MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Cartography in Virtual Environment

MASTER THESIS

Bc. Rastislav Tisovčík

Brno, 2014

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Rastislav Tisovčík

Advisor: Mgr. Jiří Chmelík, Ph.D.

Acknowledgement

First of all, I would like to thank my advisor Jiří Chmelík for his guidance and all the priceless consultations, too numerous to count. I'd also like to thank Lukáš Herman from the Faculty of Science for his valuable advice and help during the development phase.

Last but not least I would like to thank my family and friends for their continual support during my work on this thesis. I wouldn't be able to complete this thesis without you, hence thank you once again.

Abstract

The aim of this thesis is to discuss and apply knowledge of computer graphics and human-computer interaction in the field of cartography. This text covers several topics including basic principles of cartography, representation of terrain data and possible ways how to obtain the data. Furthermore, selected interaction techniques and devices which can be used to employ them are also discussed.

The implementation part of the thesis introduces *Land* ability, a new extension of the *vrecko* framework. The extension allows users to visualize and work with terrain data obtained from various sources. This includes support for real world terrain data of the Czech Republic provided by ČÚZK along with utilisation of orthoimagery that is available via web services.

Keywords

vrecko, virtual environment, cartography, interaction, terrain, land, height field, triangulated irregular network, TIN, digital elevation model, DEM, S-JTSK

Contents

1 Introduction

Since the origin of the field of computer graphics, its continual advances have led to an increased demand for scientific visualization. This demand is not related to any particular field discipline – medicine, chemistry, astrophysics, archeology and many other fields use computer graphics for the purpose of visualization on a regular basis.

One of the fields that can benefit from visualization of data in three-dimensional virtual environment is cartography. After millenia of using physical maps (both planar and globular), cartography has obtained a new set of tools which can be used for visualization of maps and other geographical data. In this text, we focus on visualization of subset of these data, namely topography of the land – i.e. the terrain model.

However – apart from visualization, we have to also focus on the topic of human-computer interaction. Our interaction with physical maps and globes in the real world is intuitive and mostly subconscious. In contrast, interaction with virtual maps – especially in three dimensions – requires us to study techniques and devices which we use to interact with the virtual models.

Next chapter of this text covers several important topics related to the field of cartography. Content of this chapter is not designed to cover all relevant topics but rather to clarify and explain those that are important in the context of the whole thesis.

In the third chapter, we discuss visualization of terrain model in virtual environment. This includes internal representation of the terrain model and ways how to obtain terrain data that correspond to real world locations.

The fourth chapter is focused on user's interaction with terrain model in virtual environment. Several significant interaction techniques are discussed, along with selected hardware devices that can be used to implement the interaction techniques.

Chapter five covers design of the *Land* ability, an extension that was created as a practical output of the thesis. This includes design goals, description of possible data sources and data flow of the extension.

The sixth and the longest chapter contains details regarding implementation of the extension. In this part, significant ideas related to the implementation of height field and TIN are covered. Chapter seven concludes the main text and proposes ideas for future work.

Appendix A is a guide that can be used to pre-process point cloud files provided by ČÚZK. Appendix B lists all input parameters that are sup-

ported by the extension while appendix C contains several simplified class diagrams that accompany extension's design discussed in chapter five. The thesis' appendices conclude with appendix D which contains results of performance tests and appendix E that lists and describes all files that are located in the thesis archive in the Masaryk University Information System.

2 Basic principles of cartography

2.1 GIS

Geographic information system (GIS) is *"(...) a configuration of computer hardware and software specifically designed for the acquisition, maintenance, and use of cartographic data."*[23].

In other words, GIS is an information system specialized for use in the fields of cartography and geoinformatics. It allows its users to access, work with and often visualize geo-referenced data of various types.

2.2 Coordinate systems

Coordinate systems form the core of every single GIS or an application that works with cartographical data. As defined by ESRI, *"A coordinate system is a reference system used to represent the locations of geographic features, imagery, and observations, such as (...) within a common geographic framework."*[6].

Each coordinate system is defined by a set of properties. These include mainly measurement framework, units of measurement and a set of optional properties such as projection, central meridian or shift in a particular direction. The usage of optional properties varies and depends entirely on the coordinate system used.

Measurement framework of a coordinate system is either geographic or planimetric. Geographic coordinate system (GCS) uses spherical coordinates measured from the Earth's centre – such as latitude and longitude. This way, any location on the Earth's surface can be addressed using spherical coordinates.

Unlike the GCS, the planimetric coordinate system (PCS) works in two dimensional plane. It is always based on a GCS and includes a map projection among its properties. This projection is used to transform a three dimensional surface (e.g. some part of the Earth) to create its representation in a two dimensional Cartesian plane [6].

Most of the time, a vertical coordinate system is also used along with GCS or PCS. Vertical coordinate system defines origin for elevation values which specify height and/or depth at a particular location. Vertical coordinate systems are usually based on the elevation of a sea level – though the reference sea and its phase (e.g. low, high or mean tide) varies across the systems which are in use.

For the purpose of identification, each coordinate system is labeled by

its own unique code. This code is called SRID (Spatial Reference System Identifier) and is assigned to the coordinate system by a GIS vendor, such as ESRI[1]. To achieve higher compatibility among vendors, many SRID have already been defined and standardized by the EPSG[2] authority[5].

Next section describes several coordinate systems that are significant for the content of this thesis. Comprehensible list of many existing coordinate systems can be found e.g. at [4], along with their definition.

2.2.1 S-JTSK

S-JTSK is a planimetric coordinate system designed by Josef Křovák in 1922. It was intended for use in former Czechoslovakia and is still officially used in its successor states Czech Republic and Slovakia.

The coordinate system is based on a reference ellipsoid derived by Friedrich Wilhelm Bessel in 1841 and uses Křovák projection to transform 3D coordinates to map plane [24].

Throughout the history, several versions of the system were created. The original system uses Ferro[3] as a prime meridian while some modified versions of the system use the Greenwich meridian instead.

X / Y direction	**Prime meridian**	
	Ferro	**Greenwich**
South / West	EPSG:2065	EPSG:5513
East / North	EPSG:5221 ESRI:102066	EPSG:5514 ESRI:102067

Table 2.1: Different versions of S-JTSK coordinate system.

Similarly, there are modifications which differ in usage of a slightly modified projection. *Křovák* projection in the original system maps the positive directions of X and Y axes to south and west, respectively – as shown in figure 2.1. However, working with this projection might not be intuitive, due to the difference from the Cartesian coordinate system. To avoid the confusion, a different version of S-JTSK with modified projection was created. The modified projection is labeled as *Křovák East North* and maps the positive directions of X and Y axes to east and north, respectively.

1. http://www.esri.com/
2. *European Petroleum Survey Group*, nowadays part of *International Association of Oil & Gas Producers* (OGP)
3. Located $17°39'46.02''$ west of the Greenwich meridian.

Several version of S-JTSK coordinate system are shown in table 2.1 along with their identifiers. All files provided by ČÚZK (see section 3.1.1) that are used in the implementation part of the thesis use the *ESRI:102067* version.

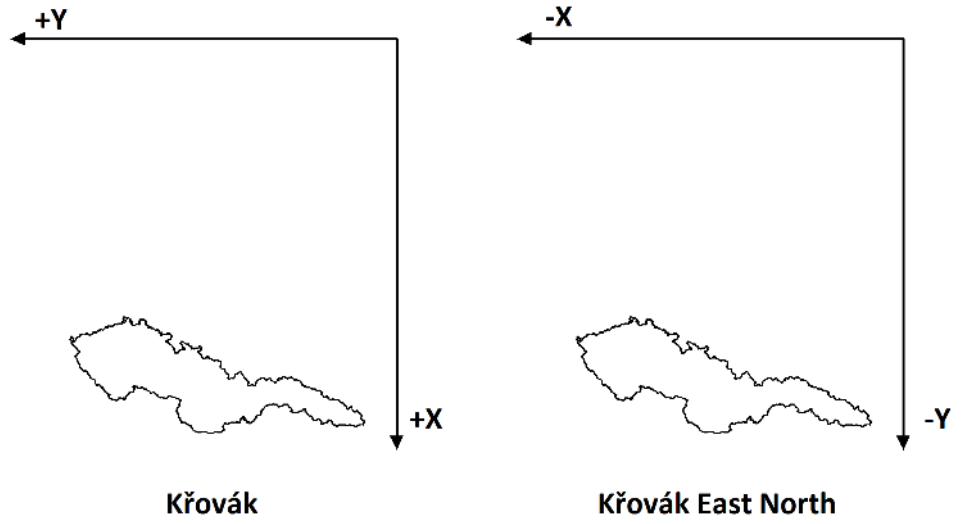


Figure 2.1: Mapping of X and Y axes in the original Křovák (left) and modified Křovák East North (right) projections.

2.2.2 Baltic vertical datum - after adjustment

Baltic Vertical Datum – After Adjustment (translated from Czech: *Výškový systém baltský – po vyrovnání* [29]) is a vertical coordinate reference system used in several European and Asian countries, including Czech Republic and Slovakia. It uses average water level at Kronstadt[4] (gauged in 1833, adjusted in 1977) as its origin. SI meter is used as a unit of measurement of height above the sea level.The system is associated with the code EPSG:5705[5].

As is the case with *S-JTSK Křovák East North*, all files provided by ČÚZK (see section 3.1.1) that are used in the implementation part of the thesis use this system for measurement of elevation.

4. Located at $60°0'0''N, 29°46'0''E$ in WGS 84 coordinate system.
5. http://epsg.io/5705

2.3 Map layers

A physical map offers only a static form of visualization – once a map is created, its content cannot be easily changed. Moreover, the amount and type of data that is displayed on a map has to be limited for the sake of clarity. For example, we cannot display elevation of terrain surface, level of noise pollution and average air temperature on the same map in a comprehensible way. Therefore, a real world solution might be to create several maps of the same area, each of them displaying a different kind of information.

On the other hand, a virtual map allows for dynamic visualization and behaviour. Apart from the ability to change the displayed area and scale, we can also utilize a concept of layers. Using some kind of interaction, a user might be able to dynamically select a subset of layers which are visualized (see figure 2.2 for example). Each layer may contain data of a different type, such as:

- *Terrain* – Visualization of terrain's topography and its surface.
- *Contour lines* – A set of curves used to display topography of the land on a two dimensional map.
- *Cadaster* – Highlighting of borders among plots of land.
- *3D models* – Models of man-made structures which can greatly increase the visualization's overall level of detail.
- *Labels* – Textual (or pictorial) markings which are used to provide user with informatiion related to a particular location on the map.
- *Networks* – Representation of various network which comprise the country's infrastructure, e.g. road and railroad systems, sewer networks or power line networks.

This thesis only deals with visualization of terrain and the content of other possible layers is out of its scope. Information regarding various cartographical data and possible ways of their visualization is thoroughly described and can be found in [10, p. 13 - 37].

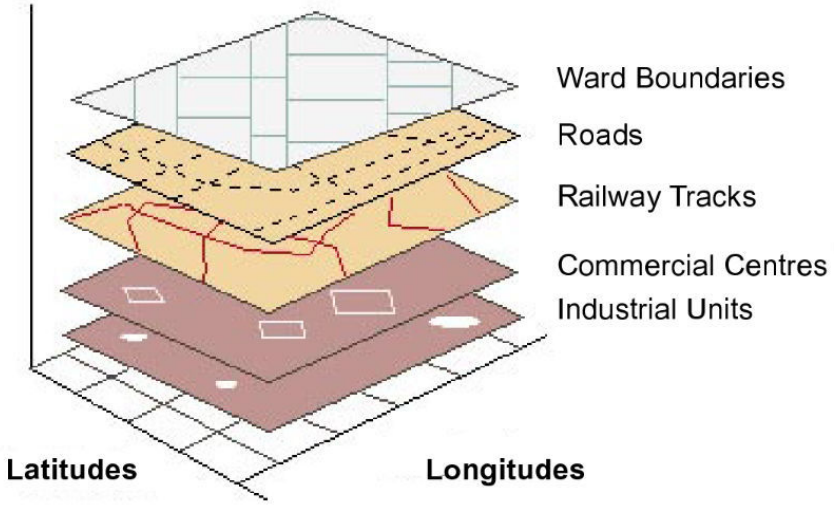


Figure 2.2: Example of possible map layers in a GIS. Source: [8, p. 9]

3 Visualization

3.1 Digital elevation model

Digital elevation model (DEM) is a digital representation of a planet's surface – in our case the Earth's. The model is most often stored as a set of samples, where each sample contains an elevation at a given location in the world.

There are two common types of DEM which differ only in the meaning of the elevation value. Unfortunately, they are frequently interchanged and there seems to be no strict definition. For the purpose of this text we will use the following terminology:

- *Digital surface model* (DSM) is a DEM which contains elevation of the terrain and all objects that were present at the time of the measurement. Depending on the quality of acquisition method, DSM can offer a highly detailed model of the landscape, including buildings, vegetation and other entities.
- *Digital terrain model* (DTM) contains elevation of the land at its ground level, excluding all other entities such as man-made objects and vegetation. For obvious reasons, DTM cannot be measured directly and is computed from a detailed DSM using a specialized algorithm (see [14] for more information).

