometric bodies, like rectangles, spheres or cylinders, possess parametric representations that can be directly used as texture coordinates. If a polygonal mesh is used to represent a surface, the texture coordinates are often given for the vertices of the mesh and interpolated across the associated mesh faces. This method is frequently used if no parametric representation of the surface is available, or in conjunction with manually modeled meshes of complex objects. In some situations, texture coordinates can be computed directly on the GPU from vertex positions, e.g., as described in section 5.3.2.

A simple way to create a color texture of \(T_u \times T_v\) texels for a surface with a given parametrization \((u, v)\) is to create a piece-wise linear approximation of the surface by using rectangular patches. These patches are aligned to the tangential planes of the surface at equally spaced distances \(\frac{1}{T_u}\) and \(\frac{1}{T_v}\) in u- respectively v-direction. Each patch is assigned the color value of the underlying part of the surface and is identified with a texel of the texture. In areas where the piece-wise linear approximation and the sampling rates \(\frac{1}{T_u}\) and \(\frac{1}{T_v}\) are insufficient, the texture will expose distortions. Figure 6.1 illustrates the creation process of a texture on a sphere and the resulting distortion.
6 Texturing Lateral Surfaces

Figure 6.1: Mapping the surfaces of geometric bodies onto a planar rectangle in order to create a texture can result in distorted images, especially in regions where the approximation by rectangular patches is insufficient, e.g., as in the case of a sphere. Towards the poles, the illustrated image of the Earth’s surface is increasingly distorted.

Orthophoto textures, as employed in our approach, are obtained from a parametrization of the ground plane of the underlying DEM, which is a planar rectangular surface. In order to store color information about a complete surface of a DEM in a single texture, the surface has to be parametrized with respect to its elevation. Depending on the range of the elevation, such a texture can require substantially more samples than the corresponding DEM, since the surface area can be much larger than the one of the ground plane. This is illustrated in figure 6.2. Furthermore, the total surface area must be computed in order to find an appropriate texture size \((T_u, T_v)\) and to normalize the parametrization.

Figure 6.2: Storing color information about the entire surface of a DEM, including lateral surfaces, may require the creation of large textures. The depicted elevation of the surface, which is illustrated in profile, is sampled at six positions (gray numbers), whereas the entire surface is larger and is sampled at 16 positions (blue numbers).

Since we strive for fast generation of DSMs, which may in addition change frequently, storing the entire surface in a single texture would be inefficient, and we therefore map colors onto lateral surfaces by means of projective textures.

### 6.2 Aspects of Projective Texturing

Projective textures can be thought of as virtual video or slide projectors. They can be used to assign color values to surfaces based on an object’s position in world space instead of its surface parametrization as described in the previous section. During rendering, the color
values of the corresponding screen pixels are determined by the texels which become projected onto the surface. We already used this approach in chapter 5 in the context of stereo matching by means of a space sweep where aerial images are projected onto the sweeping plane. Another application of projective textures is, for instance, the implementation of complex light effects in virtual environments [1, p. 222]. Shadow mapping uses a similar concept for determining the visibility of locations on a surface from the position of a light source [1, pp. 348–353].

In order to use projective texturing on a DEM, we project oblique aerial images onto lateral surfaces, and vertical aerial images onto surfaces which are parallel to the ground plane. Only images depicting parts of the DEM surface which are currently visible to the viewer in a 3D virtual environment need to be employed as textures. Hence, the selection of aerial images which are appropriate for projective texturing depends on the orientation and position of the virtual camera representing the viewer. The best rendering results can be achieved by projecting images which were captured at positions and with orientations similar to the ones of the virtual camera, because these images are likely to depict the objects which are about to be rendered.

In the context of projective texturing, we call the associated camera of an aerial image \( I_k, k \in \mathbb{N} \) projector and the projector matrix is the camera matrix \( M_k = P_k \cdot V_k \), where \( V_k \) denotes the camera’s view matrix and \( P_k \) the projection matrix.

### 6.2.1 Aerial Image Selection

In our approach, we store all aerial images according to the bounding boxes of the view pyramids of their associated cameras in the spatial index of our FCM implementation (cf. section 3.2.3).

The spatial index assists in selecting aerial images which are appropriate for texturing a DEM as follows: An axis-aligned bounding box (AABB) \( Q \) is computed for the view frustum \( F \) of the virtual camera in order to perform an intersection query on the spatial index. Thus, images with projectors which are not entirely contained inside \( Q \), but whose frustums may intersect \( Q \) are also retrieved. Since \( Q \) has usually a larger volume than \( F \), the resulting set of aerial images \( S = \{I_0, I_1, \ldots I_{n-1}\} \) may contain many elements that are not well-suited for projective texturing. These are in particular all images whose orientations and positions of their associated projectors deviate too much from the ones of the virtual camera. Figure 6.3 illustrates in 2D the relationships of the virtual camera and the projectors, as well as their frustums and the bounding boxes.

Let \( \varphi_k \) denote the angle enclosed by the viewing direction of the projector of an aerial image \( I_k \) and the viewing direction of the virtual camera. We remove all images from the initial set \( S \) of aerial images for which \( |\varphi_k| > \Phi_{\text{max}} \) where \( 0 < \Phi_{\text{max}} < \frac{\pi}{2} \) is a user-defined threshold. Projectors with \( |\varphi_k| \geq \frac{\pi}{2} \) can only depict back sides of objects from the current point of view. The projectors of the remaining images in the modified set \( S' \) and the virtual camera thus have similar orientations.

Like with stereo matching, the total number \( N \) of projective textures which can be used at a time is limited by the amount of available video memory or the number of texture units
Texturing Lateral Surfaces

Figure 6.3: The bounding box $Q$ (gray, solid) of the virtual camera’s view frustum $F$ (blue) is used to query the spatial index which stores the aerial images (red) according to the bounding boxes of their view pyramids (gray, dashed).

(cf. section 5.2.1). Therefore, if the modified set $S'$ of aerial images contains $n' = |S'| > N$ elements after the removal of the aforementioned images, it must be reduced further. Since we want to use images whose projectors are located close to the virtual camera, we sort the images according to the distances $d_k$ of their CoPs from the CoP of the virtual camera. Except for those $N$ images with smallest $d_k$, we remove all other images from the set $S'$.

In summary, our algorithm for finding a set $\tilde{S}$ of aerial images which can be used for projective texturing, where $|\tilde{S}| = N$, is as follows:

1. Create AABB $Q$ from the frustum $F$ of the virtual camera.
2. Retrieve from the spatial index of the FCM the set $S = \{I_0, I_1, \ldots, I_{n-1}\}$ of all aerial images having projectors with frustums whose AABBs intersect $Q$.
3. For each $I_k \in S$, determine the angle $\phi_k$ enclosed between the viewing direction of the projector and the viewing direction of the virtual camera.
4. If $|\phi_k| > \Phi_{\text{max}}$, remove $I_k$ from $S$.
5. If the modified set $S'$ contains $> N$ elements, sort the elements according to the distances $d_k$ from the CoPs of their associated projectors to the CoP of the virtual camera.
6. Except for those $N$ images with smallest $d_k$, remove all other images from $S'$ and obtain $\tilde{S}$.

6.2.2 Occlusions Between Lateral Surfaces

Locations on the DEM surface, which are occluded from the point of view of a projector by other elevations as illustrated in figure 6.4, are not visible within the corresponding image. Projecting occluded points on the DEM surface back into images that do not depict them might yield a valid texture coordinate, but the resulting color values must not be used for computing the color at the corresponding screen pixel. This is the same problem as determining the visibility of a point in a scene from the position of a light source in order to render shadows by means of shadow mapping [1, pp. 348–353]. Hence, a solution to
6.2 Aspects of Projective Texturing

Figure 6.4: Locations on a DEM surface can be occluded by elevations and may hence not be visible in all projected images. In the depicted situation, \(i\) is visible in image 2 but not in image 1, because it is occluded by an elevation. Image 1 depicts the front of that occluding elevation at the corresponding pixel, but not \(i\).

this problem is the following proceeding: The scene is rendered from the perspective of the projector, and the depth values of locations on the DEM are stored in a depth image, which is called shadow map in the context of shadow mapping. When the scene is rendered from the viewer’s perspective, locations on the DEM surface that are visible to the viewer are projected into the depth map. If the depth value of a certain location is greater than the value that is stored in the depth map, it is considered invisible from the projector’s perspective. The scene needs to be rendered only once per projector and the resulting depth map can be stored and reused.

With our DSM rendering approach, this technique requires one complete ray casting pass per projector, and the depth maps need to be regenerated each time the underlying DEM changes, if new elevation data become available. The video memory consumption is increased by one additional depth map per projected image. In addition, the quality of depth maps depends on their resolution and the technique may expose inaccuracies due to aliasing. More details on the subject of shadow mapping can be found in [1, pp. 348–372], for instance.

Since we use ray casting for DSM rendering, for a projected image \(I_k\) the visibility of intersection \(i\) of a ray with the DEM surface can be determined by casting a second ray from \(i\) to the CoP of \(I_k\). This secondary ray casting can be performed by using the same ray traversal scheme as presented in section 4.2.2. If the second ray intersects the DEM on its way to the CoP, \(i\) is not visible in \(I_k\). This method is potentially costly as it requires to cast up to \(n\) additional rays for each fragment, where \(n\) is the number of projected images. However, the projectors which can definitely not depict \(i\) according to their orientations and distances from \(i\) and from the virtual camera, can be ignored, and the number of secondary rays may be reduced.

### 6.2.3 Multiple Projections

A point \(i\) on a DEM surface can be depicted in multiple projected images as illustrated in figure 6.5. In this situation, the image whose associated projector is closest to \(i\) is probably the best choice for sampling color values, since it potentially depicts more details of the object than images which were captured from greater distances. Furthermore, \(i\) is more likely occluded by other elevations in images whose CoP is located at larger distances from \(i\), than in images having CoPs located closer to \(i\).
Figure 6.5: If a point $i$ on a DEM surface is visible in multiple images, the image closest to $i$ (image 3, red) is probably the best choice for determining the final color value at $i$ in the virtual scene, because this image is likely to depict the most details. $i$ may furthermore be occluded in images captured at larger distances (image 1) by an obstacle (gray, dashed), which can be part of the scene that is depicted in those images.

The image $I_k'$ whose CoP is closest to $i$, can be determined by a fragment program as follows: Let $d_k$ denote the distance from the CoP of image $I_k$ to $i$. $i$ is projected from world space into each of the images $I_k$ by means of its associated projector matrix. If the resulting 2D image coordinates of the projection of $i$ are contained in the plane section of image $I_k$, $i$ may be visible in $I_k$. Image $I_k'$ is then determined as the image which contains the projection of $i$ and has the smallest distance $d_k'$ among all images containing the projection of $i$. In order to compute the distances $d_k$ on the GPU, the positions of the CoPs of the projected images can be passed to the fragment program.

6.3 Implementation and Results

Projective texture mapping is implemented by the GLSL fragment program for DSM ray casting as described in chapter 4. The set of aerial images which is used for projective texturing is determined by the application each time the point of view of the virtual camera is modified according to the image selection algorithm presented in section 6.2.1. The images are sorted according to the distances $d_k$ of their respective CoPs from the virtual camera in increasing order, and are provided to the ray casting shader as an array of textures. In addition, the projector matrices and the CoPs of the projected images are passed to the fragment program as uniform arrays of data type \texttt{mat4} and \texttt{vec3}, respectively. This is the same proceeding which is used to pass vertical aerial images and camera matrices to our GPU programs for stereo matching as presented in chapter 5.

After the intersection $i$ of a ray with the DEM has been determined by ray casting, the fragment program computes the texture coordinate at each fragment for each projected image $I_k$. If the coordinates are not within the range $[0, 1] \times [0, 1] \subset \mathbb{R}^2$, the corresponding image is not considered for computing the color value of the fragment. Otherwise, the squared distance $d_k^2$ from $i$ to the projector’s CoP is computed. If $d_k^2$ is the current minimum of all squared distances from $i$ to the CoP of the projected texture $I_k$, the current
minimum is updated and the sampled color value is assigned to the fragment, but only if the fragment is not occluded in $I_k$. To determine whether $i$ is not occluded in $I_k$ by any elevation of the DEM surface, we use secondary ray casting from $i$ towards the CoP of $I_k$.

The GLSL source code of the complete implementation can be found in appendix A in listing A.2 in lines 874 to 917.

6.3.1 Results

Figure 6.6 gives an example of a DSM rendering using projective textures. The maximum enclosing angle between the viewing directions of the associated cameras and the virtual camera was set to $\Phi_{\text{max}} = \frac{\pi}{4}$, and the image projections were confined to areas where elevation data are present.

Figure 6.6: Examples of renderings of a DEM of an urban area with applied projective textures. The DEM was synthesized from the eight vertical aerial images in figure 5.14 and $n$ oblique aerial images of sizes $1920 \times 1080$ pixels were used as projective textures in the depicted scene.

The performance of our projective texturing approach depends on the number of projected images and the accompanying number of secondary rays that need to be casted in order to test for occlusions. On a computer with the configuration given in section 5.6, we observed during several runs with different DEMs and up to eight active projectors a decrease of the rendering frame rate of about 20% to 50%. However, the frame rate did not drop below interactive rates, even at screen resolutions of $1920 \times 1080$ pixels. Furthermore, we observed that the additional color information on lateral surfaces can help to conceal defects in DEMs (see section 5.5.1). An illustration of this effect is shown in figure 6.7.

6.4 Discussion

Using the original aerial images as projective textures in order to provide color information on lateral surfaces of a DEM has several advantages. By employing projective textures, we neither have to create nor to store additional textures, and the set of employed aerial images can be altered without having to update the corresponding DSM layer. The latter aspect is especially important, because our application targets at fast DSM creation from potentially frequently changing sets of images. If a DSM consists only of a DEM layer,
video memory is not occupied by an additional color texture layer and thus remains available for projective textures. A comparison of DEM renderings using orthophoto textures and by using projective textures is shown in figure 6.8. The six oblique aerial images which were used for projective texturing throughout this chapter are shown in figure 6.9.

