Viewshed algorithms for strategic positioning of vehicles

by

Martin Vonheim Larsen

THESIS
for the degree of

MASTER OF SCIENCE

(Master i Anvendt Matematikk)

Faculty of Mathematics and Natural Sciences
University of Oslo

May 2015

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Acknowledgements

First and foremost a special thanks to my supervisor Solveig Bruvoll for her advice and guidance throughout this thesis. I learned a lot from our discussions, and was always inspired by her questions to do better. I am grateful to FFI for lending me Solveig, without her direction this thesis would merely be an unordered set of unfocused ideas. Thanks also to my second supervisor, Martin Reimers.

Thanks to Vegard Kvernelv for his input on my work, especially with the finishing touches. Also thanks to Geir Karlsen for teaching the basics of military strategy to a civilian student. I must also mention Edward Murr and Luftforsvaret, thanks to whom I already had security clearance and could start working on the thesis right away.

I have been fortunate to be part of a large group of friends who happen to be study fellows. There are too many to mention, but you know who you are.

Most of all I must thank my parents and sister for always supporting me, and Mari for motivating me and keeping my spirits up through the rollercoaster that is a master's degree. Thank you.

Contents

Chapter 1

Introduction

Simulation and autonomous planning has become an important part of military preparations. Using modern computers we can now produce simulations with realism that simply was not possible just a few years ago. We can also create tools that help strategists making plans of unprecedented detail and quality.

In this thesis we will use a problem from military autonomous planning as motivation for looking into visibility analysis. We consider the scenario of a platoon of vehicles attacking some hostile group of vehicles. Simulating such a scenario poses several challenges. How should the platoon position themselves in order to have the best chance of defeating their enemy? Where do the hostiles position themselves in the first place, and how do they react to the attack? These are just a few issues that must be carefully handled in order for the simulation to have any value for real world use.

Studying the scenario further we find that visibility analysis, specifically viewshed calculations, is essential to any reasonable procedure attempting to solve these challenges. With the military application in mind we consider existing theory developed by Franklin et al., de Floriani et al., Cole et al., Ben-Moshe et al. and Izraelewitz et al. Building on ideas originally proposed by Franklin et al. we establish a robust procedure for comparing viewshed algorithms for specific applications. Using this scheme we compare the performance of existing algorithms in conditions typically encountered when used as part of a military planning system.

Based on these comparisons we find that the R2 algorithm due to Ray et al. is the best fit for our application, and use it as a starting point for further improvements. Analyzing some unsatisfactory performance on certain terrain types, we discover two simple modifications to the original R2 algorithm. One of the modifications reduce the error by 50%, whithout siginificantly affecting the running time. The other modification reduces the error by 75%, while only increasing the running time with 30%.

In the quest for even more accurate approximations in a more flexible algorithm that can deliver high precision on demand in exchange for increased running times, we further analyze the error committed by our improved R2 algorithm. Based on the results of this analysis we develop a multi-pass hybrid algorithm that exploits patterns in the error for predicting where the extra evaluation cycles are best spent. We show that the resulting algorithm is capable of calculating viewsheds with more than *three orders of magnitude* the accuracy of the original R2 algorithm, using barely ten times the running time. Combined with the tunability, this level of performance makes the algorithm fill the gap between approximate and accurate algorithms.

Contributions

In chapter 3 we formalize the framework used Franklin et al. in [FRM94]. We prove the correctness of the R3 algorithm in theorem 3.1 and corollary 3.4. Additionally we formalize the steps in the development of the R2 algorithm in corollary 3.6 and corollary 3.7, clearly separating the accurate and approximate parts of the algorithm.

We greatly extend some ideas proposed in [FRM94], making a robust testing framework for viewshed algorithms in chapter 4. This includes an automated procedure for selecting test observation positions, in addition to robust statistic procedures for quantifying the relative performance of algorithms. Although other authors have put some thought into this, proper testing procedures are absent in most of the existing literature on the subject.

In chapter 5 we propose two improvements to the R2 algorithm from [FRM94]. First we develop a variant of R2, which greatly reduces the error with only minor increases in running time. Secondly we propose an efficient tunable algorithm based on the improved version of R2, capable of reducing the error on demand in exchange for prolonged running times.

Chapter 2

Finding good strategic positions

Our primary motivation for looking into visibility calculations in this thesis can be found in the general scenario where a platoon is attacking an enemy group of vehicles. The overall goal is to develop some automatic procedure for planning the whole maneuver, so it can be used for simulations and in real world scenarios. This problem is, however, rather complex. For each enemy unit the procedure should carefully select attacking positions for each friendly unit, maximizing the likelihood of incapacitating the enemy without casualties. These positions depend on many factors, such as distance, relative elevation, terrain type and accessibility, just to name a few. With multiple enemies, the procedure must also find the optimal order in which to attack each unit. This order in turn affects which attack positions are optimal. The enemy is assumed to react to attack, which means that the plan must dynamically be updated as the scenario unfolds. Additionally the procedure must take the starting position of the friendly units, and how fast they move into account.

Instead of tackling this problem head-on, we will turn our focus to the sub-problem of finding good attack positions against a *single* target. At this point it is natural to also consider the problem of finding good positions for *observing* some target, as these problems are very similar. As we shall see, visibility calculations are essential to solving these problems. But first, we need some military background on the matter.

2.1 The anatomy of an attack

The basic principles of an attack in land-based warfare are the same for most types of units on the battlefield, be it infantry or vehicles.

Suppose a blue unit is mounting an attack on a red unit. An attack is typically considered to consist of three separate positions. Initially blue should be completely covered, i.e. invisible and not in danger of being attacked by red. Before being able to attack, blue must move to a position where he has visual contact in order to lock his target systems onto red. Finally, when the target systems are set, blue must move to a position from which he can fire upon red. Typically these three positions are referred to as the cover-, observation- and attack position, respectively.

The distinction between the two latter positions becomes apparent in the case of main battle tanks. For this type of vehicle, we model projectile trajectories as straight lines, although this is not entirely accurate. Assume that red is initially hidden behind a hill, and that blue is driving up this hill. This is indicated by point 1 in fig. 2.1a. At some point as blue moves up the hill, his sights, which sit high on the vehicle, will have an uninterrupted line of sight (LOS) to red. From this position blue engages the targeting systems, and prepares to fire upon red. This is point 2 in the figure. Finally, as soon as the targeting systems are ready, blue moves forward until there is an uninterrupted LOS from the barrel to a critical point on red. This is the position blue should fire from, illustrated by point 3 in the figure. On most terrains there are significantly fewer attacking positions than there are observation positions. A typical real world example can be seen in fig. 2.1b, which clearly shows that the set of attack positions is smaller than that of observation positions.

Essential to the effectiveness of the attack is how quickly blue can move from cover to observation and attack. Blue is vulnerable to attack as soon as he leaves cover, so it is important the attack position is easily accessible, while maintaining all escape options available. Naturally a swift attack means that blue can catch red by surprise, increasing the overall likelihood of success.

In the observation- and attacking positions, blue must of course accept to be vulnerable to attack from red. A good attacking position should, however, limit blue's exposure to the remainder of the terrain, where other enemies potentially might be hiding. Therefore these positions should ideally be selected in such a way that they primarily have a view in the direction of red, or areas controlled by blue units.

2.2 Observation tactics

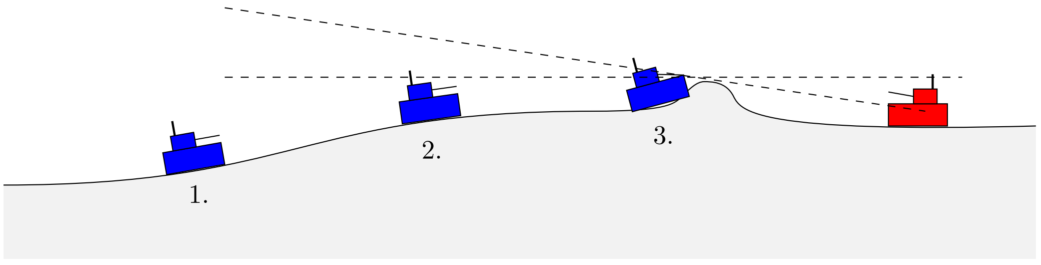
The tactics involved in observing a point of interest are by and large the same as for attack, except that we never move into attack position. If blue is to observe some point, he wants to be able to move quickly from a position of complete cover to the observation position where he has an uninterrupted LOS to the target. As for attacks it is beneficial if blue is visible only from a small region, as this reduces the risk of being spotted.

2.3 Relevant visibility calculations

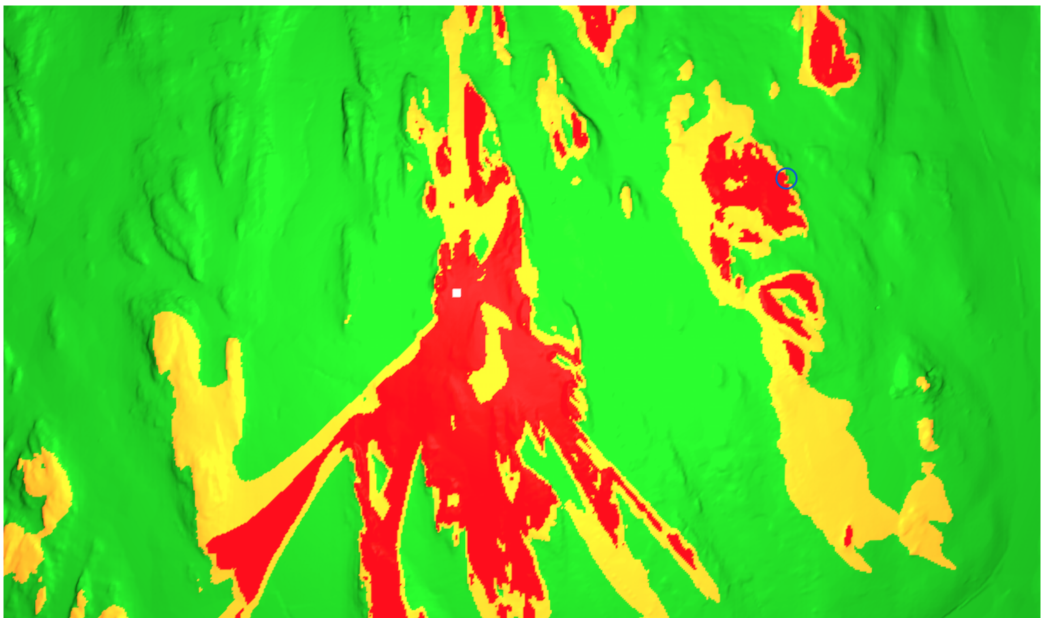
By now it should be clear that performing some type of visibility calculations will be essential in identifying good candidates for attack- and observation positions. Specifically we need to find the regions of the terrain from which there is an uninterrupted LOS to some known target point. The reverse of this problem is to find the *viewshed* of the target, i.e. finding the points that can be seen *from* the target. In practice it is not necessarily the case that red can see blue even though blue can see red. For instance, red might not be able to spot blue if he is hiding in a tree line. In this thesis however, we will assume that a LOS always can be used in both directions. This means that finding the viewshed of a target point, is the same as finding the set of points from which the target point is visible.

Say that blue's sights are situated at a height above the ground, and that the highest visible point of red is at b height above the ground. This means that we are really interested in points $\mathbf{x}$, such that there is an uninterrupted LOS from the point that is a above $\mathbf{x}$ to the point that is b above red's position. Similarly for attack, we follow the same procedure, but use the height of blue's cannon and some critical point of red for a and b. This is a generalization of the viewshed, where the *observer* height, in this case b, and the *target* height, in this case a, are extra parameters.

The higher the target and/or observer height, the larger the viewshed. As a result, there typically are fewer potential attack points than observation points. This can be seen in fig. 2.1b which shows both the attack- and observation viewsheds in a typical situation. As can be seen in the figure the



(a) A blue vehicle attacking red. Positions 1-3 indicate the cover-, observation- and attacking positions, respectively.



(b) A terrain with imposed viewsheds. Red is positioned in the small white square. Points from which blue can observe and attack red are colored yellow and red, respectively. The blue circle highlights a promising region.

Figure 2.1

attack viewshed is contained inside the observation viewshed due to lower observer- and target heights.

2.4 Viewshed boundary points

When it comes to finding *good* attack and observation positions, we are particularly interested in the *boundaries* of the relevant viewsheds. Typically the best observation positions lie only barely inside the observation viewshed. Outside this viewshed blue cannot be seen by red, and therefore has cover. Blue can observe red regardless of how deep he is inside the viewshed, so staying close to the boundary means that he has a short route to cover.

The principle is the same in an attack scenario, but here blue also needs to have a short route to cover when in the attack position. This implies that the observation- and attack positions should be chosen in a region where the boundaries of the observation- and attack viewshed are close. Such a position is highlighted by the blue circle in fig. 2.1b. Here blue can stay in cover before moving just inside the observation viewshed to prepare his targeting systems. When the systems are set he can move quickly inside the attack viewshed and fire. If something unexpected is to occur blue can at any time abort and retreat quickly back into cover.

2.5 Summary

There are many important factors that affect the quality of observation- and attack positions that we have not discussed. We know, however, that accessibility, vegetation, distance and relative elevation to the target are just a few other important factors that can render even the best candidate points useless. This is, however, beyond the scope of this thesis. Regardless, it is clear that visibility calculations, especially viewshed calculations, are at the core of any system capable of finding realistic strategic positions. This is our motivation to investigate viewshed algorithms further, and how they should be used for our application.

The goal is to use this in an even bigger system, capable of planning how an entire how an entire platoon should attack a group of targets. This is a typical two-player scenario, which might have to be explored using some variant of the minimax algorithm. If the targets are mobile, which they typically are, they will respond to the actions undertaken by the platoon. This means that each new state in the state tree requires updated knowledge about new viewsheds. It is therefore likely that such an approach will require the calculation of a large number of viewsheds.

It turns out that a brute-force approach to this is not feasible, as it is simply too slow. On terrain data sets of realistic size the running time of a brute force solution is on the order of minutes, which is not practical if it is to be used in a procedure which requires the calculation of thousands of viewsheds.

There are a few practical considerations that might cause approximate viewsheds to be acceptable. First of all, the terrain data are not an accurate depiction of the real world. This means that even using completely accurate viewshed algorithms, we might come up with an attack position from which the enemy in reality is not even visible, making it impossible to attack. This can potentially be dangerous for the unit performing the attack, since it might end up in a compromising position without any opportunity to fight back. As long as the terrain data are approximate, there is no way to fully prevent such errors. Secondly, it is generally not essential that the viewshed boundary is 100% accurate, as long as it is within a few meters. If this is the case we might still come up with impossible attack positions, but there is like to be a viable position within few meters. In the case of fig. 2.1a this corresponds to simply moving a few more meters up the hill. This means that *good* approximations can be used, without any larger risk than accurate algorithms. It is, however, clear that the quality of the resulting positions will be benefit from being based on viewsheds with low error. Therefore, there is reason to have as accurate approximations as possible within the available time frame.

Chapter 3

Overview of visibility calculations

For solving our military scenario we have seen that we will need to perform some visibility calculations, specifically *viewshed* calculations. Before we can do that, we need a theoretical framework which allows us to represent a terrain and precisely define what we mean by *visibility*.

We will build a general framework, which gives us the tools we need for proving properties of the algorithms we will consider. Our definition of visibility has the intuitive property that it is symmetric. That is, if a point a is visible from a point b, then b must also be visible from a. Building on this we will also define the *viewshed*, which is the set of points, called *targets*, that are visible from a given point, called the *observer*.

In order to use these definitions in practice, we need an efficient way to represent terrain. We will therefore consider several terrain models, and discuss how they fit our application. We will primarily focus on regular square grids (RSGs) and triangulated irregular networks (TINs), which are two commonly used terrain models.

Finally we will consider several algorithms for calculating viewsheds on these terrain models. We start by studying the brute force algorithm described in [FRM94]. Then we consider several approximate algorithms for both for RSGs and TINs, among others the *R2* algorithm due to Franklin et al., which will be the focus of chapter 5. Empirical tests of these algorithms follow in chapter 4.

3.1 Definition of visibility

We assume that the terrain at hand can be represented as a subset of $\mathbb{R}^3$. This is a definition that is general enough to contain all the various terrain representations we will consider, yet specific enough to be meaningful in our context.

We will now define the notion of *visibility* and what we mean by the term *viewshed*.

Definition 3.1.

Let $U \subset \mathbb{R}^3$ be some terrain, and $\mathbf{u} \in \mathbb{R}^3$ be a point.

*We say that two points $\mathbf{u}$ and $\mathbf{v}$ are **intervisible** wrt. U iff. the interior of the line segment between them does not contain any points from U:*

$$\{\lambda\mathbf{u} + (1-\lambda)\mathbf{v} | \lambda \in (0,1)\} \cap U = \emptyset$$

*The **viewshed** of U wrt. $\mathbf{u}$, $V(\mathbf{u})$, is given by the set of points in U that are intervisible to $\mathbf{u}$.*

The only candidates for visible points in a terrain are the ones on the terrain surface. We show that this is also the case for our formal definition.

Corollary 3.1.

Let $U \subset \mathbb{R}^3$ be some terrain, and $\mathbf{u} \in \mathbb{R}^3$ be a point.

All viewshed points of U are boundary points of U.

Proof.

Let $\mathbf{v} \in V(\mathbf{u})$, and let ∂U denote the boundary points of U.

Since $\mathbf{v} \in U$ then obviously $B(\mathbf{v}, \epsilon) \cap U \neq \emptyset$ for any $\epsilon > 0$.

Furthermore we have that:

$$\begin{array}{cc} \lambda\mathbf{u} + (1-\lambda)\mathbf{v} \in U^{\mathrm{C}} & \forall\, \lambda \in (0,1) \\ \Downarrow & \\ B(\mathbf{v}, \epsilon) \cap U^{\mathrm{C}} \neq \emptyset & \forall\, \epsilon > 0 \end{array}$$

Thus $\mathbf{v} \in \partial U$. ∎

Next, we define the *elevated viewshed* which is the viewshed we will be using in practice.

Definition 3.2.

Let $U \subset \mathbb{R}^3$ be some terrain, and denote the terrain surface by ∂U. Let $\mathbf{u} \in \mathbb{R}^3$ be a point, and $\mathbf{k}$ denote the unit vector in the vertical direction.

Given an observer height ψ and a target height ω. The ***elevated viewshed*** *of U wrt. $\mathbf{u}$, $V_{\psi,\omega}(\mathbf{u})$, is given by the points in $\mathbf{v} \in \partial U$ such that $\mathbf{u} + \psi\mathbf{k}$ and $\mathbf{v} + \omega\mathbf{k}$ are intervisible.*

For almost all practical applications we will be interested in the visibility of points that have some height above the ground. For this reason we always use the elevated viewshed, and not the basic viewshed. In the remainder of this thesis we will therefore refer to the elevated viewshed simply as the *viewshed*.

Next, we show that viewsheds are symmetric. That is, if a point $\mathbf{u}$ is in the viewshed of a point $\mathbf{v}$, then $\mathbf{v}$ must also be in the viewshed of $\mathbf{u}$.

Corollary 3.2 (Elevated viewshed symmetry)**.**

$\mathbf{v} \in V_{\psi,\omega}(\mathbf{u})$ *iff.* $\mathbf{u} \in V_{\omega,\psi}(\mathbf{v})$.

Proof. Let $\mathbf{v} \in \overline{V}_{\psi,\omega}(\mathbf{u})$. Then by definition, the line between $\mathbf{u}$ and $\mathbf{v}$ does not intersect U:

$$\{\lambda(\mathbf{u} + \psi\mathbf{k}) + (1-\lambda)(\mathbf{v} + \omega\mathbf{k}) \,|\, \lambda \in (0,1)\} \cap U = \emptyset$$

Let $\gamma = 1 - \lambda$. Then we have,

$$\begin{gathered}\{(1-\gamma)(\mathbf{u} + \psi\mathbf{k}) + \gamma(\mathbf{v} + \omega\mathbf{k}) \,|\, \gamma \in (0,1)\} \cap U = \emptyset \\ \Updownarrow \\ \{\lambda(\mathbf{v} + \omega\mathbf{k}) + (1-\lambda)(\mathbf{u} + \psi\mathbf{k}) \,|\, \lambda \in (0,1)\} \cap U = \emptyset\end{gathered}$$

By definition we now have that $\mathbf{u} \in \overline{V}_{\omega,\psi}(\mathbf{v})$. ■

This definition is flexible in that it allows us to represent a terrain as an infinite set of points. This would allow us to represent the world in infinite resolution as it *actually* is. However, this is of course not possible on a computer. This is where a *terrain model* comes into play, in that it provides us with an approximation of the world, based on a workable set of data points.

3.2 Terrain modeling

In order to work with terrain in a meaningful way on a computer, we need a model for representing it. Digital elevation models (DEM) are typically divided into two main categories; raster models and vector models. Raster models are the most intuitive and rely on elevation measurements at regular intervals that form a grid in the surface plane. Vector models operate on a higher level with mathematical objects such as lines, triangles or volumes. This can lead to a more efficient representation which better represents the features of a terrain than a simple raster model.

Typically, the source data for a terrain model is a set of elevation measurements with corresponding lateral coordinates. The purpose of the model is to provide an approximation of the *entire* terrain, based on these data.

In this thesis we will be using RSGs for our terrain model. The reasons for this are discussed throughout this chapter. But first we will take a closer look at how some of the models work, and discuss the benefits and draw-backs of the various models.

3.2.1 Triangulation

Triangulations are perhaps the most used model for representing objects in three dimensions. They consist of a set of vertices and a set of triplets, connecting the vertices in triangles. The resulting surface of triangles then represents the surface of the object. This extends trivially to terrains, where the set of data points can be used as vertices, and then the vertices can be grouped together e.g. using Delaunay triangulation. Typically, triangulations are referred to as triangulated irregular networks (TINs).

The resulting terrain surface matches the real world at the vertices, while it is only approximate elsewhere. That being said, triangulations make for an efficient way to represent good terrain approximations. In flat regions a triangle will approximate the surface very well. Thus by using a few large triangles in flat regions, and many small triangles in complex regions, we get more accuracy where it is actually needed. Triangulations also make for a flexible terrain model, in that they can represent complex structures that fold over, such as tunnels.

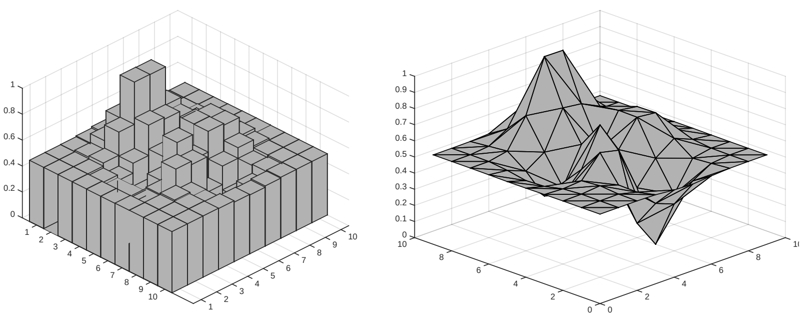
It is the case, however, that many of the viewshed algorithms we could consider using on TINs, such as the ones described in [CS89] and [FM94], actually work on *monotonic polyhedral surfaces*. Polyhedral surfaces are surfaces consisting of conjoined *flat* polygons. Thus TINs are polyhedral surfaces. *Monotonic* polyhedral surfaces, on the other hand, have the property that any vertical line must intersect the terrain in at most a single point. In other words, these algorithms do not allow the terrain tunnels or complex structures even though TINs technically can represent them.

3.2.2 Regular square grid

Assume that the terrain can be expressed as a function $f : [a, b] \times [c, d] \to \mathbb{R}$, where $a \leq b, c \leq d \in \mathbb{R}$. An intuitive way to model an approximation to this terrain is to sample the elevation at regular intervals that form squares in the $[a, b] \times [c, d]$-plane, and store the result in a two-dimensional array. This is the basis for the family of raster models known as RSGs. We shall refer to the points where the terrain is sampled as *grid points*. Four neighboring points and the space between make up what we shall call *grid cells*. The line between two adjacent grid points will be referred to as a *grid line*.

In their most basic form, RSGs do not provide any information about what the terrain looks like on the interior of the grid cells. Therefore they are typically accompanied by some interpolation scheme, in order to provide a well defined terrain surface. There are several interpolation methods that can be used for this, from crude piecewise constant interpolation to higher order interpolation with polynomial or spline basis. We will compare a few schemes and make the case why we will prefer a simple interpolation scheme to a more complex one.

In this section we will denote the elevation of the grid point (s_1, s_2) by e_{s_1,s_2}. The set of grid points will be denoted by S. To avoid confusion, we will denote the model approximated elevation in a point (x_1, x_2) by $e(x_1, x_2)$. In other words the function $e : [a, b] \times [c, d] \to \mathbb{R}$ is our model approximation of the real world terrain given by f.



(a) Nearest-neighbor (b) Triangulation

Figure 3.1

Piecewise constant interpolation

The simplest interpolation schemes are the ones based on piecewise constant interpolation. There are several ways to define them, but they all have in common that they assign the same elevation for neighborhoods in the terrain, and that the resulting model is discontinuous.

The simplest example of piecewise constant is nearest-neighbor interpolation. As the name suggests, each point in the model is assigned the same elevation as the nearest data point, that is,

$$e(x_1, x_2) = e_{\text{argmin}_{(s_1,s_2)\in S} ||(x_1,x_2)-(s_1,s_2)||}$$

Other similar piecewise constant schemes assign each point the maximum, minimum or average elevation of the four nearest data points.

The nearest-neighbor interpolation schemes provide for a simple implementation, but give a result that does not resemble the real world. Compare for instance nearest-neighbor interpolation to a triangulation of a simple terrain in fig. 3.1. Although they share the same data points, the triangulation does a much better job of representing something that we recognize as terrain. A key issue is that the piecewise constant model is discontinuous, while real world terrain is continuous, at least approximately.

Bilinear interpolation

In order to get a continuous model, we have to use a slightly more sophisticated interpolation scheme. Bilinear interpolation works by linearly interpolating a point from the four closest data in both dimensions; first along one axis on the two pairs of data points, and then between the two results along the other axis. Bilinear interpolation can be defined as follows:

Definition 3.3 (Bilinear interpolation)**.**

Given a point (x_1, x_2) *which is contained in the grid cell spanned by* (a_1, a_2) *and* (b_1, b_2) *(i.e.* $a_1 \leq x_1 \leq b_1$ *and* $a_2 \leq x_2 \leq b_2$*). Then the elevation at* (x_1, x_2) *is given by:*

$$\begin{aligned} e(x_1, x_2) = &\frac{b_1 - x_1}{b_1 - a_1}\left(\frac{b_2 - x_2}{b_2 - a_2} e_{a_1,a_2} + \frac{x_2 - a_2}{b_2 - a_2} e_{a_1,b_2}\right) \\ &+ \frac{x_1 - a_1}{b_1 - a_1}\left(\frac{b_2 - x_2}{b_2 - a_2} e_{b_1,a_2} + \frac{x_2 - a_2}{b_2 - a_2} e_{b_1,b_2}\right) \end{aligned}$$

This ensures a nice continuous surface which has a reasonable shape for a terrain. Due to their non-linear nature, bilinearly interpolated terrains make line of sight (LOS) calculations less efficient. Figure 3.2 shows a single grid cell with a bilinearly interpolated surface and a LOS that is to be tested. Given just the four grid points it is not obvious that the line should not intersect the surface. In order to check whether the line intersects the terrain, we must take the interior of the cell into account. The mathematics behind this is manageable, but it is a lot more comprehensive than for a piecewise linear model.

Although bilinear- and higher order interpolation schemes produce nice models, it is not given that they contribute to the accuracy of the model. Without more knowledge about the terrain, or more data points, there is no reason to prefer these to lower order interpolation schemes for accuracy. However, as discussed above, we typically consider real world terrain to be continuous, which suggests that a piecewise linear model is preferable to a piecewise constant one.