  Visual comparison of DSM and DTM is shown in figure 3.1.

Techniques which are used for acquisition of the elevation samples are not covered in this text. Their description along with methods which deal with processing of the measured data and computation of the DEM can be found in [14, chapter 3].

3.1.1 Providers

For most users, production of DEM is not a viable option and it is necessary to rely on external – mostly national – providers. In Czech Republic, the main provider is *Czech Office for Surveying, Mapping and Cadastre (ČÚZK)*[1] which offers a selection of geographical products including both DSM [25] and DMP [26, 27, 28].

On a global scale, USGS[2] provides DTM of the whole Earth – though naturally in a lower resolution [21].

1. http://www.cuzk.cz/
2. http://www.usgs.gov/

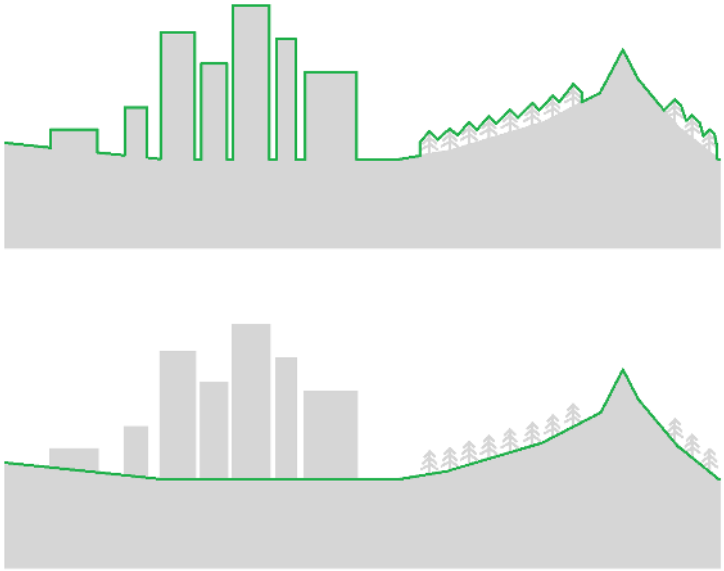


Figure 3.1: Comparison of DSM (top) and DTM (bottom).

3.1.2 Web services

When creating a desktop application, a common approach is to store its data on a local secondary storage. However, this is not an ideal solution in case of geographical data.

One of the main reasons is that the application or its user may need to work with geographical data on various locations. Moreover, the desired level of detail may change as well which results in a need to store several qualitatively different versions of the data. Therefore, this approach would require the application to store vast amounts of data, which is not suitable – especially if the application is targeted on an ordinary PC (as is the case of e.g. Google Earth[3]).

To avoid this issue, many providers offer some of their geographical data via web services. These services include e.g. WCS (*Web Coverage Service*), WMS (*Web Map Service*) and WFS (*Web Feature Service*), all of them standardized by Open Geospatial Consortium. Each service is used for different type of content – for example, WMS provides planar data in form of image files and is often used as a source of orthoimagery. Usage of web

3. http://www.google.com/earth/

services is essential for web applications which rely on geographical data.

Additional advantage of web services lies in its centred maintenance. Any data update is made only on the server side and the change is propagated to all clients once they request the data (which depends on a particular implementation of the client's cache).

3.2 Representation

Internal representation of DEM has a large influence on its properties and the way we process its data.

3.2.1 Height field

A common way to store samples of a DEM is to utilize a height field. In height field, all samples form a regular grid with uniform distance between any two adjacent samples. This regularity results in several nice properties which make height field a good choice for many applications.

Position of each sample in the grid can be inferred from its order in the array in which the samples are stored. For this operation, we need to know the dimensions of the regular grid, distance between the adjacent samples and layout of samples in the grid. Therefore, there's no need to store X and Z coordinates of the samples which results in lower memory usage.

Another advantage of height field is easier implementation of some algorithms, such as computation of normal vectors. Neighbours of any sample can be directly addressed if we modify the sample's index accordingly.

Main drawback of the height field approach lies in the sampling frequency (i.e. distance between two adjacent samples). Low frequency results in large distance between two samples which in turn causes loss of detail – this is especially troublesome in case of urban or mountainous enviroments. This issue may be alleviated by sampling at higher frequency. However, doing so might produce an unnecessary amount of samples in case of fields and lowlands – and effectively waste the memory originally saved by usage of height field.

Moreover, there's no way to model caves or overhangs of any kind – each point of the grid can store only one elevation sample.

3.2.2 TIN

Another common approach is to store DEM as a *triangulated irregular network*, i.e. TIN. Elevation samples of TIN are distributed in an irregular way

and are connected so that they form a mesh consisting of triangular faces, as seen in figure 3.2. No restriction on the location and distribution of the samples allows for adaptive density of samples according to the shape of the DEM. We can therefore use a large amount of samples to store details of the terrain in coarse areas while using only a few samples to cover large plains.

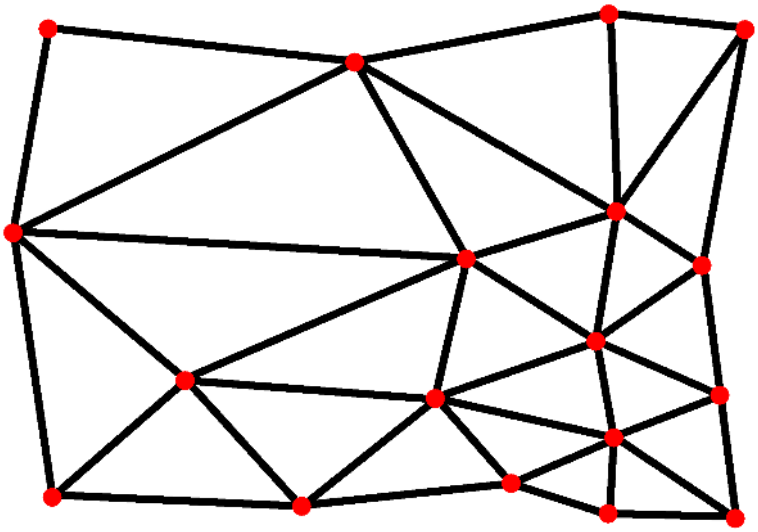


Figure 3.2: Example of triangulated irregular network (TIN).

However, usage of TIN has several drawbacks as well. Most importantly, we have to use data structures which are designed for the irregular nature of TIN such as different number samples and faces that are adjacent to a sample. This also applies to algorithms, complexity of which may grow.

Another drawback is that we have to store all 3 coordinates for each sample which may result (depending on a properties of the particular DEM) in increased memory usage when compared to height field.

Last but not least, TIN can be used to store terrain with caves or overhangs – though this solution is rarely used because of complicated implementation.

3.3 Optimization

Selection of an appropriate internal representation is important but it does not automatically guarantee us an efficient visualization. Mesh that is created from high-quality DEM may easily contain millions of vertices and

faces. To render such mesh effectively (i.e. at a sufficient and steady frame rate), it is advisable to implement one of many existing techniques designed to optimize the terrain mesh.

3.3.1 View-independent

The simplest optimization technique is to reduce number of vertices and faces of the mesh during the pre-processing phase. This approach is especially effective when applied to height fields. For this task we may use one of the vertex decimation algorithms, e.g. [18].

3.3.2 View-dependent

More complicated algorithms allow for dynamic optimization at run-time. The optimization function is called repeatedly and computes a new version of the rendered mesh according to the current position of the camera. This allows us to render close terrain in full detail and remove a large amount of vertices and faces which are located in the background at the moment. The optimization function is usually called once the camera moves to a different position which is not suitable anymore.

Dynamic optimization of terrain is covered by a wide range algorithms such as SOAR [15], ROAM [7] or Lindstrom '96 [16]. Dynamic optimization which performs most computations on GPU was also proposed [1].

4 Interaction

This chapter describes possible ways how user can interact with a terrain model in a virtual environment.

The first section lists several 3D interaction techniques which might be relevant in scenarios which include terrain model – even though the terrain itself is assumed to be a static part of the scene. The second part contains description and comparison of selected hardware devices which can be used to implement the aforementioned interaction techniques.

4.1 Interaction techniques

4.1.1 Navigation

Movement around the scene is a task which is essential for most applications which contain some kind of virtual environment. The task of navigation consists of two distinct techniques – travel and wayfinding.

- Travel is the act of movement itself – it is *"the task of performing the actions that move us from our current location to a new target location or in the desired direction"*[3, p. 183]. In other words, travel as a technique deals with mapping user's actions in a physical world (e.g. pressing a button) to movement in a virtual environment.
- Wayfinding *"is the cognitive process of defining a path through an environment, using and acquiring spatial knowledge, aided by both natural and artificial cues"* [3, p. 227]. Unlike travel, wayfinding addresses the psychological aspect of navigation. The aim of wayfinding is to help user orientate in the environment and make correct decisions regarding their movement in a virtual environment in the near future. For this purpose, a wide range of cues can be used. These include e.g. virtual map and compass, signs or motion cues.

The topic of navigation is too extensive to be discussed in-depth in this text. Detailed information related to both travel and wayfinding can be found in [3, chapters 6, 7].

4.1.2 Selection

Another interaction technique that is fundamental for most virtual environment applications is selection. In terms of interaction with terrain, it is

mainly used for selection of a particular 0- dimensional (point) or 2- dimensional (area) location on the surface of the model. The selected point or area can then be used for a variety of purposes such as

- modification of the terrain,
- placement of objects and/or labels,
- anchor point for some travel techniques (zoom, rotation),
- obtaining information related to the selected location and more.

If we allow user to select multiple points at the same time, capabilities of selection technique can be greatly expanded. For example, selection of several consecutive points can be interpreted as an outline of path between two distinct points.

However, selection technique raises an important question – which part of the screen should be selected once user does an action (presses a button, executes a gesture) that is evaluated as a selection command? Answer to this question depends on many factors including properties of the input device and type of selection – due its extent, it is out of the scope of this thesis. Detailed information regarding selection can be found in [3, chapter 5].

4.2 Input devices

4.2.1 Keyboard and mouse

Standard computer keyboard and mouse (K&M) are the most common input devices, available for use on almost any personal computer in the world. Their widespread availability and ease of use make them the most appropriate input device for an application that is targeted for use by a general public.

Interaction with application using K&M is not overly complicated to implement. The most difficult part lies in figuring out an intuitive mapping – ordinary mouse has only two degrees of freedom (DOF) which doesn't allow for unrestrained movement in a 3D scene. This issue can be alleviated by usage of keys on mouse or keyboard.

Quality of the used mapping also depends on the application's design. If the user views the scene using a first-person perspective that is associated with some virtual avatar, a common approach is to change position in the

scene by keyboard and change orientation of the view by mouse. Specifically, the position is often changed by arrow or WASD keys while mouse axes are mapped to yaw and pitch of the camera.

If the user is looking at the scene using an aerial or overhead view, a different mapping might be used. For example, a camera which is located above the scene can be moved, rotated around an anchor point or zoomed to a point by dragging the mouse in an appropriate direction while one of its buttons is pressed down. This approach is used in several map applications, e.g. *Google Earth*[1].

When using K&M, selection is most often executed by pressing a special key or a button.

4.2.2 3D Mouse

A 3D mouse is a special type of input device that was specifically designed for 3D interaction [3, p. 95]. Unlike a regular computer mouse, a 3D mouse provides 6 degrees of freedom (6-DOF) – translation along and rotation around all 3 perpendicular axes. This allows for very intuitive implementation of travel using only the mouse itself.

Similarly to the situation with K&M, a special key can be used to issue the selection command. For this purpose, we can use a 3D mouse in conjuction with keyboard or a regular mouse. If available, we may also use one of the programmable keys that are located on some 3D mice.

Unfortunately, 3D mice are not widely used due to their high cost which makes them unsuitable for most users.

4.2.3 Motion tracking devices

Another (though less common) option is to utilize input devices that detect user's motion in real world. Motion can be detected using one of several methods which include e.g. magnetic, mechanical, optical or acoustical tracking [3, p. 97]. In this section, we will focus only on selected devices based on optical tracking.

- *Leap Motion* is a small peripheral that tracks motion of hands and fingers using a set of infrared cameras and LEDs. Main advantage of the device is its high accuracy and millimetre-level precision [9]. One possible way to implement the travel technique is to measure relative deviation of user's hand from the centre of device's view frustum and

1. http://www.google.com/earth/

interpret the deviation as a relative motion in the given direction. This approach is already used in desktop version of *Google Earth*.

Main drawback of Leap Motion is that usage of the device for a longer period of time may cause a strain on user's hand.

- *Microsoft Kinect* is a motion tracking device introduced in 2010 exclusively for the Xbox 360 video game console. In 2011, it was made available for use on personal computers as well.

  The device scans an area in front of it using both regular camera and infrared laser. The reflected rays are detected by CMOS sensor and used to create a depth map [12, p. 14]. Combination of RGB and depth image allows the device to recognize figures and movement of several of users.

  The quality of tracking depends largely on the environment and other objects in the room which makes Kinect unsuitable for applications which require high level of precision. On the other hand, the device provides user a reasonable freedom of movement – which might be useful in implementation of the travel technique.