The set of projective textures for rendering a scene depends on the viewer’s current perspective and must be updated as the virtual camera moves. Transferring large amounts of data from main memory to video memory, however, temporarily degrades the performance of the rendering process. Since the original aerial images may be too large for being directly used as appropriate projective textures, it can be necessary to downscale the images in advance.

We can also enable automatic mipmap generation by graphics hardware for the employed projective textures and compute an appropriate mipmap level for sampling color values. The mipmap level computation can be integrated into the ray casting process, similar to the calculation of clip level \( l_{opt} \) which is used for ray termination (see section 4.2.3). In this way, our implementation can provide a simple level-of-detail concept for texturing lateral DEM surfaces.
Although the performance of the technique presented in this chapter appears to be sufficient for our purposes, more detailed investigations might reveal further possibilities for optimization.

Figure 6.9: The depicted six oblique aerial images of original sizes of 1920 × 1080 pixels were used for projective texturing on the DEM of an urban area. The DEM was derived from the images shown in figure 5.14 according to the method presented in chapter 5.
7 Framework Design

In this chapter we present the design and important implementation details about the framework which implements the methods for rendering and generating DSMs as presented in this thesis. The framework is implemented in C++, rendering relies on OpenGL 3.2 and GPU programs (shaders) written in GLSL version 1.5. Section 7.1 gives an overview about the organization of the framework, followed by details about the implementation of our Flexible Clipmap in section 7.2. Our implementation of the employed R*-tree is described in section 7.3. Important aspects of the design and the implementation of the two GPU programs for rendering and generating DSM data are presented in section 7.4. This chapter concludes with a brief discussion about further implementation details in section 7.5.

The UML diagrams presented in this chapter are given in UML 2 notation. The names of classes, methods, functions, variables or other programming language constructs are highlighted by using a typewriter font. In favor of readability, we usually omit the parameter lists of methods and functions, but their names are followed by a pair of opening and closing parentheses, e.g., method() or function().

7.1 Framework Overview

Our framework is structured by packages as shown in the UML package diagram in figure 7.1. The packages categorize the contained classes and functions according to their domains, and each package corresponds to a C++ namespace.

Figure 7.1: Our framework is structured according to the depicted packages which contain C++ classes and functions for different purposes. Arrows indicate dependencies among the packages.

In detail, these domains are as follows:

**common**
This package contains classes and functions for general tasks, e.g., providing base classes
for threads or for asynchronously reading and writing of files.

**database**
The components from this package provide access to the database management system in order to retrieve and store aerial images and associated metadata.

**FCM**
Components related to our FCM implementation, like classes for handling tiles, tile layers and caching, are placed in the FCM-package.

**image**
The image-package contains classes for processing 2D raster images, i.e., bitmaps, and includes classes and functions for reading and writing different image file formats. In addition, basic image processing functionality for bitmap images of different pixel formats has been implemented and placed in this package.

**math**
Classes and functions representing mathematical objects and operations, e.g., vectors, matrices and quaternions, are placed in the math-package. These classes and functions are used by numerous other components from other packages. For the remainder of this chapter, we avoid to explicitly reference this package in favor of readability.

**rendering**
This package contains classes for DSM rendering tasks and GPU computations for the stereo matching process (see section 5.3).

**spatial**
We implemented versatile C++ class templates for the R*-tree and a point kd-tree in order to use these data structures with arbitrary data types for keys and values in 2D and 3D. These two kinds of spatial indexes together with common class templates for geometric primitives like points and boxes are contained in the spatial-package.

Due to requirements in the AVIGLE project, we need to access OpenGL via the open source rendering engine “Irrlicht” [38]. We had to add several features available in OpenGL to the engine, e.g., support for 3D textures and GLSL 1.5, because Irrlicht does not support these features at the time of writing this dissertation. Rendering is therefore performed by accessing components of the Irrlicht engine instead of directly calling OpenGL API functions. Classes in our framework which utilize the rendering engine are designed to isolate code which is specific to this engine, and are placed in the rendering-package. These classes are prefixed by the letters Irr, and usually inherit from an abstract base class which may be from a different package, such as FCM or common.
7.2 Flexible Clipmap Implementation

The Flexible Clipmap is implemented by several classes and class templates in the FCM-package. Figure 7.2 shows the relationships between the most important classes. Attributes and methods, as well as nested classes, some base classes from other packages, enumerations and auxiliary structures, are omitted in favor of clarity.

In our application, an FCM tile is represented by the class LayeredTile and may contain up to \(k\) layers (see section 4.2.1), but different applications may require different types of FCM tiles. We therefore reduced the dependency of FCM’s components related to tiles on a specific type of tile by using C++ templates and by providing an unspecific base class Tile. The layers of a tile are represented by the class TextureTileLayer, which inherits from the abstract base class TileLayer, and class Image from the image-package. Cacheable provides an interface for any kind of data which need to be transferred asynchronously from secondary to main memory, and keeps track of the current location of cached data in main and secondary memory, e.g., FCM tiles and tile layers. While color texture layers are directly handled by the TextureTileLayer class, DEM layers are handled by the subclass template DEMTileLayer, which additionally stores the maximum and minimum elevation value of its contained data. These values are determined once as the DEM data are loaded or created. The template parameter of DEMTileLayer must be set according to the data type of the desired texture format, e.g., BYTE for I8 or float for F32.

Caching of FCM tiles should work in the same way for all kinds of tiles, hence the template parameter for instantiating the C++ class templates TileCache, TileArray and TileLevelCache must be a subclass of Tile. A TileCache maintains one TileLevelCache object, which in turn maintains \(L\) TileArray objects, where \(L\) denotes the current number of clip levels of the FCM. TileArray objects are 2D arrays for storing \(c_x(l) \cdot c_y(l)\) tiles from the clip area of the FCM at an associated clip level \(l\) in main memory (see section 3.3.2). Class TileLevelCache implements the FCM’s tile level cache as described in section 3.3.2 and contains one TileArray object per clip level. The TileLevelCache object and its contained TileArray objects are resized if the number of clip levels changes. An instance called LayeredTileCache of the TileCache class template for caching LayeredTile objects in our application is obtained via an appropriate C++ typedef.

Asynchronous access to secondary memory for loading and writing tile layers is controlled by the class TileCache, but is executed by an internally maintained FileStorage object. The class FileStorage inherits from the abstract class Thread class from the common-package and executes code in its own thread. In this way, potentially slow accesses to secondary memory do not overly obstruct the execution of other components. Details about the components which are executed in parallel are presented in section 7.2.1.

The numerous parameters for instantiating a FlexibleClipmap object, e.g., sizes of the clip area and the active area, tile size and descriptions of the employed tile layers, are combined in the FCMSettings structure. TileLayerDescriptor objects describe the different tile layers employed by LayeredTile objects by providing a collection of the
Figure 7.2: Overview about the relationships of the most important classes in the FCM-package. Attributes and methods, as well as some less important references to classes from other packages are omitted in favor of clarity.
properties of the layers, such as texel formats and storage locations in secondary memory, i.e., file system paths and folder names. TileLayerDescriptor objects are maintained by instances of FCMSettings structures, and are copied to an instance of the TileLayerConfiguration class which implements the software engineering design pattern Singleton [31, pp. 157–166]. The C++ enumeration LayerIndex provides symbolic constants for accessing the different layers of FCM tiles.

The class FCMState is responsible for keeping track of the state of the FCM and in particular of its current number of clip levels $L$. State changes of the FCM are usually triggered by a TileCreator object, and are propagated to other components by means of callback objects, which are not depicted in figure 7.2. A TileCreator object is responsible for creating new FCM tiles and updating existing ones by means of a LayerCreator object, as soon as new aerial images and associated metadata become available. The class LayerCreator implements the methods for creating DSM data presented in chapter 5. For this purpose, an instance of this class monitors the AerialImageIndex Singleton. The latter maintains an appropriate instantiation of the RTree class template, and provides methods for image retrieval. RTree is an implementation of the R*-tree by Beckmann et al. [6] by means of C++ class templates and is explained in detail in section 7.3. Since the creation of tiles should not obstruct ongoing rendering, the class TileCreator inherits from class Thread, so that the creation of tiles is executed in its own thread. Furthermore, the creation of additional tiles directly affects the state of the FCM and its cache. Hence, related TileCreator and FlexibleClipmap instances need to share the same FCMState and TileCache objects in order to access the data of FCM tiles in a consistent way.

The classes TileUpdateSchedule and TileUpdateJob both inherit from more general classes from the common-package and are responsible for organizing the order of execution of updating and creating tiles as described in section 3.3.3. Aerial images are delivered to the AerialImageIndex Singleton concurrently by an AerialImageCollector object. This object also executes code in a separate thread and monitors changes on the database. The communication between the different classes in response to changes of the state of the FCM due to the arrival of new aerial images is explained in detail in section 7.2.2.

Client applications that utilize our FCM implementation need to create one object each of the classes FCMSettings, FlexibleClipmap and TileCreator. An FCMSettings object can be created conveniently from FCM configuration settings stored in an XML file. Such an object is the only parameter required by the constructor of class FlexibleClipmap. A TileCreator object is only required if the FCM needs to handle DSM data of time-varying extension. In order to render a static DSM, the FCMSettings object can provide a fixed size for the FCM.

7.2.1 Multithreading

Our implementation makes use of multithreading in order to not overly obstruct the main process which is primarily engaged in rendering at real-time frame rates. The support for
multithreading depends on the operating system, and specific code is isolated in our implementation by the abstract base class `Thread` as shown in figure 7.3.

![UML class diagram of classes in our FCM implementation which execute code in separate threads. The class `Mutex` is used for synchronizing access to shared code and data via mutual exclusion.](image)

A class which inherits from class `Thread` must implement the method `run()`. Code within this method is executed in its own thread, as well as the code of other methods which are called directly or indirectly by `run()`. The execution of a thread is initiated by calling `start()` and terminates as soon as `run()` has finished. If a thread needs to run until its termination is explicitly requested by another thread, calling method `stop()` sets an internal state which causes the method `isRunning()` to return `false` on subsequent calls. Hence, loops within the method `run()` should exit if the termination of the thread is requested, so that `run()` can terminate. An example of how to implement such a loop in a fictional class `MyClass` which inherits from `Thread` is given in listing 7.1.

```cpp
void MyClass::run()
{
    while (isRunning())
    {
        // ...
    }
}
```

**Listing 7.1:** Example of an implementation of a loop within the `run()` method of the fictional subclass `MyClass` of class `Thread`. The code within this method is executed in its own thread.

The method `waitFor()` provides a simple synchronization of `Thread` objects. A caller of `waitFor()` passes a timeout period as parameter and is blocked until either the timeout period expires or the thread terminates. Hence, this method should never be called from method `run()` or any other method of the same object in order to avoid deadlocks. A `Thread` object can be suspended from execution for a given timeout period by calling the
method `sleep()` from the object’s `run()`-method or any other method that is directly or indirectly called by `run()`, without causing any deadlock.

Multiple threads may need to access the same methods and data of the same objects, and hence need to be synchronized in order to prevent memory access violations. In our application, we control the access to shared data and code by means of simple mutual exclusion, which is implemented by the `Mutex` class and depends on the operating system.

In the following, we discuss details about the three subclasses of class `Thread` which are shown in figure [7.3], because these classes enable the usage of our FCM implementation for rendering while image acquisition and DSM generation are executed in parallel.

### Class `FileStorage`

The class `FileStorage` provides access to secondary memory, because the other threads, e.g., the main thread for rendering, should not be blocked by potentially slow file access operations. Instead of accessing files themselves, other threads can delegate this task to a `FileStorage` object which maintains a queue of `FileStorageOperation` objects as shown in figure [7.4]. Classes such as `BitmapImage2D` from the `image`-package can inherit from the subclasses `ReadFileCallback` and `WriteFileCallback` of class `FileStorageCallback` and re-implement the callback methods `callbackReadFile()` and `callbackWriteFile()` in order to asynchronously transfer data between secondary and main memory. These callback methods are invoked by `FileStorageOperation` objects after the storage operation is completed. New `FileStorageOperation` objects are inserted into an internally maintained queue of a `FileStorage` object by passing the object which is about to be transferred between main and secondary memory as parameters to the methods `readFileAsync()` or `writeFileAsync()`.

![Figure 7.4: The depicted classes are used for asynchronous file transfer between secondary and main memory.](image)

The thread associated with the `FileStorage` object will be started automatically, if it is not already

```plaintext
"...
```
running. The method \texttt{run()} of \texttt{FileStorage} processes the object’s queue and executes until either the queue is empty or the thread is requested to terminate.

In order to avoid unnecessary storage operations on the same files if they are accessed multiple times in a row, e. g., by the cache of the FCM, operations are only enqueued in the follow situations:

- Read operations are only enqueued if the queue does not already contain a read operation or if an already enqueued read operation is followed by an enqueued write operation on the same file.
- Write operations are only enqueued if no write operation is already enqueued or if an existing write operation is succeeded by a read operation on the same file.

This is expressed in the truth table given in table 7.1. The columns \textit{contains read} and \textit{contains write} indicate the existence of an already enqueued read and write operation on the same file, respectively, and column \textit{read first} indicates that a reading operation on the same file is enqueued before writing to the same file.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
 contains read & contains write & read first & enqueue reading & enqueue writing \\
\hline
 true       & true       & true      & true           & false     \\
 true       & true       & false     & false          & true      \\
 true       & false      & true      & false          & true      \\
 false      & true       & false     & true           & false     \\
 false      & false      & false     & true           & true      \\
\hline
\end{tabular}
\caption{Truth table for situations which require to enqueue a storage operation on the same file at a \texttt{FileStorage} object.}
\end{table}

Note that any objects which inherit from one or more of the \texttt{FileStorageCallback} subclasses must not be deleted, as long as they are referenced by a \texttt{FileStorageOperation} object which is enqueued at a \texttt{FileStorage} object, because otherwise the callback would cause a memory access violation.