The FRM terrain model

Franklin, Ray and Mehta describe in [FRM94] a model which combines the computational simplicity of piecewise linear models with simple the repre-

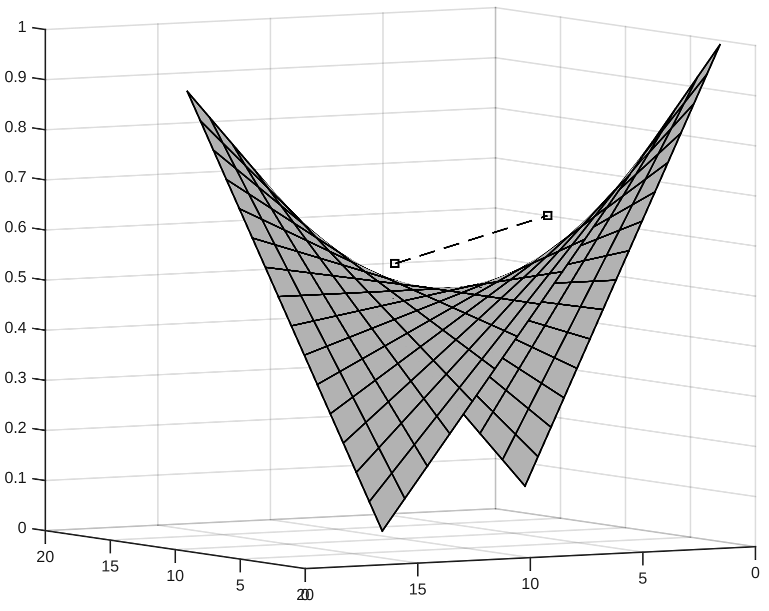


Figure 3.2: Grid cell with bilinear interpolation. The dashed line represents some LOS that is to be tested if it intersects the terrain.

sentation of RSGs. For lack of a better name we will simply refer to it as the *FRM model*.

Generally, it is not possible to represent squares linearly (as illustrated in fig. 3.2). In fact, the only primitive, other than points and lines, that is planar for arbitrary vertices, is the triangle. Planar primitives, such as triangles, have nice properties when it comes to LOS calculations. Deciding whether a LOS is obstructed by a triangle can always be done by comparing the LOS to two evaluating points on the boundary of the triangle. This is contrasted by, for instance, bilinear interpolation, where also the interior must be considered. The same issue occurs using higher order interpolations; the computations become more complicated, without obvious gains in accuracy.

Ideally we want to be able to check for LOS intersections by evaluating the LOS at each point it crosses above a grid line, and see if it intersects the terrain. To understand the implications of using this method, we will now define the FRM terrain model, which ensures its correctness. The resulting model thus allows us to work with LOSs in a highly efficient manner. Naturally, this guarantee of correctness only holds within the context of the model, as we do not know how well it matches the real world terrain. Later,

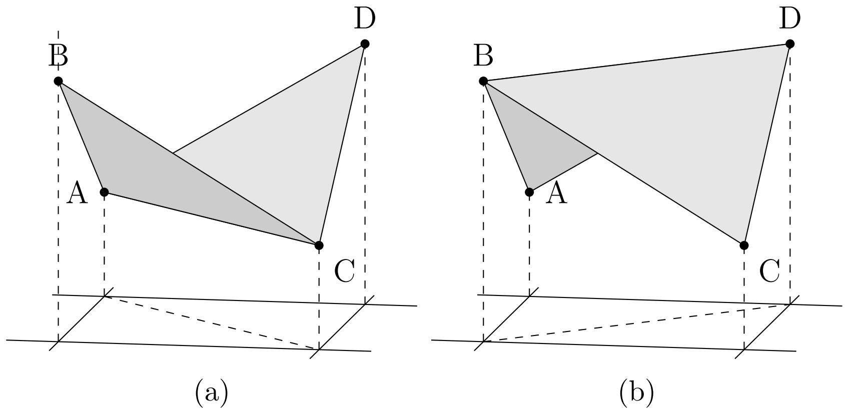


Figure 3.3: Two possible triangulations of a grid cell. The union of these triangulations is the surface of the tetrahedron spanned by the four grid points.

we will use the FRM model due to its practical properties. There is no reason to believe that it models the real world more accurately than other models, however.

We will base our model on the idea of triangulating the grid, and use this triangulation as the terrain surface. In principle, each grid cell can be triangulated in two ways. For grid cells where the corners lie in a single plane, the terrain surface for the two triangulations are exactly the same. When this is not the case, however, the two triangulations represent two different terrain surfaces. This is illustrated in fig. 3.3, which shows the two triangulations of a grid cell where the grid points lie in separate planes.

In the particular case of fig. 3.3b it is clear that the simplified LOS tests will fail, as a LOS can easily intersect the terrain on the interior without intersecting it on the boundary of the cell. Looking at fig. 3.3a, on the other hand, this method will work, as the cell boundary contains the most protruding features of the terrain. By using this "lower" triangulation in each grid cell as a representation of the terrain surface, we ensure that the simplified LOS tests are accurate.

We now give this terrain model a formal definition in terms of the tetrahedron spanned by the grid points of each grid cell. We will see why this is useful in the proof of theorem 3.1.

Definition 3.4 (The FRM terrain model)**.**

The surface of the terrain above a grid cell G is given by the bottom of the *tetrahedron spanned by the grid points of G.*

In particular, the elevation at a point $\mathbf{x} = (x_1, x_2)$ on the boundary of G between grid points $\mathbf{s}$ and $\mathbf{s}'$ is given by:

$$e(x_1, x_2) = e_{\mathbf{s}} \frac{||\mathbf{x} - \mathbf{s}'||}{||\mathbf{s} - \mathbf{s}'||} + e_{\mathbf{s}'} \frac{||\mathbf{s} - \mathbf{x}||}{||\mathbf{s} - \mathbf{s}'||}$$

This is an alternative, but equivalent, formulation of the terrain model used by Franklin et al. in [FRM94]. As suggested, this model has the property that LOSs can be tested for terrain intersections by evaluating the grid boundaries only. For completeness we will now provide a proof of this property.

Theorem 3.1 (The FRM theorem)**.**

Given a line $\ell \subset \mathbb{R}^3$ and a grid cell G representing a terrain using definition 3.4. Then ℓ intersects the terrain above G iff. there is a point $(x_1, x_2, x_3) \in \ell$ such that $(x_1, x_2, 0) \in \partial G$ and $x_3 \leq e(x_1, x_2)$.

Proof. ℓ clearly intersects the terrain above G if the conditions above are satisfied. It remains to show the converse.

Assume that ℓ has no points meeting the requirement from the theorem. Let us now consider points where $(\ell)_{xy}$ intersects ∂G. If there are zero, one or infinitely many such points, ℓ does not pass over the interior of the grid cell, and the result is evident.

If there are exactly two such points, $\mathbf{a}$ and $\mathbf{b}$, then there are points $\mathbf{a}' = (a_1, a_2, e(a_1, a_2))$ and $\mathbf{b}' = (b_1, b_2, e(b_1, b_2))$ that lie on the terrain surface. Due to our assumption ℓ lies strictly above the line through $\mathbf{a}'$ and $\mathbf{b}'$. As we know the terrain surface is upward bounded by the bottom of the tetrahedron spanned by the four grid points of G. Tetrahedra are convex, so the line segment between $\mathbf{a}'$ and $\mathbf{b}'$ is therefore on or above the terrain surface. Thus ℓ must lie strictly above the surface. ■

This theorem proves the correctness of the simplified LOS test on the model. Note that we do not use the notion of triangulations when implementing the model. In practice the FRM model behaves like a normal RSG, where we just consider the grid lines when testing for LOS intersections. The triangulations are merely a tool for understanding the implications of assuming this type of LOS testing is correct.

With the FRM model we get a terrain mode with the benefits of both nearest neighbor- and bilinear interpolation. First of all we get a continuous surface that matches what we expect a terrain should look like for a given set of grid points. Secondly we get the fast LOS calculations needed for making efficient viewshed algorithms.

3.2.3 Contour line representation

Another vector-based terrain model uses contour lines to represent the terrain. The contour lines are typically represented as closed curves. Often these are represented as piecewise linear curves or splines. The contour lines themselves do not provide any information about the terrain between the contour lines, so some interpolation is needed in order to make a complete terrain model.

The accuracy of such a model primarily depends on the resolution along the vertical axis, but also on the density of the data samples the curves are based on. Similarly to TINs this representation gives us flexibility for higher accuracy in areas of the terrain where this is needed.

Kartverket uses this type of format for much of their openly available data, as it is convenient for the purpose of rendering maps. The data sets seem to require little storage space compared to TINs and RSGs of comparable quality.

Although contour lines have some nice properties with regards to storage efficiency they are not particularly suited for visibility calculations, and hardly any of the well-known viewshed algorithms operate on contour line-based models. We will therefore not consider them further for this purpose.

3.2.4 Summary

In practice RSGs and TINs are the only viable families of terrain models for visibility applications. Contour line representation and vector models using higher order geometric objects fall short due to more complicated visibility calculations resulting in algorithms that cannot compete with their RSG and TIN counterparts.

TINs have two advantages over RSGs. The first is the ability to use varying densities of data points, which means that the model can provide higher precision where it is actually needed. In practice this typically results in a smaller memory footprint of the model, compared to a similar RSG. The second advantage is greater flexibility in what it can represent, such as tunnels or overhanging structures. As discussed, however, the algorithms we consider cannot handle such structures, which means that this is not an advantage in practice.

When implementing viewshed algorithms the inherent structure of RSGs is a huge asset in that it allows many key operations to be executed in constant time. This involves operations such as finding where a LOS intersects a grid line, or finding neighboring nodes of a LOS. On a TIN these operations typically either require logarithmic time lookups, or some preprocessing step. This means that the resulting viewshed algorithms typically run slower or use as much memory on TINs as on RSGs.

The advantages of using TINs over RSGs are therefore invalidated in viewsheds applications, which is also why RSGs are so popular in viewshed literature. For this reason we focus on RSGs throughout this thesis. More specifically we will use RSGs with the FRM model as discussed above.

3.3 Viewshed algorithms

In this section we consider several algorithms for finding elevated viewsheds on a modeled terrain. For reasons discussed in the previous section we focus on algorithms that work on the FRM model, but we will also take a quick look at a few TIN-algorithms as well for completeness. We consider both algorithms that are accurate and approximate. By *approximate* we mean that the algorithm might mislabel two points $\mathbf{u}$ and $\mathbf{v}$ as intervisible, even though the LOS between them intersects the terrain, or vice versa. Later in the thesis we also empirically compare the performance of some of these algorithms, and discuss which of them that are suitable for our application.

Common for all of the algorithms presented here is that they take as input the terrain either as an RSG or a TIN, the lateral position of the observer and the observer- and target height. Most of the algorithms classify each grid point *visible* or *not visible*, while some of them also are capable of classifying arbitrary points on the terrain. A few of the algorithms classify regions of the terrain instead of single points.

We will look at three types of algorithms for RSGs, starting with a brute force algorithm which calculates the accurate viewshed. We consider two approximate algorithms that estimate visibility by evaluating the terrain along rays spread out across the terrain. These are the *R2* and *radar-like* algorithms. We also consider *XDraw* and the *expanding circular horizon (ECH)* algorithm, that estimate the visibility by propagating an approximation of the horizon across the terrain. The radar-like- and the ECH algorithms also work for triangulations. Additionally we will briefly consider the two accurate linearithmic-time TIN-based algorithms described in [CS89] and [FM94].

Before we start, however, we need some notation. For RSG-based algorithms we will denote the set of data points by $S \subset \mathbb{R}^3$, and refer to them as *grid points*. The line between two adjacent grid points we call a *grid line*, and the area spanned by four neighboring grid points we call a *grid cell*. The set of all grid cells we shall denote by G, and the set of all grid lines, i.e. the boundaries of all grid cells, will be denoted ∂G. When discussing LOSs, we will often consider the points on grid lines which the LOS passes over. These we will refer to as *grid line crossings*. We will be looking at points projected onto the vertical axis and the grid plane. For a point $\mathbf{p}$ we will denote this by $\mathbf{p}_z$ and $\mathbf{p}_{xy}$, respectively. We will also use notation like $||\mathbf{p}||_{xy}$ to denote the norm of some projection of a point, in this particular case the projection of $\mathbf{p}$ onto the grid plane.

3.3.1 A brute force algorithm

The obvious brute force viewshed algorithm for any RSG-based terrain model is to iterate through each grid point in the grid, and test whether the LOS back to the observer intersects the terrain. Franklin et al. [FRM94] refers to this type of algorithm as the *R3 algorithm*. A similar variant is also proposed by [BMCK08]. On the FRM model this algorithm has a relatively efficient implementation, while remaining accurate. For this reason we will use it as a baseline for evaluating the accuracy and speed of the other algorithms.

Theorem 3.1 provides us with the tools we need to make an efficient implementation of this algorithm on the FRM model. To test if a LOS intersects the terrain, we only need to compare it to the terrain whenever it crosses a grid line. The idea is illustrated in fig. 3.4, where a LOS is drawn between the observer o and a target point t. The grid cells in play are shaded, and the grid line crossings are marked with crosses. These crosses are the only six points we have to compare the LOS to the terrain at in order to establish whether or not they intersect.

First, we give an efficient algorithm for finding all grid line crossings of an LOS. For the purpose of simplicity, we confine ourselves here to LOSs that have endpoints in the grid points, but this could be generalized to arbitrary points.

Algorithm 3.1 (Finding grid line crossings).
Given an LOS ℓ between points $\mathbf{o}, \mathbf{t} \in \mathbb{R}^3$, where $\mathbf{o}_{xy}, \mathbf{t}_{xy} \in U$. The following algorithm returns the set of grid cell boundary crossings, i.e. $\ell \cap \partial G \setminus \mathbf{o}$, as a list sorted from $\mathbf{o}$ to $\mathbf{t}$.

```
$\mathbf{d} = \mathbf{t} - \mathbf{o}$
$\theta = \text{atan}_2(d_y, d_x)$
$x_{dir} = \begin{cases} 1 & \textit{if } |\theta| > \frac{\pi}{2} \\ -1 & \textit{if } |\theta| < \frac{\pi}{2} \\ 0, & \textit{otherwise} \end{cases}$
$y_{dir} = \begin{cases} 1 & \textit{if } \theta > 0 \\ -1 & \textit{if } \theta < 0 \\ 0, & \textit{otherwise} \end{cases}$
$x_{max} = |d_x| - |x_{dir}|$
$y_{max} = |d_y| - |y_{dir}|$
$x, y = 0$

while $x < x_{max} \wedge y < y_{max}$
    if $\frac{x+1}{|\cos\theta|} < \frac{y+1}{|\sin\theta|}$
        $x+ = 1$
        $\mathbf{p} = (x_{dir} \cdot x, y_{dir} \cdot x|\tan\theta|)$
    else
        $y+ = 1$
        $\mathbf{p} = (x_{dir} \cdot y|\cot\theta|, y_{dir} \cdot y)$

    yield $\mathbf{p} + \mathbf{o}$
```

A sample implementation of this algorithm can be seen in listing A.4. This implementation is a somewhat modified version of the algorithm above, in order to interface efficiently with the various viewshed algorithms.

Corollary 3.3.

Algorithm 3.1 runs in $O(\sqrt{n})$ time on a square grid with $O(n)$ points.

Proof.

Given any two points $\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$ on a square grid with $O(n)$ points, then clearly $||\mathbf{b} - \mathbf{a}||_x, ||\mathbf{b} - \mathbf{a}||_y = O(\sqrt{n})$. Each iteration in the **while**-loop executes in $O(1)$ time, and moves from $\mathbf{a}$ to $\mathbf{b}$ one grid unit along one of the

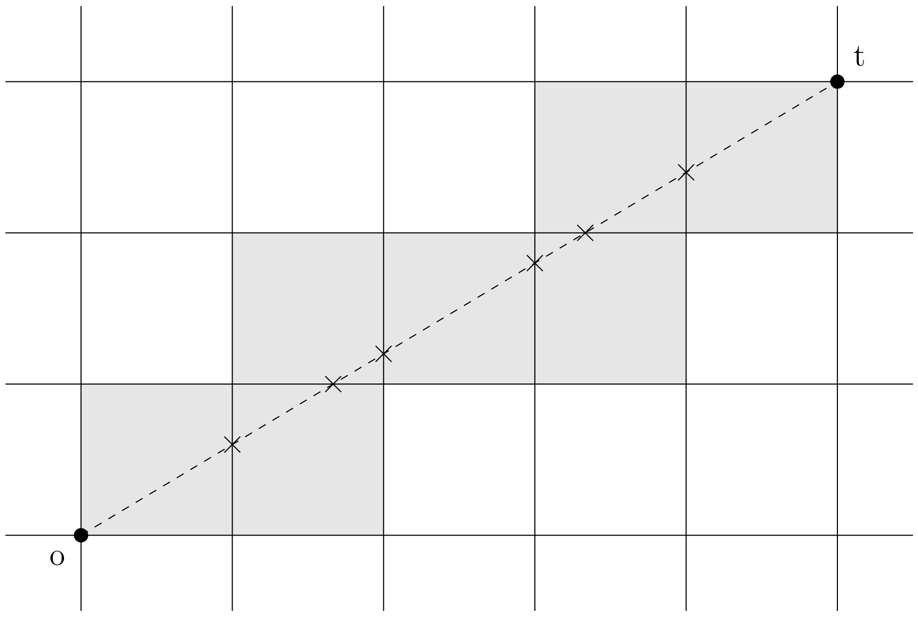


Figure 3.4: LOS with grid cell boundary crossings

axes, so there are at most $\lfloor ||\mathbf{b}-\mathbf{a}||_x \rfloor + \lfloor ||\mathbf{b}-\mathbf{a}||_y \rfloor = O(\sqrt{n})$ such steps. ■

Next, we state the complete brute force algorithm.

Algorithm 3.2 (R3).
Let $\mathbf{o} \in \mathbb{R}^3$ be the observer on the terrain surface, and let ψ and ω be the observer and target height respectively. Let S denote the set of grid points.

```
for all $\mathbf{s} \in S$
    let $\ell$ be the LOS from $\mathbf{o} + \psi\mathbf{k}$ to $\mathbf{s} + \omega\mathbf{k}$
    label $\mathbf{s}$ as visible

    for all grid line crossings $(x_1, x_2)$ of $\ell$
        if $e(x_1, x_2) \geq \ell(x_1, x_2)$
            label $\mathbf{s}$ as not visible
            break
```

We will now proof the correctness of this algorithm on the FRM model, using some of the previous results.

Corollary 3.4.

The R3 algorithm calculates the accurate viewshed.

Proof. First assume that the point $\mathbf{t}$ is intervisible with the observer $\mathbf{o}$. Then the LOS from $\mathbf{o}$ to $\mathbf{t}$ does not intersect the terrain in any of the grid cells between $\mathbf{o}$ and $\mathbf{t}$. Then by theorem 3.1 there exists no points $(x_1, x_2, x_3) \in \ell$, where (x_1, x_2) is on a grid line, such that $e(x_1, x_2) \geq \ell(x_1, x_2)$. Thus the algorithm will correctly classify $\mathbf{t}$ visible.

Next assume that $\mathbf{t}$ is *not* intervisible with $\mathbf{o}$. Then there exists a point $\mathbf{p} = (p_1, p_2, p_3)\ell$ where ℓ intersects terrain. Let G be a grid cell that contains $\mathbf{p}$. By theorem 3.1 there must then exist a point $(x_1, x_2, x_3) \in \ell$, such that $(x_1, x_2) \in \partial G$ and $e(x_1, x_2) \geq x_3$. Hence, $\mathbf{t}$ will be classified *not* visible. ■

We also show the asymptotic running time of the algorithm.

Corollary 3.5. *The R3 algorithm executes in* $O(n^{\frac{3}{2}})$ *time.*

Proof.

We can find all of the grid line crossings of any LOS in $O(\sqrt{n})$ time using algorithm 3.1. In a square grid with $O(n)$ points there are at most $O(\sqrt{n})$ such crossings underneath any LOS. The remaining operations in the inner loop clearly execute in $O(1)$ time, so the LOS can be accepted or rejected in $O(\sqrt{n})$ time.

We evaluate one LOS for each of the $O(n)$ points in the grid, so the overall time complexity for this algorithm is $O(n^{\frac{3}{2}})$. ■

3.3.2 R2

In order to improve the efficiency of the brute force algorithm, we simply have to evaluate fewer LOSs. When evaluating LOSs to targets that are far away from the observer, most of these LOSs pass through or nearby closer targets. This is illustrated in fig. 3.5. If we accept approximate results, we can use intermediate results, obtained when calculating LOSs to the far targets, to estimate the visibility of targets closer to the observer. The resulting algorithm proposed by Franklin et al. in [FRM94] is typically referred to as the R2 algorithm.

When deciding the visibility of a point $\mathbf{v}$ in R3, we simply compare the *elevation* of the corresponding LOS to the terrain at each grid line crossing. A side-view illustration of this is given in fig. 3.6, where the LOS does not intersect the terrain anywhere. These calculations cannot be reused for deciding the visibility of other points, unless they happen to lie on the exact

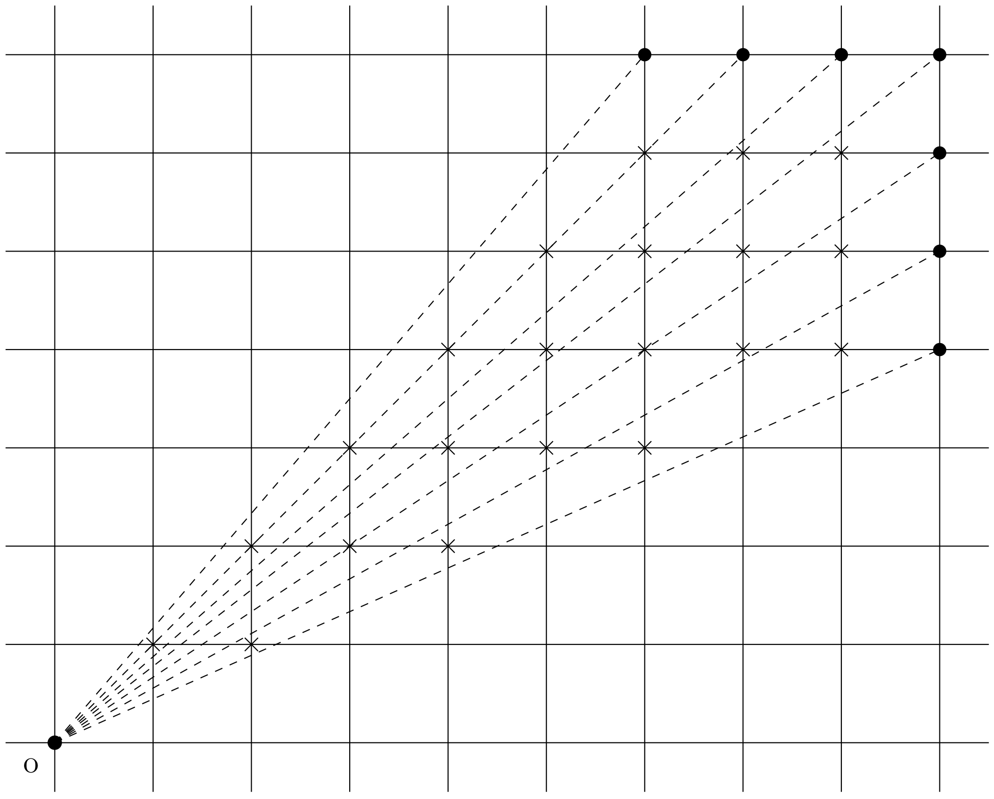


Figure 3.5: Overview of a terrain with LOSs to some of the perimeter points, which are drawn as solid circles. The idea behind R2 is to estimate the visibility of the grid points drawn with crosses from the intermediate results of these LOSs.

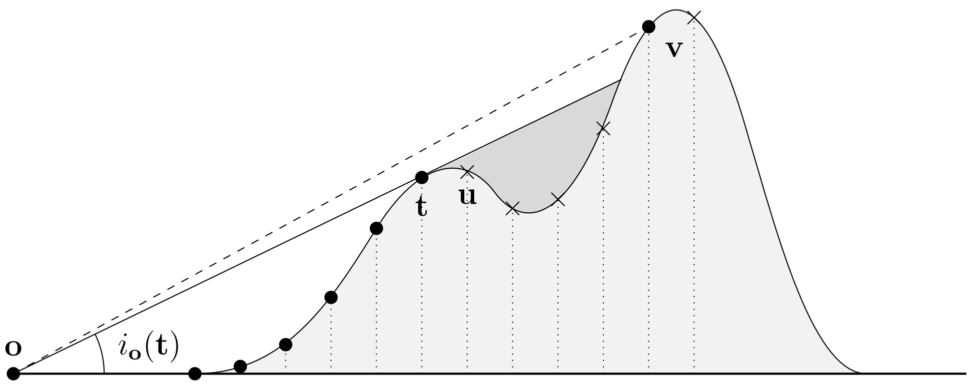


Figure 3.6: Cross section of a terrain with the observer, $\mathbf{o}$, and some grid points. Visible points are drawn with solid discs, while invisible points are drawn with crosses.

same LOS as $\mathbf{v}$. If we instead look at the *slope* of the LOS going to each of the grid line crossings, we can reuse the calculations. In fig. 3.6 for instance, the slope of the LOS going through $\mathbf{t}$ is higher than for $\mathbf{u}$. Since $\mathbf{u}$ is further away from the observer, we know that $\mathbf{u}$ is not visible. It turns out that by accumulating the maximum slope as we move away from the observer, we can easily decide the visibility of the grid line crossings as we go.

We will now formalize these ideas, and use them to make a proper algorithm. First we define exactly what we mean by *slope*, and show that we can use it as a replacement for elevation in our visibility calculations.

Definition 3.5.

The angle between between the horizontal plane and the LOS from the observer to a point $\mathbf{p}$ *we call the* ***slope*** *of* $\mathbf{p}$, $s_{\mathbf{o}}(\mathbf{p})$.

$$s_{\mathbf{o}}(\mathbf{p}) = \arctan\left(\frac{||\mathbf{p}-\mathbf{o}||_z}{||\mathbf{p}-\mathbf{o}||_{xy}}\right)$$

Corollary 3.6 (The FRM theorem for slope)**.**

Given a LOS ℓ *from* $\mathbf{o}$ *to* $\mathbf{t}$*, and a grid cell* g*. Then* ℓ *intersects the terrain above* g *iff. there is a point* $\mathbf{p} = (p_1, p_2, p_3) \in \ell$ *such that* $(p_1, p_2, 0) \in \partial g$ *with a corresponding point* $\mathbf{p}' = (p_1, p_2, e(p_1, p_2))$ *such that* $s_{\mathbf{o}}(\mathbf{t}) \leq s_{\mathbf{o}}(\mathbf{p}')$.

Proof.

Given an LOS ℓ from $\mathbf{o}$ to $\mathbf{t}$, and a grid cell G. Let $\mathbf{p} \in \ell$. First observe that

$s_{\mathbf{o}}(\mathbf{p}) = s_{\mathbf{o}}(\mathbf{t})$, since both $\mathbf{t}, \mathbf{p} \in \ell$. Next we have the following:

$$\begin{gathered}
s_{\mathbf{o}}(\mathbf{t}) \leq s_{\mathbf{o}}(\mathbf{p}') \\
s \Updownarrow \\
i_{\mathbf{o}}(\mathbf{p}) \leq s_{\mathbf{o}}(\mathbf{p}') \\
\Updownarrow \\
\arctan\left(\frac{||\mathbf{p}||_z}{||\mathbf{p}-\mathbf{o}||_{xy}}\right) \leq \arctan\left(\frac{||\mathbf{p}'||_z}{||\mathbf{p}'-\mathbf{o}||_{xy}}\right) \\
\Updownarrow \\
\arctan\left(\frac{p_3}{||\mathbf{p}-\mathbf{o}||_{xy}}\right) \leq \arctan\left(\frac{e(p_1,p_2)}{||\mathbf{p}-\mathbf{o}||_{xy}}\right) \\
\Updownarrow \\
p_3 \leq e(p_1,p_2)
\end{gathered}$$

Where we use that arctan is strictly increasing in the last step. The result now follows from theorem 3.1. ■

Next we define the notion of the *horizon* at a point, which is the property we will ultimately use to classify each grid point.

Definition 3.6.

The ***horizon*** *at a point* $\mathbf{p}$ *wrt. some observer* $\mathbf{o}$, $\hat{s}_{\mathbf{o}}(\mathbf{p})$, *is the maximum slope a point* $\mathbf{p}'$ *with the same lateral coordinates as* $\mathbf{p}$ *can have, such that the LOS from* $\mathbf{o}$ *to* $\mathbf{p}'$ *intersects the terrain in at least one point.*

$\mathbf{p}$ *is thus visible iff.* $s_{\mathbf{o}}(\mathbf{p}) > \hat{s}_{\mathbf{o}}(\mathbf{p})$.

The next results shows that in order to decide the visibility of a point, we can simply compare its slope point to the maximum slope of the grid line crossings between it and the observer.

Corollary 3.7.