- *NaturalPoint OptiTrack* consists of multiple cameras – at least six – that surround the user and track position of passive markers that are attached to the user or some other object. All cameras have to be fixed and calibrated in order to compute position of the markers in space using triangulation. Orientation of the markers can also be tracked if the markers form a rigid body [12, p. 12].

  The OptiTrack system offers both precision and freedom of movement. Tracking of position and orientation allows for intuitive implementation of travel technique. However, unlike Leap Motion or Kinect the system is targeted for professionals. For example, it is widely used in the film and video game industries for the purpose of motion capture.

All input devices which rely only on optical tracking share a common disadvantage – there's no way for user to issue commands (e.g. selection) except of using a motion gesture. Common gestures include e.g. swipe, finger tap, waving hand – all of which affect user's posture and precision in a negative way, possibly .

In case of OptiTrack, this issue can be alleviated by usage of another input device – ideally a wireless controller that doesn't restrict user's movement in real world. This solution is also applicable to Kinect if the controller

is small enough so as not to decrease quality of the motion tracking. In case of Kinect, this issue can also be avoided by usage of voice commands which are supported by the device.

5 Design

This chapter covers design of *Land* ability – the application module which was developed as a part of the thesis.

5.1 vrecko

The module was created as an extension of the *vrecko*[1] programming framework which is being continuously developed and used in HCI laboratory on Faculty of Informatics of Masaryk University.

Both *vrecko* and the newly created extension are written in C++ and use OpenSceneGraph[2] graphics library for the purpose of rendering and management of the scene graph.

Every extension of the *vrecko* framework belongs to a plug-in, which serves as a grouping of related extensions with similar functionality. Furthermore, there are two types of extensions – *Device* and *Ability*. Extensions of the *Device* type handle I/O communication with supported hardware devices while the extensions of the *Ability* type provide almost any desired functionality that is unrelated to the hardware devices.

The newly created extension was created as a part of the *Nature* plug-in which groups abilites that deal with entities and simulations related to natural phenomena.

5.2 Design goals

The new ability was designed with several main goals in mind:

1. Create an extensible interface that would allow the ability to support a wide range of input terrain data.

2. Handle the loaded data in such a way that would be independent of the type of its source and internal structure. This would also allow other abilites to easily obtain required data from the *Land* ability for their own purposes.

3. Provide an efficient rendering of the terrain data, allowing the application to work with large and detailed landscapes.

---

1. http://vrecko.cz/
2. http://www.openscenegraph.org/

5.3 Data sources

5.3.1 Real-world data

In co-operation with Laboratory on Geoinformatics and Cartography[3], ČÚZK was selected as a primary provider of the real-world terrain data of the Czech Republic. For the purpose of further research, support for two different file formats was implemented. All files provided by ČÚZK store their coordinates in the S-JTSK Křovák East North coordinate reference system (see section 2.2.1) and elevation in the Baltic Vertical Datum - After Adjustment (see section 2.2.2).

- *ZABAGED ® altimetry grid 10x10 m* is a text file which stores a DTM (see section 3.1) as an unsorted set of regularly sampled 3D points. X and Y values specify coordinates of the sample (in S-JTSK) while the Z value specifies elevation of the Earth's *terrain* at a sampled point. Distance between two neighbouring samples is 10 metres on both X and Y axes. Each grid file covers approximately 18 km^2 [28].
- *Digital Surface Model of the 1st generation* is a text file – saved in a file with an **.xyz* extension – which stores a DSM (see section 3.1) as an unsorted set of irregularly sampled 3D points. X and Y values specify coordinates of the sample (in S-JTSK) while the Z value specifies height of the *surface* at the given point. Even though a single DSM file covers a smaller area (approximately 5 km^2) than an altimetry grid, it contains far more samples due to increased level of detail, especially in urban areas [25].

  Thanks to their similar internal structure, implementation of DSM 1G also supports a Digital Terrain Model of the 4th and 5th generation [26, 27]. In this case, the elevation represents height of the terrain, as in the case of the altimetry grid.

5.3.2 Terrain generation

If the application doesn't require real-world terrain data, usage of a terrain generation algorithm might be appropriate. These algorithms are capable of creating visually more or less convincing terrain surface and are often based on fractals. Their main advantage is that the terrain data is generated on-the-fly and no data is stored on secondary storage device.

3. http://www.geogr.muni.cz/lgc/

This work uses slightly modified version of several algorithms (namely Perlin noise, midpoint displacement and random faults) that were described and implemented in [22] as another type of a possible data source, even though the algorithms are not the focus of this thesis.

5.3.3 Raster image

Another common way is to store terrain data in a form of a raster image, where each pixel represents a sample in a regular grid. Position of the sample is defined by pixel's coordinates in the image file and the pixel value specifies elevation of the sample.

However, this approach has several disadvantages – relation of the pixel value to the absolute elevation levels is not defined. The input range $[0, 255]$ might represent any interval in terms of elevation, e.g. $[-25.4, 68.3]$ or $[131.7, 958.2]$.

Another issue might be lowered precision of the stored values, caused by quantization of the pixel's values. This problem might be sufficiently supressed by a usage of all 24 bits of RGB image to store a single value or by a usage of 16-bit grayscale images.

5.4 Land ability

The ability's design reflects the fact that *Land* ability has been created with application in the field of cartography in mind. This specialization has allowed us to improve the overall performance of the extension. On the other hand, the current state of the ability doesn't support dynamic modification of the terrain data.

Core of the extension is the `LandAbility` class which is derived from `vrecko::Ability`. As is the case with all *vrecko* abilities, its instance is created automatically by the *vrecko* framework if the input file contains a paired element `<Ability>` with inner elements `<PluginName>` and `<Name>` set to "Nature" and "Land", respectively. See 5.1 for an example of usage.

Upon initialization, the instance of `LandAbility` processes the content of its element `<Parameters>`. The `<LandBlocks>` element serves as a grouping of any number of `<Block>` elements. Each `<Block>` element defines a separate land block and contains all parameters which belong to the same land block. Pointer to this XML node is passed as the only argument to the constructor of the `LandBlock` class and is then used to initialize the whole land block.

The remaining parameters are loaded as global properties (see section 5.4.5 for details).

Listing 5.1: Initialization of the *Land* ability

```
<EnvironmentObject>
    <ID>535969142</ID>
    <Ability>
        <Name>Land</Name>
        <PluginName>Nature</PluginName>
        <Parameters>
            ...     <!-- Global properties -->
            <LandBlocks>
                <Block>
                ...     <!-- Block 1 -->
                </Block>
                ...
                <Block>
                ...     <!-- Block N -->
                </Block>
            </LandBlocks>
        </Parameters>
    </Ability>
</EnvironmentObject>
```

It is important to note that no more than one instance of Land ability should be created. Existence of multiple instances may lead to an unexpected behaviour.

Furthermore, the instance of `vrecko::EnvironmentObject` which contains the *Land* ability has to use a specific ID – *535969142* – which is used by other abilities to discover *Land* ability at runtime.

Unlike most abilities which currently exist in the *vrecko* framework, the *Land* ability is not associated with any particular geometry in the scene and serves rather as a manager of all land blocks in the scene.

5.4.1 Land block

Instance of the `LandBlock` class represents an indepent block of land with all of its associated internal structures. Each land block consists of 3 different parts which are initialized in the following order:

1. Properties
2. Data
3. Mesh

First of all, a set of properties is loaded from the input XML node, using the pointer to the <Block> element that is passed to the constructor of LandBlock). In the next step, some of these properties are used to obtain raw terrain data using an appropriate data source. Finally, terrain data are used to create a mesh which is rendered on the computer screen.

In order to successfully create an instance of LandBlock, all three steps have to succeed in their actions. Any error (e.g. an invalid parameter value or running out of system memory) results in cancellation of the initialization process. Pointers to all successfully initialized land blocks are stored in std::vector in the sole instance of LandAbility.

5.4.2 Properties

The purpose of classes derived from Properties is to store all known information about a particular land block. Due to the difference among various data sources, the actual set of parameters which are loaded from the XML varies as well.

The abstract class Properties contains the most general properties which are used by every single land blocks. Its sub-classes contain specialized properties related to the internal structure of a land block (e.g. width and length of a regular grid). All other classes are fully specialized and there exists a separate sub-class for every supported type of data source. Class diagram of this hierarchical structure can be seen in appendix C.1.

However, the higher number of derived classes may result in more complicated initialization code. For that purpose the factory method design pattern is used, implemented in the form of static method Properties::CreateFromNode(). This method evaluates the XML node which is passed as an argument and selects (using the value of <DataSource> element) an appropriate sub-class instance of which is created and returned as a result. The newly created instance automatically loads all properties in its constructor which concludes the first phase of the LandBlock's initialization.

Loading properties from XML

All properties are loaded from parameters stored in the input XML node using methods from the vrecko::ReaderWriter class. In order to successfully load values of all properties, the parameters in the input XML node have to satisfy two conditions:

1. All required parameters (e.g. data source) have to be set.

2. All provided parameters – required or not – have to be valid.

If the content of the XML node fails to meet the aforementioned criteria, the initialization process is cancelled. This error is propagated (using exceptions) to the `LandAbility` and the initialization of the corresponding `LandBlock` is cancelled as well. On the other hand, successful creation of the instance of `Properties` guarantees that all properties have been properly set.

As already stated, parameters which are not required can be omitted. In such a case, default value is used instead. Detailed list of all parameters, their description and their default values is located in the appendix B.2.

5.4.3 Data

After the succesful creation of an instance of `Properties`, the initialization of the `LandBlock` continues with its next phase. The goal of this phase is to create and initialize a new instance of `LandData` – or, more specifically, one of its sub-clases – which stores the raw terrain data.

The instance of `Properties` is used as the input argument of another factory method – `LandData::CreateFromProperties`. This method selects an appropriate sub-class of `LandData` based on the value of the `DataSource` property. Current version of the *Land* ability supports two types of land data:

- `HeightFieldData` stores the terrain data in a form of regular grid with uniformly spaced samples.
- `TinData` stores the terrain data as an unsorted list of irregular samples in a 3D space. This provides higher flexibility in terms of input but also introduces multiple complications caused by a more general approach.

Once created, the instance of `LandData` is filled with raw terrain data in its constructor. Due to severe differences, the process which is used to obtain the data is described in detail in sections 6.1.1 and 6.2.1. As is the case with `Properties`, any error in initialization results in failed initialization of `LandBlock` as well.

5.4.4 Mesh

Construction of a polygon mesh is the last phase of the `LandBlock` initialization process. Its design shares multiple similarities with previous phase.

Once again, the result of a previous phase is passed to a factory method in order to create a new instance of an appropriate sub-class. In this case, an instance of `LandData` is passed as an argument to the static method `LandMesh::CreateFromLandData()` which returns a new instance of `LandMesh`.

As shown in section C.3, the derived classes are `HeightFieldMesh` and `TinMesh` instances of which are created using the input terrain data in a form of `HeightFieldData` and `TinData`, respectively.

This phase is also the only part of the ability which directly interacts with the OSG library and scene graph of the application. In general, the mesh construction part of *Land* ability addresses the following tasks:

1. Computation of normal vectors.

2. Application of textures on the surface of the mesh.

3. Segmentation of the mesh.

Given the varied internal structure of the terrain data, the mesh construction process varies as well. Different techniques used by implementations of `HeightFieldMesh` and `TinMesh` are therefore described in detail in sections 6.1.4 and 6.2.3.

It is possible to entirely skip the mesh construction phase of any block. This way the *Land* ability can be used as a tool to import or generate terrain data which are then processed by a different part of the application.

5.4.5 Global properties

It may be necessary (or useful) for some properties to share the same value among all instances of `LandBlock` in the scene. This can be achieved by utilizing the singleton design pattern which allows us to create only a single instance of a given class. Moreover, this instance is obtained via static method which allows us to access the instance from any place in the source code.

In case of the *Land* ability, globally shared parameters are stored in a singleton class `GlobalProperties`. These settings include e.g. location of the texture cache or rendering settings. See section B.1 for detailed list and explanation of the parameters.

Elevation legend

One of the global properties is an instance of `ElevationLegend`. This class is used to compute colour values which are assigned to vertices of an untextured mesh. In case of a textured mesh, white colour is used instead.

An instance of `ElevationLegend` contains a list of pairs in the form of *(elevation, colour)* which define colour of the vertex at the given elevation. Vertex colour at any requested height is then computed by linear interpolation. If the requested elevation is smaller or larger than all of the defined levels, colours at the interval bounds are used.

Figure 5.1 displays the colouring created by an elevation legend which contains 6 entries for the elevation levels 0, 400, 850, 1400, 1850 and 2500.

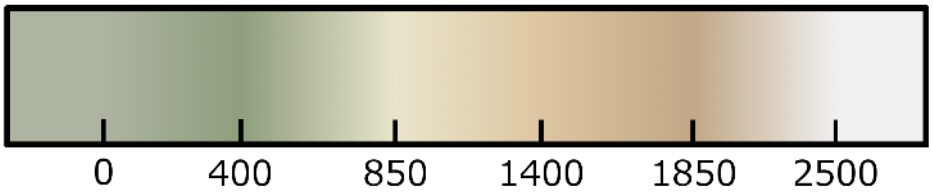


Figure 5.1: Example of an elevation legend created by linear interpolation of 6 colour values.