\textbf{Class AerialImageCollector}

The class \texttt{AerialImageCollector} is responsible for retrieving aerial images from a database. For this purpose, the images are each identified by a number (\textit{ID}) and a timestamp. Code, which is independent of the underlying database management system (DBMS), such as the method \texttt{run()}, is contained in the base class \texttt{AerialImageCollectorBase} (see figure 7.5). The method \texttt{run()} of \texttt{AerialImageCollectorBase} given in listing 7.2 calls virtual methods, which are implemented by derived subclasses and specific to the employed DBMS.
Figure 7.5: The base class AerialImageCollectorBase provides an interface for database-specific implementations which are called by the run() method.

Listing 7.2: C++ code of the run() method of class AerialImageCollectorBase. availableImages_ is an attribute for storing the state of retrieval for each of the available aerial images.

```cpp
// This typedef is used in the declaration of AerialImageCollectorBase.

//

typedef std::set<AerialImageCollectorBase::ImageState> StateSet;

void AerialImageCollectorBase::run()
{
    StateSet::iterator itNext = availableImages_.end();
    while (isRunning())
    {
        // Attempt to connect to database as long as this thread is running
        // and no connection has been established.
        //
        for (unsigned int i = 0; (isRunning() && (! isConnectedToDB())); ++i)
        {
            if (i > 0)
                sleep(500); // wait 500 ms between each two connection attempts
                connectToDB();
        }

        // If the thread is terminated while it attempts to connect to the
        // database, isRunning() will return false and we go back to the
        // beginning of the outer while-loop in order to check if the thread
        // needs to continue to execute.
        //
        if (! isConnectedToDB())
            continue;

        // The set of available images has not yet been completely retrieved.
        //
        if (itNext != availableImages_.end())
        {
            // Only try to retrieve aerial images if they have not yet been
```
const ImageState& is = *itNext;
if (! is.isRetrieved_)
{
    // Mark the image if it was successfully retrieved.
    // Otherwise, assume that something is wrong with the image
    // (maybe it became deleted meanwhile) and remove it from the
    // set of available images.
    if (fetchImage(is.imageID_, is.timestamp_, false))
    {
        // possible since isRetrieved_ is declared as mutable
        ++itNext;
    }
    else
    {
        availableImages_.erase(itNext++);
    }
}
else
{
    ++itNext;
}
else
{
    // If the iterator indicates that the set has been processed,
    // update the set of images and reset the iterator.
    if (queryAvailableImages() > 0)
    {
        itNext = availableImages_.begin();
    }
}  // while (isRunning())
disconnectFromDB();
}  // run()

Listing 7.2: C++ code of the run() method of class AerialImageCollectorBase. availableImages_ is an attribute for storing the state of retrieval for each of the available aerial images.

Class TileCreator

The implementation of the method run() in class TileCreator basically comprises the processing of its internal schedule for tile creation (cf. section 3.3.3). This class and its most important relationships to other classes are shown in the UML class diagram in figure 7.6. Since tile creation is performed on the GPU, the tile creator creates its own OpenGL rendering context by means of an IrrLayerCreator object which is directly created at the beginning of method run(). The class IrrLayerCreator implements the interface of its base class LayerCreator and performs the actual texture creation of the different DSM layers as described in section 5.3 and section 5.4 by using components of the “Irrlicht” rendering engine [38]. In this way, the main thread can continue to render, and both threads can act as if they had exclusive access to the graphics hardware. Before the method run() of class TileCreator returns, the LayerCreator object and hence the associated OpenGL rendering context are destroyed.
7.2 Flexible Clipmap Implementation

The depicted classes implement the creation of DEM and color texture layers of FCM tiles. Tiles are retrieved by TileCreator via a LayeredTileCache object, which in turn manages LayeredTile objects. The tiles and the schedule are processed in the run() method of class TileCreator.

7.2.2 Communication in Response to the Insertion of Aerial Images

The communication between the affected classes in response to the insertion of an aerial image is illustrated in the communication diagram\(^\text{10}\) in figure 7.7. An equivalent sequence diagram of the same communication is given in figure B.2 in appendix B. The involved classes and their methods are shown in figure 7.8. This figure contains additional details about the class hierarchy which are omitted in figure 7.2 in favor of readability.

![Diagram](image)

**Figure 7.7:** The communication diagram shows the interaction between different objects in response to the insertion of an aerial image at the AerialImageIndex object by calling addImage(). Images are retrieved from a database by an AerialImageCollector object, which executes code in a separate thread.

The insertion of an AerialImage object at AerialImageIndex is triggered by an AerialImageCollector object. AerialImageCollector is connected to the database for storing aerial images, and it monitors the arrival of new images. As soon as

\(^{10}\)This type of diagram was called collaboration diagram in UML versions prior to 1.5 [73, p. 27].
Figure 7.8: The classes in the depicted diagram are affected by the insertion of new aerial images at the AerialImageIndex object after addImage() is called.
new data are available, they are fetched and inserted into the spatial index maintained by the AerialImageIndex object by calling its method addImage(). In order to notify the TileCreator object about the presence of new aerial images, AerialImageIndex and TileCreator are implemented by using the Observer design pattern [31, pp. 287–300]. Directly after its creation, a TileCreator object registers itself as an observer at the AerialImageIndex Singleton, which inherits from Observable. Each time new aerial images are added to the AerialImageIndex Singleton, it notifies its registered observers by calling notifyObservers(), which in turn calls the observers’ specific implementation of the method onNotifyFromObserved(). The method onNotifyFromObserved() of class TileCreator sets an internal flag to indicate the presence of new aerial images. If this flag is present, determineLevel0Tiles() is called during the next execution of the method run() of TileCreator, and the flag is reset. The method determineLevel0Tiles() will determine those FCM tiles at the base clip level \( l = 0 \) which need to be created or updated. Whenever the creation of new tiles causes the addition of additional clip levels as described in section 3.2.5, method resize() on the FCMState object is called. The FCMState object is maintained by the FlexibleClipmap object, which registers the method onFCMStateResize() as a callback at the FCMState object. The callback is invoked in response to changes of the state of the FCM which require to resize it. Since the size of the cache of the FCM must be adjusted as well, the method onFCMStateResize() of class TileCache is called by FlexibleClipmap.

After the eventual resizing of all involved components, the method run() of TileCreator can proceed with the creation of FCM tiles by calling createLevel0Tiles(). This call will also lead to the scheduling of FCM tile creations at clip levels \( > 0 \), and each time tiles have been created TileCreator calls its method notifyObservers(). In this way, objects like FlexibleClipmap can register themselves as observers and become notified about the creation of new tiles via calls to their implementations of onNotifyFromObserved(). This is required, for instance, by class FlexibleClipmap in order to call updateTileArray(), which is responsible for initiating the transfer of modified tiles into video memory for rendering.

As shown in figure 7.8, TileCache inherits from CallbackOnResize and can hence register a callback method at an FCMState object as well. This allows to use a TileCreator object without having to create a FlexibleClipmap object, if no rendering of the created DSM is required. In this case, the required FCMState and TileCache objects must be directly created by the application, because they are usually managed by the FlexibleClipmap object and are only accessed by FCMState via references. The TileCache object can thus directly register its onFCMStateResize() method as a callback at the FCMState object and becomes notified about changes of the number of clip levels in order to initiate eventual resizing of its internal tile level cache.
7.3 R*-tree Implementation

The R*-tree by Beckmann et al. [6] is implemented by the C++ class templates shown in figure 7.9.

![Figure 7.9: We implemented the R*-tree by the depicted C++ class templates. Note that the classes EntryType, IntermediateNode and LeafNode are generated by template instantiation in RTree as indicated by the UML annotation, and that they are available only within instances of that class template.]

The four template parameters of the class template RTree representing the R*-tree itself must be used as follows: The parameter DIM indicates the dimension of the space and is DIM = 3 in the context of our work, but DIM may have any value ≥ 1. N defines the maximal number of entries per node, and the minimal number of entries is set to m = \( \frac{N}{2} \) (cf. section 3.2.3). Specifying the number of entries at compile time allows to use statically allocated arrays in the node classes for storing entries and hence simplifies memory management. The data type which is stored in the R*-tree is indicated by parameter T and is a pointer to AerialImage objects in our application. Since we want to automatically deallocate memory for objects which are removed from an instance of the class template RTree, we pass an appropriate Deallocator class template as the fourth template parameter DEALLOC. The Deallocator class template must be chosen with respect to the allocation method for data stored at the R*-tree. Therefore, we provide three classes for this task, each one for the most common memory allocation methods used in C++. The base class Deallocator does not deallocate any memory and is intended for being used with parameters T which are no pointers. DeallocatorDelete and DeallocatorDeleteArray deallocate memory by using C++ delete and delete[] operators, respectively. As indicated by the UML annotation in figure 7.9, the four template parameters used by RTree are also used for instantiating the RTreeDataEntry class template, which represents entries at the leaf nodes of the R*-tree. The meaning of these parameters is the same as with the RTree class template, and the instantiated template is made a new
type \texttt{EntryType} via a C++ \texttt{typedef}, and it is only available within instances of RTee. The memory allocated by the actual data of type \texttt{T} of an \texttt{EntryType} object is deallocated within its destructor by a local deallocator object.

The data are stored in an R*-tree only at its leaf nodes, which are represented by \texttt{LeafNode} objects. \texttt{IntermediateNode} objects are used for handling intermediate nodes, and both types are instantiations of the RTeeNode class template. The template parameter \texttt{ENTRY} is set to the data type of the node’s entries and is \texttt{EntryType} for leaf nodes. In case of intermediate nodes, the entries can be either leaf nodes or other intermediate nodes, and \texttt{ENTRY} is set to RTeeNodeBase, which is the base class template of all types of nodes. If access to the data is required, the generalized node objects are casted back to pointers of their respective subclasses, based on a node’s level within the R*-tree.

Since all nodes and entries must maintain an axis-aligned bounding box (AABB), RTeeDataEntry and RTeeNodeBase inherit from an instance of the class template RTeeChild, which provides amongst others a RHyperRect<\texttt{NUM}, \texttt{DIM}> object. The latter represents the actual AABB and consists of two \texttt{RPoint<NUM, DIM>} objects, each one for the minimum and the maximum corner. These two class templates provide the basic operations for determining geometric relationships between each other, like containment, overlap, coverage and intersection, for instance. The template parameters \texttt{NUM} and \texttt{DIM} of RHyperRect and \texttt{RPoint} must be set to a numerical data type, e.g., \texttt{int}, \texttt{float} or \texttt{double}, and the dimension of space, respectively.

Operations for manipulating the content of the R*-tree, e.g., \texttt{insert()}, \texttt{find()} and \texttt{remove()} as well as operations related to its structure are implemented by methods of the RTee class template. The operations described in \cite{35} and \cite{5} for splitting nodes, like \texttt{ChooseSplitAxis} and \texttt{SplitNode}, are implemented by the class template RTeeNode.

### 7.4 GPU Programs

In this section we discuss the most important details and functions of the GLSL vertex and fragment programs for DSM rendering and DSM synthesis. We employ GLSL version 1.5 in order to have access to texture arrays, i.e., \texttt{sampler2Darray} data types. The complete shader source codes can be found in appendix A.

Like C/C++ programs, GLSL shaders can make use of preprocessor definitions \cite[pp. 93–97]{66}. Some of these definitions are generated as textual strings at runtime by our FCM components written in C++, e.g., in order to provide constants for allocating arrays of fixed size (cf. section \ref{5.3.1}). The resulting strings are inserted in front of the shader source code before the shaders are built by the graphics driver. The additional preprocessor directives are inserted as comments at the beginning of the source code for illustrative purposes. The values of such \texttt{#define} directives are chosen according to typical values from our application.

In the GLSL code listings presented in this dissertation, the names of uniform variables contain a trailing underscore, e.g., \texttt{worldSize_}. The names of variables of fragment programs qualified as \texttt{in}, and the ones of vertex programs qualified as \texttt{out}, are prefixed
by an underscore, e.g., _texCoord0. Attributes of structures are also suffixed by an underscore, but can be easily distinguished from uniform variables, since they appear outside of struct definitions on the right-hand side of an element access operator, i.e., a dot. Furthermore, the world coordinate system that is used by the Irrlicht rendering engine \[38\] differs from the world coordinate system \(\Sigma\) as presented in section \[2.4\] by swapped labels of the y- and z-axis. Hence, positions in the ground plane are denoted by \(p.xz\) instead of \(p.xy\) and height values are stored at \(p.y\) instead of \(p.z\). In addition, the red and blue color components of RGB color textures are swapped by the rendering engine as well, so that we have to access the components of an RGB color value \(\text{texColor}\) from a texture via \(\text{texColor.bgr}\) instead of \(\text{texColor.rgb}\).

### 7.4.1 DSM Rendering

The GPU programs for DSM rendering are associated by subclasses of FlexibleClipmap, which contain code specific to rendering by means of a certain API or graphics engine. The vertex program is given in listing \[A.1\] in appendix \[A\]. Its task is to transform the vertex coordinates of the polygonal box-shaped mesh, which serves as proxy geometry for ray casting, into screen space, and to compute normalized 3D texture coordinates. The texture coordinate \((0, 0, 0)\) is located at the minimum corner and \((1, 1, 1)\) at the maximum corner. In addition, the vertex program computes the ratios of the two shorter sides to the longest side of the box. If the proxy geometry is not a cube, ray traversal during ray casting would require a non-uniform step width along the different axes. By component-wise multiplication of the direction of the ray with the ratio of the bounding box, we can use a uniform step width for ray traversal.