Let $\mathbf{o}$ *be the observer and* $\mathbf{p}$ *a point. Also let* X *be the set of grid line crossings of the LOS from* $\mathbf{o}$ *to* $\mathbf{p}$. *When using the FRM terrain model, the following two inequalities are equivalent.*

$$\begin{aligned}
s_{\mathbf{o}}(\mathbf{p}) &> \hat{s}_{\mathbf{o}}(\mathbf{p}) \\
s_{\mathbf{o}}(\mathbf{p}) &> \max_{\mathbf{x}\in X} s_{\mathbf{o}}(\mathbf{x})
\end{aligned}$$

Proof.

From the definition of the horizon it is obvious that $s_{\mathbf{o}}(\mathbf{p}) > \max_{\mathbf{x}\in X} s_{\mathbf{o}}(\mathbf{x})$ if $s_{\mathbf{o}}(\mathbf{p}) > \hat{s}_{\mathbf{o}}(\mathbf{p})$.

Let ℓ be the LOS from $\mathbf{o}$ to $\mathbf{p}$. Assume that $s_{\mathbf{o}}(\mathbf{p}) \leq \hat{s}_{\mathbf{o}}(\mathbf{p})$. Then there exists a point $\mathbf{x} \in \ell$ on the terrain that intersects ℓ. Then by corollary 3.6 there is also a grid line point, $\mathbf{x}'$, that lies on or above ℓ such that $s_{\mathbf{o}}(\mathbf{p}) \leq s_{\mathbf{o}}(\mathbf{x}')$. Thus $s_{\mathbf{o}}(\mathbf{p}) \leq \max_{\mathbf{x}\in X} s_{\mathbf{o}}(\mathbf{x})$. ■

We already know that we can decide the visibility of any point by comparing its slope to the horizon. Corollary 3.7 shows that when making such comparison, we can use the maximum slope of LOS grid line crossings instead of the horizon, without risk of making any misclassifications. As suggested this maximum can be accumulated as we move away from the observer, giving us an efficient method for classifying the grid line crossings. From now on we will refer to this maximum simply as the horizon, since they are equivalent for our purposes.

In order to classify grid points, which after all is what we are interested in, we can use the horizon of the nearest grid line crossing as an estimate of the horizon at the grid point. This forms the basis for the R2 algorithm as formulated by Franklin et al. in [FRM94]. The R2 algorithm works in two passes. First the algorithm calculates horizons by evaluating LOSs to points along the perimeter of the grid. The results are stored as estimates of the horizon for the corresponding grid points. In the second pass the algorithm classifies each point by comparing the estimated horizon to the actual slope.

Algorithm 3.3 (R2).
Let $\mathbf{o}' \in \mathbb{R}^3$ *be the observer on the terrain surface, and let* ψ *and* ω *be the observer and target height respectively. Let* S *denote the set of grid points. If* $\mathbf{x}$ *is a point on a grid line, then* $neig(\mathbf{x})$ *denotes the two grid points at the ends of said grid line.*

```
for all $\mathbf{s} \in S$
    set $\mathbf{s}.dist = \infty$

let $\mathbf{o} = \mathbf{o}' + \psi\mathbf{k}$

for all $\mathbf{p} \in S$, st. $\mathbf{p}$ is on the perimeter of $S$
    let $\ell$ be the LOS from $\mathbf{o}$ to $\mathbf{p}$
    $h = -\infty$

    for all grid line crossings $(x_1, x_2)$ of $\ell$
        $\mathbf{x} = (x_1, x_2, e(x_1, x_2))$

        for all $\mathbf{s} \in neig(\mathbf{x})$
            if $||\mathbf{x} - \mathbf{s}||_{xy} < \mathbf{s}.dist$
                $\mathbf{s}.dist = ||\mathbf{x} - \mathbf{s}||_{xy}$
                $\mathbf{s}.h = h$

        $h = \max\{h, s_{\mathbf{o}}(\mathbf{x})\}$

for all $\mathbf{s} \in S$
    if $s_{\mathbf{o}}(\mathbf{s} + \omega\mathbf{k}) > \mathbf{s}.h$
        label $\mathbf{s}$ as visible
    else
        label $\mathbf{s}$ as not visible
```

As a final result we also show the asymptotic running time of the R2 algorithm.

Corollary 3.8.

The R2 algorithm runs in $O(n)$ *time on a square grid with* $O(n)$ *points.*

Proof.

The perimeter of a square grid with $O(n)$ points consists of $O(4\sqrt{n} - 4) = O(\sqrt{n})$ points. neig$(\mathbf{x})$ contains at most two grid points for any $\mathbf{x}$, so the procedure of evaluating an LOS and updating each neighborhood point still only takes $O(\sqrt{n})$ time. Thus the first step of the algorithm runs in $O(n)$ time.

The second step of the algorithm consists of simply iterating through each grid point and performing a simple comparison of two numbers, which obviously can be done in $O(n)$ time. ■

As we see the R2 algorithm should be considerably faster than R3. As discussed this algorithm will give us an approximation of the viewshed. Thanks to corollary 3.7 we know that any grid points intersecting a LOS always will be classified correctly. In particular this means that the grid points on the axes and diagonals relative to the observer are correctly classified. For the remaining grid points we can offer no such guarantee, but as we shall see in chapter 4 the estimates work well in practice. In chapter 5 we will also consider some modifications of this algorithm for further improving the accuracy.

3.3.3 The radar-like algorithm

One of the algorithms proposed by Ben-Moshe et al. in [BMCK08] is referred to as the *radar-like* algorithm, and it shares several similarities with the R2 algorithm. This algorithm is originally proposed used on TINs, but has a trivial extension to RSGs. Like R2, this algorithm works by evaluating the terrain along a set of LOSs. Instead of sending a LOS to each grid point on the perimeter of the terrain, the radar-like algorithm first sends a set of LOSs in evenly distributed directions, dividing the terrain into sectors. The cross sections of the terrain is compared in each pair of adjacent LOSs, and the sector is subdivided until all adjacent cross sections are sufficiently similar. Once the sectors have been settled, the algorithm estimates the visibility of their interior using a special interpolation technique. We will not go into the specifics of this interpolation technique here, other than referring to the original article and our implementation listing A.21. The primary difference is, however, that the radar-like algorithm *first* classifies the points along each LOS, and then interpolates using the classification result itself. Whereas R2 calculates the horizon for points along each LOS, interpolates these values, and *then* classifies the grid points.

Ben-Moshe et al. also suggests a variant of this algorithm, referred to as the *fixed radar-like* algorithm. In this case the sector subdivision is omitted, and only the predefined uniformly distributed LOSs are considered. The only difference between this algorithm and R2, apart from the interpolation schemes, is that the R2 directs LOSs points, while the fixed radar-like algorithm sends them in uniformly distributed directions, not necessarily hitting any points. A nice feature of this algorithm is that its accuracy can easily be adjusted by changing the number of evaluated LOSs. For time-critical applications we can reduce the running time by evaluating fewer LOSs, and we can also boost the accuracy in exchange for running time, should that be desirable.

In principle this algorithm has a similar complexity as R2, so there is reason to believe that it is possible to create comparably efficient implementations the two algorithms. Our implementation of R2 has received a lot more attention in terms of optimization for speed, so we have not been able to make a fair comparison in terms of running time. Our primary interest in this algorithm is, however, its ability to boost the accuracy on demand.

A secondary interest is to compare the two interpolation schemes. As we shall see in chapter 4, R2 is more accurate than the radar-like algorithm when evaluating the same number of LOSs. However, this is not a fair comparison of the interpolation schemes. The radar-like algorithm typically evaluates fewer grid line crossings than R2 for the same number of LOSs. This is because a smaller portion of the area covered by the LOSs used by radar-like lie inside the terrain. Instead we should compare the two interpolation schemes on the exact same LOSs, in which case it turns out that we are not able to show any statistically significant difference in performance between the two schemes. We shall see in chapter 5, however, that with minor modifications to the R2 scheme, we get an interpolation scheme that outperforms the scheme proposed by Ben-Moshe et al.

3.3.4 XDraw

As we have seen with the R2 algorithm, there is a close relationship between the horizon of points far away from and close to the observer. In R2 we exploit this relationship out-to-in, by evaluating LOSs from the distant points, and then leveraging the intermediate results to estimate the horizon for points on the interior of the grid.

Another natural approach to this is to evaluate the horizons in-to-out. We do this by establishing the horizon of the grid points closest to the observer, and then propagating outwards, estimating the horizon of each new grid point from the horizon of the neighboring grid points between itself and the observer. Algorithms based on this notion belong to a family typically referred to as XDraw in the literature, e.g. by [Ray94], [Izr03], [XY09] and

[KZ02] to name a few.

We shall now turn to fig. 3.7 for an illustration of how the XDraw algorithm works. As we saw with R2, we can calculate the accurate horizon of all points along any of the four diagonals and axes by evaluating a total of eight LOSs; one to each corner and midpoint of the grid perimeter. In the figure all of these points are marked with crosses and have a white background. For the majority of points, however, the horizon must be estimated. Therefore we will in this algorithm work with an *horizon estimate*, denoted $\tilde{s}_{\mathbf{o}}$.

We begin estimating the horizon of the points that are closest to the observer, i.e. the points on light gray background in the figure. For the point $\mathbf{t}$ we see that the horizon depends on the horizon of $\mathbf{x}$. Since x is on the grid line between $\mathbf{n_1}$ and $\mathbf{n_2}$, we will estimate its horizon, $\tilde{s}_{\mathbf{o}}(\mathbf{x})$, from the horizons of $\mathbf{n_1}$ and $\mathbf{n_2}$. As we shall see, this estimation can be done in several ways.

Once we have estimated the horizon of all points in the light gray area, we repeat this process on each "shell" of grid points, moving further away from the observer. Each of these shells will have their horizon estimated from the previous shell. So in order to estimate the horizon of the points in the dark gray area, we use the estimated horizons of the points in the light gray area.

We now give a formal description of the XDraw algorithm.

Algorithm 3.4 (XDraw)**.**
Let $\mathbf{o} \in \mathbb{R}^3$ be the observer on the terrain surface, and let ψ and ω be the observer and target height respectively. Let S denote the set of grid points, and let $\tilde{s}_{\mathbf{o}}$ be some function that estimates the horizon of a point.

```
let $\mathbf{o} = \mathbf{o}' + \psi\mathbf{k}$

for all $\mathbf{p} \in S$
    let $\ell$ be the LOS from $\mathbf{o}$ to $\mathbf{p}$
    let $\mathbf{n}_1, \mathbf{n}_2 \in S$ be the endpoints of the grid line that intersects $\ell$ closest
to $\mathbf{p}$ at some point $\mathbf{x}$

    $\mathbf{p}.h = \max\{s_{\mathbf{o}}(\mathbf{p}), \tilde{s}_{\mathbf{o}}(\mathbf{x}, \mathbf{n}_1, \mathbf{n}_2)\}$

    if $s_{\mathbf{o}}(\mathbf{p} + \omega\mathbf{k}) > \mathbf{p}.h$
        label $\mathbf{p}$ as visible
    else
        label $\mathbf{p}$ as not visible
```

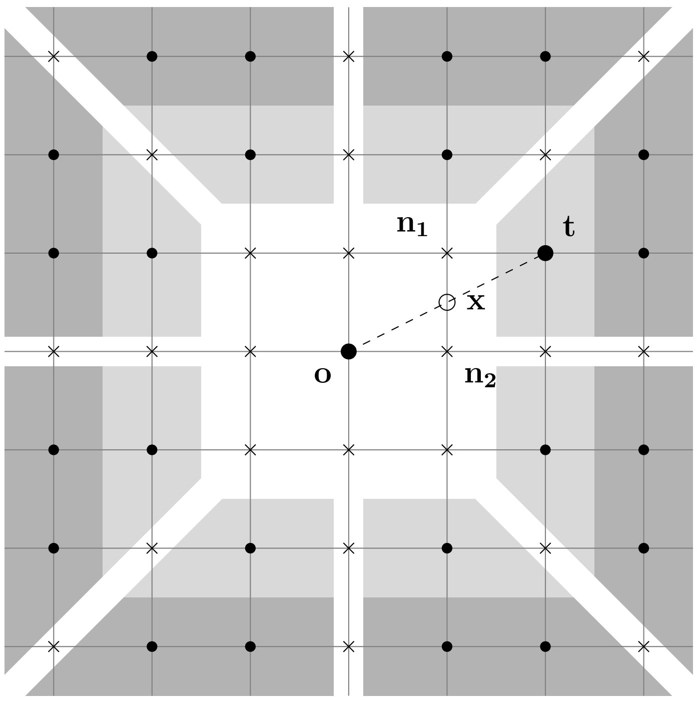


Figure 3.7: XDraw estimates the horizon of $\mathbf{t}$ using only the estimated horizon at $\mathbf{n}_1$ and $\mathbf{n}_2$.

Next, we also show that the running time of XDraw is, as expected, $O(n)$.

Corollary 3.9.

Provided that $\tilde{s}_{\mathbf{o}}$ runs in $O(1)$ time, algorithm 3.4 runs in $O(n)$ time on a square grid with n points.

Proof.

E.g. using ideas from algorithm 3.1 we can find $\mathbf{n}_1$ and $\mathbf{n}_2$ in $O(1)$ time. The rest of the operations in the loop are clearly $O(1)$ time operations. ■

When it comes to the choice of estimator, $\tilde{s}_{\mathbf{o}}$, there are several possibilities. Some exotic variants exist, such as the one proposed in [Izr03], but the four *standard* estimators used in the literature are:

$$\tilde{s}_{\mathbf{o}}^{\max}(\mathbf{x}, \mathbf{n}_1, \mathbf{n}_2) = \max\{\mathbf{n}_1.h, \mathbf{n}_2.h\} \tag{3.1}$$

$$\tilde{s}_{\mathbf{o}}^{\min}(\mathbf{x}, \mathbf{n}_1, \mathbf{n}_2) = \min\{\mathbf{n}_1.h, \mathbf{n}_2.h\} \tag{3.2}$$

$$\tilde{s}_{\mathbf{o}}^{\text{mean}}(\mathbf{x}, \mathbf{n}_1, \mathbf{n}_2) = \frac{\mathbf{n}_1.h + \mathbf{n}_2.h}{2} \tag{3.3}$$

$$\tilde{s}_{\mathbf{o}}^{\text{int}}(\mathbf{x}, \mathbf{n}_1, \mathbf{n}_2) = ||\mathbf{x} - \mathbf{n_2}||_{xy}\mathbf{n}_1.h + ||\mathbf{x} - \mathbf{n_1}||_{xy}\mathbf{n}_2.h \tag{3.4}$$

As our analysis will show, the linearly interpolated estimator described in eq. (3.4) has superior classification accuracy compared to the other three. Although the max and min estimators may seem crude, we shall see in chapter 5 that these have the special property that they hardly commit any type 1 and type 2 errors, respectively. That is, XDraw with the max estimator almost never classifies a point as visible if it is not. Vice versa for the min variant. [Ray94] suggest the max, min and mean estimators have an advantage in terms of efficiency over the interpolated estimator. In our analysis, however, we shall see that this advantage is negligible.

We shall also see that XDraw is significantly less accurate than the R2 algorithm. It does have the same time complexity as R2, but it is a simpler algorithm which has a significantly more streamlined implementation. The actual running time is typically around a third of that of R2. This, combined with the nice properties of the max and min estimators, means that XDraw might have an advantage over R2 in some applications.

3.3.5 The ECH algorithm

Ben-Moshe et al. proposes another algorithm in [BMCK08] that in some sense is similar to the XDraw algorithm. This is referred to as the *ECH algorithm*, or *ECH* for short. Similarly to XDraw, it works by propagating an estimate of the horizon outward, classifying the terrain as it moves by. Instead of doing this at each grid point, ECH does maintains an approximation to the horizon of the terrain inside a circle that is expanded in a series of steps.

More specifically this is done by considering the horizon along rays in a predefined set of headings $\{\alpha_i\}_{i=0}^{k}$. In each step, the algorithm tries to increase the radius of the current horizon from r_{old} to r_{new}. The slope of the terrain at r_{new} is compared to the horizon in order to determine the visibility in r_{new} for each azimuth direction α_i. The visibility of the remaining points on the circle, i.e. the ones with a heading different from all α_i, is then interpolated using nearest-neighbor interpolation. The visibility of the terrain along r_{new} is then compared to the visibility along r_{old}. If there is *sufficient* correspondence in visibility along these to circles, the visibility of the points on the interior of the annulus between the circles is interpolated using a similar correlation scheme as in the radar-like algorithm. If the visibility along the two circle is too inconsistent the algorithm retries with an r_{new} that is closer to r_{old}.

This algorithm is designed to be used on TINs, but there is nothing that prevents it from working on RSGs as well. The article provides no theoretical time- nor accuracy analysis of this algorithm. The authors do provide some empirical tests, however, indicating that ECH is outmatched by the radar-like algorithm both in terms of running time and accuracy. For this reason, and since ECH is rather similar to XDraw, which is significantly faster than the radar-like algorithm, we will not study this algorithm further.

3.3.6 The Cole-Sharir algorithm

Cole and Sharir present in [CS89] a data structure that can be used to calculate accurate viewsheds in $O(n\alpha(n)\log n)$ time on TINs with n faces. $\alpha(n)$ here is the inverse of the Ackerman function, which in practice is no larger than 4, so this algorithm runs in $O(n \log n)$ time for any case that is feasible to solve on a computer. The proposed data structure, referred to as an *horizon tree* in the literature, can be built in $O(n\alpha(n)\log n)$, and allows us to query the first intersection between a ray from the given observer and the terrain in $O(\log n)$ time. By comparing the distance to the first intersection to the distance to some target point along the given ray, we can easily establish the visibility of the target point. By repeating this process for each of the $O(n)$ target points in the terrain, we can obtain the full viewshed in a total of $O(n\alpha(n)\log n)$ time.

This type of algorithm works on what the authors refer to as *polyhedral terrains* or *monotonic polyhedral surfaces*. A polyhedral surface is a surface consisting of *flat* polygons. By *monotonic* they mean surfaces where any vertical line intersects the surface in at most one point. TINs are polyhedral surfaces, so the algorithm will work on those as long as the monotonicity criterion is met. RSGs in general are not polyhedrons, since the face of each primitive is not necessarily flat. However, RSGs based on FRM model will work with this algorithm. The key to this algorithm is that we can find partial horizons simply by considering the edges in the terrain. For the FRM model we have established that this is the case.

As [FM94] points out, due to a complicated implementation this algorithm is mostly of theoretical interest. Attempts at efficient implementations have failed to execute faster than the brute force algorithm for typical test cases. Thus we will not consider this algorithm further, other than giving a quick overview of how it works. The algorithm is thoroughly described in [CS89], so we will only go through the essentials here with an emphasis on special considerations that must be made in order to use it with RSGs.

If we consider the edges in a terrain with distance less than r, and project them onto the unit sphere centered at the observer, the partial horizon at r is given by the upper envelope of these projected edges. The resulting envelope consist of at most $O(n)$ non-overlapping smooth segments. Thus we can check if a ray lies above or below a given envelope in $O(\log n)$ time by binary searching for the relevant segment, and then comparing it to the ray. The primary idea behind this algorithm is to store a carefully selected set of such horizons in a binary tree, aptly named horizon tree, such that we can search the binary tree to decide which edge in the terrain that block the ray in question. Doing so allows us to find this edge in $O(\log^2 n)$ time. A technique reducing the running time of this search is described in [CS89], but we will not describe it here.

Before building the horizon tree, we need a particular ordering of the edges in the terrain. The ordering can be defined partially as follows, where $\mathbf{o}$ denotes the observer. If an edge A comes before an edge B, then there exists no point where $(B)_{xy}$ is closer to $(\mathbf{o})_{xy}$ than $(A)_{xy}$. It requires some work to obtain such an ordering for TINs, but it can be done in $O(n \log n)$ time. For

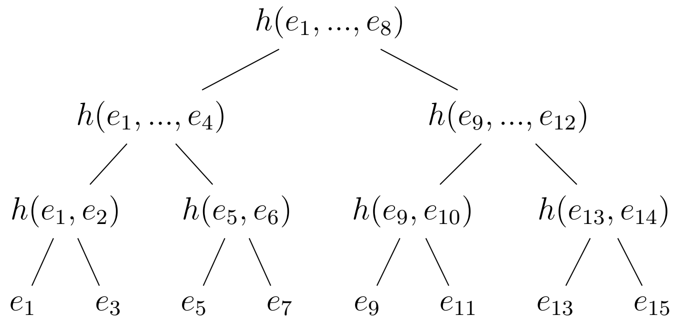


Figure 3.8: The structure of an horizon tree for $n = 16$.

RSGs, however, this corresponds to, loosely speaking, ordering the edges by distance from the observer. This ordering can be obtained in $O(n)$ time by iterating the edges breadth-first, starting at the observer.

Once an ordering of the edges $\{e_i\}_{i=1}^n$ has been obtained, we can build the horizon tree. We start by assigning a subset of the edges to each node. The root gets the full set of nodes, the left node gets the first half of this set, and the right gets the second half. This process is repeated until we reach nodes that are assigned only two edges. For each node we calculate the horizon of the *first half* of the corresponding set of edges. An illustration of this can be seen in fig. 3.8. These partial horizons happen to be the intermediary steps of the algorithm due to Atallah for finding the upper envelope of n smooth curves. This algorithm was first described in [Ata85], and runs in $O(n\alpha(n)\log n)$ time.

Assume that we want to find the first edge intersecting a specific LOS. On the unit sphere around the observer this LOS corresponds to a single point. We can now search the horizon tree for the edge in the following manner. Starting at the root, we move right if the LOS lies above the horizon of the current node, left otherwise. We continue in this manner until a leaf node is reached. The leaf node contains a single edge, e_i. If the LOS lies below e_i, then e_i is the first intersecting edge. Otherwise it is e_{i+1}.

Using this technique for finding the viewshed is straight-forward. For each grid point we find the LOS to the observer. We then find the first edge intersecting the LOS. The grid point is visible iff. this edge lies further away from the observer than the point itself.

3.3.7 Summary

We have presented a wide range of algorithms, most of which are approximate, while some are accurate. We have seen how the accurate Cole-Sharir algorithm can be used on RSGs, capable of calculating the viewshed in essentially $O(n \log n)$ time on a grid with n points. This algorithm has a rather complicated implementation, however, so in practice we will use the standard brute force algorithm for obtaining accurate viewsheds. The brute force algorithm runs in $O(n^{\frac{3}{2}})$ time, and will be useful for validating the results produced by the approximate algorithms.

As discussed, these algorithms are likely to be used as part of a more complex algorithm which might require evaluating a large number of viewsheds. This warrants the use of fast approximate algorithms, as the brute force algorithm will be too slow. We will compare the performance of three such algorithms; R2, XDraw and the radar-like algorithm. Based on existing literature we expect R2 and the radar-like algorithm to be the most accurate of the three, while XDraw is expected to be significantly faster.

Chapter 4

Benchmarking viewshed algorithms

In this chapter we first aim to establish a method for empirical comparison of viewshed algorithms. Once we have such a robust method in place we will use it for comparing the various viewshed algorithms, both in general and for our specific application. Later, in chapter 5, we shall also apply these techniques in order to improve the original R2-algorithm.

4.1 A motivating example

An intuitive and straight-forward approach to evaluating the performance of some viewshed algorithm can be summed up as follows:

1. Select some relevant terrain data
2. Select some observation point randomly
3. Run both the algorithm in question and R3 from the observation point
4. Quantify the error made by the algorithm

The steps 2-4 can be repeated in order to increase the accuracy of the performance estimate.

To see why this might be a bad idea, we will try this testing procedure on a sample regular square grid (RSG) terrain consisting of 1024×1024 points. The goal of the test is to compare the performance of two algorithms A and B, to decide which is better.

First we run the algorithms on 36 randomly chosen observation points. The relative error obtained in each run is shown as a boxplot in fig. 4.1a. Algorithm B seems to perform slightly better better than A, but the figure does a really poor job illustrating the difference. This can to some extent be mended by using a paired t-test. In this particular test, however, the apparent difference is insignificant, as it fails the t-test with a p-value of 0.81.

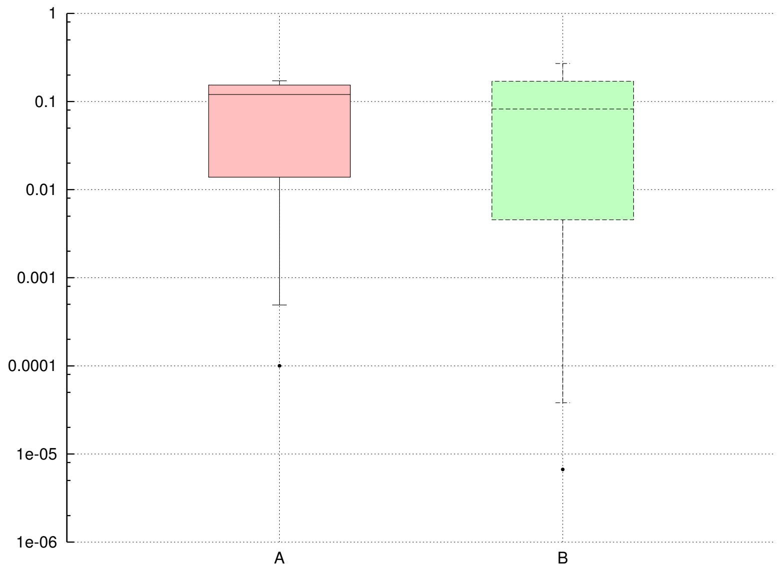
Next we run the algorithms on a test set consisting of 36 hand-picked points. The points are selected as local peaks in the terrain. In this case the figure shows clearly that algorithm A is more accurate than B. The t-test also successfully asserts A as more accurate than B with a p-value of 10^{-11}.

This example illustrates two problems using randomly selected observation points. Firstly the results obtained from such tests tend to differentiate algorithms poorly, which means we do not get statistically significant results. Secondly, we could have arrived at opposite conclusions about whether A or B is better, depending on which of the two tests we chose to emphasize. What makes the algorithms perform so differently in the two tests? How should we test to ensure we get robust results?

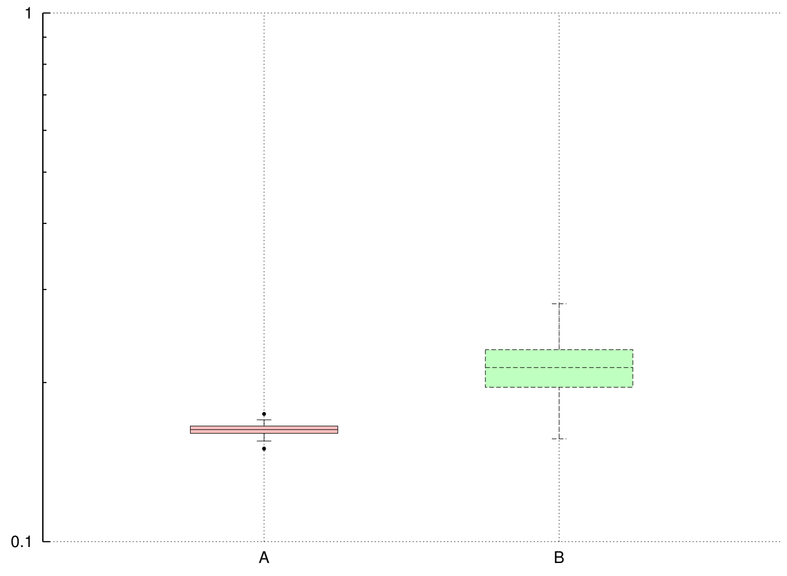
4.2 Error metrics

The most straight-forward metric for viewshed accuracy is the misclassification metric. Using e.g. the R3 algorithm we can find the accurate viewshed for a particular test case. The absolute classification error of some approximate viewshed can then be obtained by counting the number of misclassified points. By dividing by the total number of points in the grid, we obtain the relative error. This metric is widely used in the literature, e.g. [Ray94], [BMCK08] and [Izr03].

Another metric proposed by Franklin et al. in [FRM94] is to measure the error of the estimated horizon at each point. By *horizon* we here mean the figure defined in definition 3.6. We can obtain this error by comparing the estimate to the horizon calculated by the R3 algorithm. This metric clearly only applies to algorithms that operate with some notion of horizon. This includes the algorithms proposed by Franklin et al., but also some horizon-based algorithms, such as the randomized algorithm due to de Floriani described in [FM94]. Other algorithms, such as the expanding circular horizon (ECH) and the radar-like algorithm from [BMCK08] do not provide any estimate of



(a) 36 randomly selected observation points



(b) 36 hand-picked observation points

Figure 4.1: The relative classification error of two viewshed algorithms (lower is better).

the horizon, and thus cannot be used with this metric.