5.5 Textures

In order to increase the degree of realism in our scene, we will utilize texture mapping which is without a doubt one of the most important techniques designed for this purpose. However, to achieve the desired effect, it is necessary to consider the source of the terrain data and use textures with an appropriate content.

If our terrain data match an area in the real world, it is highly advisable to use orthoimagery of the area as a primary texture source. This is the case of terrain data loaded from the altimetry grid or digital surface model, both of them provided by ČÚZK.

Another way is to compute the texture dynamically, using shader programs. In this case, we use several different textures which represent various biomes or materials (such as grass, sand, rock or snow). These textures are then used to compute the final blended texture at run time, according to a pre-defined set of rules (e.g. "use grass texture in areas with lower elevation"). Source textures should be tileable, evenly lit and shouldn't contain

any distinct object so as to avoid visual artifacts or repeated patterns. This option is more suitable for terrain data generated by an algorithm or for real world data for which the orthoimagery is not available.

For the purpose of texturing, the *Land* ability uses orthoimagery provided by ČÚZK and noise pollution data provided by INSPIRE[4]. Both services are provided via WMS (Web Map Service) without any fees for non-commercial purposes. However, textures from these sources are only used in land blocks which are created from real world data (altimetry grid and digital surface model). Land blocks which are created from another type of data source are coloured by the global elevation legend (see section 5.4.5) instead.

5.5.1 LandTexture class

The `LandTexture` is an abstract class which contains a texture along with a rectangular area (defined by boundaries in S-JTSK coordinate system) which is covered by the texture. During its initialization, the instance of `LandTexture` builds two strings. One of them is a unique identifier which is used as a local file name for the downloaded texture (see section 5.5.2). The other string is URL which is used to directly access the texture on a distant server via WMS. For data transfer via HTTP, the C/C++ version of the *libcurl*[5] library is used.

The derived classes `OrthoPhoto` and `NoiseTexture` differ in a way the URL is built – orthoimagery and noise pollution data are not provided by the same WMS. Moreover, each texture type is obtained using a slightly different set of parameters.

Each WMS can restrain the maximum dimensions of the image that is downloaded using the `GetMap` request [11, p. 23]. In case of both WMS used by `LandTexture`, maximum size of the image is limited to $2,500 \times 2,000$ pixels. To optimize GPU's performance, the `LandTexture` downloads only images with dimensions equal to a value which is power of two. Combined with limitations of `GetMap`, the maximum size of an image downloaded by `LandTexture` is 1024×1024.

5.5.2 Cache

All textures are downloaded to a specific cache folder on the disk, where each texture is stored as a separate JPEG (for orthoimagery) or PNG (for

4. http://geoportal.gov.cz/web/guest/wms/
5. http://curl.haxx.se/libcurl/

noise pollution data) image file.

File name of the image comprises coordinates of the area which is cover by the texture, size of the texture and its type. This way, the file name serves as a hash – if the texture with required properties (area, size and type) already exists in the cache folder, it is loaded from the secondary storage. If not, it is downloaded via WMS and stored in cache for a possible – and very likely – repeated usage.

6 Implementation

6.1 Height field

Support for land blocks which are based on regular height field is implemented in classes `HeightFieldData` and `HeightFieldMesh`. As stated in sections 5.4.3 and 5.4.4, these classes are derived from abstract classes `LandData` and `LandMesh`.

6.1.1 Data storage

An instance of the `HeightFieldData` class contains terrain data sampled at a uniform rate. All samples are stored in a single one-dimensional array the elements of which cover a rectangular grid. The dimensions of the grid – labeled as *width* and *length* – are saved in the instance of `HeightFieldProperties` which is associated with an instance of `HeightFieldData`. Values of *width* and *length* are fixed and cannot be modified once the grid is created.

6.1.2 Height point

Each elevation sample is stored in an instance of `HeightPoint`. This class contains a single floating point value which represents an elevation of the sample in meters above sea level. Thanks to the regular nature of the height field, it is not necessary to store coordinates of the sample in a grid – the coordinates can be trivially inferred from the sample's position in the one-dimensional array.

Invalid samples (e.g. those with no set value) use special value which represents positive infinity – more specifically, the value provided by `std::numeric_limits<float>::infinity()`.

The `HeightPoint` class also contains implementation of arithmetical and relational operators which handle a possible invalid value in an appropriate way, along with operator which provides implicit conversion to `float`.

Due to the very high number of instances, it is important to examine the issue of memory usage. On most architectures, 4 bytes of memory are used by a variable of the `float` type. The same applies to a variable of the `HeightPoint` class – its only attribute being a single `float` variable.

Should we decide to add another variable, this will no longer apply. Addition of a `bool` flag that would be used to mark invalid samples (instead

of the infinity) would result in a usage of 8 consecutive bytes. In this case, 5 bytes would be used for storage of the data itself while 3 more bytes would be used by the compiler for proper alignment of variables in the system memory.

6.1.3 Data provider

As stated in section 5.3, the *Land* ability is designed to support multiple data sources. However, inclusion of multiple independent algorithms in the class itself would result in needless increase of complexity. For this purpose, the task to obtain the terrain data is therefore delegated to a different class – namely `HeightFieldProvider` and all classes derived from it.

`HeightFieldProvider` is an abstract class which contains a single public method `ProvideData()`. Implementation of this method in a derived class has two main tasks. First of all, it allocates the grid which belongs to the instance of `HeightFieldData` which is using the data provider. In the next step, the method is used to obtain the terrain data and fill the grid with it. In order to a gain a more exclusive access to private attributes of `HeightFieldData`, the `HeightFieldProvider` is is set as its friend class.

Each class derived from `HeightFieldProvider` supports a different data source. The current version of Land ability supports five different data providers, as seen in the class diagram shown in appendix C.4. An appropriate sub-class is selected and instantiated using the factory method design pattern in the constructor of `HeightFieldData`.

Terrain generators

The derived classes `MidpointDisplacementGenerator`, `PerlinNoiseGenerator` and `RandomFaultsGenerator` implement terrain generation algorithms which are described in [22]. These algorithms can be used to generated a land block of any size (which is specified by XML parameters) using a specific set of parameters – see appendix B.2.

Raster image importer

As its name suggests, the `RasterImageImporter` is used to load terrain data from raster image files. Dimensions of the grid are set automatically to the dimensions of the image, so that each pixel corresponds to a single elevation sample.

The image file is loaded as an RGBA image with 32 bits per pixel. The elevation of each sample is computed from the pixel's colour using one of preset functions such as average or sum of RGB channels (see appendix B.2 for list of all available functions).

File format support depends on the list of available OSG plug-ins. In case of *vrecko*, all common image formats (such as BMP, PNG, JPEG or TIFF) are supported. It is highly recommended to use only lossless compression to avoid any errors caused by loss of detail.

Zabaged grid importer

The `ZabagedGridImporter` loads terrain data from the altimetry grid described in section5.3. Text files provided by ČÚZK contain approximately 185,000 elevation samples per file. This number of samples covers approximately 18 square kilometers of land.

Each sample is defined by three values – X and Y coordinates in the S-JTSK coordinate reference system (see section 2.2.1) and elevation of the sample in meters above sea level. It is important to note that all samples lie in a discrete grid in which the distance between two 4-adjacent samples is 10 meters.

Figure 6.1 visualizes the coverage of samples in a demo file available at [28]. It can be seen that the rectangular area covered by samples is slightly rotated and its sides are therefore not aligned to the axes of the S-JTSK coordinate system. In this particular case the sampled area is rotated by approximately $7.5°$ – though the angle varies and is based on a specific area which is covered by samples in a file.

This rotation, along with the discrete nature of the sampled area pose an implementation problem. In order to store the samples in a regular grid (i.e. two-dimensional array), we must first select an appropriate strategy. For example, we can deal with the issue using one of these two options:

(a) Transform the samples using translation and rotation so that the transformed area is aligned with axes of the S-JTSK coordinate system.

(b) Store the samples in the regular grid along with all points which are part of the axis-aligned bounding box even though their value is not defined.

The first approach works well theoretically but its application in practice is prone to numerical errors. These may occur during several steps of the transformation process – computation of an approximate value of the angle,

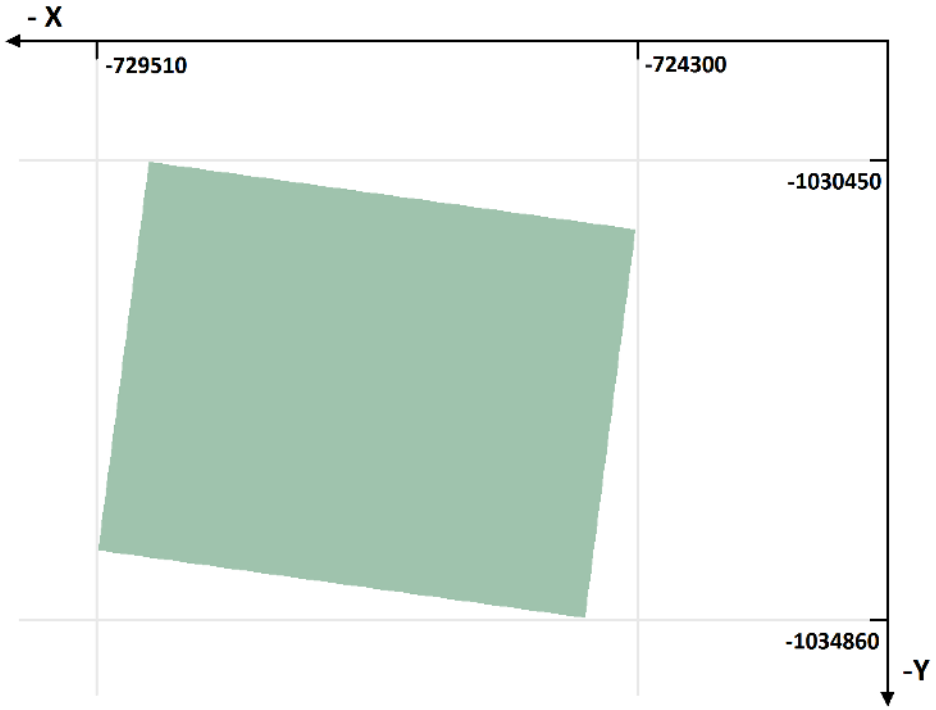


Figure 6.1: Coverage of samples in the demo file [28].

rotation of the original samples or computation of the new samples using interpolation.

Another option (and the one used by `ZabagedGridImporter`) is to enclose all samples in a bounding box which is aligned to the axes of the S-JTSK coordinate system. Using this method, the height field grid stores elevation of all points located inside of this bounding box. Points with no defined elevation (i.e. points with coordinates that are not located in the input file) use a special value, as stated in section 6.1.2.

The main advantage of this approach is that the samples require no additional transformation and their original elevations can be used directly. On the other hand, this technique requires larger grid with more elements which also results in an increased memory usage.

In case of the demonstration file, the original altimetry grid contains 181,505 sampled points. Axis aligned bounding box of this area is 522 samples wide and 442 samples long. The final grid consists of 230,724 elements and this technique therefore results in a memory usage increased by $\sim 27.12\%$.

For practical reasons, the imported samples store only their elevation, while the instance of `Properties` stores the coordinates of the bounding box in an instance of the `Area` class. If necessary, the S-JTSK coordinates of any point can be computed from its relative location in the grid and the S-JTSK coordinates of the bounding box.

6.1.4 Mesh creation

As stated in section 5.4.4, the last phase of the land block initialization is construction of the mesh. In case of land blocks based on regular height field, a new instance of `HeightFieldMesh` is built using content of a corresponding instance of `HeightFieldData`.

In the first part of the construction process, we use the input data to create a new vertex array. For each sample, we create a new vertex (x, y, z) where x and z are coordinates of the sample in the input grid and y is the sample's elevation. Invalid samples are skipped and no vertices with their x and z coordinates are created.

Normal computation

In the next phase, we compute normal vectors of the mesh – one normal vector for each vertex of the mesh. This step is largely simplified by regular property of the height field grid. It allows us to compute the normal vectors without the use of more complicated data structures [20, p. 131] which are designed for applications that work with general meshes.

We may compute the normal vector associated with a vertex v using its neighbouring vertices (see figure 6.2 for reference) and the following steps:

1. Label the 8 neighbouring vertices in a clock-wise order starting with any vertex and going from v_0 to v_7.
2. Repeat for all $x \in [0...7], x \in \mathbb{N}$. Note that $v_8 \equiv v_0$.
   (a) Check whether vertices v_x and v_{x+1} both exist. If they do, compute normal vector of the face f_x which they form along with vertex v.
3. Compute a vertex normal by averaging all face normals computed in step 2.
4. Normalize the vertex normal so that its length is equal to 1.