The function main() of the GLSL fragment shader for surface rendering in our FCM implementation and the definition of the employed RayPosition structure are given in listing \[7.3\]. The information that is stored at the current position of a ray in a RayPosition structure is indicated by the comments in listing \[7.3\]. In line \[969\] a preprocessor definition added by the client program determines whether the FCM uses DEM ray casting or not, and the shader either executes the function rayCasting() or planarFCM(). In the latter case, the FCM can be used for texturing a planar surface as described in chapter \[3\]. Both functions return a RayPosition structure which contains the final grid position, and the required clip levels for sampling the color texture layer. The attribute hit indicates whether the grid position is located on the DEM surface. The check in line \[983\] is used to discard fragments at positions outside the domain of the FCM, e.g., if the ray has left the boundaries of the proxy geometry during ray casting.

If DEM ray casting is employed by using rayCasting() in line \[978\], the texture coordinate which is passed by the vertex program via _texCoord0 encodes the position where the corresponding ray exits the proxy geometry. The CoP of the virtual camera is transformed by the vertex program into the proxy geometry’s normalized texture coordinate system and is passed to the fragment shader via _camPosTextureCoordinates. The check in line \[971\] verifies that the coordinates of the camera position are in range.
Listing 7.3: The main() function of the FCM fragment program for DSM rendering and the employed structure RayPosition.

[0, 1] × [0, 1] ⊂ R^2 and that the camera is thus located inside the proxy geometry. In this case, ray casting starts at the position of the camera. Otherwise, the camera is located outside the box, and the corresponding entry points of the ray on the proxy geometry are computed by the function getBoxIntersection() in line 974. The direction of the ray is then computed in line 977 and is multiplied component-wise by the ratio of the box.

If the FCM is used without DEM ray casting by calling planarFCM() in line 980, the FCM is located in the ground plane at the bottom of the proxy geometry box. In this case, positions for sampling its color texture layers can be thought of as the intersections of rays with a completely flat surface. Hence, planarFCM() merely transforms the normalized texture coordinate into the grid coordinate system by multiplying it by the grid size as used for ray casting and determines the optimal clip level that best avoids aliasing as described in section 3.3.4. This transformation into the grid coordinate system allows to handle the
subsequent computation of the final fragment color and the Z-Buffer value by the function `determineFragmentValues()` in line 986 in the same way as with ray casting.

The computation of the final fragment color is controlled by the preprocessor definition `COLORING_MODE`, which causes the function `determineFragmentValues()` to call code for either sampling the color texture layer, sampling projective textures, or applying diffuse shading with optional shadow casting. If no DEM ray casting is employed, sampling the color texture layer is the only available option. The computation of shadows relies on the same occlusion test by means of secondary ray casting, which is also used to determine the visibility of a location on the DEM surface in a projective texture (see section 6.2.2).

The code for initializing ray casting is shown in listing 7.4.

```glsl
uint getDirectionCode(const in vec3 dir)
{
    // prevent loss of sign by truncations to zero for small values
    ivec3 iDir = ivec3(dir * 1.0e8);
    uint dirCode = uint(((iDir.x >> 31) & 1)); // sign of x-component (bit 0)
    dirCode |= uint(((iDir.y >> 30) & 2)); // sign of y-component (bit 1)
    dirCode |= uint(((iDir.z >> 29) & 4)); // sign of z-component (bit 2)
    return dirCode;
}

RayPosition rayCasting(const in vec3 p, const in vec3 dir)
{
    RayPosition rp;
    rp.pGrid_ = (p * g_gridSize);
    rp.levelCurrent_ = MAX_CLIPLEVEL;
    rp.levelLowest_ = 0u;
    rp.levelIdeal_ = 0u;
    rp.numSteps_ = 0u;
    rp.hit_ = false;
    // Determine the target octant in space for the ray.
    // Since the intersection tests with the DEM surface are different
    // for rays directed upward and downward, use specialized functions.
    uint dirCode = getDirectionCode(dir);
    if ((dirCode & 2u) != 0u)
        castRayDownward(rp, dir, dirCode);
    else
        castRayUpward(rp, dir, dirCode);
    return rp;
}
```

Listing 7.4: GLSL source code for initializing DEM ray casting.

After the initialization of a `RayPosition` structure, the direction of the ray is encoded by the bit-mask `dirCode`, which is created by the function `getDirectionCode()`. This bit-mask is used for efficient branching during ray traversal via `switch`-statements. If the x-, y- or z-component of the ray’s direction vector are negative, the lower three bits of `dirCode` are set accordingly and thus indicate the octant in 3D space into which the
ray is casted. Since computing the intersection of a ray with the DEM depends on whether the ray is directed upwards or downwards (cf. equation 4.2 in section 4.2.2), the function `rayCasting()` branches accordingly.

The two functions `castRayDownward()` and `castRayUpward()` essentially differ only in the intersection test. `castRayDownward()` is given in listing 7.5.

**Listing 7.5: GLSL source code from the ray casting branch for downward directed rays.**

```glsl
void castRayDownward(inout RayPosition rp, const in vec3 dir, const in uint dirCode)
{
    uint n = 0u; // counter for ray casting steps at the current clip level
    while (((! rp.hit_) && (isInsideRange(rp.pGrid_, vec3(0.0), g_gridSize)))
    {
        ++rp.numSteps_;
        vec4 pNext = getNextCell(rp, dir, dirCode);
        // If an intersection is detected, compute the exact hit point
        // on the box of the grid cell.
        // The w-component contains the DEM value at the next grid cell.
        // The y-component is the current height of the ray above the ground plane.
        if ((pNext.y - pNext.w) <= EPSILON_HIT)
        {
            rp.pGrid_ += dir * max(((pNext.w - rp.pGrid_.y) / dir.y), 0.0);
        } // Determine the different clip levels. 
        rp.levelLowest_ = getTileMapEntry(rp.pGrid_);
        float idealLOD = computeIdealLOD(rp.pGrid_);
        rp.levelIdeal_ = uint(idealLOD);
        rp.lodBlend_ = fract(idealLOD);
        // If the corresponding box of the grid cell at the current hit point
        // is from a clip level greater than the clip level which
        // would expose the least aliasing, descend one level.
        // Otherwise, ray casting can terminate, and surface can be
        // refined.
        uint bestLevel = max(rp.levelLowest_, min(rp.levelIdeal_, MAX_CLIPLEVEL));
        if (rp.levelCurrent_ > bestLevel)
        {
            --rp.levelCurrent_;
            n = 0u; // reset counter for ray casting steps at current level
        }
    } // defined(SURFACE_REFINEMENT) && (SURFACE_REFINEMENT == SURFACE_REFINEMENT_LINEAR))
    else
        rp.hit_ = refineSurfaceLinear(rp, dir, dirCode);
    } else (defined(SURFACE_REFINEMENT) && (SURFACE_REFINEMENT == SURFACE_REFINEMENT_BICUBIC))
    else
        rp.hit_ = refineSurfaceBicubic(rp, dir, dirCode);
    else
        rp.hit_ = true;
    } // SURFACE_REFINEMENT
}
else
    // the DEM was not hit by the ray => proceed at the next grid cell
```
Listing 7.5: GLSL source code from the ray casting branch for downward directed rays.

The while-loop terminates if the ray has either hit the DEM surface, or if it leaves the domain of the FCM. Based on the current grid position and the direction of the ray, the next grid cell for ray traversal as described in section 4.2.2 is determined by the function getNextCell(). This function furthermore computes the position for sampling the DEM layer texture and retrieves the corresponding DEM value, which is stored at the fourth component (w-component) of the returned vec4. If the check in line 690 indicates that the ray intersects the DEM, the exact position on the grid cell’s box surface is computed and the clip levels are determined according to the description in section 4.2.3. If the current clip level is greater than the clip level which best avoids aliasing, the level is decreased and ray casting continues from the current hit point. Otherwise, the final intersection of the ray with the surface has been found, and ray casting can be terminated. The surface refinement methods as presented in section 4.2.4 are applied only if the corresponding preprocessor definition SURFACE_REFINEMENT is present, and if its value is either as in line 712 or as in line 715. This directive is added to the shader source code by class FlexibleClipmap, if it is requested by the user of the client application. Since the surface refinement methods may invalidate the final hit point on the box surface of the grid cell, ray casting is resumed from that position if refineSurfaceLinear() respectively refineSurfaceBicubic() return false.

If the DEM is not hit by the ray at the current fragment, the ray is advanced to the next grid cell in line 726 and the counter for ray casting steps at the current level is increased. After four steps without any further intersection of the ray with the DEM, the level is increased again if it is not the top-most level. In this way, the effective size of the grid cell and thus the step width for ray traversal is increased (cf. section 4.2.2).
7.4.2 DSM Synthesis

The complete GPU programs for generating the different DSM layers of FCM tiles as described in chapter 5 are given in listing A.3 and listing A.4 in appendix A. The shaders create the DEM and the color texture layer of an FCM tile simultaneously by emitting two color values. Aerial images which are projected onto the sweeping plane and their associated projector matrices are passed to the programs via the uniform arrays projectedTextures_ and mProjectors_.

The vertex program computes the texture coordinates for each projected image at the vertex positions of the rectangular mesh which represents the sweeping plane. These coordinates are passed on to the fragment program via the array _texCoords and are automatically interpolated across the sweeping plane. In the fragment program, the coordinates are transformed from homogeneous coordinates to Cartesian ones. Listing 7.6 shows the structure ImageInfo for storing the information about each projected image as indicated by the source code comments. The fragment shader’s most important function computeMatchingCosts() is given in listing 7.7.

Listing 7.6: Structure ImageInfo for storing information about an aerial image at the current fragment from the fragment program for DSM synthesis.

```cpp
struct ImageInfo
{
    vec4 colorValue_; // color value from corresponding image
    float costs_; // matching costs for image
    float luminance_; // luminance value (derived from RGB texel value)
    uint imageIndex_; // index of the image
    bool covers_; // true if the image covers this fragment
}; // struct ImageInfo
```

Listing 7.7: Function computeMatchingCosts for computing the aggregated matching costs at each fragment in the fragment program for DSM synthesis.

```cpp
float computeMatchingCosts(out vec4 color)
{
    float summedCosts = 0.0;
    color = vec4(0.0, 0.0, 0.0, 1.0);
    ImageInfo info[NUM_PROJECTORS];

    // Determine the coverage by projective images at this fragment and
    // initialize the ImageInfo structure at the lowest mipmap level for
    // matching cost aggregation.
    float mipmapLevel = max(float(minSupportWindowLevel_), 0.0);
    uint coverage = 0u;
    for (uint i = 0u; i < NUM_PROJECTORS; ++i)
    {
        coverage += initImageInfo(i, mipmapLevel, info);
        if (info[i].covers_)
            color = info[i].colorValue_;
    }

    // Change to (coverage == N) to get visually most appealing results.
    if (coverage > 1u)
    {
        // add costs from previous pass with smallest support window size
```
for (uint i = 0u; i < NUM_PROJECTORS; ++i)
{
    if (info[i].covers_)
    {
        for (uint j = 0u; j < NUM_PROJECTORS; ++j)
        {
            if (info[j].covers_)
                info[i].costs_ += abs(info[i].luminance_ - info[j].luminance_);
        }
    }
} // for (i

// add costs from larger support windows
for (int l = (minSupportWindowLevel_ + 1); l <= maxSupportWindowLevel_; ++l)
{
    // update luminances
    float mipmapLevel = float(l);
    for (uint i = 0u; i < NUM_PROJECTORS; ++i)
    {
        if (info[i].covers_)
            info[i].luminance_ = getLuminanceUnchecked(i, mipmapLevel);
    }
} // for (l

// same as above
for (uint i = 0u; i < NUM_PROJECTORS; ++i)
{
    if (info[i].covers_)
    {
        for (uint j = 0u; j < NUM_PROJECTORS; ++j)
        {
            if (info[j].covers_)
                info[i].costs_ += abs(info[i].luminance_ - info[j].luminance_);
        } // for (j
    }
} // for (i
} // for (l

#ifdef USE_BETTER_HALF
    // sort matching costs in increasing order
    selectionSort(info, N);
    // use color from image with least matching costs
    color = info[0].colorValue_;

    // sum up costs
    uint k = 0u;
    for (uint i = 0u;
        {((info[i].costs_ < INFINITY) && (i < (coverage / 2u))); ++i, ++k)
        summedCosts += info[i].costs_;

    // normalize costs to simplify comparison with threshold
    summedCosts /= float(k);
#else
    color = vec4(0.0);
    for (uint i = 0u; (i < N); ++i)
    {
        // sum up only valid matching costs (there is at least one valid value)
        if (info[i].costs_ < INFINITY)
        {
            summedCosts += info[i].costs_;
        }
#endif
7.4 GPU Programs

Listing 7.7: Function `computeMatchingCosts` for computing the aggregated matching costs at each fragment in the fragment program for DSM synthesis.

The function `computeMatchingCosts()` returns the aggregated matching costs at the current fragment and additionally assigns a color value to the `out` variable `color`. If all images covering the fragment are taken into account for matching cost computation, the final color value is computed as the averaged pixel value. Otherwise, only those images with lowest matching costs are considered, and the color value is obtained from the image with lowest matching costs. The matching costs can be aggregated over a contiguous range of mipmap levels, which is passed to the shader via uniform variables and corresponds to cost aggregation over different support window sizes (cf. section 5.1.2).

The aggregated matching costs are computed in detail as follows. For each projected image, the function initializes one element of an array of `ImageInfo` structures via `initImageInfo()` and simultaneously determines the number of images that cover the current fragment. The function `initImageInfo()` already samples the projected images at the lowest mipmap level given in `mipmapLevel` for cost aggregation. Furthermore, it converts color values into luminance values, which are stored in the attribute `luminance_` of the passed `ImageInfo` structure. If the fragment is not covered by at least two images, the matching costs at the fragment are set to `INFINITY` in line 219. Otherwise, the SAD of the intensities obtained for the images covering the fragment at the lowest mipmap level are computed and are aggregated in the `costs_` attribute in line 154. The coverage of image $I_k$ at the current fragment is indicated by the boolean attribute `covers_`, and projected images not covering that fragment are assigned `INFINITY` as matching costs. If the costs are aggregated over multiple mipmap levels, i.e., over different sizes of support windows, for each additional level the luminance values for the images are updated in the for-loop starting at line 161. In lines 172 to 182 the additional costs are computed and aggregated in the same way as at the lowest mipmap level.