All we ultimately care about is whether we can trust that the classification produced is sufficiently correct. An algorithm can produce quite good horizon estimates, and still misclassify a large number of points. This suggests that the misclassification metric better quantifies what we are interested in. The horizon metric, however, can provide some valuable insights to the inner workings of an algorithm. It can also help us understand why a given algorithm performs well or poorly on particular test cases.

Since we are primarily interested in the resulting viewshed, and because we want to compare several types of algorithms, we will use the misclassification metric as our primary metric of viewshed accuracy.

4.3 Statistically robust performance measures

We are primarily interested in comparing algorithms in terms of how accurate they are, and how fast they execute. These data will be obtained by running the algorithms on a series of test cases, resulting in a set of error- and running time measurements for each algorithm. This section addresses how we should compare these results in a robust manner.

4.3.1 Comparison of accuracy

When comparing the accuracy of two algorithms on a test set, it might seem natural to just use the mean or median error to establish which algorithm is better. In the case of fig. 4.1a, however, the weaknesses of this method become apparent. Here algorithm A has a lower mean error than B, while the opposite is true for the median. We need a more robust way to compare algorithm performance.

The error produced by an algorithm can be viewed as a random variable, and we can therefore use standard statistic methods to test for the significance of a given test result. We will not assume that the error results have the normal distribution, but instead rely on theory that does not require this property.

Given test results for two algorithms from the same test set, the error measurements have a natural pairing on each test case. This enables us to use paired t-tests for deciding which algorithm is better. The t-test assumes that sample means have the normal distribution. Due to the central limit theorem, this is the case if we use sufficiently large samples.

The paired t-test provides us with a much more robust tool for asserting one of two algorithms as better. In the case of fig. 4.1a, using the one-sided paired t-test to see if A is better than B, the test fails with a p-value of 0.41. This means that assuming A is *not* in fact better than B, there would still be a 41% chance of observing the results we just did. In other words, the given results were not significant in showing that A is better than B. Similarly, asserting B as better fails with a p-value of 0.59.

Most of the algorithms we will compare, however, will *not* have similar performance. There will be no question which algorithm is better, but rather *how much* better it is. Once again it might be tempting to use the ratio of the mean or median errors to make claims like "algorithm A makes only $x\%$ of the error B does". Unfortunately, this is just as weak a result as using the mean or median to decide which algorithm is better. Instead we should one-sided confidence intervals to make sure we *underestimate* how good the better of the two actually is. This way, we avoid overoptimistic claims about the performance of the algorithms.

Specifically, the conservative estimate of the improved error can obtained as follows.

Let A_{err} and B_{err} be the error of algorithms A and B represented as random variables. Next, define

$$R = \frac{A_{\text{err}}}{B_{\text{err}}}$$
$$\mu_R = E\left(\frac{A_{\text{err}}}{B_{\text{err}}}\right)$$

Let $\{a_i\}_{i=1}^{n}$ and $\{b_i\}_{i=1}^{n}$ be two samples from A_{err} and B_{err} respectively, such that a_i and b_i come from the same test case. Next consider the mean of the paired ratio:

$$\overline{R} = \frac{1}{n}\sum_{i=1}^{n} \frac{a_i}{b_i}$$

Assume n sufficiently large. Then by the central limit theorem $\overline{R} \sim N(\mu_R, \frac{\sigma}{\sqrt{n}})$.

Thus

$$Z = \frac{\overline{R} - \mu_R}{\frac{\sigma}{\sqrt{n}}} \sim N(0,1)$$

This allows us to find an upper limit of μ_R with p confidence.

$$\begin{gathered} P(Z \geq z) = p \\ \Updownarrow \\ \frac{\overline{R} - \mu_R}{\frac{\sigma}{\sqrt{n}}} \geq \Phi^{-1}(1-p) \\ \Updownarrow \\ \mu_R \leq \overline{R} - \frac{s}{\sqrt{n}}\Phi^{-1}(1-p) \end{gathered}$$

Thus the upper limit for μ_R with p confidence is:

$$\hat{\mu}_R = \overline{R} - \frac{s}{\sqrt{n}}\Phi^{-1}(1-p)$$

We will use $\hat{\mu}_R$ as a measure of the ratio of the error made by two algorithms, A and B. If we see that A is more accurate than B, then we will say that the expected error ratio (EER) of A and B is $\hat{\mu}_R$ with p confidence. Of course, what we really should say is that *the expected error ratio of A and B is no larger than $\hat{\mu}_R$ with p confidence*, but this seems somewhat tedious. It is common to use confidence levels of 95% or 99%. Since we easily can generate as much test data as we need, we will use the 99% confidence level throughout this thesis. The EER claim will therefore almost certainly be an understatement, so in practice the observed error ratio will typically be lower. This conservative behavior is, however, exactly what we want.

A good example of this conservative behavior can be found in the data behind fig. 4.1b. In this case the mean relative error of algorithm A and B is 16.3% and 21.6%, respectively. From this one might jump to the conclusion that A makes only about 75% of the error of B. However, using the method above, we see that the EER of A and B is around 82%. Applying the method to the dataset behind fig. 4.1a this falls apart with an EER of A and B of more than 400%, giving us a clear indication that the test results are insignificant.

4.3.2 Comparison of running time

In contrast to the error measurements, running time measurements are not deterministic. Although paired on each test case, running time measurements are affected by the state of the computer at the time they were executed, which adds some noise to the results. This is, however, exactly what we aim to handle with the methods established for the accuracy measurements. The only effect of the noise added to the timing results is that the significance of the t-test and the EER become smaller and larger, respectively. This means that the results might be less impressive, but they are indeed valid.

4.4 Choosing observation points

As fig. 4.1 illustrates, the choice of test observation positions clearly affects the performance of the algorithms. To get an understanding of why this happens, consider fig. 4.2 which shows a 3D-rendering of the viewsheds of two observer points, fig. 4.2a uses one of the observer points from the randomized test, while fig. 4.2b uses one of the observer points from the scenario. Figure 4.2a illustrates an issue that arises surprisingly often when using randomly selected points; the viewshed becomes very small.

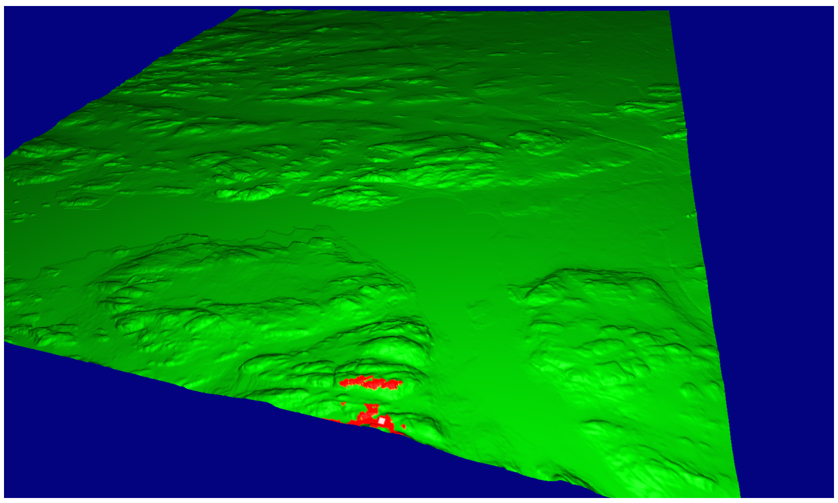
Some points in any given viewshed are trivial to classify correctly for most algorithms, because they are so obviously visible or invisible. If a viewshed consists mostly of such points, then it will fail to properly differentiate algorithms, which is exactly what happens in fig. 4.1a. It turns out that the majority of small viewsheds have this property, making them less suited for testing purposes.

4.4.1 Properties of trivial viewsheds

For performance comparison it seems to be good idea to avoid observers with small viewsheds. We will now try to understand exactly what it is about small viewsheds that make them unsuitable for benchmarking viewshed algorithms.

The number of points in the viewshed is the same as the number of points from which the observer in question is visible, and is a measure of how *visible* the observer is in general. The size of the viewshed of a point, and sometimes also the relative size, is referred to as the *visibility index* in the literature.

Consider first an observer situated in a deep pit, like the one in fig. 4.3. The points around the perimeter of the pit, $\mathbf{b}_1$ and $\mathbf{b}_2$ in the figure, force the horizon up very close to the observer, which means that most of the


(a) Randomly chosen observer

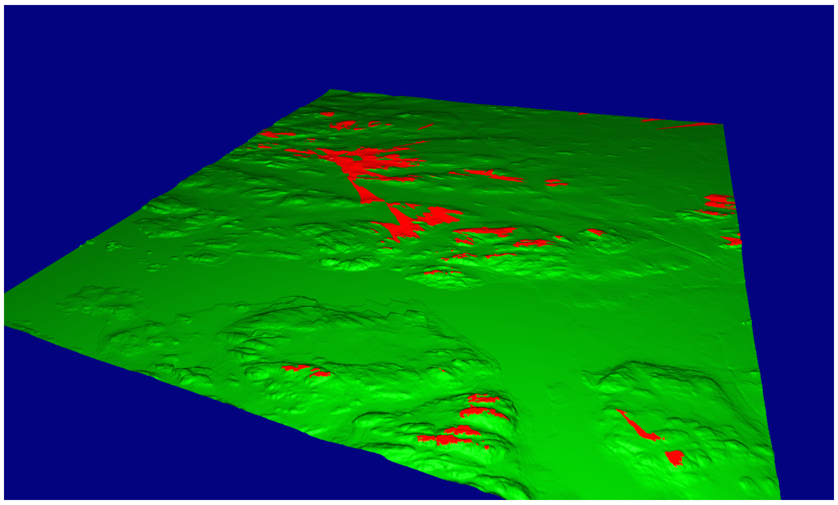

(b) Observer from scenario

Figure 4.2: R2-estimated viewsheds of two selected observer points from the tests. Points that are visible to the observer are colored red, and the invisible points are colored green.

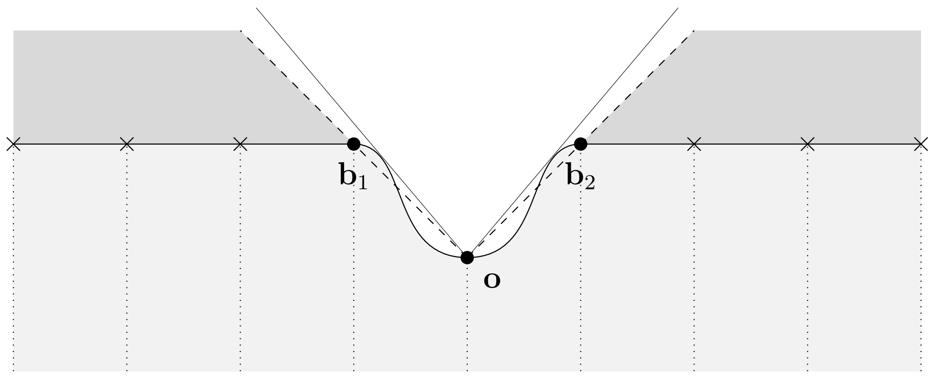


Figure 4.3: Observer in a pit. The solid lines depict the actual horizon, while the dashed lines show the horizon as estimated by R2 and XDraw.

remaining points in the terrain will be classified as invisible and do not affect the horizon any further. As soon as the difference between the horizon and the slope of the remaining points on the terrain becomes large, the error in the estimate of the horizon produced by algorithms such as R2 and XDraw is negligible. In fact, the only points in this figure that risk being misclassified are the points on the perimeter of the pit, $\mathbf{b}_1$ and $\mathbf{b}_2$. In general this means that except for a very small number of points, the viewshed is trivial for most algorithms to calculate, which allows minor details in the terrain to tip the conclusion of benchmark tests in either direction.

Next consider an observer situated on top of a hump above an otherwise flat terrain, as illustrated in fig. 4.4. In this case the horizon is strictly increasing with distance from the observer. Here the visibility index is 100%, but the viewshed can still be trivially determined by most algorithms. It is important to point out that real-world terrains seldomly contain points with a ground-level visibility index of 100%. Even a visibility index of 50% is rare. However, many applications use the elevated viewshed, where the observer is situated at some height above the ground. For instance when working with aircraft, this height can be substantial. In these cases highly visible observers with trivial viewsheds are much more common.

4.4.2 A method for finding hard viewsheds

To avoid observers yielding this type of trivial viewsheds, Ray suggests in [Ray94] as a rule of thumb that observers should be placed at points with high visibility index. This, in combination with some sampling pattern that ensures observation points that are spread out across the terrain, should form

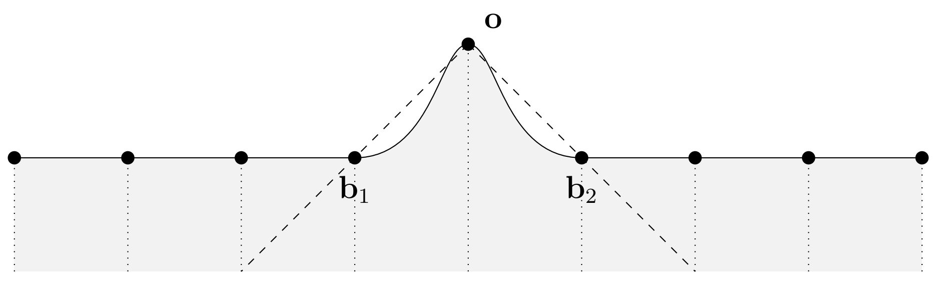


Figure 4.4: Observer on a hump. The dashed lines show the horizon at $\mathbf{b}_1$ and $\mathbf{b}_2$ as estimated by R2 and XDraw.

a good basis for evaluating algorithm performance. Ray suggests to do this by dividing the terrain into a coarse grid of cells, and then select the point with the highest visibility index within each such cell. As fig. 4.4 illustrates however, this is not always enough, because in some applications many of the highly visible observers have trivial viewsheds nevertheless.

Looking at fig. 4.3 and fig. 4.4, we see that it is typically in the transition from visible to invisible or vice versa that the estimation accuracy of these algorithms is put to the test. The points that are situated well inside or outside the viewshed and thus far away from a visibility transition are typically much easier to classify. Additionally, the interpolation techniques used in the various algorithms typically struggle in areas where the viewshed is inhomogeneous. This suggests that we should look for observers with viewsheds that have complex shapes, and that ideally have several visibility transitions along any ray.

To meet these criteria we shall use a similar approach to that of Ray. But instead of using the area of the viewshed, we will use the *circumference*, i.e. the area of the boundary. This preserves the first criterion of having a large viewshed, since small viewsheds also have short boundaries. It also preserves the criterion of a complex viewshed, as viewsheds with large homogeneous areas have relatively short boundaries. Once we have a relatively complex viewshed we also have several visibility transitions along most rays.

In general our procedure for selecting n test observation points is then as follows:

1. Divide the terrain into n cells
2. Select the point in each cell which has the largest viewshed circumference

Following this procedure using the R3 algorithm for finding the circumference of the viewshed for each point in a terrain is typically not feasable. A much more practical approach is to randomly select k points in each cell, and choose the best fit from that selection. Since we are interested in finding points that make for *reasonable* test candidates, it is also perfectly acceptable to use a fast algorithm like XDraw for finding the viewshed circumferences.

We will sketch this as an algorithm.

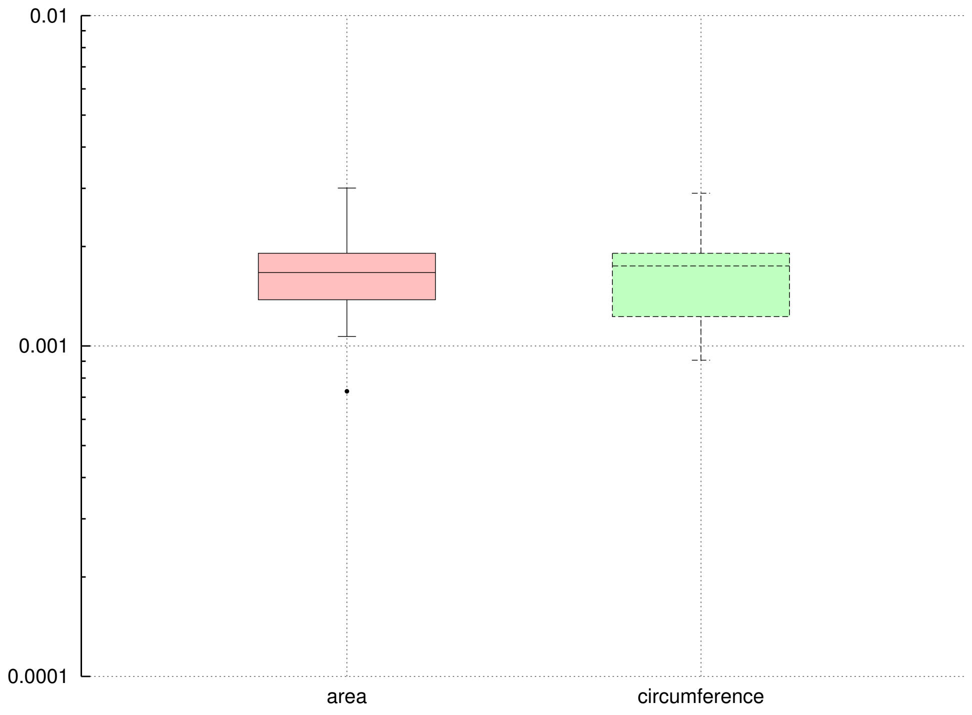
Algorithm 4.1 (Finding observation points with hard viewsheds).

```
Let $n$ be the number of observation points
Let $k$ the number of viewshed evaluations per observation point

Divide the terrain into $n$ cells

for all such cells $c$
    Select $k$ points randomly within $c$
    Use the point with the largest estimated circumference as an observation point
```

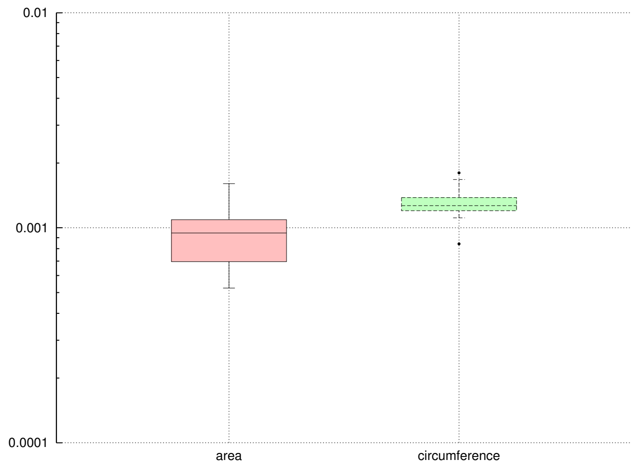
Figure 4.5 illustrates the effect of using this method compared to that proposed by Ray. Each plot shows the error made by R2 on 16 observation points chosen using the two methods at various observer and target heights. For both methods the terrain is divided into 16 cells, then 16 points are randomly selected in each cell, and the one with the largest viewshed area and circumference is selected for the two methods respectively. Note that the selection of the 16 points within each cell is the same for both methods.

As we see in fig. 4.5a, the difference between the two methods is insignificant for the low observer/target height at 3m. At an observer- and target height of 20m, as shown in fig. 4.5b, we see indications that the error at the observers chosen using the circumference method is higher. At 50m, as shown in fig. 4.5c, the difference is clearly significant with the viewsheds of the observation points selected using the circumference method being considerably harder to classify.

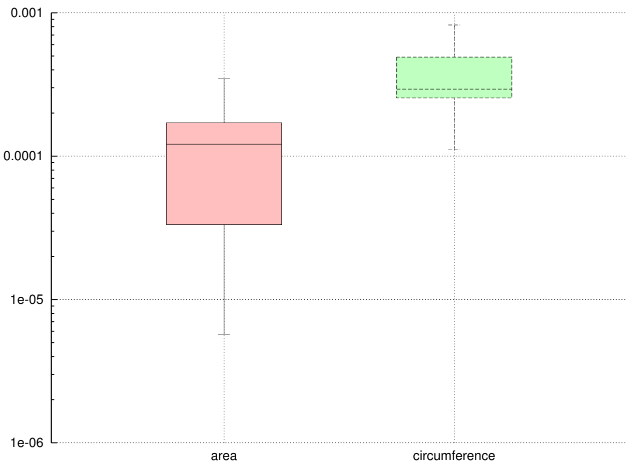
These test results are consistent with the observation that the situation illustrated in fig. 4.4 seldomly occurs in real world terrains unless a high observer height is used. In the case of low observer heights the circumference method seems to perform at least as good as the visibility index method. In terms of computational complexity calculating the circumference of the viewshed is analogous to calculating the area of the viewshed.



(a) Observer- and target height of 3m



(b) Observer- and target height of 20m



(c) Observer- and target height of 50m

Figure 4.5: The relative error of R2 on 16 observers chosen with maximum viewshed area and circumference, at three different observer- and target heights. Higher means means more difficult test cases, which is what we try to achieve.

4.4.3 A method for finding average viewsheds

It is not always meaningful to only use the points with the hardest viewsheds. Therefore we would like to have a method for finding observation points with viewsheds of arbitrary difficulty. Generalizing the method for finding hard viewsheds, we can instead of choosing the point with the *largest* viewshed circumference in each cell, pick the one closest to some percentile.

Algorithm 4.2 (Finding observation points with average viewsheds)**.**

```
Let $n$ be the number of observation points
Let $k$ the number of viewshed evaluations per observation point
Let $i$ be some percentile

Divide the terrain into $n$ cells

for all such cells $c$
    Select $k$ points randomly within $c$
    Use the point nearest the $i$th percentile wrt. estimated circumference
as an observation point
```

4.4.4 Assembling a complete testset of observation points

If the goal of the benchmark is to establish the average performance of the algorithm in some sense, then the selection of observation points should reflect the population of observation points in the intended application. This is rarely the case for some selection consisting only of observation points with hard viewsheds. A reasonable testing population of observation points for a general average performance benchmarking should therefore contain meaningful portions of each of the following categories of points:

- Points with medium viewsheds
- Points with hard viewsheds
- Points that are typical for the application

Here, the medium and hard points can be found using algorithm 4.2 around the 50th and 100th percentile respectively. The application-specific points must be found by hand.

Remember that points with trivial viewsheds in general should be avoided. As discussed they are rarely able to differentiate most of the algorithms in any meaningful manner, and thus do not contribute to the accuracy of the benchmarking. Even in applications that involve finding points with low visibility index, points with trivial viewsheds should not be leveraged in algorithm benchmarking.

Emphasis should be put on the points that are typical for the application. But points with medium and hard viewsheds are important for establishing a baseline of viewshed algorithm performance, as well as for uncovering flaws in algorithm implementation that might be present.

4.4.5 The role of the observer- and target height

The difficulty of a viewshed also seems to be affected by the observer- and target height. From fig. 4.5 it appears as if the difficulty decreases when the observer- and target height is increased. This is consistent with our discussion of fig. 4.4, since a large observer height mimics a situation where the observer is on the top of a narrow peak. Similarly, a very low observer height to some extent mimics the situation in fig. 4.3, where the observer is down in a pit. This might suggest that extreme observer- and target heights make it more difficult to find good observation points.

Although the results clearly are affected by the choice of observer- and target height, the effect is much smaller than what we saw when choosing observers randomly. As long as these heights are kept within a reasonable range, this does not seem to compromise the benchmarking result. There is, however, no reason to use extreme observer/target heights unless it is required by the application.

4.5 Choosing test terrain

As discussed, both the choice of test terrain and observation points can affect the benchmark results in ways that can be hard to predict. In general this means that it is always recommendable to choose terrains that mimic those encountered in the actual application. However, some types of artificial terrains are interesting from a theoretical point of view, and can provide valuable insights when analyzing viewshed algorithm behavior.

4.5.1 Synthesizing a hard terrain

In an attempt to get an impression of how much the terrain affects the performance of viewshed algorithms, we will try to create a terrain that is especially difficult to classify using approximate methods. To do this we will analyze the R2 algorithm, and look at some situations where it performs particularly bad. Hopefully the resulting terrain will also be somewhat hard to classify for the other algorithms as well.

As we know, R2 works by sending a line of sight (LOS) to each point on the perimeter of the terrain. Along each LOS we obtain the accurate horizon, and therefore also an accurate classification for all grid points that intersect any of these LOSs. Erroneous classifications can only occur at grid points that do not fall on any LOSs, since the horizon at these is estimated using the nearest LOS.

Figure 4.6 illustrates this situation, where a point **p** falls between two adjacent LOSs, as shown in fig. 4.6a. When using the FRM terrain model we know that the horizon at **p** depends only on the grid lines within the gray sector of fig. 4.6a. We can now plot the slope of each point on these lines as a function of azimuth. This is done in fig. 4.6c and fig. 4.6b for two different terrains. The image of a line under this projection is a smooth curve that typically is *almost* linear. The upper envelope of these piecewise smooth functions is drawn in bold red on the plots. From corollary 3.7 we know that this is the horizon of the points along the red arc in fig. 4.6a.

The dashed lines in fig. 4.6 indicate the nearest neighbor estimate R2 obtains for each point. As the plots show, this can cause R2 to both over- and underestimate the horizon, which can potentially lead to classification errors. These estimates seems to be worse when the horizon is complex and irregular. Intuitively this occurs more frequently in terrains with large numbers of narrow spikes. Large changes in elevation tend to reduce this effect, however. This is illustrated in fig. 4.6d, which shows an horizon plot of the same terrain as fig. 4.6c, but extending all the way to $\mathbf{p}'$ and the blue line in fig. 4.6a. Here, one of the grid lines between **p** and $\mathbf{p}'$ completely dominates the horizon, in effect "resetting" the complexity of the horizon. Without these large elevation changes, the complexity of the horizon accumulates with distance from the observer.

In order to maximize the chance of misclassifications, the terrain should therefore contain a large number of spikes, but overall no large elevation changes, such as a hill. Gaussian white noise has both of these properties, and as we shall see, it is a good basis for generating difficult terrains. It should be duly noted that since the white noise terrain is specifically designed to be difficult for R2, it is useless for making unbiased comparisons. We will, however, use it for illustrating the difference in performance between easy and difficult terrains.

4.5.2 The effect of different terrain types

In order to illustrate the effect different terrain types has on the performance of some of the algorithms, consider fig. 4.8. The figure shows the error of R2, radar and XDraw run on the terrain near Larvik, Alta and on the white noise terrain.

Figure 4.7 shows a 3D rendering of the Larvik and Alta terrains. As the figure illustrates, these terrains are quite different. The Larvik dataset has a total elevation range of 107m, and is overall relatively flat. The Alta dataset has an elevation range of 875m, due to several high mountains. The white noise terrain is as we know extremely rough with lots of sharp peaks. In this test we have used Gaussian noise with $\mu = 0$ and $\sigma = 0.5$. This means that approximately 95% of all points lie in the range -1m to 1m.

Considering the overall performance of the three algorithms, Larvik and Alta seem to be of comparable difficulty. Looking at the white noise terrain, on the other hand, all three algorithms seem to perform significantly worse. Perhaps even more interesting is that the algorithms are affected differently by the different terrains. On Larvik, the radar-like algorithm performs clearly better than XDraw, while on the white noise terrain, XDraw is clearly better. Similarly R2 is better than radar by a margin on Larvik, but on Alta the difference is much less obvious.

From this we can only conclude that some algorithms perform better on certain types of terrain, and that the effect is not always easy to predict. Therefore it is important to test algorithms on terrain that is similar to that they will face when in use.