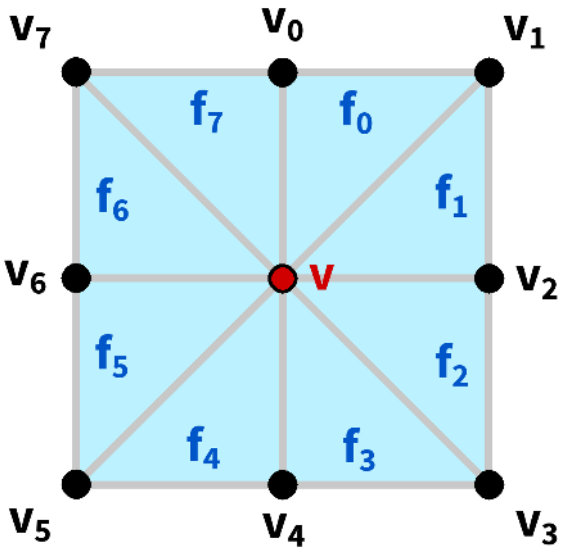


Figure 6.2: Neighbourhood of vertex v which is used to compute its normal vector.

Using this approach, we can design a one-pass algorithm that computes values of all normal vectors. Its main advantage is that no additional memory (array) is used to store temporary data. This is crucial in situations in which we deal with large amount of samples.

The drawback lies in an increased computation time caused by some values – mainly the normal vectors of each face – which are computed repeatedly. This does not, however, presents any significant problem in the initialization phase. If necessary, the computation time can be reduced by running the algorithm in parallel. The vertex array is only used for reading operations and the write operation in the normal array doesn't affect its neighbours.

Segmentation

Instead of creating one large block, the `HeightFieldMesh` internally builds a set of smaller blocks called segments. Each segment is created as an instance of `HeightFieldSegment` and contains its own data, independent of other segments which belong to the same mesh.

Creation and handling of individual segments is completely managed by the instance of `HeightFieldMesh` to which they belong. The segmentation process is hidden from other classes – it is therefore possible to work

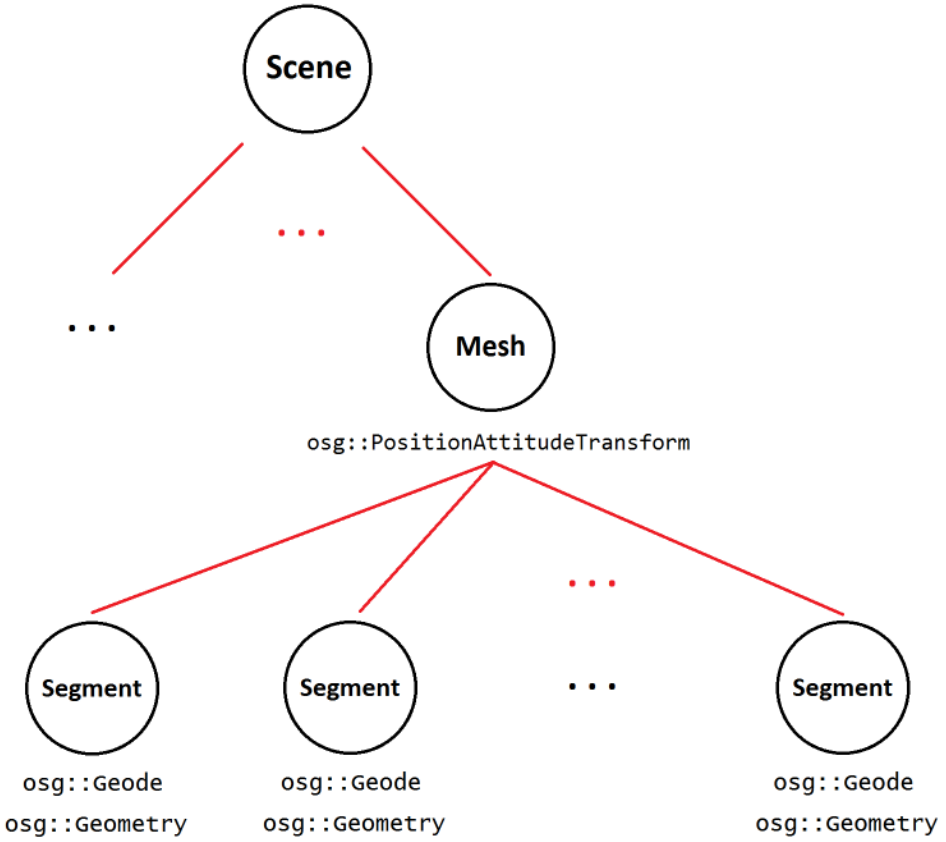


Figure 6.3: Mesh and its segments shown as apart of a scene graph.

with the `HeightFieldMesh` as if it were a single large mesh.

In terms of a scene graph, each segment represents a single leaf node. All segments that belong to the same block (and contain at least one face) are connected under the mesh node which is located in the `HeightFieldMesh` and which is accesible via public method. The resulting mesh node is then added to the scene graph – this way the rendering process is automatically handled by OSG library in its *Draw* phase. See figure 6.3 for reference.

There are two main reasons why we split mesh in many smaller parts:

1. Division of the mesh into smaller segments allows for easier texture mapping.
2. Index array which contains too many elements may cause rendering issues on some GPUs.

In the first case, we're limited by the maximum resolution of a single texture. Even though majority of current GPUs theoretically support textures with resolution up to 16384^2, we have experienced practical difficulties while handling textures larger than 4096^2 – which may not be enough to cover larger areas (in our case 18 square kilometres).

Another approach might be to use multi-texturing and map a different textures to a various parts of mesh. Again – each GPU is limited by a maximum number of texture units which can be used on a single mesh. In OpenGL, this value can be obtained by calling the library function `glGetIntegerv()` with an appropriate value as a parameter (in this case `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`) [19, p. 355].

Segmentation of the mesh allows us to map a different texture on each small segment of the large mesh. In this case, the only practical limitation is the amount of video memory available to the application.

The second reason for segmentation of the mesh is to avoid rendering problems. We have observed that if the element array of a vertex array object contains more than 2^{16} indices, visual artifacts might occur on some systems. These range from occasional missing primitive to not rendering the mesh at all.

As a result, every segment is a separate vertex array object with its own vertex, normal, colour and texture coordinate arrays.

Segment size

Vertices of the mesh are split into segments based on the location of their corresponding samples in the data grid. All segments of the land block cover a square area and have the same size $n \in \mathbb{N}$ which represents number of vertices which constitute a side of the square. Figure 6.4 shows segmentation of the demonstration file (see section 6.1.3) using $n = 50$. This divides the mesh in 99 segments. Segments that are coloured in red contain no vertices and are not created at all.

Segment size across the land blocks may vary, though the minimum (and default) size of any segment is 32. See appendix B.2 for information on how to override this value.

It is important to note that the size of the segment is automatically increased by 1. This increase in size is used by the segment to copy data from neighbouring segments in the X and Z directions. This allows the segment to render faces on the border between adjacent segments and produce an illusion of a single, seamless mesh.

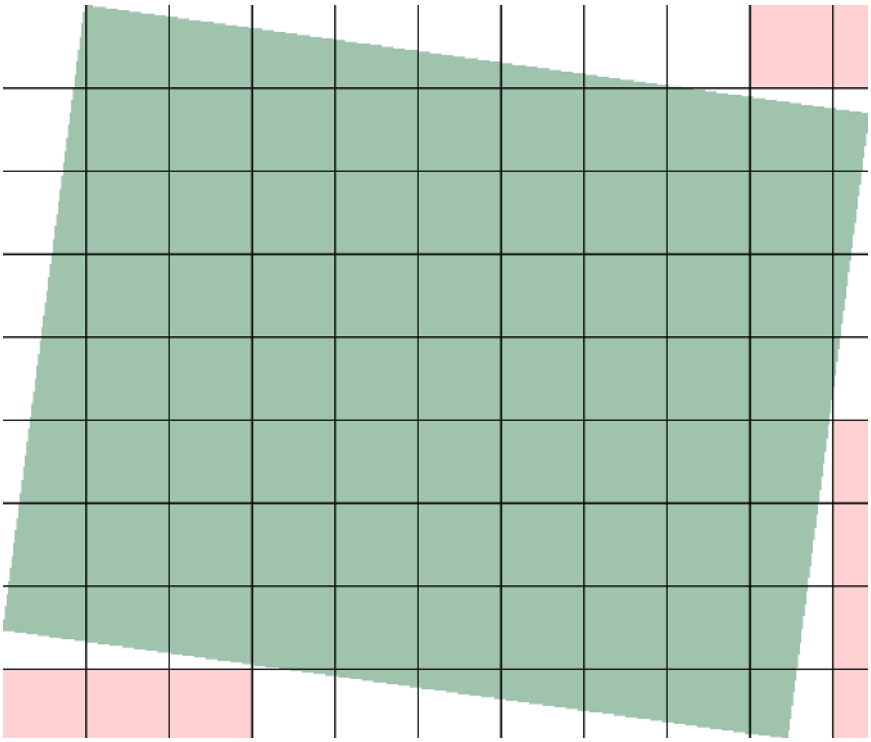


Figure 6.4: Segmentation of the data grid which contains terrain samples from the demonstration altimetry grid. Value $n = 50$ is used as a segment size. Segments coloured in red contain no vertices.

6.2 TIN

Beside land blocks that are based on regular height field, the *Land* ability also provides support for triangulated irregular networks (TIN), described in section 3.2.2. As stated in sections 5.4.3 and 5.4.4, this functionality is implemented in the classes `TinData` and `TinMesh`.

Similarly to the implementation of height field, an instance of `TinData` is used to obtain the terrain data and the corresponding instance of `TinMesh` uses it to construct mesh which is rendered on user's screen.

6.2.1 Data storage

Similarly to the implementation of height field, the `TinData` also stores terrain data in a one-dimensional array. However, the samples' coordinates in the *XZ* plane are not limited to a particular subset of $\mathbb{R}^2$ (as is the case with

height field). Each elevation sample is stored as a point (x, y, z) in three-dimensional space where x and z define the sample's position in the *XZ* plane and y specifies the sample's elevation.

The current version of *Land* ability supports only one type of data source – specifically TIN files provided by ČÚZK. These files are available in 3 versions:

- Digital Terrain Model of the 4^{th} generation *(DMR 4G)* [26]
- Digital Terrain Model of the 5^{th} generation *(DMR 5G)* [27]
- Digital Surface Model of the 1^{st} generation *(DMP 1G)* [25]

Difference between the terrain and surface model is described in section 3.1. *DMR 4G* and *DMR 5G* differ only in their level of precision and availability – *DMR 4G* covers the entire area of the Czech Republic while *DMR 5G* offers higher precision.

Each TIN file is stored in a text file with the `*.xyz` extension. This file contains an unsorted set of elevation samples which cover an area of 5 square kilometres (2.5×2 km). As is the case with the altimetry grid, all samples are defined by 3 values – two of them contain position of the sample in the S-JTSK coordinate reference system while the third contains elevation in meters above sea level.

6.2.2 Face computation

The TIN files provided by ČÚZK contain only points (vertices) of the TIN and no further information regarding its structure. Unfortunately, information regarding faces is essential if we wish to construct and render TIN as a triangular mesh. For the purpose of clarity, the vertex data which comprise the input file will be referenced as *point cloud* in the further text.

This situation poses a problem, as the computation of normals and faces from the point cloud is by no means a trivial process. We can use one of several different techniques to obtain the required data from the initial point cloud.

Triangulation

The most direct approach is to compute a triangulation of the point cloud. For this task, we can use the well known *Delaunay triangulation* which is thoroughly described in scientific literature, e.g. in [2, p. 191 - 215].

The Delaunay Triangulation is designed for use in a plane but the algorithm can be extended for use in 3D [17]. However, drawback of this modified version lies in is its increased time complexity.

Still, there's a way to use the regular Delaunay triangulation in 2D. Majority of elevation samples have unique coordinates in the *XZ* plane. If we project the elevation samples to this plane, we will encounter only a few collisions – most of them being building walls with one sample at the ground level and another at roof. Once the computation is complete, we can "raise" the vertices their original elevation and compute the missing triangulation of the walls.

Main advantage of the triangulation is preservation of the original samples, i.e. number of vertices.

Surface reconstruction

Another approach we can take is to utilize one of the algorithms which deal with reconstruction of the surface from a point cloud. These algorithms are designed to create mesh from a point cloud which is usually produced by a 3D scanning device. One of the more popular algorithms used for this task is Poisson surface reconstruction [13].

Disadvatage of this technique is that the surface reconstruction algorithms create a completely new set of points. Vertices of the new mesh most often don't match the initial point cloud in neither location nor their count. This may lead to an undesired loss of detail, especially visible in case of sharp edges.

Pre-processing

Due to the high number of vertices in the point cloud (often as high as 2 million points or more), the computation is quite demanding – both in terms of performance and time. For this reason, each point cloud file is pre-processed before actual usage.

Because of their complexity, an implementation of any of the aforementioned algorithms is outside of the scope of this thesis. The point cloud has to be pre-processed outside of the *vrecko* framework using several tools. In short, the following course of action is taken:

1. *XYZ preparation utility* is used to convert point cloud to the OBJ[1] file format. The *XYZ preparation utility* has been created and is provided

1. OBJ specification is available at: http://www.martinreddy.net/gfx/3d/OBJ.spec

as a part of this thesis.