After the computation of the matching costs for each projected image for the current fragment, the costs are summed up to obtain the aggregated matching costs. If the GLSL preprocessor definition `USE_BETTER_HALF` exists, the array of `ImageInfo` structures is sorted by the function `selectionSort()` in line 187 in increasing order of matching
costs. Since images whose projections do not cover the current fragment have matching costs of `INFINITY`, which is implemented as a very large constant value, these images do not contribute to the summed costs. In addition, the color value sampled from the image with least matching costs is assigned to the `out` variable `color` and thus determines the final value of the corresponding texel in the color texture layer of the DSM.

If all images are considered for computing matching costs, images with matching costs of `INFINITY` are not taken into account during cost aggregation in line 207 because they do not cover the current fragment. In this case, the final value of the color texture layer is computed in line 208 and line 213 as the averaged color value of pixels from images covering the fragment.

### 7.5 Further Implementation Details

A detailed class diagram of the most important class `FlexibleClipmap` is presented in figure B.1 in appendix B.

The classes and functions from the packages `common`, `database`, `math` and `rendering` shown in figure 7.1 basically support those classes, which have been presented so far. Classes and functions from the packages `database` and `rendering` are furthermore specific to the underlying database system and rendering engine, respectively, and possess external dependencies on different code libraries, but their implementations are straightforward. The `math`-package essentially comprises classes and functions for representing mathematical objects like vectors, matrices and quaternions, but makes extensive use of C++ templates. These components are thus very versatile, but their implementations are rather lengthy and a little complex. Most of the classes from the `common` package have already been presented, and the remaining ones have straightforward implementations as well. Hence, we do not enter on further details about the implementation of the elements from these four packages. But since processing bitmap images is an important aspect in our application, the relationships of the classes and class templates for image processing from the `image`-package are shown in the UML diagram in figure B.3 in appendix B.
8 Conclusions

This dissertation presents techniques for managing and rendering digital surface models by using texture data of time-varying extension. The FCM has proved useful not only for managing and rendering color texture data, but also for accelerated ray casting of DEM data which are managed in the same way as color textures. Since the FCM allows to handle textures whose extensions may change in the course of time, it can be used for already rendering while the acquisition of the underlying data is still in progress. The bounding volume hierarchy provided by the FCM allows to render elevation data by means of ray casting at real-time frame rates on commodity graphics hardware. Ray casting provides a lot of flexibility, because the data can be rendered without extensive preprocessing and without generating polygonal meshes, which is especially important in the case of frequently changing data as in our application. The methods for surface refinement as presented in this work allow to improve the visual quality of the resulting surface renderings.

Adding color information to renderings of digital elevation models in 3D virtual environments by using orthophoto textures results in a lack of information on lateral surfaces of the elevations. In order to overcome this limitation, we presented a solution based on projective textures. By directly projecting images which depict the lateral faces of surfaces such as oblique aerial images, we can avoid creating large color textures for the entire surface area of a DEM. Since such textures would have to be modified each time the underlying elevation data and images change, projective textures offer more flexibility.

We furthermore presented a simple method for generating DSM data from simulated aerial images captured at low altitudes in an unsorted manner by means of a GPU-based space sweep for stereo matching. With the employed input images, this method allows to obtain credential results within few seconds, while the resulting surface model can be rendered simultaneously. More sophisticated and optimized method can be expected to yield elevation data of even higher quality, although they may be computationally more expensive. The DSM creation process and projective texturing rely on a 3D spatial index such as an R*-tree which supports intersection queries, in order to efficiently retrieve the required input images. In conjunction with the caching techniques of the FCM, this allows to create, manage and render DSMs of almost arbitrary sizes.

Our approach mainly targets at the rendering of DSMs and their creation by means of aerial photographs, which are captured at low altitudes by small unmanned aerial vehicles and are directly transferred to a ground station by wireless networks in order to provide instant feedback about the underlying surface. Since we do not yet have a reliable source of such images available, our results are currently based on simulated aerial images. This is not a limitation for our methods for managing and rendering DSMs by using textures.
of time-varying extension as presented in chapters 3 and 4, because these techniques are independent of the employed image acquisition technique. Other applications in which time-varying data about surfaces are directly available for rendering may benefit from the managing and rendering capabilities of the FCM as well. The technique for texturing the rendered surfaces by means of projective textures as presented in chapter 6 is also independent of the source of images, but strongly relies on accurate position and orientation information about the employed images. In the case of simulated aerial images, these metadata are perfectly accurate, and the resulting renderings therefore barely expose displacements of the textures on the surfaces. The position and orientation data of real aerial images are probably less precise and hence might require preprocessing by means of bundle adjustment or similar techniques, in order to improve the accuracy of these metadata. The accuracy of the metadata of real aerial images could probably also influence the quality of the results of the stereo matching approach as presented in chapter 5 and adjustments of the technique for such images might be necessary.

The runtime performance of our techniques appears to be completely sufficient for our intended applications, but might be further improved by careful optimizations. It would also be interesting to obtain direct comparisons with competitive rendering techniques which are based on rasterization of polygonal meshes. For instance, our FCM implementation could be extended in such a way that it can be used like a Geometry Clipmap [47] for handling triangular meshes of time-varying extension. As tessellation shaders become more available on commodity graphics hardware, the FCM might also be useful for directly rendering DSMs that are represented by bicubic patches stored in texture data instead of using ray casting with applied bicubic surface refinement (see chapter 4.2.4). However, these extensions and further comparisons were not required in the context of our application in the AVIGLE project and have thus been omitted in favor of the development of the techniques presented in this work.

The space sweep approach for stereo matching produces a dense set of 3D points as required for DSM synthesis, but other techniques for stereo matching, like feature-based methods, would also be worthwhile to investigate. We would also have liked to extensively evaluate our DSM creation process for a wider range of input data, but unfortunately had no such data available at the time of writing this thesis.
The GPU programs listed in this appendix were developed and tested on a desktop computer with a NVidia GeForce 470 GTX graphics adapter, 64-bit Windows 7 operating system and NVidia graphics drivers version 306.92.

A.1 Flexible Clipmap Shaders

The following GLSL vertex and fragment programs are employed by our Flexible Clipmap implementation for rendering DSMs by means of ray casting according to chapter 4.

Listing A.1: The complete GLSL vertex program of our FCM implementation.

```glsl
#version 150

uniform mat4 modelMatrix_;  
uniform mat4 viewMatrix_;   
uniform mat4 projectionMatrix_; 
uniform vec3 worldOffset_;   
uniform vec3 worldSize_;     
uniform vec3 cameraPosition_; 

in vec3 vertexPos;           
in vec3 vertexNormal;        
in vec2 vertexTexCoord0;    
in vec4 vertexColor;         

out vec3 _texCoord0; // exit point of a ray on proxy geometry 
out mat4 _modelViewProj;    
out vec3 _camPosTextureCoordinates; // camera position in texture coordinates 
out vec3 _boxRatio; 

vec3 getBoundingBoxRatio() 
{
    vec3 ratio = worldSize_; 
    if ( ((ratio.x >= ratio.y) && (ratio.x >= ratio.z)) ) 
        ratio.yz /= ratio.x; 
        ratio.x = 1.0; 
    else if ( ((ratio.z >= ratio.x) && (ratio.z >= ratio.y)) ) 
        ratio.xy /= ratio.z; 
        ratio.z = 1.0; 
    else 
        ratio.xz /= ratio.y; 
        ratio.y = 1.0; 
    }
```
A GPU Programs

Listing A.1: The complete GLSL vertex program of our FCM implementation.

```cpp
return ratio;
}

void main()
{
_texCoord0 = (vertexPos - worldOffset_) / worldSize_;  
camPosTextureCoordinates = (cameraPosition_ - worldOffset_) / worldSize_;  
_boxRatio = getBoundingBoxRatio();  
_modelViewProj = projectionMatrix_ * viewMatrix_ * modelMatrix_;  
gl_Position = _modelViewProj * vec4(vertexPos, 1.0);
}
```

Listing A.2: The complete GLSL fragment program of our FCM implementation.

```cpp
// The following defines are added by the class FlexibleClipmap after the file containing this code is read.
// The given values are only for illustrative purposes.

#define NUM_CLIPLEVELS
#define TILE_SIZE vec2(512.0, 512.0)
#define SAMPLES_PER_UNIT vec3(4.0, 4.0, 4.0)
#define MAX_FILTERING_ANISOTROPY 16.0
#define EPSILON_HIT 0.0
#define MAX_NUM_PROJECTORS 8u
#define COLORING_MODE COLOR_MODE_SHADED_WITH_SHADOWS
#define FCM_USE_DEM_RAY_CASTING
#define SURFACE_REFINEMENT SURFACE_REFINEMENT_LINEAR

// Indicates the scaling factor for different texel formats of textures storing DEM layers: I8 ~ BYTE = 255.0, F32 ~ float = 1.0

#define SAMPLE_SCALE

// These two arrays only change as the number of clip levels of the FCM changes. Hence their content is generated in form of GLSL code by class FlexibleClipmap and the shader is rebuilt each time the number of clip levels changes.
uniform vec2 numTilesAtLevel_[NUM_CLIPLEVELS] = vec2[NUM_CLIPLEVELS]();
uniform int tileArrayOffsets_[NUM_CLIPLEVELS] = int[NUM_CLIPLEVELS]();

#line 30

#define COLOR_MODE_TEXTURING 1
#define COLOR_MODE_TEXTURING_PROJECTIVE 2
#define COLOR_MODE_SHADED 3
#define COLOR_MODE_SHADED_WITH_SHADOWS 4
#define SURFACE_REFINEMENT_LINEAR 1
#define SURFACE_REFINEMENT_BICUBIC 2

#ifndef FCM_USE_DEM_RAY_CASTING
#undef MAX_NUM_PROJECTORS
#undef SURFACE_REFINEMENT
#endif
```
A.1 Flexible Clipmap Shaders

// ### in/out variables ###

in vec3 _texCoord0; // uniform texture coordinate on proxy geometry
in mat4 _modelViewProj; // combined model, view and projection matrix

// Position of camera relative to the left lower corner of the proxy geometry box.
in vec3 _camPosTextureCoordinates;

// Ratios of the two shorter sides of the proxy geometry box and its longest side. Required for facilitating ray traversal.
in vec3 _boxRatio;

out vec4 fragColor; // the final fragment color

// ### uniform variables ###

uniform vec2 windowSize_; // The current size of the rendering window; required for LOD calculations.
uniform vec3 worldOffset_; // Offset and size of the grid/proxy geometry in world coordinates.
uniform vec3 worldSize_; // uniform sampler2D tileMap_;

#ifndef MAX_NUM_PROJECTORS
uniform sampler2DArray colorLayerArray_;
#endif

#ifdef FCM_USE_DEMRAY_CASTING
// only relevant if DEM ray casting is used
uniform sampler2DArray demLayerArray_;
uniform float heightScaling_; // uniform vec3 dirLight_; // only relevant if projective texturing is used
uniform int numProjectors_;
uniform sampler2D projectiveTextures_[MAX_NUM_PROJECTORS];
uniform mat4 mProjectors_[MAX_NUM_PROJECTORS];
uniform vec3 projectionCenters_[MAX_NUM_PROJECTORS];
#endif // FCM_USE_DEMRAY_CASTING

#endif // FCM_USE_DEM_RAY_CASTING

// ### global variables and constants ###

vec3 g_gridSize = (worldSize_ * SAMPLES_PER_UNIT);
vec3 g_gridOffset = (worldOffset_ * SAMPLES_PER_UNIT);
const uint MAX_CLIPLEVEL = ((NUM_CLIPLEVELS > 0u) ? (NUM_CLIPLEVELS - 1u) : 0u);
A GPU Programs

```c
#ifdef MAX_NUM_PROJECTORS
const mat4 g_mScaleProjector = mat4(0.5, 0.0, 0.0, 0.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.5, 0.5, 0.5, 1.0);
#endif // MAX_NUM_PROJECTORS

uint N = min(uint(numProjectors_), MAX_NUM_PROJECTORS);

// ### structures ###
struct RayPosition
{
  vec3 pGrid_;    // current grid position
  float lodBlend_; // parameter for blending LODs, range [0, 1]
  uint levelCurrent_; // current clip level at ray position
  uint levelLowest_; // lowest clip level available at ray position
  uint levelIdeal_; // clip level of least aliasing at ray position
  uint numSteps_; // current number of ray casting steps
  bool hit_;     // indicates whether the ray has hit the DEM surface
}; // struct RayPosition

// ## functions ##

/**
 * Returns the texel coordinate and the index of a tile within the tile array
 * at the given grid location at the given clip level.
 *
 * @param level Clip level where the tile is located.
 * @param pGrid Grid position of a location.
 */
ivec3 getTileCoord(const in uint level, const in vec2 pGrid)
{
  if (level >= NUM_CLIPLEVELS)
    return ivec3(-1);
  ivec3 pTile;    // xy = uniform texture coordinate, z = tile array index
  float k = float(1u << level); // 2^level
  // Compute the offset of the area covered by the FCM at the grid position
  // in texels. The offset is different at top-most clip level, since the
  // tile at that level covers the entire area of the FCM.
  vec2 offset = floor(g_gridOffset.xz / k);
  if ((level + 1u) >= NUM_CLIPLEVELS)
    offset -= vec2(0.5 * TILE_SIZE);
  // Compute the coordinates of the grid position at the tile in texels.
  vec2 coord = (pGrid / k) + offset;
  pTile.xy = ivec2(mod(coord, TILE_SIZE));
  // Compute the index of the tile in the tile array.
  vec2 tileID = floor(coord / TILE_SIZE);
  vec2 n = numTilesAtLevel_[level];
  // GLSL apparently uses the mathematical modulus function
  // mod(a, b) = a - floor(a / m) * m
  // instead of the symmetrical one which is
  // mod(a, b) = a - div(a, m) * m
  // where div(x, y) = sign(x) * sign(y) * floor(abs(x) / abs(y))
}
```
A.1 Flexible Clipmap Shaders

```cpp
// as used in C/C++. Therefore the expression below can be simplified.
//
//vec2 i = floor(abs(tileID) / n.xy);
//vec2 r = mod(((tileID + (i + vec2(1.0)) * n.xy)), n.xy);
vec2 r = mod(tileID, n);

pTile.z = tileArrayOffsets_[level] + int(r.y * n.x + r.x);
return pTile;
}

/**
* Returns the uniform texture coordinate and the index of a tile within the
* tile array at the given grid location at a given clip level.
* THE COMPUTATIONS OF THIS FUNCTION ARE THE SAME AS IN FUNCTION getTileCoord()
* EXCEPT FOR THE MARKED TWO LINES.
*/
vec3 getTileCoordUniform(const in uint level, const in vec2 pGrid)
{
vec3 pTile;
float k = float(1u << level);
vec2 offset = floor(g_gridOffset.xz / k);
if ((level + 1u) >= NUM_CLIPLEVELS)
    offset -= vec2(0.5 * TILE_SIZE);
vec2 coord = (pGrid / k) + offset;
pTile.xy = (mod(coord, TILE_SIZE) / TILE_SIZE); // division, no casting
vec2 tileID = floor(coord / TILE_SIZE);
vec2 n = numTilesAtLevel_[level];
vec2 r = mod(tileID, n);

// cast to float
pTile.z = float(tileArrayOffsets_[level]) + (r.y * n.x + r.x);
return pTile;
}

/**
* Returns true if all components of r are within the given range.
*/
bool isInsideRange(const in vec2 r, const in vec2 lower, const in vec2 upper)
{
    return all(greaterThanEqual(r, lower)) && all(lessThanEqual(r, upper));
}
```
bool isInsideRange(const in vec3 r, const in vec3 lower, const in vec3 upper)
{
    return all(greaterThanEqual(r, lower)) && all(lessThanEqual(r, upper));
}