4.6 Benchmarking running times

Benchmarking the running time of viewshed algorithms require much of the same consideration for picking test data as the accuracy testing. Some algorithms, like R3 and the non-fixed version of the radar-like algorithm, have

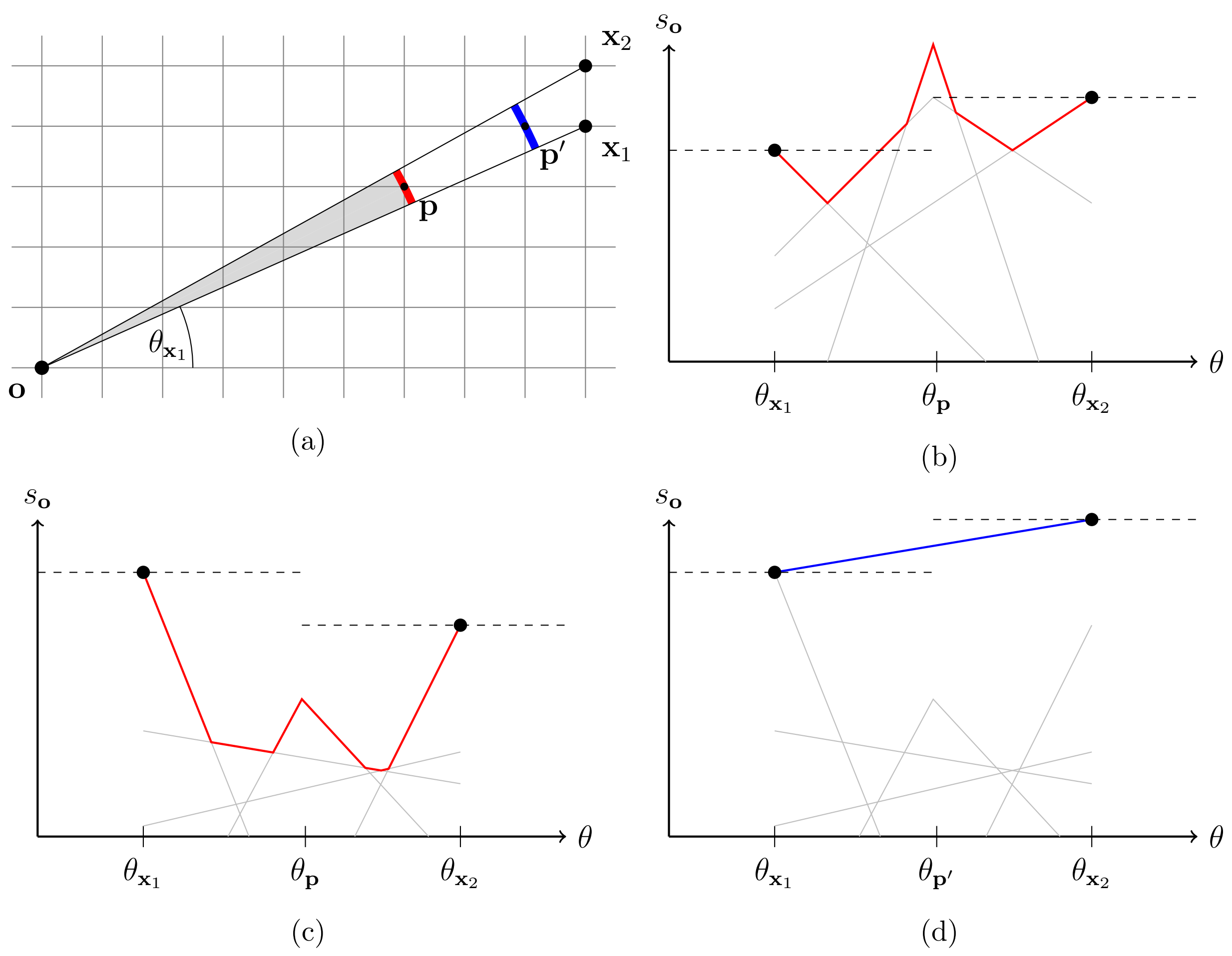
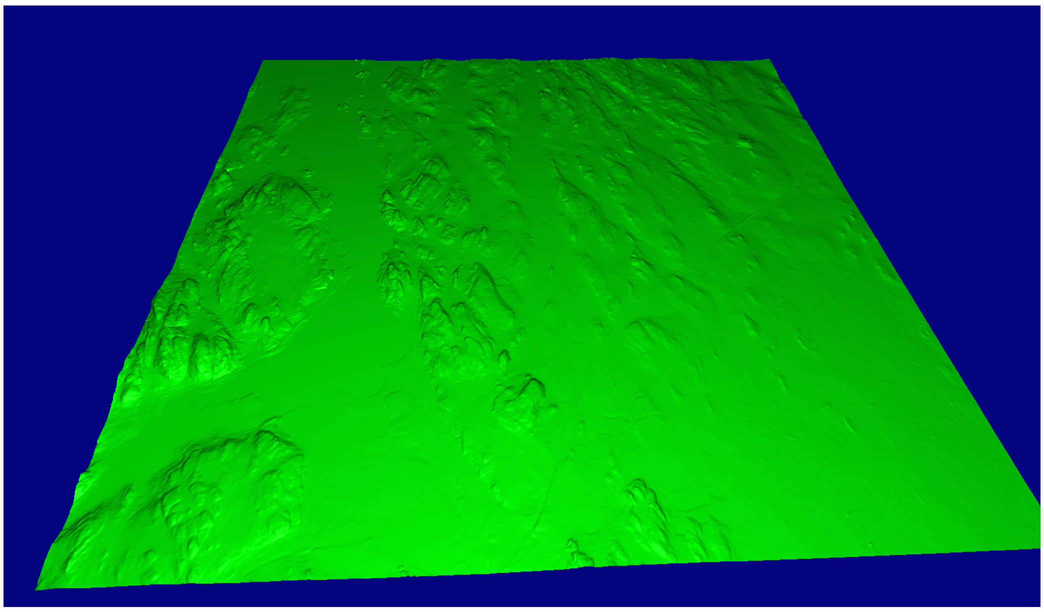
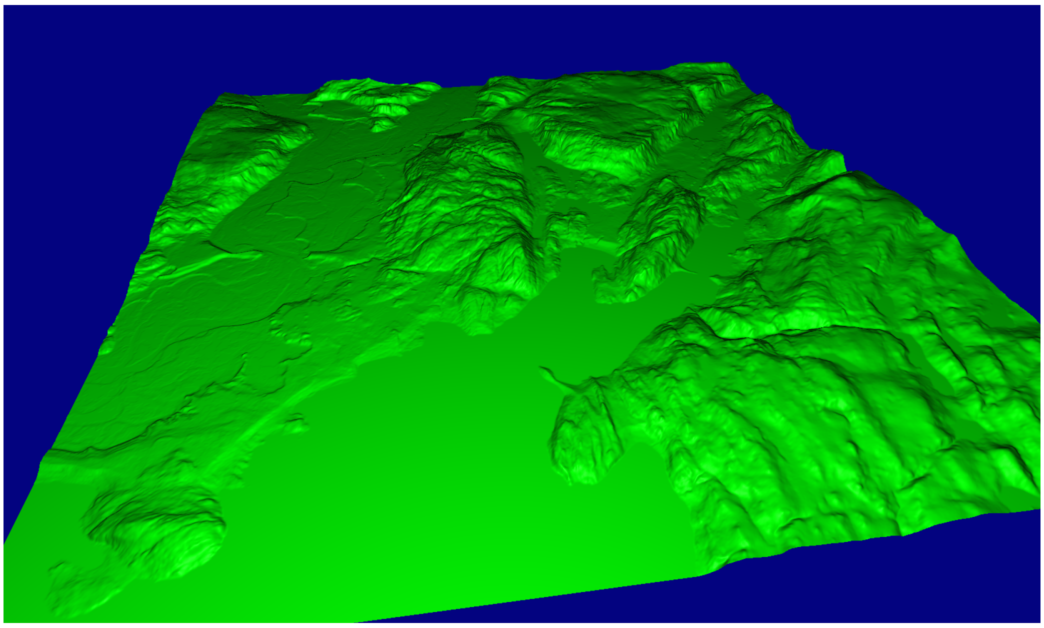


Figure 4.6: Overview of a situation that typically leads to errors in R2-type algorithms is illustrated in fig. 4.6a, where the point $\mathbf{p}$ lies far away from the nearest ray. Figure 4.6b and fig. 4.6c plots some of the grid lines in the grey sector of fig. 4.6a, for two different terrains. Figure 4.6d is a plot of the same terrain as in fig. 4.6c, but also includes some of the grid lines from the sector between $\mathbf{p}$ and $\mathbf{p}'$. The slope of the points on the lines, as defined in definition 3.5, is plotted as a function of azimuth. The upper envelope of the projections, plotted in bold color, is thus the horizon of the points along the red and blue arcs in fig. 4.6a respectively. The dashed lines show the estimated horizon as obtained by R2 using nearest neighbor interpolation.

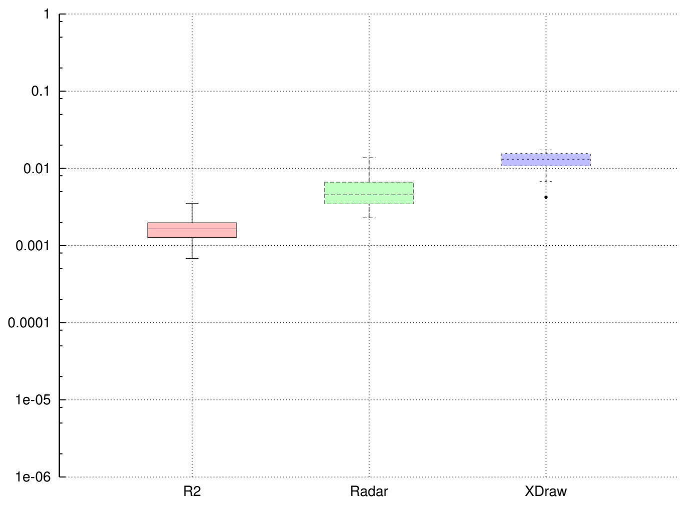


(a) Larvik

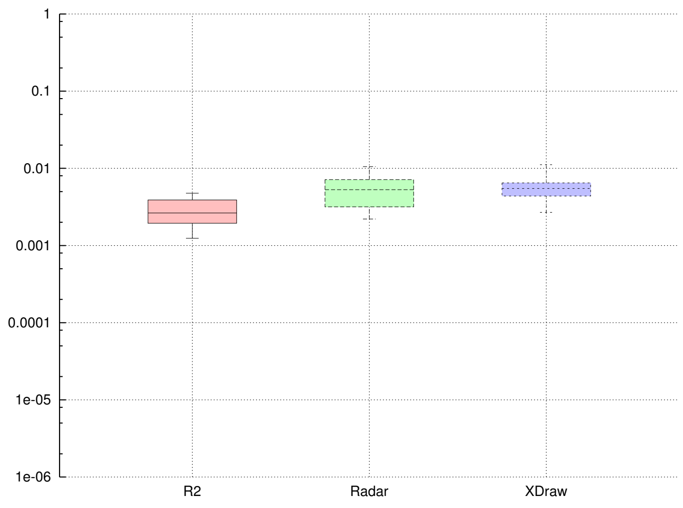


(b) Alta

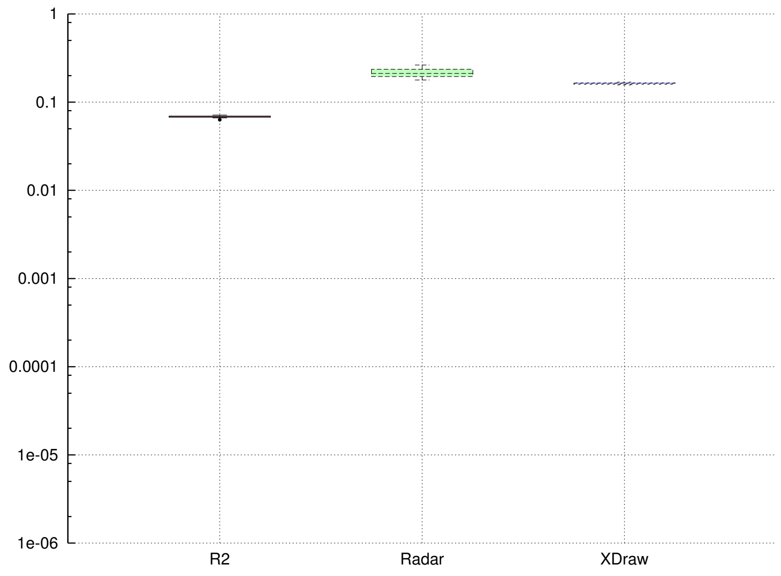
Figure 4.7: 3D renderings of the Larvik and Alta data sets.



(a) Larvik



(b) Alta



(c) White noise

Figure 4.8: Comparison of terrains. The plots show the relative error obtained by the R2, radar-like and XDraw algorithms on the Larvik, Alta and white noise terrains (lower is better). 16 observers are chosen using algorithm 4.2 with $n = k = 16$, and $i = 100$. The tests are run with an observer- and target height of 1m.

an obvious running time dependency on the complexity of the terrain and/or resulting viewshed. Other algorithms like R2 and XDraw do not. Therefore it is key to test using terrains and observation points that yield realistic viewsheds. This is achieved by following the procedure discussed in the previous sections.

4.7 Evaluation of existing algorithms

We will now put the above theory into practice by benchmarking a set of algorithms for use in the military scenario discussed in chapter 2. For this we need to create a suitable test setup by selecting terrain data and observation points.

The goal of these tests are two-fold: We want to determine which algorithm works best for this type of application. Also, we want to compare the results of using observation points that are hand-picked by experts to those selected automatically.

4.7.1 Terrain data

We want to find an algorithm that performs well on a variety terrain types that can be found in Norway, ranging from the flat-land to mountainous regions. Therefore we will perform the tests on both the Larvik and Alta datasets.

Both terrains are represented as RSGs of size 1024×1024 and a vertical resolution of $1m$. The Larvik dataset has a horizontal resolution of approximately $4.9m$, while the Alta dataset has a resolution of $15m$.

4.7.2 Observation points

Looking back to the military background from chapter 2, we are particularly interested in two types of viewsheds: We want to know where we can observe an enemy from, and where we can attack it from.

Assume that both the sights and the highest visible point on the enemy are situated 3m above ground. Then the points we can observe from are given by the viewshed with the enemy as observer, and both an observer- and target height of 3m. Similarly, assume that both the barrel and the highest critical point on the enemy are situated 1m above the ground. Then we can attack from any point in the viewshed with the enemy as observer, and an observer- and target height of 1m. These heights are chosen somewhat arbitrary as they depend on vehicle type, but the principle remain clear.

Since all relevant observers in this scenario are enemy positions, we should leverage points where an enemy unit typically would be positioned, when benchmarking the algorithms.

The enemy positions are typically situated in such a way that they are only vulnerable to attack from few directions, while maintaining a good view. The enemies never position themselves such that they can be seen from far away in many directions at once. This means that the points with the largest viewsheds are unlikely choices in this context. The viewsheds as described here will be medium in size, with a few narrow "fans" extending in the directions the unit is set to watch over.

According to the method established above we should therefore test with a selection of observers where the majority have medium difficult viewsheds. We should include hand-picked observers that are known to be typical, and we should also include some difficult viewsheds. Since the hardest viewsheds don't occur in the application, we will choose the hardest in the test at the 80th percentile instead of the 100th. Additionally, experts will select some points that are typical positions that enemies will take. These will make up the hand-picked portion of the test set.

In summary we will use the following recipe for choosing observers:

- 9 hand-picked observers with height 1m
- 9 hand-picked observers with height 3m
- 9 observers with height 1m, from 50th percentile
- 9 observers with height 3m, from 50th percentile
- 9 observers with height 1m, from 80th percentile

4.7.3 Algorithms

The algorithms we will be testing are some of the ones discussed in chapter 3:

- R2

- XDraw interpolated
- XDraw maximized
- XDraw minimized
- XDraw averaged
- Fixed radar-like

The XDraw variants are obtained using the corresponding estimators described in eq. (3.4).

Throughout the tests we will be using the fixed version of the radar-like algorithm proposed by Ben-Moshe et al., although we will refer to it simply as *the radar algorithm*.

The R2 and XDraw implementations are optimized for speed to a comparable level. The implementation of the radar algorithm, however, is not optimized to the same level. This algorithm can evaluate an adjustable number of LOSs. Since R2 and radar have relatively similar modes of operation we will set the radar algorithm to evaluate the same number of LOSs as R2 does, and consider them as comparable in terms of speed.

4.7.4 Expectations

The results of Franklin et al. indicate that the interpolated version is the most accurate of the XDraw variants. [Ray94] and [Izr03] consider R2 to be significantly more accurate than XDraw. Therefore it is natural to expect that we will see the same trends.

No known prior comparisons of R2 and the radar algorithm have been made. Except for the interpolation techniques used, these algorithms have a relatively similar mode of operation, so it seems reasonable to assume that they should have similar performance.

As for the case of running times the results of Franklin et al. indicate that XDraw is significantly faster than R2, sometimes as much as an order of magnitude. As discussed, the radar implementation has not been optimized as much for speed, so it should be expected to run considerably slower than the other algorithms.

Algorithm	Median rel. err.	1st quartile	3rd quartile	Mean run. time
R2	8.0×10^{-4}	5.4×10^{-4}	1.1×10^{-3}	107.6 ms
Radar	1.6×10^{-3}	1.0×10^{-3}	2.5×10^{-3}	327.2 ms
XDraw interp.	6.1×10^{-3}	3.2×10^{-3}	1.0×10^{-2}	32.8 ms
XDraw max	4.8×10^{-2}	3.4×10^{-2}	8.0×10^{-2}	31.1 ms
XDraw mean	4.4×10^{-2}	2.5×10^{-2}	6.7×10^{-2}	31.4 ms
XDraw min	1.6×10^{-1}	1.0×10^{-1}	2.5×10^{-1}	31.2 ms

Table 4.1: Larvik, full set of observers

Algorithm	Median rel. err.	1st quartile	3rd quartile	Mean run. time
R2	1.3×10^{-3}	8.3×10^{-4}	1.9×10^{-3}	117.3 ms
Radar	1.6×10^{-3}	1.0×10^{-3}	2.6×10^{-3}	342.7 ms
XDraw interp.	4.1×10^{-3}	2.7×10^{-3}	5.8×10^{-2}	35.6 ms
XDraw max	6.8×10^{-2}	3.4×10^{-2}	7.4×10^{-2}	32.7 ms
XDraw mean	3.8×10^{-2}	2.9×10^{-2}	4.8×10^{-2}	33.9 ms
XDraw min	1.1×10^{-1}	8.5×10^{-2}	1.5×10^{-1}	33.3 ms

Table 4.2: Alta, full set of observers

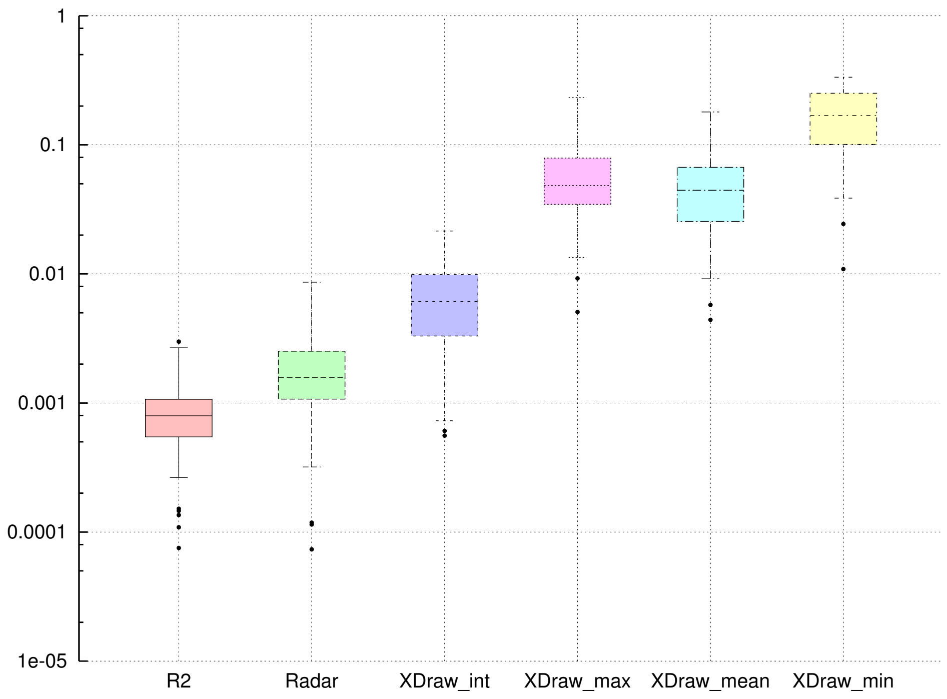
4.7.5 Results

We first ran the tests using the full set of observation points as described above. The results are shown in table 4.1 and table 4.2. Figure 4.9a and fig. 4.9b show the relative error of the full test as a boxplot.

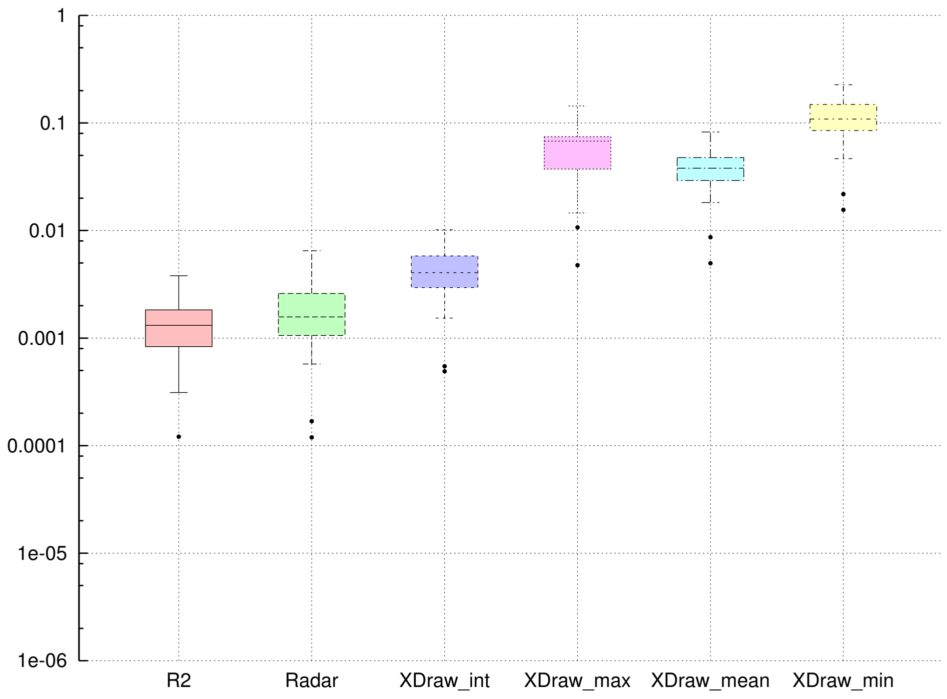
We also ran a test using only the hand-picked observation points for each terrain. The results can be seen in tabular form in table 4.3 and table 4.4. Boxplots are shown in fig. 4.10a and fig. 4.10b.

Algorithm	Median rel. err.	1st quartile	3rd quartile	Mean run. time
R2	7.1×10^{-4}	1.5×10^{-4}	1.1×10^{-3}	117.0 ms
Radar	1.5×10^{-3}	3.2×10^{-4}	2.1×10^{-3}	370.9 ms
XDraw interp.	6.1×10^{-3}	2.3×10^{-3}	1.2×10^{-2}	35.1 ms
XDraw max	4.0×10^{-2}	1.4×10^{-2}	7.5×10^{-2}	32.6 ms
XDraw mean	3.8×10^{-2}	1.4×10^{-2}	6.4×10^{-2}	34.7 ms
XDraw min	2.0×10^{-1}	8.2×10^{-2}	2.5×10^{-1}	33.2 ms

Table 4.3: Larvik, hand-picked observers

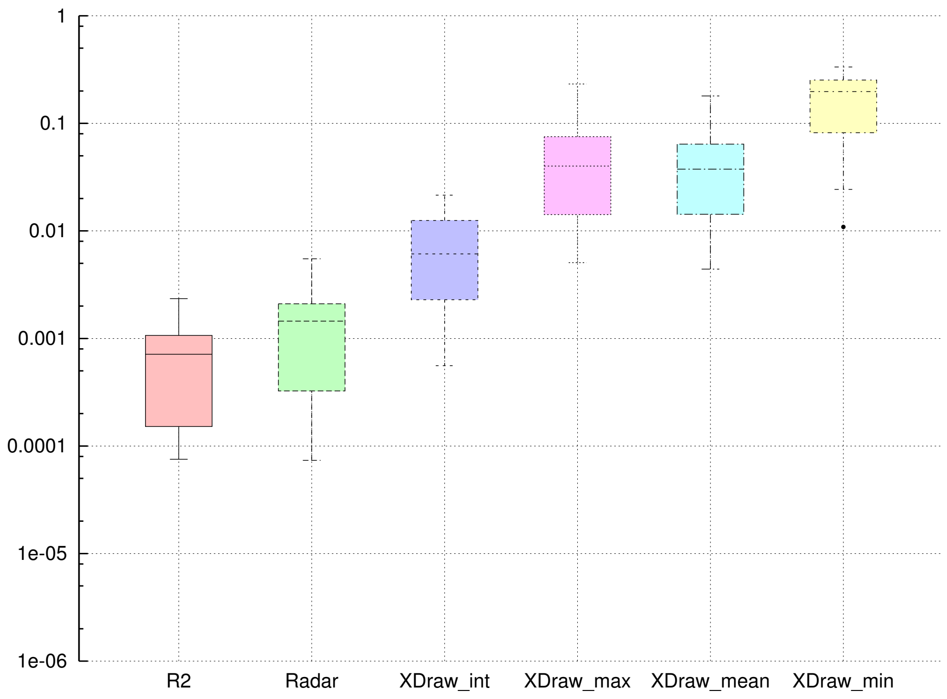


(a) Larvik

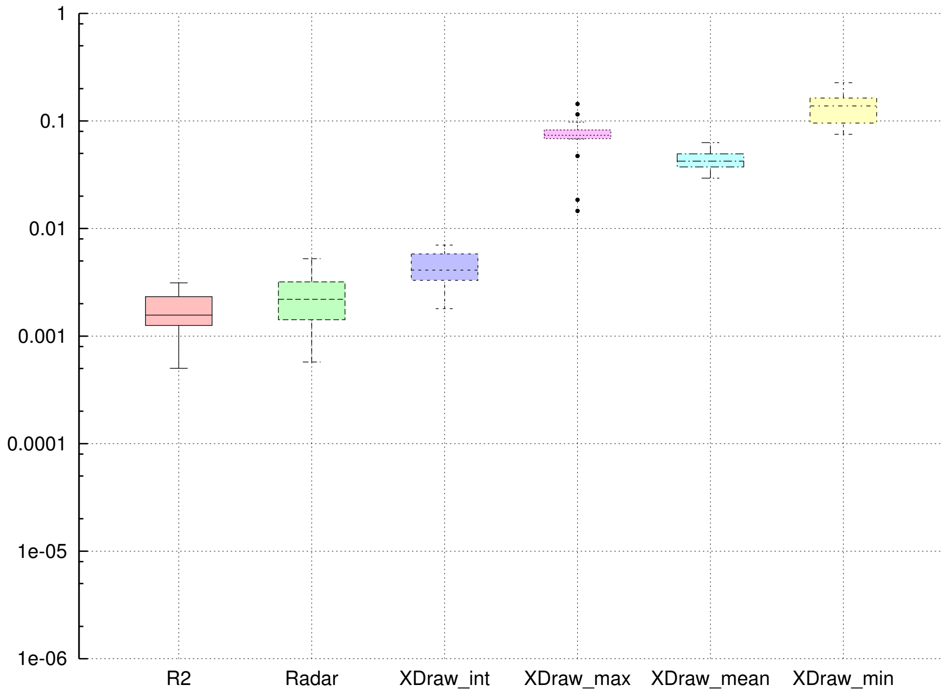


(b) Alta

Figure 4.9: Comparison of the relative error of some existing algorithms (lower is better). Results are obtained using the full set of observers described in section 4.7.2.



(a) Larvik



(b) Alta

Figure 4.10: Comparison of the relative error of some existing algorithms (lower is better). Results are obtained using only the hand-picked set of observers described in section 4.7.2.

Algorithm	**Median rel. err.**	**1st quartile**	**3rd quartile**	**Mean run. time**
R2	1.6×10^{-3}	1.3×10^{-3}	2.3×10^{-3}	116.5 ms
Radar	2.2×10^{-3}	1.4×10^{-3}	3.2×10^{-3}	339.7 ms
XDraw interp.	4.1×10^{-3}	3.3×10^{-3}	5.8×10^{-3}	35.4 ms
XDraw max	7.4×10^{-2}	6.8×10^{-2}	8.2×10^{-2}	32.7 ms
XDraw mean	4.2×10^{-2}	3.7×10^{-2}	5.0×10^{-2}	32.3 ms
XDraw min	1.4×10^{-1}	9.6×10^{-2}	1.6×10^{-1}	33.4 ms

Table 4.4: Alta, hand-picked observers

4.7.6 Verifying the implementations

Since several similar experiments have been made by others, it is natural to compare our results to the available data. In order to verify the correctness of our implementations we should expect that our results are not drastically worse than others, unless differences in the testing procedures should warrant weaker results.

XDraw

The authors do not provide measurements of the accuracy of XDraw in the original article [FRM94]. Fortunately, Izraelevitz proposes a variant of interpolated XDraw in [Izr03], in which he includes empirical tests comparing it to the original algorithm. The tests consist of 16 samples, obtained using uniformly spaced observers on a terrain data set that seems comparable to our Alta set in terms of resolution and terrain type.