During this step, all points are translated so that their x and z coordinates lie in the interval $[0, 2500]$ and $[0, 2000]$, respectively. S-JTSK coordinates specifying the real-world area which is covered by the point cloud are stored in a separate text file with the `*.area` extension.

2. The transformed point cloud is loaded into *MeshLab* and new mesh is created from it using Poisson surface reconstruction algorithm.

3. In the last step, *Blender* (or any other suitable 3D modelling application) is used to remove excess parts of the mesh – mainly vertices which lie outside of the area covered by the original point cloud.

Detailed information including step-by-step guide is located in appendix A.

The `*.obj` and `*.area` files are the result of the pre-processing phase. These files are loaded by the *Land* ability at runtime, with their content stored in an instance of `TinData`.

Implementation of an automatical pre-processing using a third-party library such as *PCL*[2] or *CGAL*[3] should be considered in the future. Unforunately, this option was not feasible in this version due to time constraints.

6.2.3 Mesh creation

Once the input OBJ file is loaded, an instance of `TinData` contains all information that are needed to build a mesh. Because of that, the initialization of `TinMesh` is a rather straightforward process.

Geometry primitives are divided in multiple element arrays so that no array contains more than 2^{16} indices. This measure is taken because of the problem that may occur if we use an index array which contains too many elements – as previously described in section 6.1.4.

When compared to `HeightFieldMesh`, the real world area that is covered by the mesh is $3.6\times$ smaller (5 square kilometers instead of 18). This makes that the application of textures is easier as well. The highest resolution in which the orthoimagery is available uses 16 pixels per square meter. In total, the most detailed orthophoto covering the mesh's area would use

2. http://pointclouds.org/
3. https://www.cgal.org/

$10,000 \times 8,000$ pixels. It is clear that there's no need to use texture with size larger than 8192^2. This allows us to use just a single texture for the whole mesh and avoid any kind of segmentation, as seen in section 6.1.4.

As stated in section 5.5.1, the `LandTexture` downloads only textures up to the size of 1024^2. However, we can join several smaller textures to create a desired, larger texture. For example – to obtain a texture which contains 4096^2 pixels, we connect 16 smaller textures downloaded separately using `LandTexture`. Texture of this size is contains enough details to sufficiently cover whole surface of the mesh.

Finally, construction of the mesh is also simplified in terms of scene graph. Geometry of the whole mesh is contained inside of a single leaf node which is in turn connected to a parent node which translates and scales the mesh using the content of `Properties`. The parent node is then directly connected to the root node of the scene.

7 Conclusion

In this thesis, we studied visualization of cartographical data in virtual environment. For this purpose, we discussed several topics related to the subject such as coordinate reference systems, representation of terrain data and selected interaction techniques.

Practical result of this work is *Land* ability, a new extension of the *vrecko* framework. The extension is able to work with terrain data from various sources, including digital elevation model of the Czech Republic provided by ČÚZK. Furthermore, the extension is able to download an appropriate orthoimagery using specialized web service and texture the terrain model with it.

The features of *Land* ability were designed in co-operation with researchers from Faculty of Science and Faculty of Arts, Masaryk University. The extension is currently in a test phase and is and is expected to be soon used in research along with several other parts of the *vrecko* framework.

Future improvements of the ability may include e.g. optimization of the mesh using an appropriate algorithm, implementation of additional input formats and support for web services which provide data other than orthoimagery and noise pollution. Automatical pre-processing of the point cloud is also a significant feature implementation of which should be considered.

Literature

[1] Arul Asirvatham and Hugues Hoppe. *Terrain Rendering Using GPU-Based Geometry Clipmaps*. URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter02.html (visited on May 15, 2014).

[2] Mark de Berg et al. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN: 3540779736, 9783540779735.

[3] Doug A. Bowman et al. *3D User Interfaces: Theory and Practice*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004. ISBN: 0201758679.

[4] Howard Butler et al. *Spatial reference systems*. URL: http://spatialreference.org/ (visited on May 7, 2014).

[5] ArcGIS Resource Center. *What is an SRID?* URL: http://resources.arcgis.com/en/help/main/10.1/index.html#//006z000000w6000000 (visited on May 7, 2014).

[6] ArcGIS Resource Center. *Working with spatial references*. URL: http://help.arcgis.com/en/sdk/10.0/arcobjects_net/conceptualhelp/index.html#//0001000002mq000000 (visited on May 7, 2014).

[7] Mark Duchaineau et al. "ROAMing Terrain: Real-time Optimally Adapting Meshes". In: *Proceedings of the 8th Conference on Visualization '97*. VIS '97. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997, pp. 81–88. ISBN: 1-58113-011-2. URL: http://dl.acm.org/citation.cfm?id=266989.267028.

[8] Shahab Fazal. *GIS Basics*. New Delhi, India: New Age International Pvt Ltd Publishers, 2008. ISBN: 978-8122423761.

[9] Jože Guna et al. "An Analysis of the Precision and Reliability of the Leap Motion Sensor and Its Suitability for Static and Dynamic Tracking". In: *Sensors* 14.2 (2014), pp. 3702–3720. ISSN: 1424-8220. DOI: 10.3390/s140203702. URL: http://www.mdpi.com/1424-8220/14/2/3702.

[10] Lukáš Herman. *Vizualizace 3D modelů měst na webu*. Rigorózní práce. 2014. URL: http://is.muni.cz/th/222752/prif_r/.

[11] Open Geospatial Consortium Inc. *OpenGIS® Web Map Server Implementation Specification*. Version 1.3.0. Mar. 2006.

[12] Miroslava Jarešová. *Zachycení a vizualizace pohybu v reálném čase*. Bachelor thesis. 2012. URL: http://is.muni.cz/th/324777/fi_b/.

[13] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. "Poisson Surface Reconstruction". In: *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*. SGP '06. Cagliari, Sardinia, Italy: Eurographics Association, 2006, pp. 61–70. ISBN: 3-905673-36-3. URL: http://dl.acm.org/citation.cfm?id=1281957.1281965.

[14] Zhilin Li, Qing Zhu, and Christopher Gold. *Digital Terrain Modeling: Principles and Methodology*. Boca Raton, Florida: CRC Press, 2005. ISBN: 9780203486740.

[15] Peter Lindstrom and Valerio Pascucci. "Visualization of Large Terrains Made Easy". In: *Proceedings of the Conference on Visualization '01*. VIS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 363–371. ISBN: 0-7803-7200-X. URL: http://dl.acm.org/citation.cfm?id=601671.601729.

[16] Peter Lindstrom et al. "Real-time, Continuous Level of Detail Rendering of Height Fields". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 109–118. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237217. URL: http://doi.acm.org/10.1145/237170.237217.

[17] Pavel Maur. "Delaunay Triangulation in 3D". PhD thesis. Pilsen, Czech Republic: University of West Bohemia, 2002.

[18] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. "Decimation of Triangle Meshes". In: *SIGGRAPH* 26.2 (July 1992), pp. 65–70. ISSN: 0097-8930. DOI: 10.1145/142920.134010. URL: http://doi.acm.org/10.1145/142920.134010.

[19] Mark Segal and Kurt Akeley. *OpenGL 4.4 Core Profile*. Mar. 2014. URL: http://www.opengl.org/registry/ (visited on May 1, 2014).

[20] Colin Smith. "On Vertex-vertex Systems and Their Use in Geometric and Biological Modelling". AAINR19574. PhD thesis. Calgary, Alta., Canada, 2006. ISBN: 978-0-494-19574-1.

[21] United States Geological Survey. *Global 30 Arc-Second Elevation (GTOPO30)*. URL: https://lta.cr.usgs.gov/GTOPO30 (visited on May 11, 2014).

[22] Rastislav Tisovčík. *Generation and Visualization of Terrain in Virtual Environment*. Bachelor Thesis. 2012. URL: http://is.muni.cz/th/359691/fi_b/.

[23] Dana Tomlin. *Geographic information systems and cartographic modeling*. Englewood Cliffs, New Jersey, USA: Prentice Hall, 1990. ISBN: 9780133509274.

[24] Bohuslav Veverka. "Křovák's projection and its use for the Czech Republic and the Slovak Republic". In: *50 years of the Research Institute of Geodesy, Topography and Cartography*. Zdiby, Prague, Czech Republic: Research Institute of Geodesy, Topography and Cartography, 2005, pp. 173–179. URL: http://www.vugtk.cz/odis/sborniky/sb2005/.

[25] Geoportal ČÚZK. *Digital Surface Model of the Czech Republic of the 1st generation (DMP 1G)*. Mar. 2014. URL: http://geoportal.cuzk.cz/(S(bqcw1e551ngkrw55tbd5vy55))/Default.aspx?mode=TextMeta&side=vyskopis&metadataID=CZ-CUZK-DMP1G-V&head_tab=sekce-02-gp&menu=303 (visited on May 3, 2014).

[26] Geoportal ČÚZK. *Digital Terrain Model of the Czech Republic of the 4th generation (DMR 4G)*. Mar. 2014. URL: http://geoportal.cuzk.cz/(S(bqcw1e551ngkrw55tbd5vy55))/Default.aspx?mode=TextMeta&side=vyskopis&metadataID=CZ-CUZK-DMR4G-V&head_tab=sekce-02-gp&menu=301 (visited on May 3, 2014).

[27] Geoportal ČÚZK. *Digital Terrain Model of the Czech Republic of the 5th generation (DMR 5G)*. Mar. 2014. URL: http://geoportal.cuzk.cz/(S(bqcw1e551ngkrw55tbd5vy55))/Default.aspx?mode=TextMeta&side=vyskopis&metadataID=CZ-CUZK-DMR5G-V&head_tab=sekce-02-gp&menu=302 (visited on May 3, 2014).

[28] Geoportal ČÚZK. *Fundamental Base of Geographic Data of the Czech Republic (ZABAGED®) - altimetry - grid 10x10 m*. Jan. 2014. URL: http://geoportal.cuzk.cz/(S(f0l5nn55iobeovqde200ioyh))/Default.aspx?mode=TextMeta&side=vyskopis&

metadataID=CZ-CUZK-ZABAGED-VG&head_tab=sekce-02-gp&menu=305 (visited on Apr. 28, 2014).

[29] Terminologická komise ČÚZK. *Terminologický slovník zeměměřictví a katastru nemovitostí*. URL: https://www.vugtk.cz/slovnik/4106_vyskovy-system-baltsky---po-vyrovnani-(bpv) (visited on May 19, 2014).

A Pre-processing of point cloud

This appendix describes a sequence of steps which can be used to create a triangulated mesh from a point cloud provided by ČÚZK. The following software is used across this guide:

- *XYZ preparation utility*, created and provided as a part of this thesis.
- *MeshLab v1.3.3 (64-bit)*[1]
- *Blender 2.68a (64-bit)*[2]

The transformation process will be demonstrated on the point cloud file `pp_HLIN04_1g.xyz`, available at [25].

A.1 Point cloud preparation

1. Move the point cloud file `pp_HLIN04_1g.xyz` to a folder which contains the *XYZ preparation utility* executable.
2. Run the *XYZ preparation utility* executable and input the name of the point cloud file (without its extension): `pp_HLIN04_1g`.
3. The application will automatically quit once it completes the computation.

The folder now contains two new files – `pp_HLIN04_1g.obj` and `pp_HLIN04_1g.area`. The `*.obj` file contains the transformed point cloud in OBJ file format – the Y and Z axes are swapped and all vertices have been translated so that their X and Z coordinates lie in the interval $[0, 2500]$ and $[0, 2000]$, respectively.

The small `*.area` file contains four values which specify a geographical area which is covered by the point cloud data. The values represent borders of a rectangular, axis-aligned area using S-JTSK coordinates.

A.2 Surface reconstruction

1. Open *MeshLab* and load the OBJ file `pp_HLIN04_1g.obj`.

   Menu: *File -> Import Mesh...*

1. Available at http://meshlab.sourceforge.net/
2. Available at http://www.blender.org/

2. Compute the normal vectors using the parameters shown in figure A.1 and the following table:

   Menu: *Filters -> Point Set -> Compute normals for point sets*

Neighbour num	`20`
Smooth Iteration	`0`
Flip normals w.r.t. viewpoint	`Yes`
Viewpoint Pos.	`1250 5000 1000`
Get	`Camera Pos`

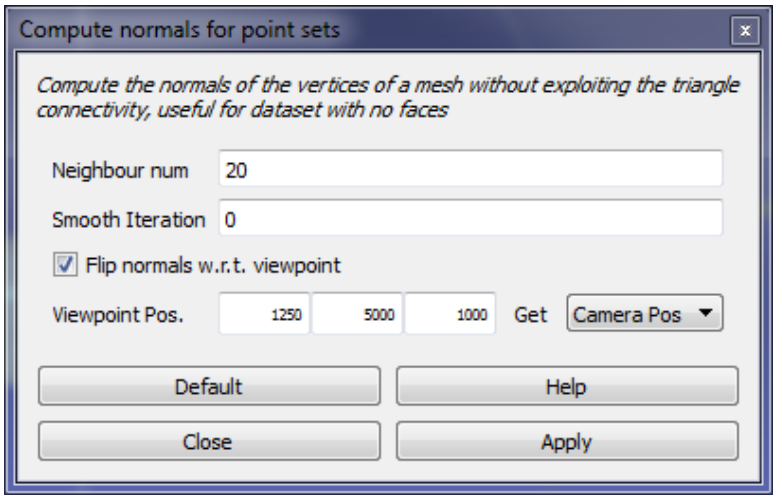


Figure A.1: Parameters used for normal computation.