/**
 * Compute the ideal clip level for texture sampling based on the distortion
 * of one pixel in screen coordinates if no DEM ray casting is used.
 */
float computeIdealLODPlanar(const in vec3 pGrid)
{
    vec2 ddx = dFdx(pGrid.xz);
    vec2 ddy = dFdy(pGrid.xz);
    float lddf = dot(ddx, ddx);
    float lddf = dot(ddy, ddy);

    float level;
    if (MAX_FILTERING_ANISOTROPY <= 1.0)
        level = log2(max(lddf, lddf)) * 0.5; // = log2(sqrt(max()))
    else
        level = log2(min(pMax / pMin, MAX_FILTERING_ANISOTROPY));

    return max(level, 0.0);
}

RayPosition planarFCM(const in vec3 p)
{
    RayPosition rp;
    rp.numSteps_ = 1u;
    rp.pGrid_ = (p * g_gridSize);
    rp.hit_ = ((p.y <= 0.0) && isInsideRange(rp.pGrid_, vec3(0.0), g_gridSize));
    if (rp.hit_)
        rp.levelLowest_ = getTileMapEntry(rp.pGrid_);
    float idealLOD = computeIdealLODPlanar(rp.pGrid_);
    rp.levelIdeal_ = uint(idealLOD);
    rp.levelBlend_ = fract(idealLOD);
    rp.levelCurrent_ =
        max(rp.levelLowest_, min(rp.levelIdeal_, MAX_CLIPLEVEL));
}

#ifdef FCM_USE_DEM_RAY_CASTING
/**
 * Determines where a ray given by a point r on the box (specified by boxMin
 * and boxMax) and a direction vector dir will intersect the box a second time.
 */
vec3 getBoxIntersection(const in vec3 r, const in vec3 dir,
    const in vec3 boxMin, const in vec3 boxMax)
{
    vec3 hit = r;
}
hit.x = (dir.x >= 0.0) ? boxMax.x : boxMin.x;
hit.y = (dir.y >= 0.0) ? boxMax.y : boxMin.y;
hit.z = (dir.z >= 0.0) ? boxMax.z : boxMin.z;

vec3 k = (hit - r) / dir;
if ((k.x <= k.y) && (k.x <= k.z))
hit.yz = r.yz + dir.yz * k.x;
else if ((k.y <= k.x) && (k.y <= k.z))
hit.xz = r.xz + dir.xz * k.y;
else
hit.xy = r.xy + dir.xy * k.z;
return hit;

/** Texel-precise sampling of DEM at the given clip level and grid position. */
float sampleDEM(const in uint level, const in vec2 pGrid)
{
    ivec3 tileCoord = getTileCoord(level, pGrid);
    // sample height value and scale according to data type & program settings
    float h = texelFetch(demLayerArray_, tileCoord, 0).r
            * SAMPLE_SCALE * heightScaling_;
    // transform from world coordinates into grid coordinates
    return (((h - worldOffset_.y) * g_gridSize.y) / worldSize_.y);
}

/** Sample the DEM layer at the given clip level and grid position using
 * bilinear interpolation. */
float sampleDEMFiltered(const in uint level, const in vec2 pGrid)
{
    vec3 pTile = getTileCoordUniform(level, pGrid);
    // sample height value and scale according to data type & program settings
    float h = texture(demLayerArray_, pTile, 0).r
            * SAMPLE_SCALE * heightScaling_;
    // transform from world coordinates into grid coordinates
    return (((h - worldOffset_.y) * g_gridSize.y) / worldSize_.y);
}

/** Compute surface normal by means of simple forward differences. */
vec3 getSurfaceNormal(const in vec3 pGrid, const in uint level)
{
    const vec3 du = vec3(1.0 / SAMPLES_PER_UNIT.x, 0.0, 0.0);
    const vec3 dv = vec3(0.0, 0.0, 1.0 / SAMPLES_PER_UNIT.z);
    float center = sampleDEMFiltered(level, pGrid.xyz);
    float right = sampleDEMFiltered(level, pGrid.xyz + vec2(1.0, 0.0));
    float upper = sampleDEMFiltered(level, pGrid.xyz + vec2(0.0, 1.0));
    return normalize(vec3((center - right), 1.0, (center - upper)));
}

/** Compute the ideal LOD of the DEM surface at the given grid position. */
float computeIdealLOD(const in vec3 pGrid)
{
    const vec3 du = vec3(1.0 / SAMPLES_PER_UNIT.x, 0.0, 0.0);
    const vec3 dv = vec3(0.0, 0.0, 1.0 / SAMPLES_PER_UNIT.z);
    // transform from grid into world coordinates
vec3 pWorld = (floor(pGrid) / SAMPLES_PER_UNIT) + worldOffset_;  
pWorld.y *= heightScaling_;  

// transform into screen space  
vec4 r = _modelViewProj * vec4(pWorld, 1.0);  
vec4 s = _modelViewProj * vec4(pWorld + du, 1.0);  
vec4 t = _modelViewProj * vec4(pWorld + dv, 1.0);  
vec4 u = _modelViewProj * vec4(pWorld.x, 0.0, pWorld.z, 1.0);

// transform to normalized device coordinates in the range [0, 1]  
r.xyz = (r.xyz / r.w) * 0.5 + vec3(0.5);  
s.xyz = (s.xyz / s.w) * 0.5 + vec3(0.5);  
t.xyz = (t.xyz / t.w) * 0.5 + vec3(0.5);  
u.xyz = (u.xyz / u.w) * 0.5 + vec3(0.5);

// three corners of grid cell's bounding box  
vec2 a = (s - r).xy * windowSize_;  
vec2 b = (t - r).xy * windowSize_;  
vec2 c = (u - r).xy * windowSize_;  

// the two sides of the parallelogram  
float l1 = a.x * a.x + b.x * b.x + c.x * c.x;  
float l2 = a.y * a.y + b.y * b.y + c.y * c.y;  

float minification = sqrt(min(l1, l2));  
float level = -log2(minification);  

return max(level, 0.0);  

}
/**
 * Returns the grid position on the boundary of the next cell at the current grid position stored in the given RayPosition struct in direction dir.
 * The w-component of the returned vec4 contains the DEM height in grid coordinates at the next cell.
 */

vec4 getNextCell(const in RayPosition rp, const in vec3 dir, const in uint dirCode)
{
  float k = float(1u << rp.levelCurrent_); // = 2^level

  vec4 pNext;
  vec2 samplePos; // position for sampling DEM layer at the next cell

  switch (dirCode)
  {
    case 2u:
    case 0u: // only bit 1 is set => (dir.x >= 0.0) && (dir.z >= 0.0)
      pNext.xyz = getNextCellRU(k, rp.pGrid_, samplePos);
      break;
    case 3u:
    case 1u: // bit 0 is set => dir.x < 0.0
      pNext.xyz = getNextCellLU(k, rp.pGrid_, samplePos);
      break;
    case 6u:
    case 4u: // bit 2 is set => dir.z < 0.0
      pNext.xyz = getNextCellRL(k, rp.pGrid_, samplePos);
      break;
    case 7u:
    case 5u: // bit 0 and bit 2 are set
      pNext.xyz = getNextCellLL(k, rp.pGrid_, samplePos);
      break;
  } // switch (dirCode

  float h1 = sampleDEM(rp.levelCurrent_, samplePos);

  // Use blending between DEM LODs only if no surface refinement is employed.
  #ifndef SURFACE_REFINEMENT
  if (rp.levelCurrent_ == max(rp.levelLowest_, min(rp.levelIdeal_, MAX_CLIPLEVEL)))
  {
    float h2 = sampleDEM(min((rp.levelCurrent_ + 1u), MAX_CLIPLEVEL), samplePos);
    pNext.w = mix(h1, h2, rp.lodBlend_);
  }
  #else
  #endif
  pNext.w = h1;

  // Compute the exact position on the boundary of the next grid cell.
  vec3 t = ((pNext.xyz - rp.pGrid_) / dir);

  if (abs(t.x) <= abs(t.z))
  {
    pNext.yz = rp.pGrid_.yz + (dir.yz * t.x);
  } else
  {
    pNext.xy = rp.pGrid_.xy + (dir.xy * t.z);
  }

  return pNext;
}
// ### functions for surface refinement ###
#ifdef SURFACE_REFINEMENT
#ifdef SURFACE_REFINEMENT_LINEAR
bool refineSurfaceLinear(inout RayPosition sample, const in vec3 dir, const in uint dirCode)
{
    float k = (1u << sample.levelCurrent_); // = 2^level
    vec2 dir_proj = normalize(dir.xz);
    float maxComp = max(abs(dir.x), abs(dir.z));
    if (abs(dir.y) > (2 * maxComp))
        return true;

    float delta_proj = (k * 0.5) / maxComp;
    float delta = delta_proj / maxComp;
    vec2 samplePos = sample.pGrid_.xz + dir_proj * delta_proj;
    vec3 pB = sample.pGrid_ + dir * delta;
    float hB = sampleDEMFiltered(sample.levelCurrent_, samplePos);
    if (hB < pB.y)
    {
        sample.pGrid_ = getNextCell(sample, dir, dirCode).xyz;
        return (! isInsideRange(sample.pGrid_, vec3(0.0), g_gridSize));
    }

    samplePos = sample.pGrid_.xz - dir_proj * delta_proj;
    vec3 pA = sample.pGrid_ - (dir * delta);
    float hA = sampleDEMFiltered(sample.levelCurrent_, samplePos);
    float a = abs(hA - pA.y);
    float b = abs(hB - pb.y);
    sample.pGrid_ = mix(pA, pB, a / (a + b));
    return true;
}
#elif (SURFACE_REFINEMENT == SURFACE_REFINEMENT_BICUBIC)
/**
 * +----+----+----+ 0: alpha
 * | NW | N | NE | 1: beta
 * +----2----3----+ 2: gamma
 * | W | .p | E | 3: delta
 * +----0----1----+
 * | SW | S | SE |
 * +----+----+----+
 * + Hermite bicubic geometry matrix (transposed!!!):
 * +
 * + [ alpha gamma ddv(alpha) ddv(gamma) ]
 * + [ beta delta ddv(beta) ddv(delta) ]
 * + [ ddu(alpha) ddu(gamma) ddu(ddv(alpha)) ddu(ddv(gamma)) ]
 * + [ ddu(beta) ddu(delta) ddu(ddv(beta)) ddu(ddv(delta)) ]
 * +
* mat4 getHermiteBicubicGeometryMatrix(const in vec2 r, const in uint level)
{
    float k = (1u << level); // = 2^level
    float samples[9]; // 0 = SW, 1 = S, 2 = SE, 3 = W, ..., 8 = NE
for (int i = 0; i < 9; ++i)
{
    vec2 off = vec2(mod(i, 3) - 1, (i / 3) - 1) * k;
    samples[i] = sampleDEM(level, r + off);
}

float s = min(samples[4], samples[1]);
float w = min(samples[4], samples[3]);
float n = min(samples[4], samples[7]);
float e = min(samples[4], samples[5]);

mat4 G_y; // first index: column, second index: row
G_y[0][0] = min(min(s, w), samples[0]); // alpha
G_y[1][0] = min(min(w, n), samples[6]); // gamma
G_y[1][1] = min(min(e, n), samples[8]); // delta
s = samples[4] + samples[1]; // C + S
w = samples[4] + samples[3]; // C + W

// ddu(alpha) = 0.5 * (C + S - (W + SW))
G_y[0][2] = 0.5 * (s - (samples[3] + samples[0]));
// ddv(alpha) = 0.5 * (C + w - (S + SW))
G_y[2][0] = 0.5 * (w - (samples[1] + samples[0]));
// ddu(ddv(alpha)) = C - W - S + SW