The results obtained by Izraelevitz match ours very well. In his tests the relative error lies in the range of about 1×10^{-3} - 1×10^{-2}. Our results for interpolated XDraw lie in the range of about 5×10^{-4} - 1×10^{-2}. Thus there is no reason to suspect our implementation is any less correct than that of Izraelevitz.

The difference in code between the other variants of XDraw is trivial, as can be seen in e.g. listing A.18. It is therefore unlikely that these have errors the interpolated version does not have.

R2

Franklin et al. only briefly describes the relative error of R2 in [FRM94], as their primary concern is running time. The full details of their testing procedure used are somewhat difficult to make out, but [Ray94] contains a slightly more thorough description.

The terrain used in the tests is referred to as *the south-west quadrant of DTED level 1 cell N37E127*. This is a terrain data set centered at 37°N and 127°E, which is in the mainland of South Korea. DTED level 1 terrain is sampled at a resolution of 3 arc secs, which corresponds to about $83m$ in this area. The test is executed by choosing a single observer close to the center of the south-west quadrant. The observer is chosen to have high visibility, which means that it should have a *difficult* viewshed as we have previously discussed. There is no description of the observer or target height used.

In the test R2 has a relative error of 1.2%, which is significantly worse than any of our real-world terrain tests. Therefore there is no reason to believe that our implementation is any less correct than that of Franklin et al. However, as we have seen, the test terrain can significantly impact the performance of the algorithms. We should therefore investigate this closer.

Unfortunately, DTED cell N37127 does not seem to be openly available anymore. However, the USGS GTOPO30 model covers the desired area with $30m$ resolution, and can be obtained freely through the USGS earth explorer. We can then produce a data set of a region to the south west of 37°N and 127°E with $83m$ resolution using interpolation and decimation on the GTOPO30 data set. The resulting region has a size of 600×600 points, matching fairly well the region described in [Ray94]. A 3D rendering of parts of this terrain can be seen in fig. 4.11. This illustration provides an immediate explanation for the poor results of Franklin et al. The terrain has a large amount of spikes, possibly an artifact due to the low resolution, making it resemble the white noise terrain. From what we have discussed earlier, it should therefore come as no surprise that the algorithms perform worse on this terrain.

A full test of R2, among other algorithms, on this terrain can be found in chapter 5. The worst samples have a relative error of about 0.8%, which is not too dissimilar to that seen by Franklin et al. We therefore consider our implementation of R2 to be validated.

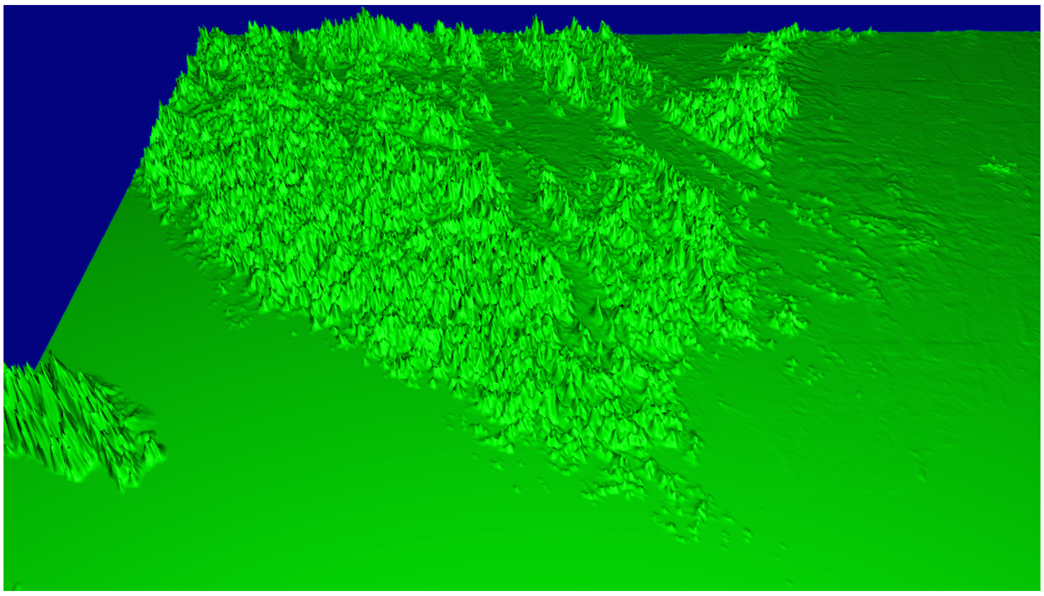


Figure 4.11: Parts of the terrain from DTED cell N37E127

The radar-like algorithm

The only source of empirical data for the radar-like algorithm is the original article by Ben-Moshe et al. [BMCK08]. These results are based on a triangulated irregular network (TIN) terrain model, making it close to impossible to reproduce the results with similar LOS density and terrain characteristics. Additionally, since the radar-like algorithm is able to classify *regions* and not only points, the provided results give relative error of the viewshed *area*, which is not easily translated to our error measures.

For the radar-like algorithm we must therefore rely on hand-crafted test cases for validating the implementation.

4.7.7 Evaluating the results

The trend in all four tests is that R2 is more accurate than radar, which in turn is more accurate that XDraw. The interpolated version is by far the most accurate of the XDraw variants, and is the only of the XDraw algorithms that is comparable to R2 and radar in terms of accuracy. This is consistent with our predictions based on earlier work.

We must keep in mind, however, that we here use the fixed version of the

	XDraw mean	**XDraw int**	**Radar**
XDraw int	1.4×10^{-1}		
Radar	5.0×10^{-2}	4.3×10^{-1}	
R2	3.3×10^{-2}	3.0×10^{-1}	7.8×10^{-1}

Table 4.5: EER analysis of four existing algorithms. The cell in column i and row j contains the EER of algorithms i and j with 99% confidence. Blank cells indicate that the test to show that algorithm i is significantly better than algorithm j failed.

radar-like algorithm. As we know, the non-fixed version of radar typically obtains a lower error rate for a given number of LOS evaluations. Therefore it seems fair to conclude that R2 and non-fixed radar are roughly comparable in terms of accuracy for this use, albeit with a slight edge to R2.

Turning to running time the radar algorithm is by far the slowest, as expected. As discussed this is an implementation that is not optimized for speed, so this result should not be emphasized. An interesting point is that the running time of the interpolated version of XDraw is negligibly slower than the other versions. This in contrast to the results of Franklin et al. in [FRM94], which suggest that the interpolated version should be considerably slower. Based on these results there is no reason to choose the averaged version over the interpolated version of XDraw.

Comparing the R2 algorithm to interpolated XDraw it is clear that R2 is significantly more accurate, while XDraw is significantly faster. In time-critical applications requiring real-time performance, XDraw might therefore be only option. If this is not the case, R2 should generally be preferred.

Combining the full Larvik and Alta data sets, we can use EER from section 4.3 to further analyze the data. Since the max- and min-variants of XDraw are not of particular interest in this context, we focus on the averaged and interpolated variants, in addition to radar and R2. The result can be seen in table 4.5.

The EER analysis shows that interpolated XDraw makes less than 14% of the error averaged XDraw does. R2 and radar are more than 3.3 and 2.3 times as accurate as interpolated XDraw, respectively. Finally, we also see that R2 makes less than 78% of the error radar does. Although we expect the non-fixed version of radar to perform better, it seems reasonable to prefer R2 for our application.

Chapter 5

Improving the R2 algorithm

Based on the results obtained in chapter 4 the R2 algorithm stands out as a good choice in terms of accuracy for our application. In this chapter we aim to improve the R2 algorithm, making it even more suitable for our application. First we conduct a more thorough analysis of some of the weaknesses of R2, and propose how to mend them through simple modifications of the original algorithm.

Next, we propose a more flexible variant of R2, inspired by the radar-like algorithm. We want to be able to increase the accuracy of R2 on demand, in exchange for increased running time, as we can with radar. A natural way to achieve higher accuracy in R2, while accepting longer running times is to evaluate more lines of sight (LOSs). By letting the number of evaluated LOSs be a parameter of the algorithm, we get the adjustable behavior we want. In this chapter we analyze the properties of this algorithm, and investigate further where the extra LOSs should be sent in order to increase the potential of correcting errors.

5.1 R2 side slope performance

By carefully comparing some of the test results from chapter 4, it seems that R2 performs worse compared to the other algorithms on the Alta terrain than Larvik. This is especially clear in the Larvik and Alta plots from fig. 4.8.

Focusing on one of the Alta test cases where R2 performs especially bad compared to the other algorithms, we can investigate further what is going on. Figure 5.1 shows both the correct viewshed and the viewshed computed by R2 for such a test case. Inside the red circle the R2 viewshed has some obvious artifacts. Looking at the region between the observer and the red circle there are no obstructions that would suggest the lines of invisible terrain in the red circle. As we see from the correct viewshed, most of these points are well inside the boundary of the viewshed, and should therefore be relatively easy to classify. It seems as if the side slope itself is enough to R2 make errors on these trivial points.

To test this hypothesis we can synthesize a terrain with a deep and smooth valley, and run the algorithm on an observer situated on the bottom of the valley. We will generate the terrain as follows.

Let k denote the center coordinate of the terrain on the x-axis. Then the elevation of each grid point s_1, s_2 is given by:

$$e_{s_1,s_2} = (s_1 - k)^2 \tag{5.1}$$

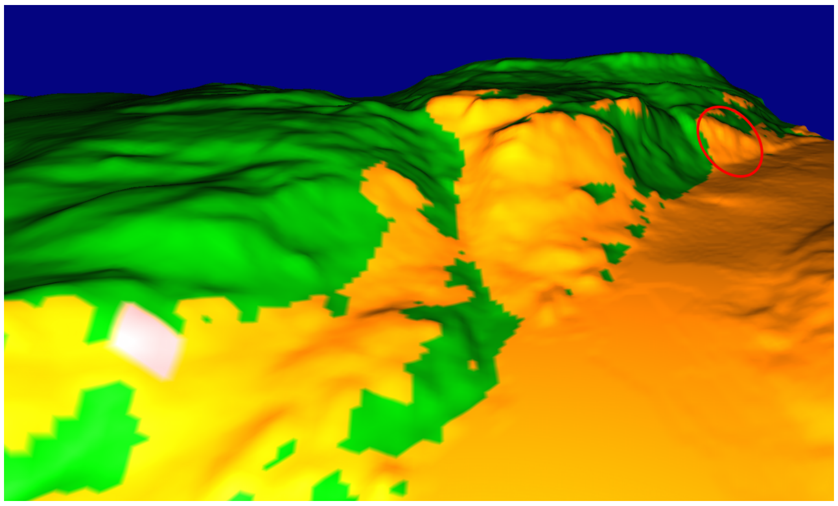
A 3D rendering of this terrain with the result from R2 can be seen in fig. 5.2, where the observer is situated at the bottom of the valley. Since the valley is convex, all points are visible to the observer. However, as the figure shows, R2 misclassifies many of the points on the side slope. This viewshed should be trivial to classify, as demonstrated by XDraw in fig. 5.2b, which classifies all points without error.

A more detailed view of this situation can be seen in fig. 5.3. Since the surface of the terrain is convex, the horizon is strictly increasing as R2 propagates each ray outward. The horizon between two such rays, therefore always takes the shape of a smooth curve with at most one kink. Figure 5.3b shows a typical such horizon as it occurs along the gray arc in fig. 5.3a. As the figure shows, the nearest neighbor interpolation used by R2 does a particularly bad job at estimating the horizon at the point p, as the error ϵ gets really large. The lines with artifacts we see in fig. 5.2a and in the red circle in fig. 5.1b are results of round-off errors in R2's nearest neighbor interpolation.

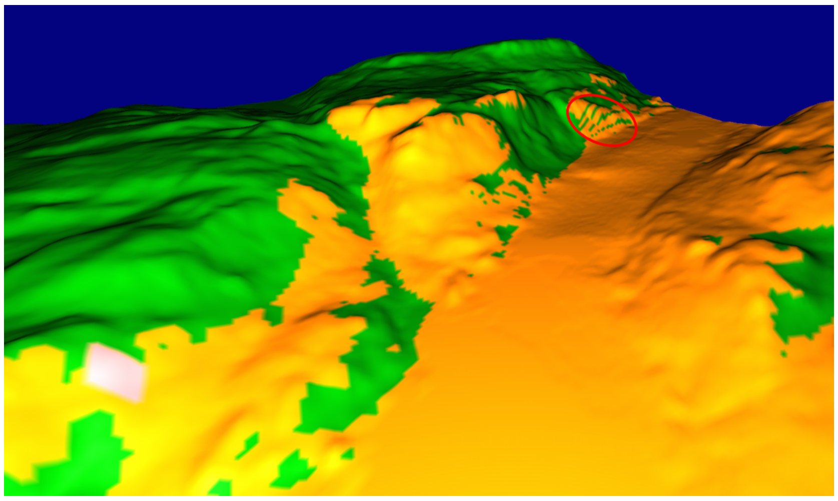
Comparing Alta to the Larvik terrain, we know that Larvik has no high mountains or severe slopes. This explains why R2 performs so much better on Larvik, since these side slope estimation errors seldom occur.

5.2 The generalized R2 algorithm

A natural way to mend the issues with side slopes is to replace the nearest neighbor interpolation with some higher order estimation leveraging several

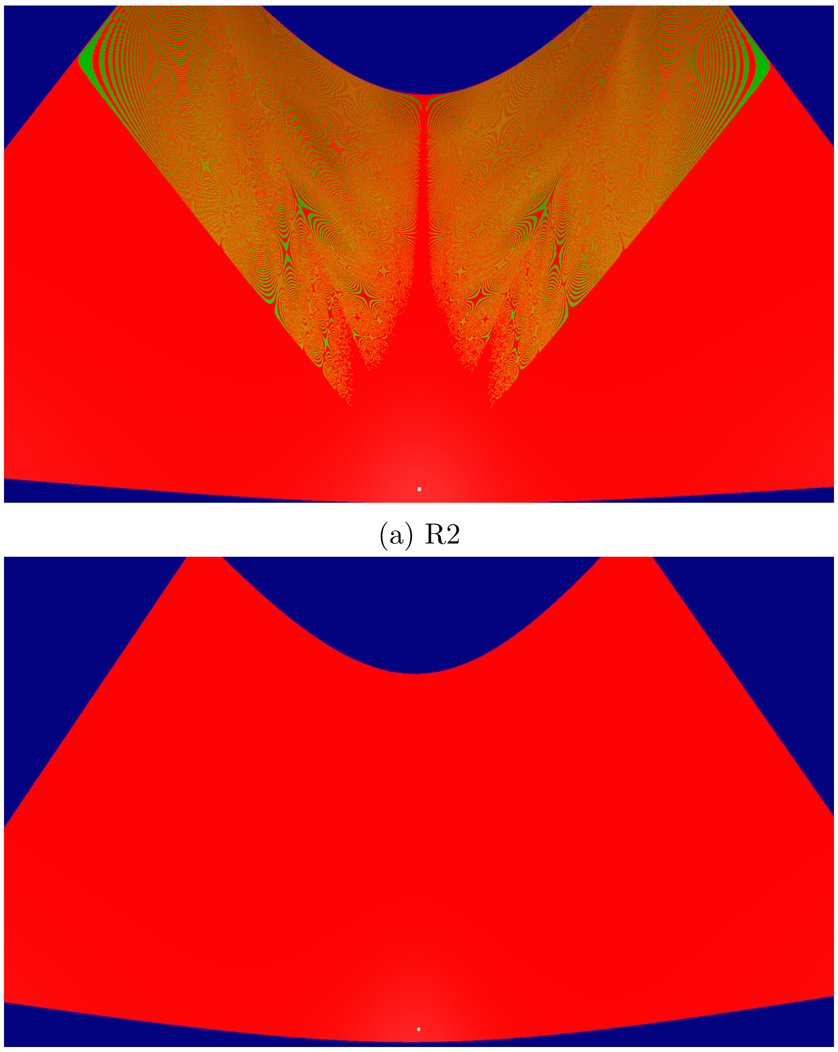


(a) Correct viewshed



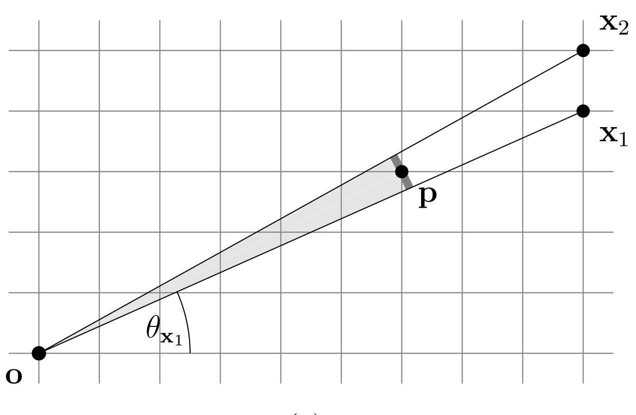
(b) Viewshed computed by R2

Figure 5.1: The viewshed of some observer on a side slope. Yellow represents points that are visible. The observer is indicated by the white square in the lower left corner of the figures. Inside the red circle the viewshed computed by R2 has some obvious artifacts.

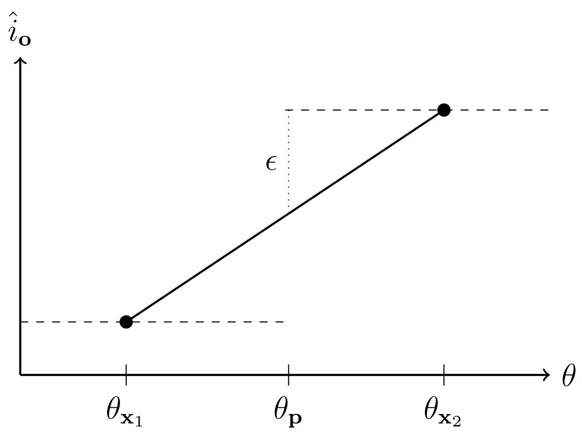

(a) R2

(b) XDraw

Figure 5.2: The viewsheds at a smooth valley as calculated by R2 and XDraw. Green and red points indicate invisible and visible points respectively. The observer is represented by the white spot situated at the bottom of the valley in the bottom of the figure.



(a)



(b)

Figure 5.3: Overview of the side slope situation from fig. 5.2. Since the terrain is convex the horizon is at all times given as a single smooth curve, or as a curve that has exactly one kink. A typical horizon between two rays of R2 is depicted in fig. 5.3b, where ϵ denotes the error in the horizon estimate made by R2.

of the rays passing through the neighborhood of a point. In order to test the effect of this we will make a generalized version of the R2-algorithm allowing us to replace the nearest neighbor interpolation with some other estimation scheme.

Algorithm 5.1 (R2 generalized).

Let est be some estimator, $\mathbf{o}$ *be some observer point, and let* ψ *and* ω *denote the observer- and target height, respectively.*

```
for all $\mathbf{p} \in S$, $\mathbf{p}$ perimeter point of $S$
    let $\ell$ be the LOS from $\mathbf{o} + \psi\mathbf{k}$ to $\mathbf{p} + \omega\mathbf{k}$
    $h = -\infty$

    for all grid line crossings $(x_1, x_2)$ of $\ell$
        $\mathbf{x} = (x_1, x_2, e(x_1, x_2))$

        $est.train(\mathbf{x}, h)$
        $h = \max\{h, s_{\mathbf{o}}(\mathbf{x})\}$

for all $\mathbf{s} \in S$
    if $s_{\mathbf{o}}(\mathbf{s} + \omega\mathbf{k}) > est.estimate(\mathbf{s})$
        label $\mathbf{s}$ as visible
    else
        label $\mathbf{s}$ as not visible
```

The only difference from algorithm 3.3 here being that we use a general object *est* referred to as an *estimator*, that in some way estimates the horizon at each grid point. For nearest neighbor estimation *est* must keep track of the nearest grid line crossing of each grid point. We refer to the first double *for*-loop as the *training step* of the algorithm, while the second *for*-loop we call the *classification* step.

Corollary 5.1.

If est.train and est.estimate runs in $O(1)$ *time algorithm 5.1 runs in* $O(n)$ *time on a square grid with* n *points.*

Proof. This follows directly from corollary 3.8. ■

5.3 Estimators for generalized R2

We will now consider a few possible implementations of the horizon estimator *est* discussed in the previous section. In the original paper [FRM94], the authors simply use the horizon of the nearest grid line crossing as the estimate for each grid point. Our primary goal is to replace this with some higher order estimator that handles side slopes better than the original algorithm. For this

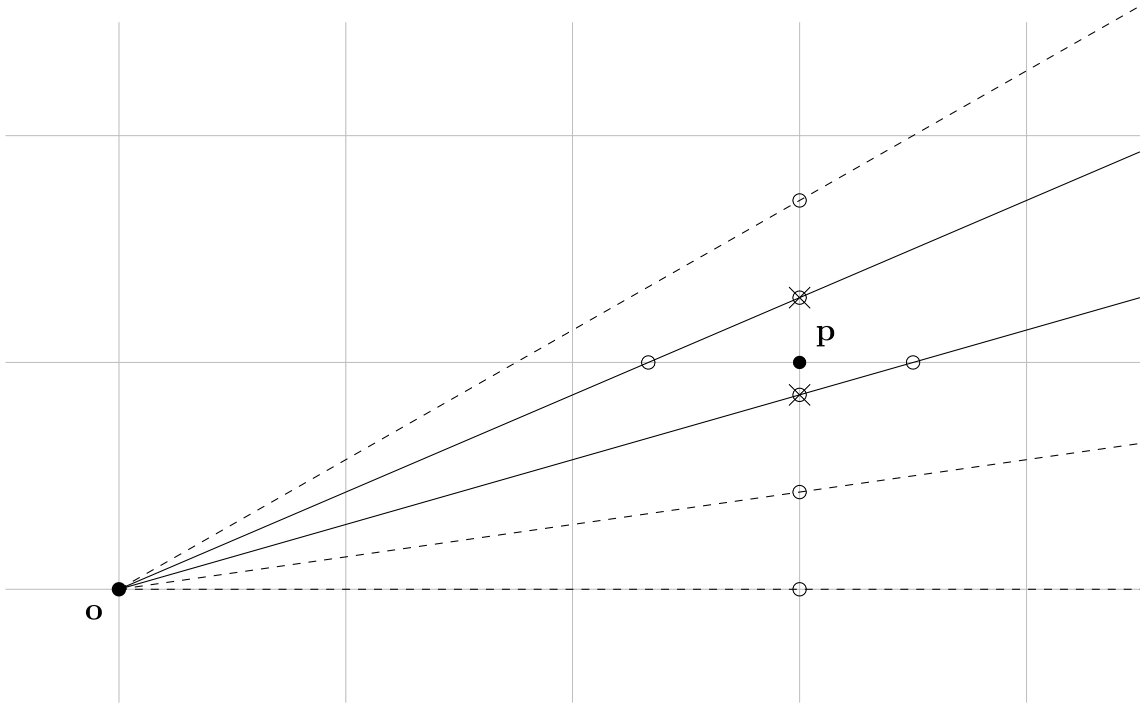


Figure 5.4: Estimation of the horizon of some point $\mathbf{s}$. The circles indicate all grid line crossings in the neighborhood of $\mathbf{s}$. The solid lines are the two rays used by the linear estimator, $est^{(\text{linear})}$. The crosses represent the grid line crossings it uses in the interpolation.

purpose we shall consider two first order estimators; what we shall call the *weighted* and the *linear* estimator. We shall also consider the *maximum* and *minimum* estimators, to see if we can achieve similar behavior as the max- and min-variants of XDraw.

Figure 5.4 shows an overview of the situation where the estimator comes into play. By evaluating numerous LOSs radiating from the observer, we have obtained the horizon at several grid line crossings in the neighborhood of a grid point $\mathbf{s}$. These are highlighted by circles in the figure. The purpose of the estimator is to make an estimate of $\hat{s}_{\mathbf{o}}(\mathbf{p})$ using this data.

5.3.1 Candidate estimators

The max- and min- and nearest neighbor estimators simply choose a single grid line crossing within the neighborhood of $\mathbf{s}$, and use its horizon as an estimate for that of $\mathbf{s}$. Unsurprisingly the max- and min estimators here choose the grid line crossing with the highest or lowest horizons, respectively. The nearest neighborhood estimator chooses the grid line crossing closest to $\mathbf{s}$.

One way to obtain a first order estimate of $\hat{s}_{\mathbf{o}}(\mathbf{s})$ is to weigh each grid line crossing by some function of its distance to $\mathbf{s}$. The estimate can then be calculated as a weighted average of the horizons of grid line crossings in the neighborhood. This type of estimator we call *weighted* estimators. Since the neighboring grid line crossings have a distance to $\mathbf{s}$ of at most 1, a natural weighting function is $(1 - d)$, where d is the distance to $\mathbf{s}$. As we shall see the *weighted* estimator has an efficient implementation which is almost as fast as the nearest neighbor estimator, while performing much better on side slopes.

Another way to obtain first order estimates is to linearly interpolate the horizons of the nearest ray on *both sides* of $\mathbf{s}$. These are the solid rays in fig. 5.4. Ideally we should use the horizon of the points along the rays that have the same distance to $\mathbf{o}$ as $\mathbf{s}$ has. However, as we know the horizon is constant between grid lines. Therefore, we will use the horizon of the nearest grid line crossing on both sides of $\mathbf{s}$. These two points are marked with crosses in the figure.

We will now give a precise definition of the *estimate*-method of these estimators.

Definition 5.1 (R2 visibility threshold estimators)**.** *Let* $\mathbf{o}$ *be some observer,* $X_{\mathbf{s}}$ *be the set of grid line crossings in the neighborhood of* $\mathbf{s}$*. Also let* $\underline{\mathbf{x_s}}$ *and* $\overline{\mathbf{x_s}}$ *be the two closest grid line crossings to* $\mathbf{s}$ *on either side of the ray running through* $\mathbf{s}$ *and the observer.*

$$est^{(near)}.estimate(\mathbf{s}) = \hat{s}_{\mathbf{o}}(\text{argmin}_{\mathbf{x}\in X_{\mathbf{s}}} ||\mathbf{x} - \mathbf{s}||) \tag{5.2}$$

$$est^{(max)}.estimate(\mathbf{s}) = \max_{\mathbf{x}\in X_{\mathbf{s}}} \hat{s}_{\mathbf{o}}(\mathbf{x}) \tag{5.3}$$

$$est^{(min)}.estimate(\mathbf{s}) = \min_{\mathbf{x}\in X_{\mathbf{s}}} \hat{s}_{\mathbf{o}}(\mathbf{x}) \tag{5.4}$$

$$est^{(weight)}.estimate(\mathbf{s}) = \frac{\sum_{\mathbf{x}\in X_{\mathbf{s}}} (1 - ||\mathbf{x} - \mathbf{s}||_{xy})\, \hat{s}_{\mathbf{o}}(\mathbf{x})}{\sum_{\mathbf{x}\in X_{\mathbf{s}}} (1 - ||\mathbf{x} - \mathbf{s}||_{xy})} \tag{5.5}$$

$$est^{(linear)}.estimate(\mathbf{s}) = \frac{||\overline{\mathbf{x_s}} - \mathbf{s}||\hat{s}_{\mathbf{o}}(\underline{\mathbf{x_s}}) + ||_{xy}\underline{\mathbf{x_s}} - \mathbf{s}||_{xy}\hat{s}_{\mathbf{o}}(\overline{\mathbf{x_s}})}{||\overline{\mathbf{x_s}} - \mathbf{s}||_{xy} + ||\underline{\mathbf{x_s}} - \mathbf{s}||_{xy}} \tag{5.6}$$

Note that using the generalized R2 algorithm with the nearest neighbor estimator is equivalent to using the original R2 algorithm. To avoid confusion we will from now on call this R2 near.

5.3.2 Implementation notes

As suggested the estimators should accumulate information obtained through the *train* method, and use this information for estimating the horizon. An efficient way to implement this is using one or more two-dimensional arrays of the same size as the terrain.

Using this technique the max- and min estimators can be implemented using one such array, where the value at index (i, j) represents the maximum or minimum horizon seen so far in the neighborhood of the grid point $s_{i,j}$. See listing A.14 for a sample implementation of this.

The nearest neighbor estimator needs two such arrays, one for storing the distance to the so far nearest grid line crossing, and the other for storing its horizon. The weighted estimator also needs two arrays, one each for accumulating the numerator and the denominator from eq. (5.5).

The most complicated estimator is the linear one, which needs four two-dimensional arrays. As for the nearest neighbor estimator two arrays is needed to keep track of the nearest grid line crossing, but in this case we need to do it for crossings on both sides.