3. Create a mesh from the vertices and normals using the Poisson surface reconstruction algorithm. Please note that this step is highly performance demanding. In our experience, a desktop PC with *at least* 4 GB of RAM is recommended. Moreover, multiple CPU cores can speed up the final computation time. Recommended parameters can be seen in the following table and figure A.2.

   Menu: *Filters -> Point Set -> Surface Reconstruction: Poisson*

Octree Depth	`13`
Solver Divide	`8`
Samples per Node	`1`
Surface offsetting	`1`

   The *octree depth* is the most important parameter regarding level of detail of the final mesh. Higher values produce mesh which contains

more vertices and details – and which takes far more time to compute. The following table presents results for a given *octree depth* on a desktop PC with Intel Core i5-2500K and 4 GB of system memory.

Octree Depth	# of vertices	# of faces	Time
6	2,867	5,686	1.142 s
7	10,550	21,050	1.933 s
8	40,411	80,774	4.584 s
9	162,038	324,016	17.796 s
10	452,119	904,146	50.798 s
11	724,376	1,448,514	85.146 s
12	787,395	1,574,354	93.813 s
13	859,252	1,717,666	124.437 s

Note: The source point cloud consists of 884,726 vertices.

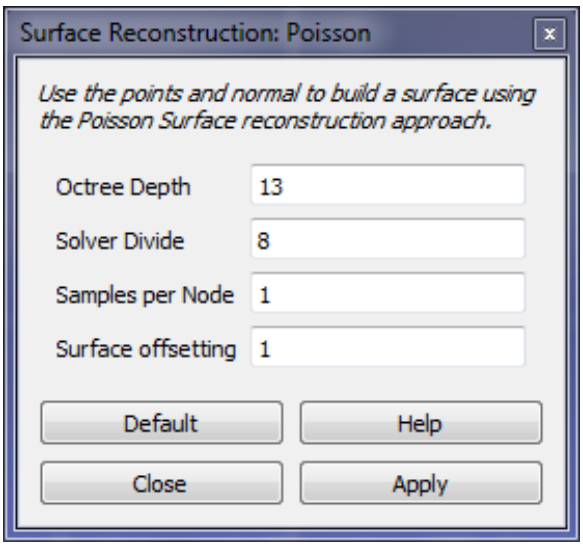


Figure A.2: Parameters used for surface reconstruction.

4. Open the *Layer dialog*.

   Menu: *View -> Show Layer Dialog*

5. Select the newly created mesh (should be labeled as "Poisson Mesh"), and export it as an OBJ file. In our case, we will overwrite the input file `pp_HLIN04_1g.obj`.

   Menu: *File -> Export Mesh As...*

6. Close *Meshlab*.

A.3 Removal of excess parts

The goal of this last step is to remove excess parts of the mesh which were created by the Poisson surface reconstruction algorithm. Unfortunately, the decision which vertices should be removed is not automated and so the removal process has to be done manually.

1. Open *Blender* and create a new scene.

   Menu: *File -> New...*

2. Remove *Camera*, *Cube* and *Lamp* objects from the scene.

3. Import `pp_HLIN04_1g.obj` to the scene, using options shown in figure A.3.

   Menu: *File -> Import -> Wavefront (.obj)*

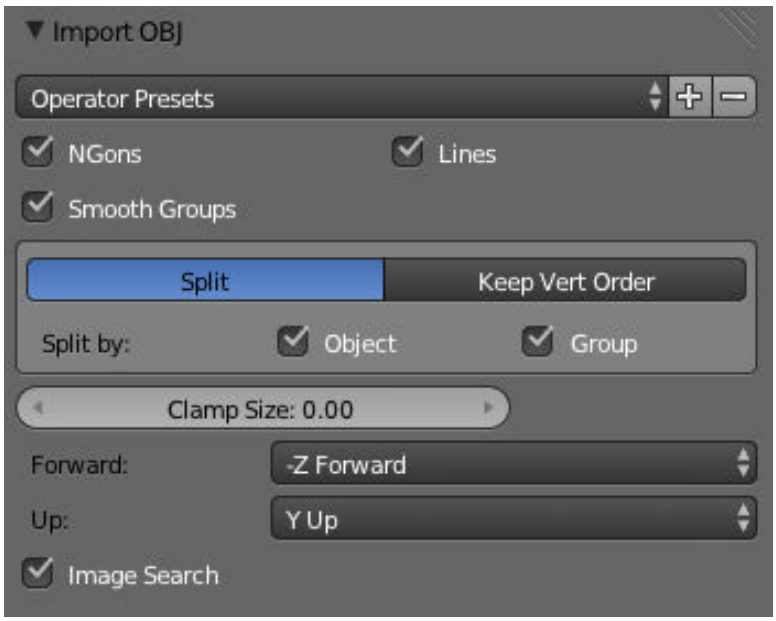


Figure A.3: Options used to import OBJ in Blender.

4. With mouse cursor above 3D view and press the **N** key. In the *View* section, set distance to the far clipping plane (*End* parameter) to 10,000 units.

5. With mouse cursor above the 3D view, press the **Z** key to switch rendering mode to wireframe.

6. Select the imported mesh and switch from *Object Mode* to *Edit Mode*.

7. Remove all vertices which you deem not appropriate to be part of the final TIN. For example – all vertices which have X and Y coordinates outside of the range $[0, 2500]$ and $[0, 2000]$, respectively, can be removed.

8. With all excess vertices removed, return back to *Object Mode*.

9. Select the mesh and export it to OBJ using options shown in figure A.4. In our case, we will overwrite the input file.

   Menu: *File -> Export -> Wavefront (.obj)*

10. Close *Blender*.

To complete the whole procedure, move the `*.obj` (exported from *Blender*) and `*.area` (created in appendix A.1) files to the target directory, e.g. `bin/Data/Nature/Land` in the *vrecko* installation directory.

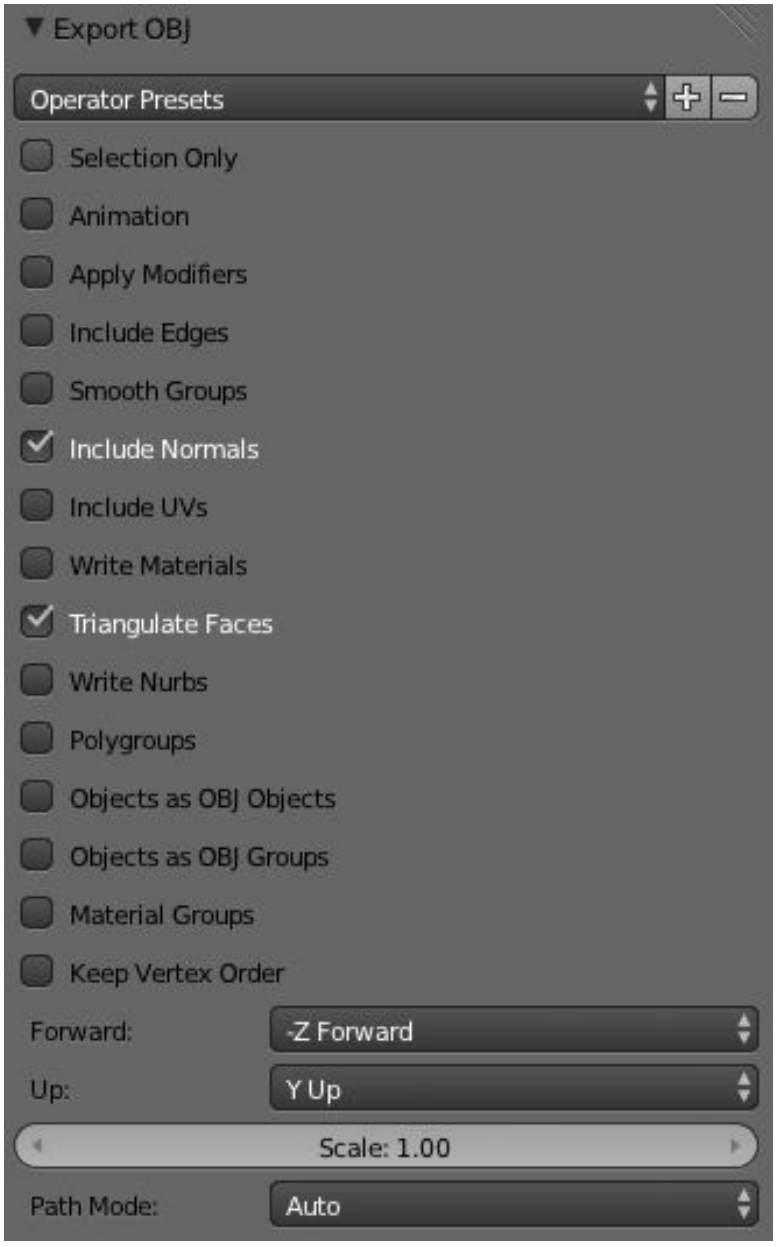


Figure A.4: Options used to export OBJ from Blender.

B XML Properties

This appendix describes the XML elements which are used to initialize the *Land* ability. Majority of properties are required and their omission results in an initialization failure. If a property has a pre-defined default value, it is not required and can be omitted.

B.1 Global properties

All global properties are placed directly inside the `Parameters` element of the *Land* ability.

- `TextureSize` – Size of the texture image which is downloaded from the Internet and stored in cache. Has to be $2^n, n \in 1...10$. Default value is **1024**.
- `TextureCachePath` – Location of the texture cache on the secondary storage (see section 5.5.2). Both relative and absolute paths are supported.
  Default value is **"data/Nature/Land/Cache"**.
- `UseWireframe` – Empty element which turns on wireframe rendering if it's found. Default value is **false**.
- `ElevationLegend` – Defines colouring of the vertices (see section 5.4.5 for detailed information) using any non-zero number of `Colour` elements. Each `Colour` element specifies a single *(elevation, colour)* pair using two values – `Height` and `Value`. Usage of default value results in a single colour (white) being applied to all vertices.
  - `Height` – Elevation at which the colour value is applied. Serves as a map key and its value has to be unique.
  - `Value` – Colour of the vertex at the given height. Specified by 3 integer values from the $[0, 255]$ interval, each representing one of the RGB channels.

B.2 Land block properties

This section provides textual description of all parameters which can be set **by user** inside the `<Block>` element in the input XML file. The description

is followed by a table which summarizes metadata of all properties.

B.2.1 Description

- `DataSource` – Defines the source of terrain data and specifies which subset of parameters is loaded from the input XML file. Only one of the following values can be used:
  - `ZabagedGrid` (ZG)
  - `ZabagedTin` (ZT)
  - `RasterImage` (RI)
  - `PerlinNoise` (PN)
  - `MidpointDisplacement` (MD)
  - `RandomFaults` (RF)

  Acronym in parentheses is used as a reference in further text.
- `Position` – Location of the land block in the scene.

  Used by all land blocks, not related to any data source.
- `Scale` – Scale of the land block.

  Used by all land blocks, not related to any data source.
- `Render` – Optional parameter which disables construction of `LandMesh` and its rendering. This way, the ability may be used as a relatively simple importer of terrain data which can be then used for other purposes. The parameter accepts all following values, with their meaning being obvious – `on`, `off`, `true`, `false`, `yes`, `no`, `1` and `0`.
- `StretchInterval` – Specifies value range in which the elevation samples are linearly stretched. Useful e.g. for perlin noise which produces very small values. The stretch operation is performed iff the first value is smaller than the second one.

  Used by all land blocks, not related to any data source.
- `ExportMeshPath` – Specifies path to file in which the block's mesh is exported. Supports both relative and absolute path. If the path is not valid, no mesh is exported.

  Used by all land blocks, not related to any data source.

- `ExportImagePath` – Specifies path to RGBA image file in which the terrain data (stored as a height field) is exported. All elevation values are stretched to the interval $[0, 255]$ and stored in RGB channels with alpha values set to 255. Invalid points are saved with zero value in all 4 channels. If the path is not valid, no image file is exported.

  Used by ZG, RI, PN, MD and RF.

- `MeshSegmentSize` – Size of height field segment. Represents number of height samples on X and Z axes. Smaller values increase quality but decrease overall performance.

  Used by ZG, RI, PN, MD and RF.

- `Width` – Size of the height field (number of samples) on the X axis.

  Used by PN, MD and RF.

- `Length` – Size of the height field (number of samples) on the Z axis.

  Used by PN, MD and RF.

- `FilePath` – Path to the input file which stores the terrain data. Both absolute and relative path is supported.

  Used by ZG and ZT, though in a slightly different way. In case of ZG, the input file is an altimetry grid provided by ČÚZK, stored in a text file with `*.txt` extension. In case of ZT, the parameter contains path to an OBJ file created from the digital surface model using sequence of steps described in appendix A. Moreover, an area file with the same name but a different extension (`*.area` instead of `*.obj`) is also loaded from the same location.