// ddu(beta) = (E + SE) - s
G_y[0][3] = 0.5 * ((samples[5] + samples[2]) - s);
// ddv(beta) = e - (SE + S)
G_y[2][1] = 0.5 * (e - (samples[2] + samples[1]));
// ddu(ddv(beta)) = E - C - SE + S

// ddu(gamma) = n - (NW + W)
G_y[1][2] = 0.5 * (n - (samples[6] + samples[3]));
// ddv(gamma) = (N + NW) - w
G_y[3][0] = 0.5 * ((samples[7] + samples[6]) - w);
// ddu(ddv(gamma)) = N - NW - C + W

// ddu(delta) = (NE + E) - n
G_y[1][3] = 0.5 * ((samples[8] + samples[5]) - n);
// ddv(delta) = (NE + E) - e
G_y[3][1] = 0.5 * ((samples[8] + samples[7]) - e);
// ddu(ddv(delta)) = NE - N - E + C

return G_y;
}
vec3 pNext;
vec2 samplePos;

vec2 off; // offset for mapping map (u, v) to [0.0, 1.0] x [0.0, 1.0]

switch (dirCode) {
    case 0u:
        // (dir.x >= 0.0) && (dir.z >= 0.0)
        pNext = getNextCellRU(k, rp.pGrid_, samplePos);
        off = floor(rp.pGrid_.xz / k);
        break;

    case 1u;
    case 3u: // dir.x < 0.0
        pNext = getNextCellLU(k, rp.pGrid_, samplePos);
        off.x = ceil(rp.pGrid_.x / k) - 1.0;
        off.y = floor(rp.pGrid_.z / k);
        break;

    case 4u;
    case 6u: // dir.z < 0.0
        pNext = getNextCellRL(k, rp.pGrid_, samplePos);
        off.x = floor(rp.pGrid_.x / k);
        off.y = ceil(rp.pGrid_.z / k) - 1.0;
        break;

    case 5u;
    case 7u: // (dir.x < 0.0) && (dir.z < 0.0)
        pNext = getNextCellLL(k, rp.pGrid_, samplePos);
        off = ceil(rp.pGrid_.xz / k) - 1.0;
        break;
}

// Compute the exact position on the boundary of next grid cell.
vec2 t = ((pNext.xz - rp.pGrid_.xz) / dir.xz);
if (abs(t.x) <= abs(t.y))
    pNext.yz = rp.pGrid_.yz + (dir.yz * t.x);
else
    pNext.xy = rp.pGrid_.xy + (dir.xy * t.y);

// Get the Hermite geometry matrix for DEM.
mat4 g_y = getHermiteBicubicGeometryMatrix(samplePos, rp.levelCurrent_);
float delta = length(pNext - rp.pGrid_) / NUM_STEPS;

bool hit = false;
for (uint n = 0u; n < NUM_STEPS; ++n) {
    // Map grid position at current level to [0.0, 1.0] x [0.0, 1.0].
    // Cannot use fract() here, because when looking along negative
    // directions, the border would contain the fraction of an integral
    // value which is 0.0, but it must be equal to 1.0.
    //
    vec2 uv = (rp.pGrid_.xz / k) - off;

    // Result of Hermite base matrix after multiplication with vectors
    // [u"3, u"2, u, 1] and [v"3, v"2, v, 1], respectively.
    //
    //a = vec4(2.0 * uuu - 3.0 * uu + 1.0, 3.0 * uu - 2.0 * uuu,
    //uuu - 2.0 * uu + uv.x, uuu - uu);
    //b = vec4(2.0 * vvv - 3.0 * vv + 1.0, 3.0 * vv - 2.0 * vvv,
    //vvv - 2.0 * vv + vv.y, vvv - vv);
vec2 i1 = uv * uv;
vec2 i2 = i1 + uv;
vec2 i3 = vec2(2.0) * i2 - vec2(3.0) * i1;
vec2 i4 = i2 - i1;
vec4 a = vec4(i3.x + 1.0, -i3.x, i4.x - i1.x + uv.x, i4.x);
vec4 b = g_y * vec4(i3.y + 1.0, -i3.y, i4.y - i1.y + uv.y, i4.y);

float h = dot(a, b);
if ((rp.pGrid_.y - h) <= EPSILON_HIT) {
    n = NUM_STEPS;
    hit = true;
    rp.pGrid_.y = h;
} else {
    rp.pGrid_ += dir * delta;
} // for (n
if (! hit) rp.pGrid_ = pNext;
if (! isInsideRange(rp.pGrid_, vec3(0.0), g_gridSize)) hit = true;
return hit;
}
#endif // SURFACE_REFINEMENT

/** Performs ray casting for rays directed downwards, i.e., dir.y < 0. */
void castRayDownward(inout RayPosition rp, const in vec3 dir, const in uint dirCode)
{
    uint n = 0u; // counter for ray casting steps at the current clip level
    while (((! rp.hit_) && (isInsideRange(rp.pGrid_, vec3(0.0), g_gridSize)))
    {
        ++rp.numSteps_;
        vec4 pNext = getNextCell(rp, dir, dirCode);
        // If an intersection is detected, compute the EXACT hit point
        // on the box of the grid cell.
        // The w-component contains the DEM value at the next grid cell.
        // The y-component is the current height of the ray above the ground
        // plane.
        if ((pNext.y - pNext.w) <= EPSILON_HIT) {
            rp.pGrid_ += dir * max(((pNext.w - rp.pGrid_.y) / dir.y), 0.0);
            // Determine the different clip levels.
            rp.levelLowest_ = getTileMapEntry(rp.pGrid_);
            float idealLOD = computeIdealLOD(rp.pGrid_);
            rp.levelIdeal_ = uint(idealLOD);
            rp.lodBlend_ = fract(idealLOD);

            // If the corresponding box of the grid cell at the current hit
            // point is from a clip level greater than the clip level which
            // would expose the least aliasing, descend one level.
            // Otherwise, ray casting can terminate, and surface can be
            // refined.
            uint bestLevel = max(rp.levelLowest_, min(rp.levelIdeal_, MAX_CLIPLEVEL));
            if (rp.levelCurrent_ > bestLevel)
A GPU Programs

```cpp
{
  --rp.levelCurrent_;
  n = 0u; // reset counter for ray casting steps at current level
}
#endif // SURFACE_REFINEMENT
}
else
  rp.hit_ = refineSurfaceLinear(rp, dir, dirCode);
#else
  rp.hit_ = refineSurfaceBicubic(rp, dir, dirCode);
#endif // SURFACE_REFINEMENT
else
  rp.hit_ = true;
#endif // SURFACE_REFINEMENT
}
else
  // the DEM was not hit by the ray => proceed at the next grid cell
  rp.pGrid_ = pNext.xyz;
  ++n; // increase the number of ray casting steps at this level

  // After four or more ray casting steps at the current clip
  // level, increase the level again (if possible) in order to
  // iterate over larger grid cells. Four is an empirical value.
  //
  // if ((n >= 4u) && (rp.levelCurrent_ < MAX_CLIPLEVEL))
  {
    ++rp.levelCurrent_;
    n = 0u; // reset step counter
  }
  // while ( ...

  // fix potential round-off errors which might create *holes* in the ground
  if (rp.pGrid_.y <= 0.0)
  {
    rp.hit_ = true;
    rp.pGrid_.y = 0.0;
  }
  //
}

/**
 * Performs ray casting for rays directed upwards, i.e., dir.y >= 0.
 * Except for the computation of the intersection with the DEM, this function
 * is essentially the same as castRayDownward().
 */
void castRayUpward(inout RayPosition rp, const in vec3 dir, const in uint dirCode)
{
  uint n = 0u;
  while ((! rp.hit_) && (isInsideRange(rp.pGrid_, vec3(0.0), g_gridSize)))
  {
    ++rp.numSteps_;
    vec4 pNext = getNextCell(rp, dir, dirCode);
    if ((rp.pGrid_.y - pNext.w) <= EPSILON_HIT)
    {
      rp.levelLowest_ = getTileMapEntry(rp.pGrid_);
      float idealLOD = computeIdealLOD(rp.pGrid_);
    }
  }
  // while ( ...
```
A.1 Flexible Clipmap Shaders

```
rp.levelIdeal_ = uint(idealLOD);
uint bestLevel =
    max(rp.levelLowest_,
        min(rp.levelIdeal_, MAX_CLIPLEVEL));

if (rp.levelCurrent_ > bestLevel)
{
    --rp.levelCurrent_;
    n = 0u;
}

#ifdef SURFACE_REFINEMENT
    #if (SURFACE_REFINEMENT == SURFACE_REFINEMENT_LINEAR)
        else
            rp.hit_ = refineSurfaceLinear(rp, dir, dirCode);
    #elif (SURFACE_REFINEMENT == SURFACE_REFINEMENT_BICUBIC)
        else
            rp.hit_ = refineSurfaceBicubic(rp, dir, dirCode);
    #endif
#else
    else
        rp.hit_ = true;
#endif // SURFACE_REFINEMENT

else
{
    rp.pGrid_ = pNext.xyz;
    ++n;
    if ((n >= 4u) && (rp.levelCurrent_ < MAX_CLIPLEVEL))
    {
        ++rp.levelCurrent_;
        n = 0u;
    }
}
```
rp.hit_ = false;

// Determine the target octant in space for the ray.
// Since the intersection tests with the DEM surface are different
// for rays directed upward and downward, use specialized functions.
uint dirCode = getDirectionCode(dir);

if ((dirCode & 2u) != 0u)
castRayDownward(rp, dir, dirCode);
else
castRayUpward(rp, dir, dirCode);

return rp;

/

/**
* Determines whether the given grid location is occluded by other elevations
* along the given direction. This function is used for casting shadows and
* for determining the visibility of fragments in projective textures.
*/
bool isOccluded(const in vec3 pGrid, const in vec3 dir)
{
    RayPosition sample;
    sample.pGrid_ = pGrid;
    sample.levelCurrent_ = 0u;  // only for getting to next grid cell
    sample.levelLowest_ = 0u;
    sample.levelIdeal_ = 0u;
    sample.numSteps_ = 0u;
    sample.hit_ = false;

    // Do not directly start at the give location, but advance one cell
    // before checking for occlusion. Otherwise, the initial location would
    // be considered as an intersection of the ray with the DEM.
    uint dirCode = getDirectionCode(dir);
    sample.pGrid_ = getNextCell(sample, dir, dirCode).xyz;
    sample.levelCurrent_ = MAX_CLIPLEVEL;

    if ((dirCode & 2u) == 0u)
castRayUpward(sample, dir, dirCode);
else
castRayDownward(sample, dir, dirCode);

    return sample.hit_;}

/// ### functions for computing color values ###

#ifdef MAX_NUM_PROJECTORS
bool projectiveTexturing(const in vec3 pWorld, const in vec3 pGrid, inout vec3 color)
{
    if (pWorld.y <= 0.1)
        return false;

    bool isValid = false;
    float minDistSq = 1.0e8;
    for (uint i = 0u; (i < N); ++i)
    {
        // Compute texture coordinate in projected images.
        vec4 tmp = g_mScaleProjector * mProjectors_[i] * vec4(pWorld, 1.0);
        tmp.xy /= tmp.w;
        tmp.y = 1.0 - tmp.y;

        // ...
vec2 projTexCoord = tmp.xy;

// Vector pointing from DEM intersection towards CoP of projector.
vec3 dirProj = (projectionCenters_[i] - pWorld);

// If the projective texture coordinate is valid...
if (isInsideRange(projTexCoord, vec2(0.0), vec2(1.0))) {
    // ...compute distance from CoP of projector to DEM intersection...
    float distSq = dot(dirProj, dirProj);
    // ...and check if the distance is smaller than the current minimum.
    // Only perform the occlusion test, if the distance is smaller,
    // since the test is computationally expensive.
    //
    if (distSq <= minDistSq && (! isOccluded(pGrid, normalize(dirProj)))) {
        // If the tests are passed, update fragment color and current
        // minimal distance.
        minDistSq = distSq;
        color = texture(projectiveTextures_[i], projTexCoord).bgr;
        isValid = true;
        //color = getLevelColor(i + 1u);
    }
} // if (insideRange
} // for (i

return isValid;

vec4 determineFragmentValues(const in RayPosition rp, out float zBuffer)
{
    vec4 color = vec4(0.0, 0.0, 0.0, 1.0);
    // transform grid position to world space
    vec3 pWorld = (floor(rp.pGridĢ / SAMPLES_PER_UNIT) + worldOffsetĢ;

    // compute surface normal + dot-product with directional light source
    vec3 n = getSurfaceNormal(rp.pGrid_, rp.levelCurrent_);
    float nDotL = clamp(dot(n, -dirLight_), 0.0, 1.0);

    // use diffuse shading by default
    color.rgb = clamp(nDotL * surfaceColorDiffuse_, 0.0, 1.0);

    // use normal map with lighting
    if (defined(COLORING_MODE) && (COLORING_MODE == COLOR_MODE_SHADED_WITH_SHADOWS))
        color.rgb = surfaceColorAmbient_;
else
vec3 pTile = getTileCoordUniform(rp.levelCurrent_, rp.pGrid_.xz);
color.rgb = texture(colorLayerArray_, pTile).bgr;
#endif // FCM_USE_DEM_RAY_CASTING

// Compute the correct depth value at the fragment for correct "interaction"
// with polygonal data, i.e., occlusion, depth-values and depth-tests.
// transform world position to normalized device coordinate
vec4 pNDC = _modelViewProj * vec4(pWorld, 1.0);
pNDC.xyz = (pNDC.xyz / pNDC.w) * 0.5 + vec3(0.5);
zBuffer = pNDC.z;

return color;

// ### main function ###
void main()
{
vec3 exit = _texCoord0;
RayPosition rp;