Based on this it seems fair to assume that the max- and min estimators will be the fastest. Also we should expect the nearest neighbor- and weighted estimators to be comparable in terms of speed, while the linear estimator should be the slowest estimator.

5.3.3 Max- and min estimator performance

We will now assess the performance of the max- and min estimators for the generalized R2 algorithm. The purpose of these estimators is not to improve overall classification accuracy, so we do not expect them to perform better than the nearest neighbor estimator. Instead the goal of the max- and min estimators is to obtain an inner and outer boundary, respectively, for the actual viewshed.

Since the max estimator has a bias towards overestimating the horizon, it is less likely to commit false positives. That is, wrongfully label a point as

Algorithm	**Median rel. err.**	**Median rel. type 1 err.**	**Max rel. type 1 err.**	**Mean run. time**
R2 near	8.0×10^{-4}	4.1×10^{-4}	1.5×10^{-3}	116.0 ms
R2 max	4.9×10^{-3}	5.7×10^{-6}	3.1×10^{-5}	94.5 ms
XDraw max	4.8×10^{-2}	0	9.5×10^{-7}	32.6 ms

Table 5.1: R2 max and XDraw max performance on the Larvik full test set from chapter 4. For each test run the number of points erroneously classified as *visible* is counted and divided by the total number of points, giving the relative type 1 error. The test set consists of 45 runs, and the result of worst of these runs is used for the max type 1 statistic.

Algorithm	**Median rel. err.**	**Median rel. type 2 err.**	**Max rel. type 2 err.**	**Mean run. time**
R2 near	8.0×10^{-4}	3.9×10^{-4}	1.5×10^{-3}	116.0 ms
R2 min	4.0×10^{-3}	1.0×10^{-5}	5.7×10^{-5}	95.8 ms
XDraw min	1.7×10^{-1}	9.5×10^{-7}	1.3×10^{-5}	35.2 ms

Table 5.2: R2 min and XDraw min performance on the Larvik full test set from chapter 4. For each test run the number of points erroneously classified as *not visible* is counted and divided by the total number of points, giving the relative type 2 error. The test set consists of 45 runs, and the result of worst of these runs is used for the max type 2 statistic.

visible. Similarly we expect the min estimator to commit fewer false negatives.

It cannot be shown that the max- and min estimators commit *no* false positives and negatives, respectively. Figure 4.6b and fig. 4.6c illustrate why this is not case. Regardless of how we estimate the horizon, we do not know what it is like *between* the two nearest LOSs we have evaluated. Even though we take the maximum of the horizon at the two LOSs in the max estimator, we cannot guarantee that the horizon is no higher between the LOSs. Similarly, we cannot give any guarantees for the min estimator either.

The results of a performance test of these estimators on the Larvik test set from chapter 4 can be seen in table 5.1 and table 5.2. In the tests both the R2 max- and min estimators perform reasonably well, although the XDraw variants perform significantly better in terms of one-sided accuracy. One might ask why we should even consider to use the R2 variants, when the corresponding XDraw variants have higher one-sided accuracy, and are close to three times faster. The reason for this is that the XDraw variants are *much* more conservative in their estimates. Looking at the median overall error committed by these, we see that they are more than an order of magnitude less precise than the R2 variants. If the XDraw bounds are nowhere near tight, it greatly affects their usefulness.

The effect of this is illustrated in fig. 5.5, where the max- and min viewsheds are compared to the correct one. As the figure clearly shows, the viewsheds produced by the R2 variants coincide very closely with the correct viewshed. The viewsheds produced by XDraw are on the other hand either much smaller or much larger than the corresponding correct viewshed.

From these results it is clear that the R2 variants are much more useful for identifying a small region where the boundary of the actual viewshed is likely to be.

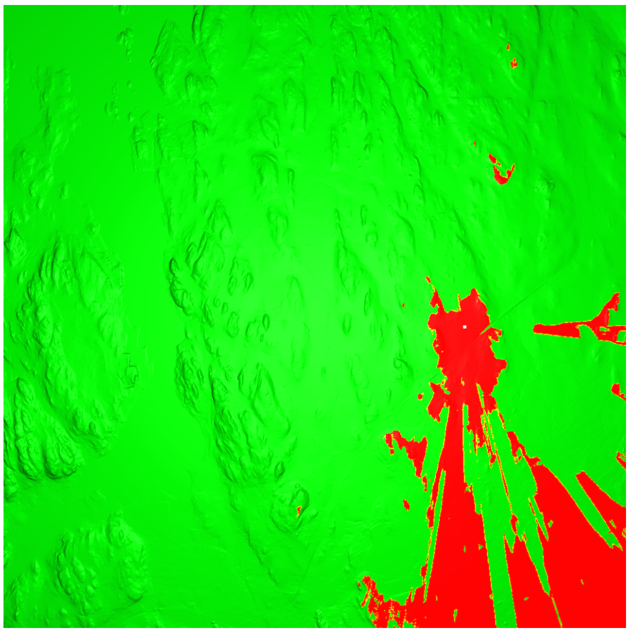
5.3.4 First order estimator performance

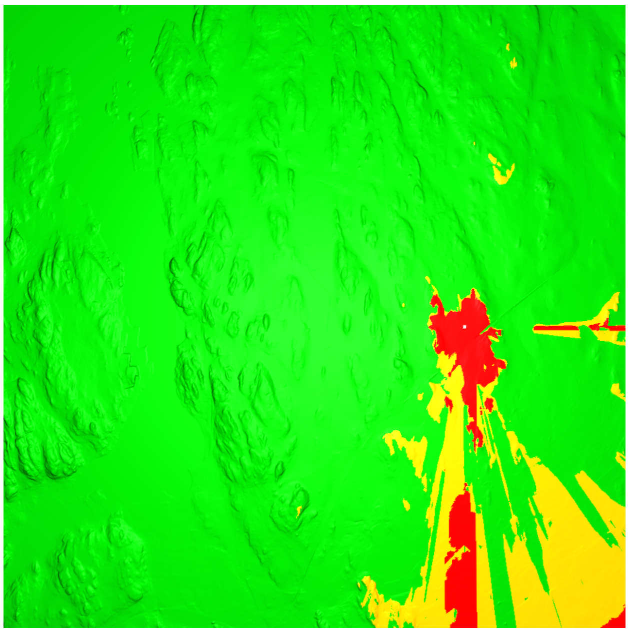
We have reason to believe that the weighted and linear estimators should improve the performance of the R2 algorithm, due to the shortcomings of the nearest neighbor estimator we have pointed out. It is now time to put these estimators to the test, by running through the test procedure we established in chapter 4.

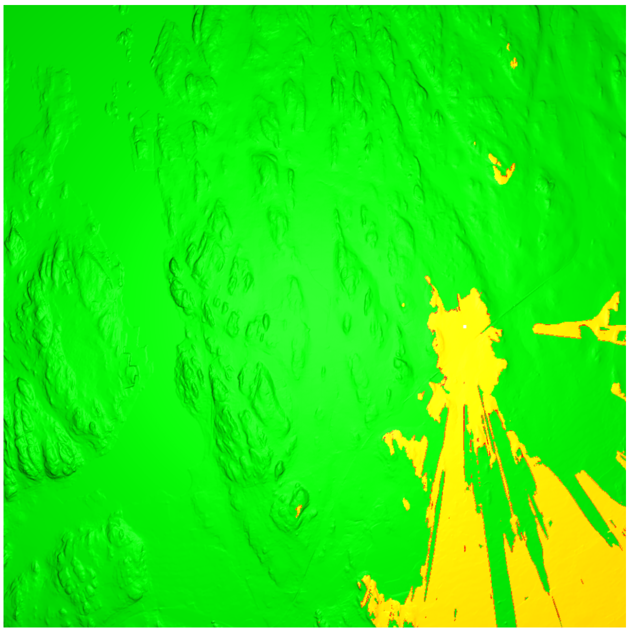
In order to verify that these estimators actually solve the side slope issues, we first re-run the test from fig. 5.2. The result of this can be seen in fig. 5.6. As the figure shows, these estimators perform much better than the nearest neighbor estimator. The linear estimator calculates this viewshed without error. The weighted estimator misclassifies a total of eight points, although not visible in the figure. This in stark contrast to the nearest neighbor estimator which misclassifies a total of 284 345 points in this test. From this it seems fair to conclude that both the weighted and the linear estimators solve the side slope issues.

Next we repeat the full testing procedure used in chapter 4 in order to see how the weighted and the linear estimators perform on more realistic terrains. The results can be seen in fig. 5.7, table 5.3 and table 5.4.

As the plots show, both the weighted and linear estimators perform significantly better than the original nearest neighbor estimator in terms of accuracy. R2 linear performs by far the best of the three, with a median relative error close to a quarter of R2 near, and roughly half of R2 weight.


(a) R2 max


(b) XDraw max


(c) R2 min

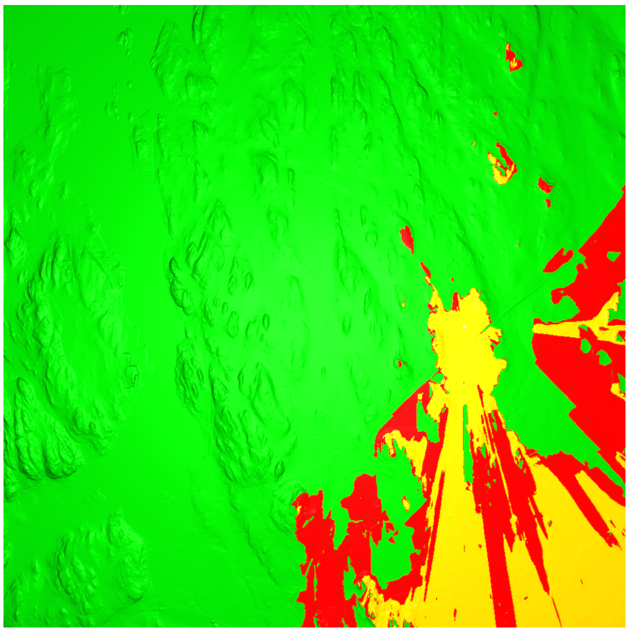
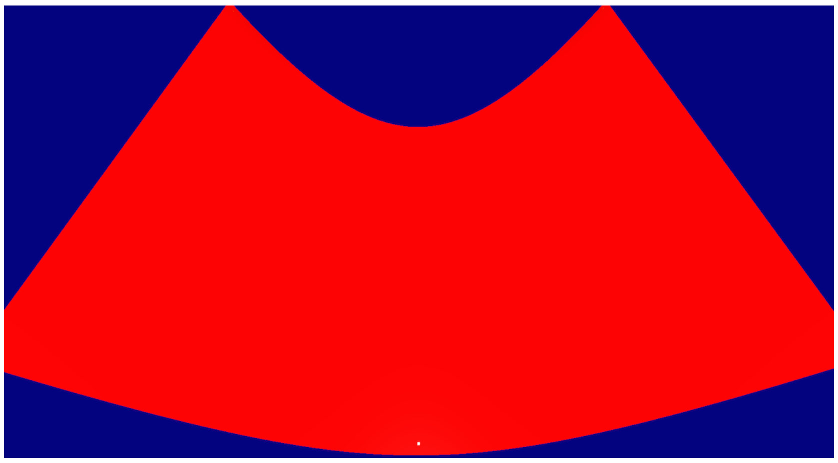

(d) XDraw min

Figure 5.5: Max- and min viewsheds of some observer on the Larvik terrain. The correct viewsheds are colored yellow, while the estimated viewsheds are colored red. In the max figures the estimated viewshed is smaller than the correct viewshed, so the estimated viewshed is rendered on top of the correct one. Opposite on the min figures.


(a) R2 weighted

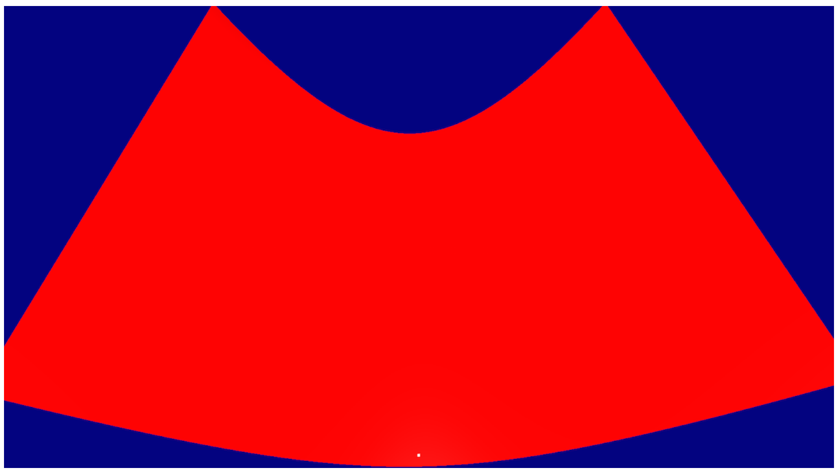

(b) R2 linear

Figure 5.6: The viewsheds at a smooth valley as calculated by R2 using the weighted and linear estimators, respectively. Green and red points indicate invisible and visible points respectively. The observer is represented by the white spot situated at the bottom of the valley in the bottom of the figure. The viewshed calculated by the linear estimator is perfect, i.e. all points colored red. The viewshed from the weighted estimator has eight misclassified points, although not visible in this figure.

Algorithm	**Median rel. err.**	**1st quartile**	**3rd quartile**	**Mean run. time**
R2 near	8.0×10^{-4}	5.5×10^{-4}	1.1×10^{-3}	126.2 ms
R2 weighted	4.3×10^{-4}	3.1×10^{-4}	5.7×10^{-4}	128.7 ms
R2 linear	2.3×10^{-4}	1.3×10^{-4}	2.9×10^{-4}	150.0 ms

Table 5.3: Larvik first order estimator performance test

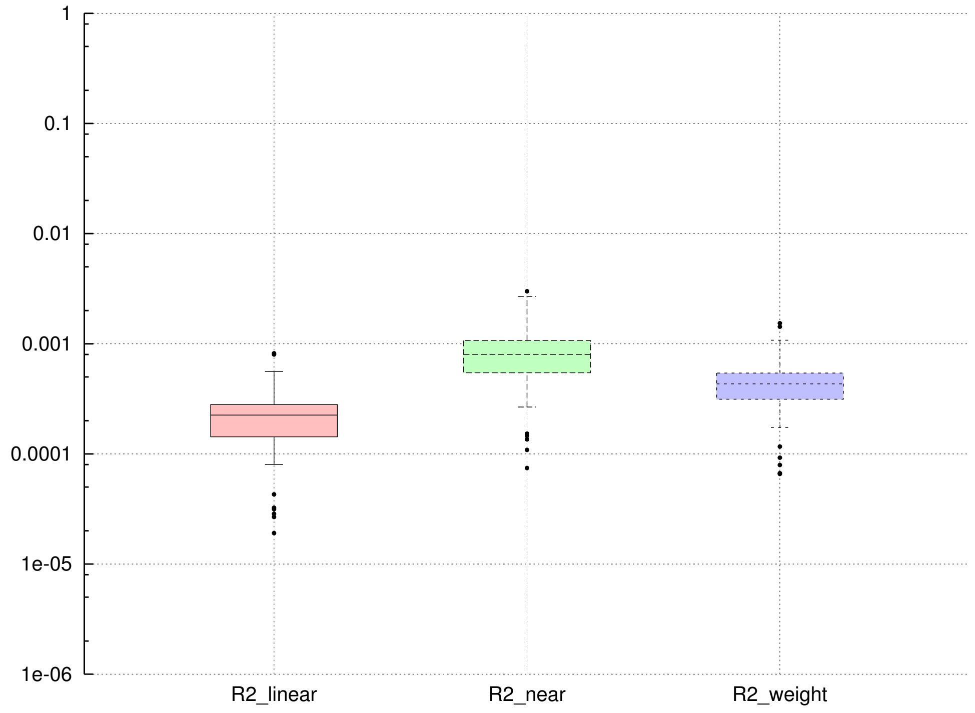
Algorithm	**Median rel. err.**	**1st quartile**	**3rd quartile**	**Mean run. time**
R2 near	1.3×10^{-3}	8.3×10^{-4}	1.9×10^{-3}	127.7 ms
R2 weighted	5.3×10^{-4}	3.8×10^{-4}	7.5×10^{-4}	129.8 ms
R2 linear	2.5×10^{-4}	1.7×10^{-4}	3.5×10^{-4}	146.7 ms

Table 5.4: Alta first order estimator performance test

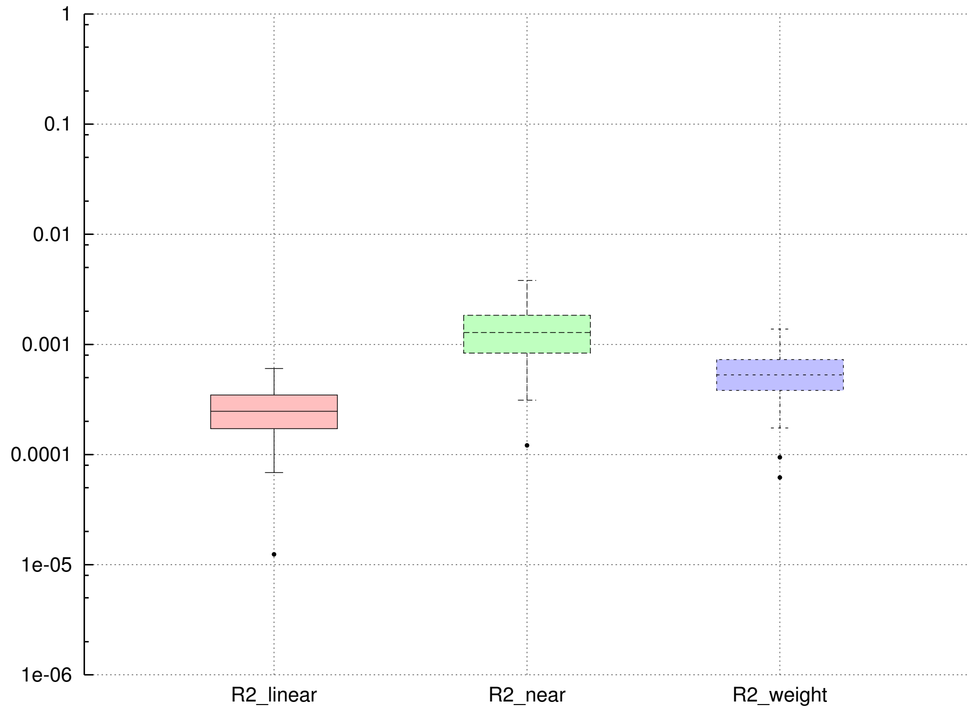
Observe that R2 weight is only barely slower than R2 near. In both the Larvik- and Alta tests the difference fails to be significant with a one-sided paired t-test p-value of 8.9% and 12.2%, respectively. Keeping in mind that the only real difference between these estimators are minor details in how the estimate is calculated, it is perhaps not surprising that the difference is this small. R2 linear seems to run $10 - 20\%$ slower than the other two, as expected due to the increased memory usage.

Also, note that the linear estimator has very similar performance on the Larvik and Alta terrains. This is contrasted by the nearest neighbor estimator, whose median relative error is about 50% larger on Alta than on Larvik. The median relative error of the weighted estimator is slightly less than 25% larger on Alta than on Larvik, which might indicate that it is also somewhat affected by side slopes on the Alta terrain.

Using the techniques discussed in chapter 4 we obtain from these results that the expected error ratio (EER) of the linear estimator and the nearest neighbor estimator is 0.24 with 99% confidence. For the weighted- and nearest neighbor estimator the EER is 0.53. Based on these results it is fair to claim that the weighted- and linear estimators are respectively twice and four times as accurate as the nearest neighbor estimator on these types of terrains and observer points. Since these boosts in accuracy come with modest increases in running time, there is no reason to test further with the nearest neighbor estimator.



(a) Larvik



(b) Alta

Figure 5.7: The relative error of the viewsheds as calculated using R2 with nearest neighbor-, weighted- and linear estimators. The observers used in this test are the same as the ones used in the full set tests in chapter 4.

5.4 The uniform R2 algorithm

As discussed in the beginning of this chapter, the next step is now to modify the R2 algorithm to evaluate an adjustable number of LOSs. The emphasis in this section is more on accuracy than efficiency, so we will consider algorithms that run several times slower than the original R2 algorithm. However, we do have a brute force algorithm from which we can obtain an upper bound on the running times of *useful* algorithms.

We will first consider the most obvious way to modify R2 for evaluating more LOSs. For this algorithm our primary interest is to investigate how the error changes as a function of running time.

The easiest way to handle any number of LOSs, is to evaluate LOSs at fixed angle intervals, instead of send a LOS to each point on the boundary of the terrain. The result is an algorithm that in some sense is similar to the fixed radar algorithm, in that it evaluates LOSs in uniformly distributed directions. Unlike the radar algorithm we still use the horizon of nearby LOSs for estimating the horizon of each grid point. As before we can use any type of estimator that fits into the generalized R2 scheme, but we will focus on the weighted- and the linear estimator as these seem to have superior performance.

Algorithm 5.2 (R2 uniform).

Let est be some estimator, **o** *be some observer point, C the number of LOSs the algorithm is allowed to evaluate, and let ψ and ω denote the observer- and target height, respectively.*

```
Set $\delta = \frac{2\pi}{C}$
for all $i = 1...C-1$
    let $\ell$ be some LOS from $\mathbf{o} + \psi\mathbf{k}$ with horizontal direction $i\delta$
    $h = -\infty$

    for all grid line crossings $(x_1, x_2)$ of $\ell$
        $\mathbf{x} = (x_1, x_2, e(x_1, x_2))$

        $est.train(\mathbf{x}, h)$
        $h = \max\{h, s_{\mathbf{o}}(\mathbf{x})\}$

for all $\mathbf{s} \in S$
    if $s_{\mathbf{o}}(\mathbf{s} + \omega\mathbf{k}) > est.estimate(\mathbf{s})$
        label $\mathbf{s}$ as visible
    else
        label $\mathbf{s}$ as not visible
```

First we compare this algorithm to the generalized R2, with both the weighted- and the linear estimator. As before we do this by running the Alta test from chapter 4. The results of this test can be seen in fig. 5.8 and in table 5.5. From the results we see that the two algorithms are comparably fast, which is expected since they evaluate the same number of LOSs. What is more interesting in these results is that the uniform variants of the R2 algorithm perform noticeably worse than the original variants in terms of accuracy. As we know the R2 algorithm with both the weighted and linear estimators make no error at points that lie exactly on an LOS. Some explanation to these results might be offered by the fact that the uniform version of the algorithm matches fewer point exactly with an LOS, resulting in more points with uncertain estimates.

Next we re-run the tests against a range of configurations of the uniform algorithm, to see how it performs using a larger number of LOSs. Since our radar implementation is not properly optimized for speed, we will use the number of evaluated LOSs as a measure of running time, in order to make a fair comparison. We test the algorithms using $2, 3, 4, 6, 8, 10$ and 16 times as many LOSs as the original R2 algorithm. When evaluating arbitrary precision algorithms like these, the *mean* relative error is actually a better

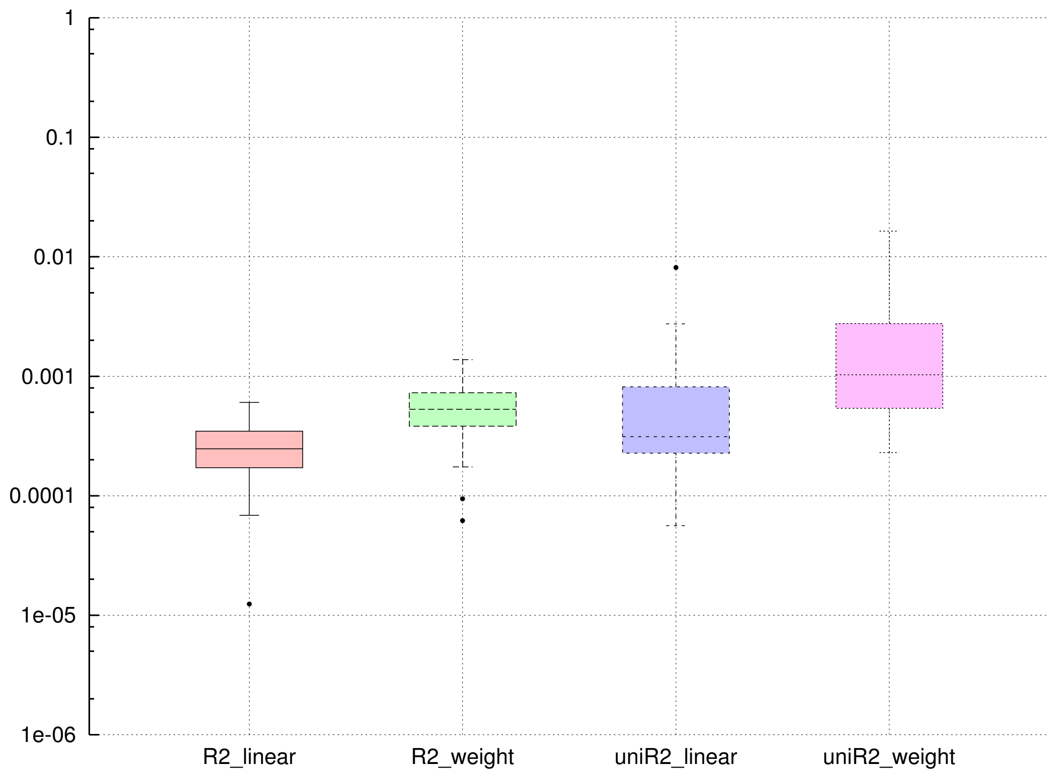


Figure 5.8: Comparison of the generalized and the uniform R2 algorithm with the same number of rays used on both algorithms. The tests are performed on the full Alta test set as described in chapter 4.

Algorithm	**Median rel. err.**	**1st quartile**	**3rd quartile**	**Mean run. time**
R2 linear	2.5×10^{-4}	1.7×10^{-4}	3.5×10^{-4}	146.6 ms
Uniform R2 linear	3.1×10^{-4}	2.2×10^{-4}	8.3×10^{-4}	133.9 ms
R2 weighted	5.3×10^{-4}	3.8×10^{-4}	7.5×10^{-4}	131.8 ms
Uniform R2 weighted	1.0×10^{-3}	5.3×10^{-4}	2.8×10^{-3}	124.1 ms

Table 5.5: Generalized and uniform R2 performance test

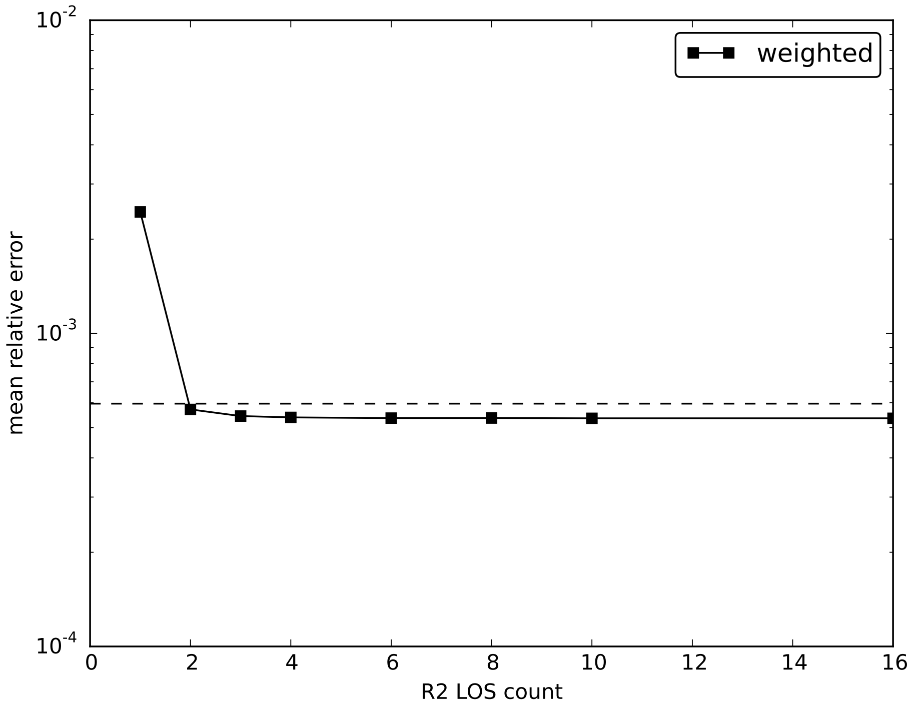


Figure 5.9: The performance of uniform R2 with the weighted estimator as a function of the number of LOSs evaluated. The x-axis shows the number of LOSs in multiples of evaluations used by the original R2 algorithm. The dashed line indicates the mean performance of the generalized R2 algorithm using the weighted estimator.

measure of the performance than the median relative error. The reason for this is that when the precision is increased, typically only the hardest test cases see any error improvement as the easy ones already are close to perfectly classified. At some point this means that the median error stops improving, making it seem like the error reduction has converged. In reality the algorithm might still be improving the error, but only on the hardest test cases. The mean relative error, on the other hand, captures this very well. Therefore we will for each test find the *mean* relative error, and plot it as a function of the number of evaluated LOSs.