- `SampleDistance` – Distance between two 4-adjacent samples (in meters).

  Used only by ZG.

- `TextureSource` – Type of the texture which is used on the created mesh. Only one of the following values can be used:

  - `OrthoPhoto`
  - `Noise-Day`
  - `Noise-Night`

  Used by ZG and ZT.

- `TextureQuality` – Specifies quality (size) of the texture which is applied to rendered TIN mesh. There are four available options - each of them representing a texture with different dimensions:

`Low`	1024^2
`Medium`	2048^2
`High`	4096^2
`Ultra`	8192^2

  Support for large textures varies among different GPUs. However, modern desktop GPUs should safely handle textures up to 4096^2.

  Used only by ZT.

- `ImagePath` – File path to image which is used as a terrain data source. Both absolute and relative paths are supported.

  Used only by RI.

- `ColourFunction` – Specifies function which is used to compute the height h of each sample from its input RGBA colour value c. The following options are available:

`R`	$h = c_r$
`G`	$h = c_g$
`B`	$h = c_b$
`Sum`	$h = c_r + c_g + c_b$
`Average`	$h = \frac{1}{3} \cdot (c_r + c_g + c_b)$

  Used only by RI.

- `RandomSeed` – Seed value used to initialize the pseudo random number generator using the `std::srand()` function.

  Used by PN, MD and RF.

- `OctaveCount` – Number of octaves in Perlin noise algorithm that are summed up together to the composite noise. See [22] for details.

  Used only by PN.

- `Persistence` – Persistence of the Perlin noise. See [22] for details.

  Used only by PN.

- `Interpolation` – Type of interpolation which is used in the Perlin noise algorithm. Only two values are allowed – `linear` and `cosine`. See [22] for details.

  Used only by PN.

- `FractalDimension` – Fractal dimension of the land. See [22] for detailed description.

  Used only by MD.

- `CornerValueRange` – Value range from which the initial value of each surface corner is pseudo randomly generated. See [22] for details.

  Used only by MD.

- `Iterations` – Number of iterations computed by random faults algorithm. See [22] for details.

  Used only by RF.

- `FaultValueRange` – Value range from which the initial fault value is pseudo randomly generated. See [22] for details.

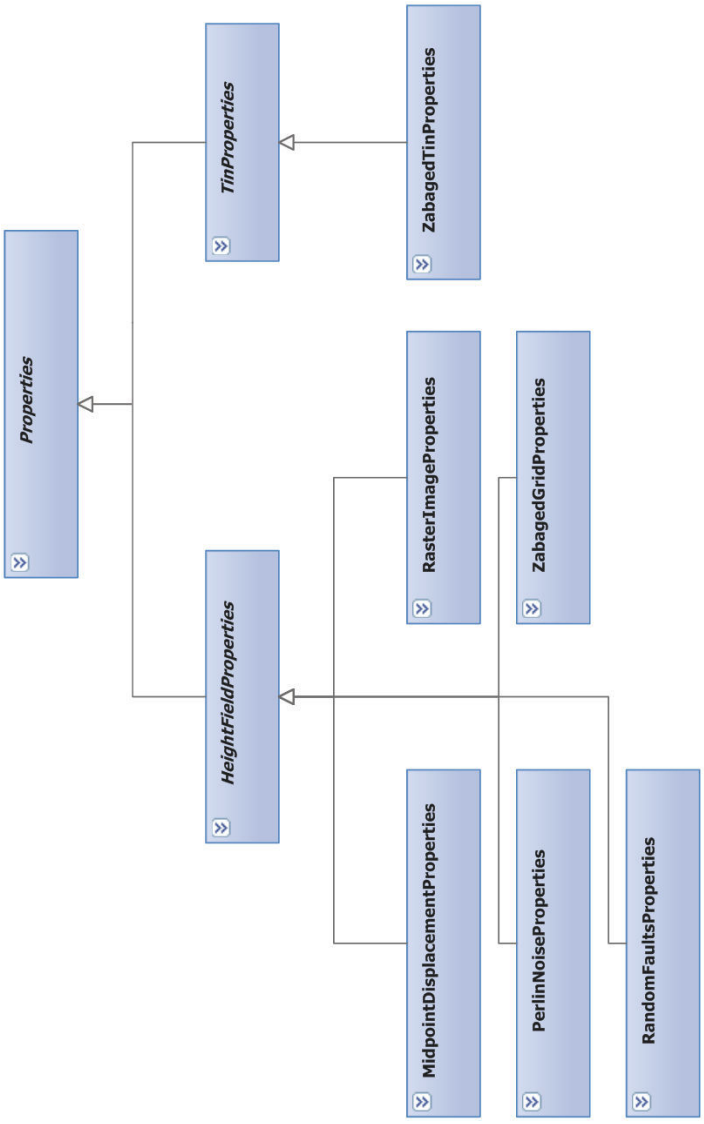
  Used only by RF.

B.2.2 Metadata

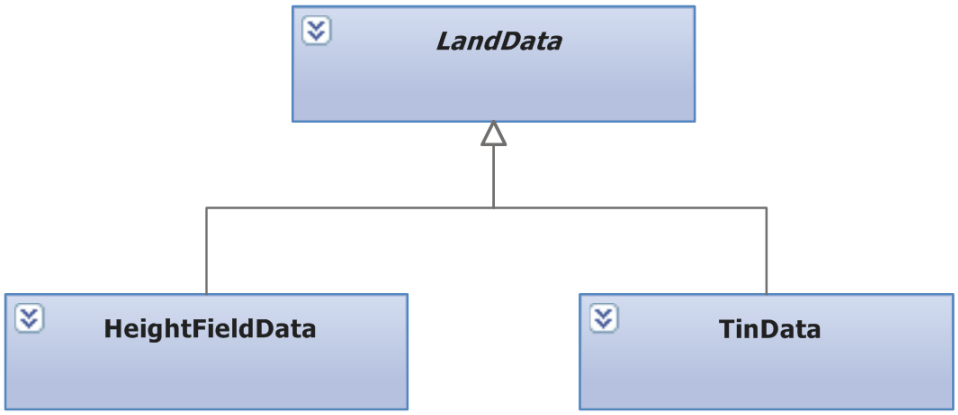
Name	Type	Default value	Allowed values
DataSource	Land::DataSource	—	See B.2.1
Position	osg::Vec3f	0 0 0	$x, y, z \in \mathbb{R}$
Scale	osg::Vec3f	1 1 1	$x, y, z \in \mathbb{R} - \{0\}$
Render	bool	true	See B.2.1
StretchInterval	osg::Vec2f	0 0	$x, y \in \mathbb{R}$
ExportMeshPath	std::string	*(empty)*	path to file
ExportImagePath	std::string	*(empty)*	path to file
MeshSegmentSize	int	32	$x \geq 32$
Width	int	—	$x \geq 2$
Length	int	—	$x \geq 2$
FilePath	std::string	—	path to file
SampleDistance	float	—	$x > 0$
TextureSource	Land::TextureSource	—	See B.2.1
TextureQuality	Land::TextureQuality	—	See B.2.1
ImagePath	std::string	—	path to file
ColourFunction	Land::ColourFunction	—	See B.2.1
RandomSeed	unsigned int	current time	$x \in \mathbb{N}_0$
OctaveCount	int	—	$x \geq 1$
Persistence	float	0.5	$x \in (0.0, 1.0)$
Interpolation	Land::InterpolationType	cosine	See B.2.1
FractalDimension	float	—	$x \in [2.0, 3.0]$
CornerValueRange	osg::Vec2f	—	$x \leq y$
Iterations	int	—	$x \geq 0$
FaultValueRange	osg::Vec2f	—	$0 \leq x \leq y$

C Class diagrams

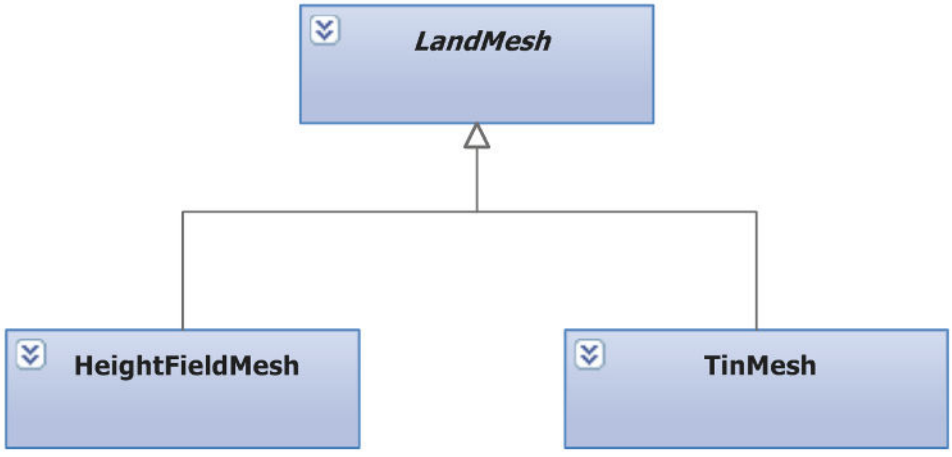
C.1 Properties



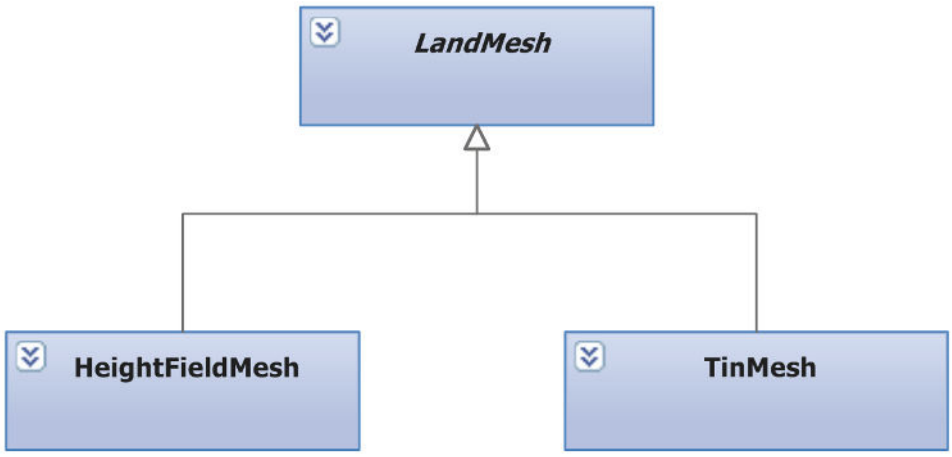
C.2 LandData



C.3 LandMesh



C.4 HeightFieldProvider



D Performance tests

D.1 Hardware configurations

	Config. 1	Config. 2	Config. 3
CPU	Intel Core i5-2500K	Intel Core i5-2500K	Intel Core i5-2430M
	3.30 GHz	3.30 GHz	2.40 GHz
RAM	4 GB	4 GB	4 GB
GPU	AMD HD 6870	AMD HD 6870	AMD HD 6630M
Storage	WD Caviar Black	Intel SSD 330	Hitachi Travelstar Z7K320
	3.5″, 7200 rpm	2.5″	2.5″, 7200 rpm
OS	Win 7 (64-bit)	Win 7 (64-bit)	Win 7 (64-bit)

D.2 Test data

	File size	# of vertices	# of normals	# of faces
ZACL51_5g	327,716 KB	1,689,250	3,370,171	3,373,882
ZACL61_5g	476,860 KB	2,434,611	4,857,706	4,864,478
ZACL62_5g	408,272 KB	2,096,634	4,184,889	4,188,180
PARD80_1g	332,576 KB	1,726,101	3,403,796	3,448,649
pp_HLIN04_1g	159,224 KB	844,774	1,675,033	1,686,464

D.3 Results

	Config. 1	Config. 2	Config. 3
ZACL51_5g	22.596 s	22.055 s	28.365 s
ZACL61_5g	32.495 s	32.304 s	40.145 s
ZACL62_5g	28.063 s	27.963 s	34.527 s
PARD80_1g	23.940 s	22.508 s	28.241 s
pp_HLIN04_1g	11.091 s	10.914 s	13.848 s

Table D.1: Time required to load TIN from OBJ file.

E Thesis archive in IS MU

This appendix contains brief description of files that accompany this text in thesis archive in Masaryk University Information System.

- `Thesis.pdf` – Text of this thesis.
- `vrecko.zip` – Trimmed down version of *vrecko*. This release contains only executable *vreckoApp.exe* along with *Land* ability and content required by it. To simplify execution of the application, several example batch files are prepared in the subfolder

  `vrecko/Examples/Nature/Land`.
- `Source codes.zip` – All source codes and projects related to the *Land* ability, *XYZ preparation utility* and text of the thesis written in LaTeX.

  Note: Source codes of the *Land* ability can only be compiled along with the rest of the *vrecko* framework. See the project's homepage[1] for instructions on how to download and compile the framework.
- `Diagrams.zip` – Class diagrams of the *Land* ability, created in Microsoft Visual Studio 2010.

1. http://vrecko.cz/