#ifdef FCM_USE_DEM_RAY_CASTING
vec3 entry = _camPosTextureCoordinates;
if (!isInsideRange(entry, vec3(0.0), vec3(1.0)))
{
    vec3 tmpDir = normalize(entry - exit);
    entry = getBoxIntersection(exit, tmpDir, vec3(0.0), vec3(1.0));
}
vec3 dir = normalize((exit - entry) * _boxRatio);
rp = rayCasting(entry, dir);
#else
rp = planarFCM(exit);
#endif // FCM_USE_DEM_RAY_CASTING

if (! rp.hit_)
discard;

fragColor = determineFragmentValues(rp, gl_FragDepth);
} // main()

Listing A.2: The complete GLSL fragment program of our FCM implementation.
A.2 DSM Synthesis Shaders

The following GLSL vertex and fragment programs are employed by our implementation for creating DSM layers as described in chapter 5.

```glsl
//#version 150
//
#define NUM_PROJECTORS 8u

#line 5

// Required to unroll loops when using NVidia graphics adapters and drivers
// up to version 306.92 (or greater?).
#pragma optionNV(unroll all)

const mat4 g_mScaleProjector = mat4{
    0.5, 0.0, 0.0, 0.0, // 1st row
    0.0, 0.5, 0.0, 0.0,
    0.0, 0.0, 0.5, 0.0,
    0.5, 0.5, 0.5, 1.0};

uniform mat4 modelMatrix_;  
uniform mat4 viewMatrix_;  
uniform mat4 projectionMatrix_;  
uniform mat4 mProjectors_[NUM_PROJECTORS];
uniform int numActiveProjectors_;  

in vec3 _vertexPos;  
in vec3 _vertexNormal;  
in vec2 _vertexTexCoord0;  
in vec4 _vertexColor;  

out vec4 _texCoords[NUM_PROJECTORS];  
out vec3 _worldPos;

void main()  
{
    mat4 modelViewMatrix = viewMatrix_ * modelMatrix_;  
    mat4 modelViewProjMatrix = projectionMatrix_ * modelViewMatrix;
    uint N = min(uint(numActiveProjectors_), NUM_PROJECTORS);  
    for (uint i = 0u; i < NUM_PROJECTORS; ++i)
    {
        if (i < N)
            _texCoords[i] = g_mScaleProjector * mProjectors_[i] * vec4(_vertexPos, 1.0);
        else
            _texCoords[i] = vec4(0.0, 0.0, 0.0, 1.0);
    }
    gl_Position = modelViewProjMatrix * vec4(_vertexPos, 1.0);  
    _worldPos = _vertexPos;
}
```

Listing A.3: The complete GLSL vertex program for DSM synthesis by means of a space sweep.

147
Listing A.4: The complete GLSL fragment program for DSM synthesis by means of a space sweep.

```glsl
// The following defines are generated by the class <code>LayerCreator</code>
// according to client program settings.
/*#version 150
#define NUM_PROJECTORS 8
#define CREATION_MODE CREATE_DEM_AND_COLOR
//#define USE_BETTER_HALF*/
#line 8

// Required to unroll loops when using NVidia graphics adapters and drivers
// up to version 306.92 (or greater?).
#pragma optionNV(unroll all)

#define CREATE_DEM_AND_COLOR 0
#define CREATE_COLOR_PLANAR 1

// ### in/out variables ###
in vec4 _texCoords[NUM_PROJECTORS]; // coordinates in projective textures
in vec3 _worldPos; // fragment’s position in world space
out vec4 _demValue;
out vec4 _fragColor;

// ### uniform variables ###
uniform int numActiveProjectors_;
uniform int minSupportWindowLevel_;
uniform int maxSupportWindowLevel_;
uniform float costsThreshold_;
uniform float minDEMHeight_;
sampler2D projectedTextures_[NUM_PROJECTORS];

// ### global variables and constants ###
uint N = min(uint(numActiveProjectors_), NUM_PROJECTORS);
vec2 g_texCoords[NUM_PROJECTORS];
const vec3 g_LuminanceFactors = vec3(0.212671, 0.715160, 0.072169);
const float INFINITY = 1.0e8;

// ### structures ###
struct ImageInfo {
  vec4 colorValue_; // color value from corresponding image
  float costs_; // matching costs for image
  float luminance_; // luminance value (derived from RGB texel value)
  uint imageIndex_; // index of the image
  bool covers_; // true if the image covers this fragment
}; // struct ImageInfo

// ### functions ###
```
/** Returns true if all components of \( v \) are within the given range. */

```cpp
bool isInsideRange(const in vec2 v, const in vec2 lower, const in vec2 upper)
{
    return (all(greaterThanEqual(v, lower)) && all(lessThanEqual(v, upper)));
}
```

/** Selection sort for sorting the first \( R \) elements of the given array. */

```cpp
void selectionSort(inout ImageInfo arr[NUM_PROJECTORS], const in uint R)
{
    for (uint i = 0u; i < R; ++i) // (R - 1u) would be sufficient
    {
        uint minIndex = i;
        for (uint j = i + 1u; j < R; ++j)
        {
            if (arr[j].costs_ < arr[minIndex].costs_)
                minIndex = j;
        } // for (j
        // swap elements
        ImageInfo tmp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = tmp;
    } // for (i
}
```

/** Returns the luminance of the projective texture at the given index which projects onto this fragment. For efficiency reasons, no check for is performed whether the projector’s texture coordinate is valid. */

```cpp
float getLuminanceUnchecked(const in uint index, const float mipmapLevel)
{
    vec3 clr = textureLod(projectedTextures_[index], g_texCoords[index], mipmapLevel).bgr;
    return dot(clr, g_LuminanceFactors);
}
```

/** Initialize the ImageInfo structure for this fragment at the given index with all information available. */

```cpp
uint initImageInfo(const in uint index, const float mipmapLevel, out ImageInfo info[NUM_PROJECTORS])
{
    // projective texture covers this fragment, if index is valid and texture coordinates are within the range [0, 1] x [0, 1]
    info[index].covers_ = ((index < N) && insideRange(g_texCoords[index], vec2(0.0), vec2(1.0)));
    if (info[index].covers_)
    {
        info[index].imageIndex_ = index;
        info[index].colorValue_ = textureLod(projectedTextures_[index], g_texCoords[index], mipmapLevel).bgra;
        info[index].luminance_ = // convert RGB value to luminance
            dot(info[index].colorValue_.rgb, g_LuminanceFactors);
        info[index].costs_ = 0.0;
        return 1u; // for computing the total coverage at the fragment
    }

    info[index].colorValue_ = vec4(0.0, 1.0, 0.0, 1.0);
    info[index].luminance_ = 0.0;
}
info[index].costs_ = INFINITY;
return 0u;
}

float computeMatchingCosts(out vec4 color)
{
  float summedCosts = 0.0;
  color = vec4(0.0, 0.0, 0.0, 1.0);
  ImageInfo info[NUM_PROJECTORS];

  // Determine the coverage by projective images at this fragment and
  // initialize the ImageInfo structure at the lowest mipmap level for
  // matching cost aggregation.
  float mipmapLevel = max(float(minSupportWindowLevel_), 0.0);
  uint coverage = 0u;
  for (uint i = 0u; i < NUM_PROJECTORS; ++i)
  {
    coverage += initImageInfo(i, mipmapLevel, info);
    if (info[i].covers_)
      color = info[i].colorValue_;  
  }

  // Change to (coverage == N) to get visually most appealing results.
  if (coverage > 1u)
  {
    // add costs from previous pass with smallest support window size
    for (uint i = 0u; i < NUM_PROJECTORS; ++i)
    {
      if (info[i].covers_)
      {
        for (uint j = 0u; j < NUM_PROJECTORS; ++j)
        {
          if (info[j].covers_)
            info[i].costs_ += abs(info[i].luminance_ - info[j].luminance_);
        }
      }
    }

    // add costs from larger support windows
    for (int l = (minSupportWindowLevel_ + 1); l <= maxSupportWindowLevel_; ++l)
    {
      // update luminances
      float mipmapLevel = float(l);
      for (uint i = 0u; i < NUM_PROJECTORS; ++i)
      {
        if (info[i].covers_)
          info[i].luminance_ = getLuminanceUnchecked(i, mipmapLevel);
      }
    }
  }

  // same as above
  for (uint i = 0u; i < NUM_PROJECTORS; ++i)
  {
    if (info[i].covers_)
    {
      for (uint j = 0u; j < NUM_PROJECTORS; ++j)
      {
        if (info[j].covers_)
          info[i].costs_ += abs(info[i].luminance_ - info[j].luminance_);
      }
    }
  }  // for (j
```cpp
A.2 DSM Synthesis Shaders

#define USE_BETTER_HALF

// sort matching costs in increasing order
selectionSort(info, N);

// use color from image with least matching costs
color = info[0].colorValue_

// sum up costs

uint k = 0u;
for (uint i = 0u; (i < N); ++i)
{
    // sum up only valid matching costs (there is at least one valid value)
    if (info[i].costs_ < INFINITY)
    {
        summedCosts += info[i].costs_;  
        color += info[i].colorValue_;  
    }
}

// normalize costs to simplify comparison with threshold
summedCosts /= float(k);

else
    color = vec4(0.0);
for (uint i = 0u; (i < N); ++i)
{
    // sum up only valid matching costs (there is at least one valid value)
    if (info[i].costs_ < INFINITY)
    {
        summedCosts += info[i].costs_;  
        color += info[i].colorValue_;  
    }
}

// normalize costs to simplify comparison with threshold
summedCosts /= float(coverage);
color /= float(coverage); // use averaged color values
color.a = 1.0; // fix potential round-off errors

// if (coverage
if (coverage
    summedCosts = INFINITY;
return summedCosts;
}

void createDSM(out float fragDepth, out vec4 dsmValue, out vec4 color)
{
    // Compute the SAD to rate the quality of matching. The smaller the SAD,
    // the better the matching. The SAD is output as depth value of the
    // fragment in order to make subsequent z-Buffer tests consider only
    // fragments that have better quality, i.e., lower SADs.
    float matchingCosts = computeMatchingCosts(color);
    if (matchingCosts >= costsThreshold_)
    {
        // must be < 1.0; otherwise the fragment would be discarded
        fragDepth = 0.999999;
        dsmValue.rgba = vec4(vec3(minDEMHeight_), 1.0);
        color = vec4(0.0, 0.0, 0.0, 1.0);
        return;
    }
    fragDepth = matchingCosts;
    dsmValue.rgba = vec4(vec3(_worldPos.y), 1.0);
```
vec4 createPlanarColorLayerLeastDistance()
{
    float minDist = INFINITY;
    vec4 color = vec4(0.0, 0.0, 0.0, 1.0);
    for (uint i = 0u; (i < N); ++i)
    {
        if (isInsideRange(g_texCoords[i], vec2(0.0), vec2(1.0)))
            vec2 delta = (g_texCoords[i] - vec2(0.5));
        float dSq = dot(delta, delta);
        if (dSq < minDist)
            minDist = dSq;
        color = texture(projectedTextures_[i], g_texCoords[i]).bgra;
    } // for (i
    return color;
}

/**
* Transform all coordinates in projective textures from homogeneous
* coordinates to Cartesian coordinates.
*/
void transformTextureCoordinates()
{
    for (uint i = 0u; i < NUM_PROJECTORS; ++i)
    {
        vec4 tmp = (_texCoords[i] / _texCoords[i].w);
        tmp.y = 1.0 - tmp.y; // flip horizontally due to Irrlicht
        g_texCoords[i] = tmp.xy;
    } // for (i
}

Listing A.4: The complete GLSL fragment program for DSM synthesis by means of a space sweep.
B Additional UML Diagrams

The class diagram in figure B.1 contains details about the interface of class Flexible-Clipmap. Methods which contain the word upload call virtual methods which are implemented by derived subclasses, like IrrFlexibleClipmap, in order to initiate the transfer of texture data from main memory to video memory. The two structures ClipLevelInfo and TileArrayInfo are both declared as private in class FlexibleClipmap and store information about a clip level such as the number of tiles, sizes of active area and clip area, etc., and the state of presence of a tile in the tile array, respectively. The method update() is responsible for initiating updates of the active areas and clip areas as the virtual camera is moved and must therefore be called before a frame is rendered. This method takes the current position of the virtual camera and the clip center as parameters.

Figure B.2 illustrates the communication between the affected objects in our FCM implementation in response to the insertion of an aerial image at the AerialImageIndex object. The depicted communication is the same as shown in the communication diagram in figure 7.7 in chapter 7.

Figure B.3 shows the relationships of the classes in the image-package of our FCM implementation. The components of this package provide basic raster image processing functionality such as filtering, re-sampling, loading and storing different file formats and converting between different pixel formats.
Figure B.1: Details about the class FlexibleClipmap from our FCM implementation.
Figure B.2: A sequence diagram showing the communication of affected objects in response to the insertion of an aerial image at the AerialImageCollector object.
Figure B.3: Relationships between the classes in the "image-package" for image processing.


157


<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AABB</td>
<td>axis-aligned bounding box</td>
</tr>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>BVH</td>
<td>bounding volume hierarchy</td>
</tr>
<tr>
<td>CCD</td>
<td>charge-coupled device</td>
</tr>
<tr>
<td>CoP(s)</td>
<td>center(s) or projection</td>
</tr>
<tr>
<td>DEM</td>
<td>digital elevation model</td>
</tr>
<tr>
<td>DSM</td>
<td>digital surface model</td>
</tr>
<tr>
<td>FCM</td>
<td>Flexible Clipmap</td>
</tr>
<tr>
<td>FBO</td>
<td>frame buffer object</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>GPU</td>
<td>graphics processing unit</td>
</tr>
<tr>
<td>GLSL</td>
<td>OpenGL Shading Language</td>
</tr>
<tr>
<td>IMU</td>
<td>inertial measurement unit</td>
</tr>
<tr>
<td>LOD</td>
<td>level of detail</td>
</tr>
<tr>
<td>(m)UAV</td>
<td>(miniature) unmanned aerial vehicle</td>
</tr>
<tr>
<td>SAD</td>
<td>sum of absolute differences</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UTM</td>
<td>Universal Transverse Mercator (coordinate system)</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>