As fig. 5.9 illustrates, the uniform R2 with the weighted estimator does not perform as expected in these tests. Instead of a steady improvement of the accuracy we see that the error rate quickly stabilizes at about the same level as the standard version of weighted R2, regardless of how many LOSs are

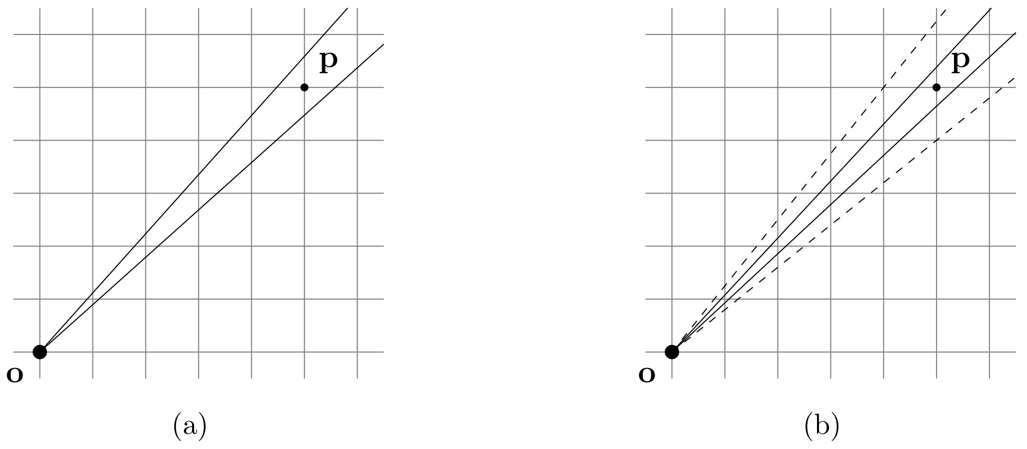


Figure 5.10: The effect of denser LOS sampling in the uniform R2 algorithm. Figure 5.10a shows the two LOSs passing through the neighborhood of a grid point **p** when the uniform R2 algorithm is used with a low LOS density. Figure 5.10b shows the same situation with higher LOS density, such that four LOSs pass through the neighborhood of **p**. When using the weighted estimator, the solid lines, which are closer to p, improve the accuracy of the estimate. The dashed lines, however, tend to affect the accuracy of the estimate negatively.

evaluated. Based on this it seems that the weighted estimator fails completely to exploit denser LOSs to increase the accuracy.

Upon closer inspection, the horizon estimates seem to fluctuate as the LOS density is increased. Some estimates become better, but others become worse. For a possible explanation for this behavior consider fig. 5.10, where we look at the LOSs in the neighborhood of some grid point **p**. When the LOS density is increased, the distance from **p** to the closest LOS decreases, which is good for the accuracy of the horizon estimate. These LOSs are indicated by the solid lines in the figure. Additionally, more LOSs intersect the neighborhood of **p**, so the average distance from the LOSs to **p** remains relatively unchanged. It is possible that the negative effect of adding the distant LOSs zeroes out the positive effect of the closer LOSs.

In an attempt to mend this, we define the *p-weighted* estimator, which allows us to adjust the estimators bias towards the nearest LOSs.

Definition 5.2 (The p-weighted estimator for R2)**.**

Let $\mathbf{o}$ *be some observer and* $X_{\mathbf{s}}$ *be the set of boundary crossings in the neighborhood of* $\mathbf{s}$*.*

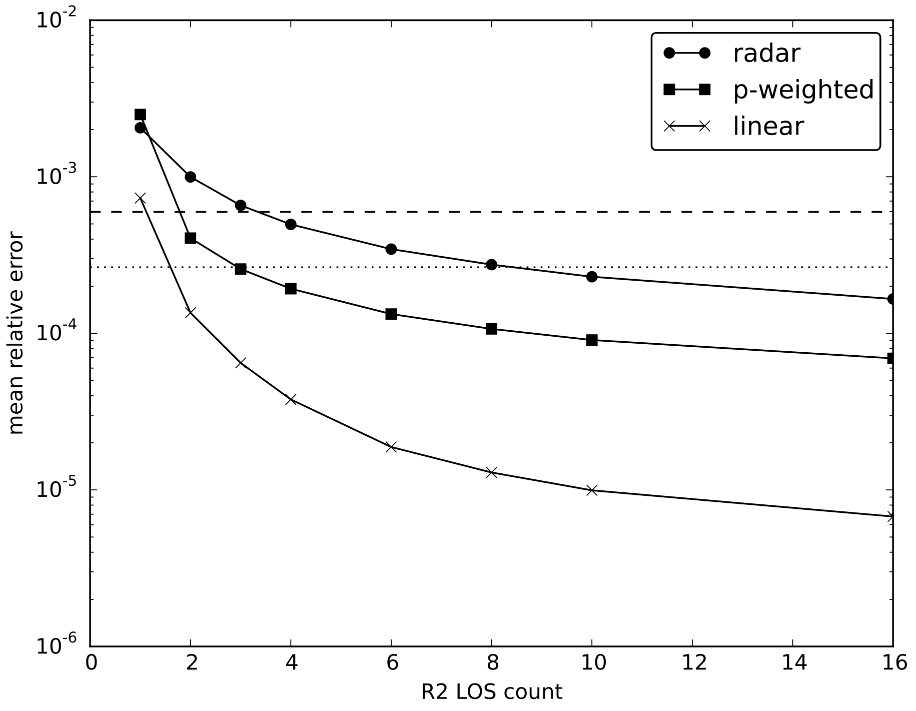


Figure 5.11: Algorithm performance as a function of the number of LOSs evaluated. The x-axis shows the number of LOSs in multiples of LOSs used by the original R2 algorithm. The dashed- and dotted line indicate the mean performance of the generalized R2 algorithm using the weighted- and linear estimators respectively.

$$est^{(\text{p-weight})}.estimate(\mathbf{s}) = \frac{\sum_{\mathbf{x}\in X_{\mathbf{s}}} (1 - ||\mathbf{x}-\mathbf{s}||_{xy})^p \, \hat{s}_{\mathbf{o}}(\mathbf{x})}{\sum_{\mathbf{x}\in X_{\mathbf{s}}} (1 - ||\mathbf{x}-\mathbf{s}||_{xy})^p} \tag{5.7}$$

Experiments indicate that using a value for p that is too high hurts accuracy as much as using $p = 1$. The best results seem to be achieved by letting p increase with LOS density. Therefore we re-run the test from fig. 5.9 with $p = 2k$, where k is LOS-count relative to that of the original R2 algorithm. I.e. when the algorithm is run with four times as many LOSs as the original R2, $p = 2 \cdot 4 = 8$.

The results of the full test can be seen in fig. 5.11. As the plot shows, all three algorithm variants now behave as expected, in that the error is significantly reduced as the number of LOSs evaluated is increased. It is clear, however, that the linear variant is superior to the other two. Although the p-weighted estimator performs clearly better than the radar algorithm, the choice of $p = 2k$ is here somewhat arbitrary, and it is unlikely that this is the optimal setting for p. However, it does not seem fruitful to optimize the weighted estimator further when the linear estimator performs so much better.

With the uniform R2 algorithm and the linear estimator we have successfully made an algorithm that is significantly more accurate than the generalized R2 algorithm. Using a LOS density that is 16 times higher than that of the original algorithm, we have reduced the median error to only a handful points. Therefore we will consider this level of accuracy to be satisfactory for terrains of this size and difficulty.

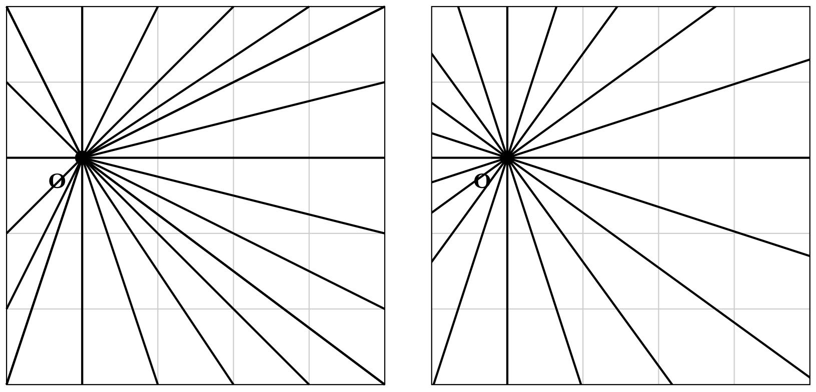
5.5 A tunable hybrid algorithm

With the uniform R2 algorithm we have established a baseline for highly accurate, albeit slow, algorithms. Next we consider a different approach for achieving the same level of accuracy while evaluating fewer LOSs. We do this by putting more knowledge about the nature of terrains and viewsheds into the algorithm. In order to make sure we do not overfit the algorithm to some specific terrain, we will develop our algorithm using the Larvik terrain only, and then validate its performance on the seemingly tougher Alta terrain.

5.5.1 Targeting high uncertainty points

Looking at fig. 5.11 we see that the uniform R2 performs worse than the generalized R2 when evaluating the same number of LOSs. Comparing the LOS-patterns of the two variants, illustrated in fig. 5.12, we will try to understand why this is the case. First of all, we see that uniform R2 sends more of its LOSs "outside" of the grid than generalized R2. In practice this means that generalized R2 evaluates more grid line intersections. Secondly, every LOS of generalized R2 runs exactly through at least one grid point. This is good, because R2 makes no estimation errors along the LOS, so these points are always classified correctly. Uniform R2 hits a few grid points by chance, but not nearly as many as generalized R2.

Based on this, it seems like a good starting point to use the same scheme



(a) Generalized R2 (b) Uniform R2

Figure 5.12: Overview of the distribution of LOSs for generalized- and uniform R2 on a 6×6 regular square grid (RSG). Notice how generalized R2 better covers the grid, and hits *through* more grid points than the uniform variant.

as generalized R2, and then add more LOSs that run *through* points when higher accuracy is needed. Since any error at points that intersect a LOS is guaranteed to be corrected, we want to send the supplementary LOSs through points that are likely to be misclassified in the first place. As discussed in chapter 4 the points with the highest probability of being misclassified are typically the ones that are close to the boundary of the viewshed. A simple count of the errors made by R2 linear on the Larvik test set shows that typically $99-100\%$ of the errors lie just inside or just outside the correct viewshed. Naturally, we do not know the exact boundary of the viewshed, but we can obtain decent estimates using variants of the generalized R2 algorithm.

The most obvious estimate of the exact viewshed boundary can be obtained simply by using the boundary of the viewshed as estimated by generalized R2 with linear estimator. On the test cases from the Larvik test set $91-96\%$ of the errors made lie on the boundary of the estimated viewshed. The boundary on these test cases contain $1.5-2.5\%$ of the points of the entire terrain, which corresponds to about $4-6$ times as many LOSs as generalized R2.

As suggested earlier we can also use max- and min variants of generalized R2 for obtaining a limited region which contains most of the boundary of the exact viewshed. In the Larvik tests this region contains $90-93\%$ of the error and spans $0.5-1\%$ of the terrain.

According to these numbers we have isolated most of the error to a very limited region of the terrain. Thus on Larvik it seems we typically should be able to improve the error one order of magnitude by evaluating less than ten times as many LOSs. We will have to test to see how this holds up on other terrains, however.

We now define more precise algorithms based on these ideas.

Algorithm 5.3 (Hybrid bound)**.**
Let $\mathbf{o}$ *be the observer and* ψ *be the observer height. Let est be the linear estimator. If* V *is a viewshed, then* $\overline{V}$ *denotes the boundary of* V*, which is defined as follows: Let* $\mathbf{s}$ *be a grid point, then* $\mathbf{s} \in \overline{V}$ *iff.* $\mathbf{s}$ *has at least one grid point 8-neighbor with a different classification than* $\mathbf{s}$ *itself.*

```
Execute the training step of algorithm 5.1 on est
Find $V_{linear}$ using the classification step of algorithm 5.1 on est

for all $\mathbf{s} \in \overline{V}_{linear}$
    Train est on the LOS running from $\mathbf{o} + \psi\mathbf{k}$ through $\mathbf{s}$

Execute the classification step of algorithm 5.1 using est
```

Algorithm 5.4 (Hybrid min/max)**.**
Let $\mathbf{o}$ *be the observer and* ψ *be the observer height. Let est,* $est^{(max)}$ *and* $est^{(min)}$ *be the linear-, max- and min estimators, respectively.*

```
Execute the training step of algorithm 5.1 on est, $est^{(max)}$ and $est^{(min)}$
Find $V_{max}$ and $V_{min}$ using the classification step of algorithm 5.1 on $est^{(max)}$
and $est^{(min)}$

for all $\mathbf{s} \in V_{min} \setminus V_{max}$
    Train est on the LOS running from $\mathbf{o} + \psi\mathbf{k}$ through $\mathbf{s}$

Execute the classification step of algorithm 5.1 using est
```

Figure 5.13 and table 5.6 shows the performance of these two algorithms compared to the generalized R2 algorithm with linear estimator. In terms of accuracy the improvement is tremendous for both techniques. The accuracy of the min/max hybrid variant seems to be close to two orders of magnitude better than R2, while the boundary variant makes almost no error at all. As expected the running times have increased, but by less than one order of magnitude.

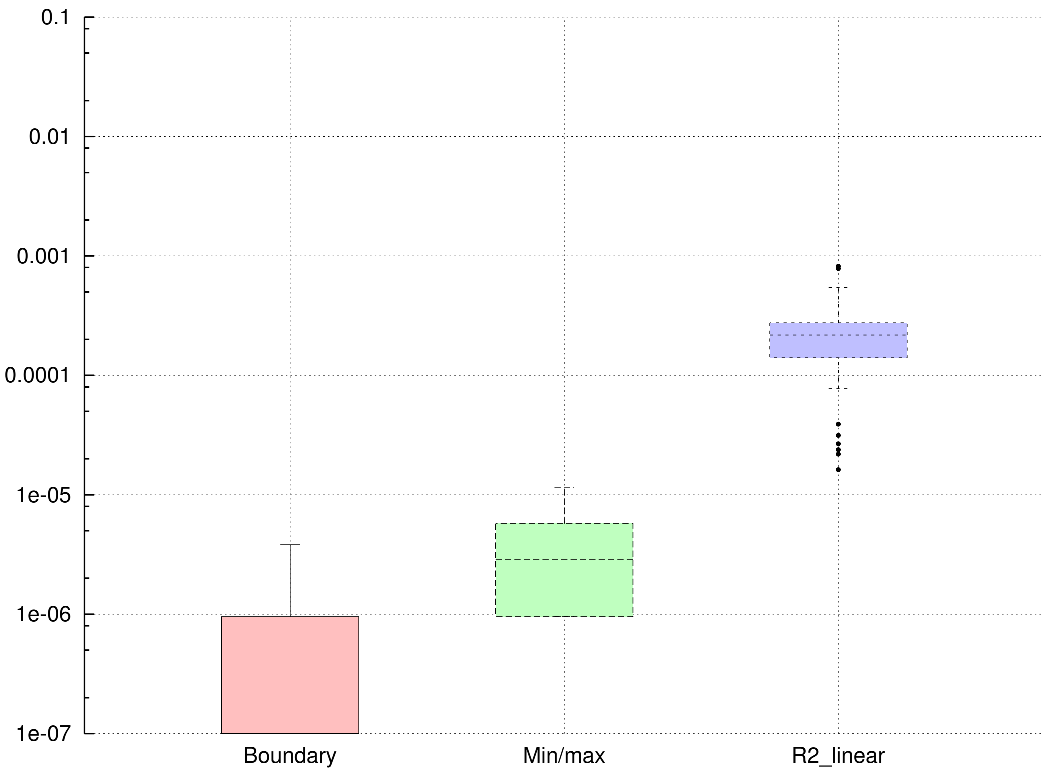


Figure 5.13: The relative error of R2 and the boundary- and min/max hybrid variants on the Larvik test set.

The EER of the min/max hybrid and linear R2 is less than 2.3% with a confidence of 99%. Similarly the EER of bounded hybrid and linear R2 is less than 0.5%. So these algorithms do indeed increase the accuracy by two orders of magnitude on this test set. The error ratio measure is not well-defined for comparing the two hybrid variants, as they both make 0 error on several test cases. Using a one-sided paired t-test we can, however, assert that the boundary variant is the most accurate with a p-value of 2×10^{-8}.

Algorithm	**Median rel. err.**	**1st quartile**	**3rd quartile**	**Mean run. time**
R2 linear	2.2×10^{-4}	1.3×10^{-4}	2.9×10^{-3}	134.2 ms
Boundary	0	0	9.5×10^{-7}	958.4 ms
Min/max	2.8×10^{-6}	9.5×10^{-7}	6.0×10^{-6}	678.2 ms

Table 5.6: Larvik fixed hybrid performance test.

5.5.2 Limiting the running time

Having achieved performance that is at least on par with the uniform R2 algorithm in terms of accuracy, and significantly faster in terms of speed, we will now attempt to make tunable versions of the fixed hybrid algorithms.

At this point we know that the misclassified points are relatively dense on the estimated boundary and in the difference between the R2 min- and max viewsheds. For brevity we will refer to these as *highlighted points*. The only way to reduce the running time of the proposed algorithms is to reduce the number of evaluated LOSs. This means that we have to select a portion of the highlighted points. Ideally we would have some way to prioritize these points, which reflects their respective likelihood of being misclassified. We could try to find a set of easily obtained features that highlight points that typically are misclassified. For a point $\mathbf{p}$ this might be features such as $s_{\mathbf{o}}(\mathbf{p}) - est.\text{estimate}(\mathbf{p})$, or the distance to the nearest LOS from $\mathbf{p}$. Finding such a set of features should be done using some machine learning algorithm, and is beyond the scope of this thesis.

A trivial procedure to reduce the number of LOSs is simply to select a subset of the highlighted points at random. Inspecting how the points from the boundary of the viewshed are oriented wrt. the observer, it seems this is far from uniformly distributed. A typical plot of the density of orientations can be seen in fig. 5.14, which has several narrow peaks. This is a good thing, since we then by sending a single LOS in the direction where the boundary points are dense potentially can correct multiple errors. By randomly sampling the set of highlighted points, we effectively sample the distribution of orientations. This means that we are likely to send LOSs in the directions where this distribution has peaks, which is exactly what we want.

The highlighted points are typically found in no particular order. Thus we can sample approximately uniformly simply by picking each kth point. By sorting the points by orientation before doing this we get a truly uniform sampling. In practice the latter method actually seems to be *faster*, despite the extra effort needed to sort the highlighted points.

These steps can be summed up as the following algorithm:

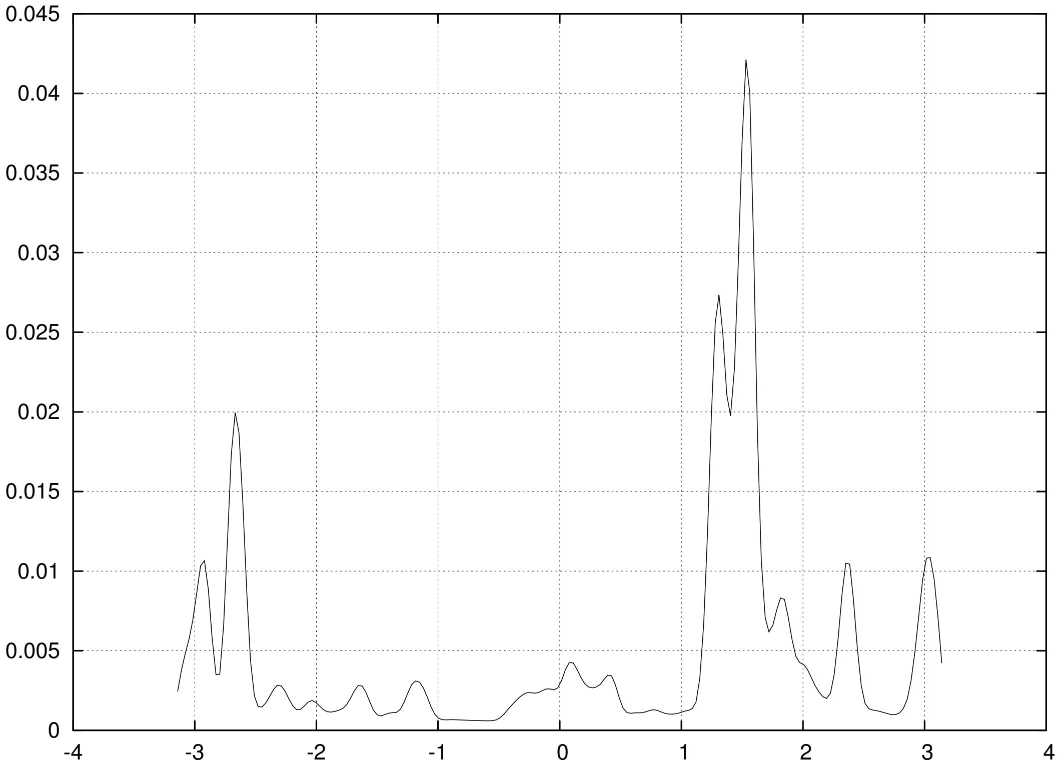


Figure 5.14: The density of viewshed boundary points by orientation wrt. the observer for a typical Larvik test case.

Algorithm 5.5 (Constrained Hybrid)**.** *Let* **o** *be the observer and* C *the number of LOSs the algorithm is allowed to evaluate in addition to the ones evaluated by a single R2 pass.*

```
Execute the first training steps of one of the Hybrid algorithms
Let $M$ be the resulting set of highlighted points

Sort $M$ by orientation relative to $\mathbf{o}$
Set $c = \frac{|M|}{C}$

for all $i = 1...C$
    Train est on the LOS running from $\mathbf{o} + \psi\mathbf{k}$ through $M_{ci}$

Execute the classification step of R2
```

We will now evaluate the performance of this algorithm, both using the boundary and min/max variants. We test it with a range of LOS-counts, and see how the accuracy changes as a function of *running time*. For each LOS-count we will run the algorithm against the full Larvik test set, and use the mean error of all those tests as a performance measure for the respective configuration. For these algorithms the mean error is a better measure than the median error. This is because when the LOS-count reaches above some level, only the hardest viewsheds are improved by increasing the LOS density further, and the median might fail to pick up on this change.

The results of the test can be seen in fig. 5.15. The figure shows that uniform R2 outperforms the hybrid variants for low LOS-counts. It is, however, clear that both of the hybrid variants converge much faster to higher accuracy than uniform R2. Observe also that for any given amount of running time, hybrid bound outperforms hybrid min/max in terms of accuracy. For this reason we abandon the min/max scheme, and will refer to the boundary variant as the *hybrid algorithm* from now on.

5.5.3 Verifying the results

With the techniques we have developed in this section we have been able to drastically reduce the error of the estimated viewshed, with moderate increases in running time. In the process we have used some properties of the viewshed that might depend on the terrain, such as the spacial distribution of misclassified points. As discussed in the beginning of the section it is therefore important that we verify the results on some different terrain, in

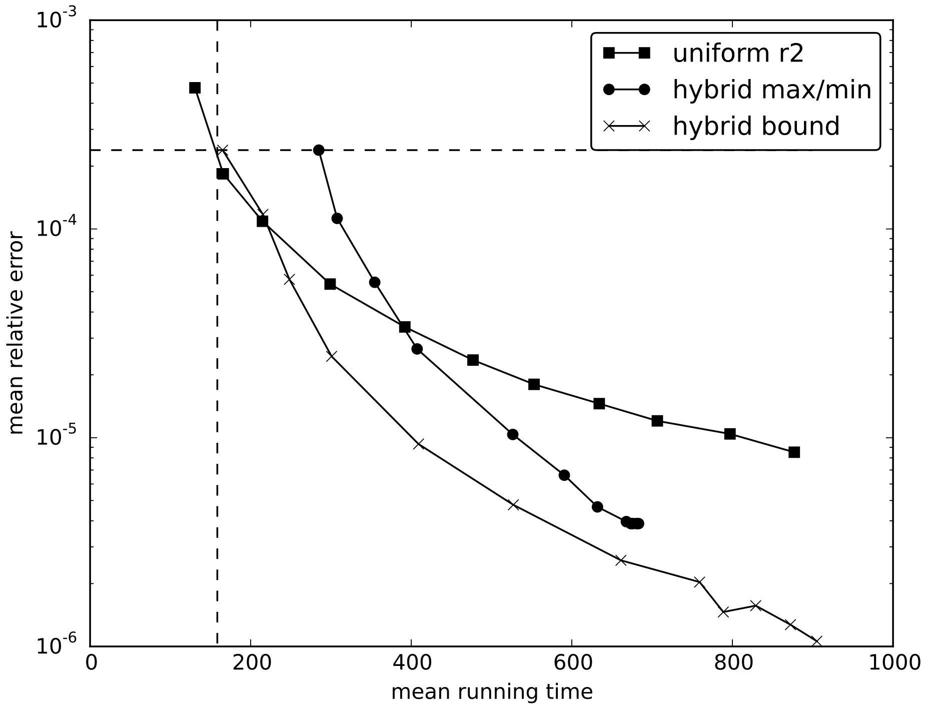


Figure 5.15: The mean relative error of the two hybrid variants and uniform R2 as a function of running time. The dashed lines indicate the accuracy and running time of the generalized R2 using the linear estimator.

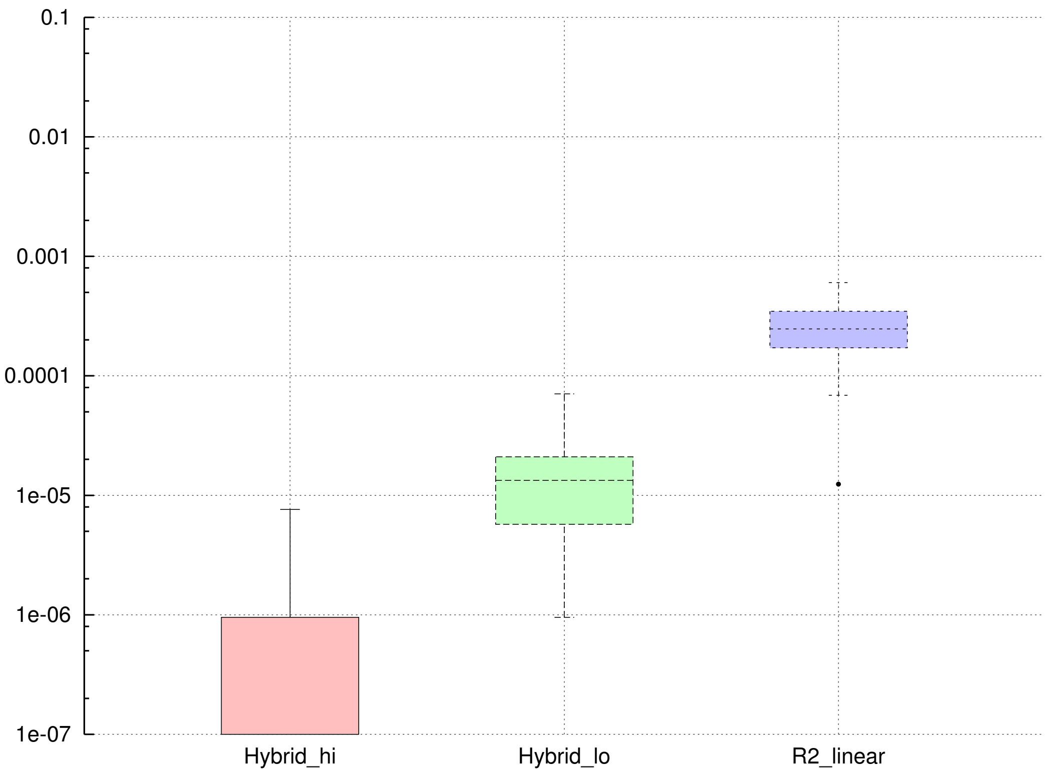


Figure 5.16: Alta hybrid performance test. The *lo* variant of hybrid is constrained to evaluating three times as many LOSs as R2. The *hi* is unconstrained. The median of *hybrid_hi* is 0, which is why its box appears cut on the log scale.

order to make sure we have created an algorithm that not only performs well on the Larvik test case.

We therefore repeat the tests once more, using the full Alta test set. In the test we include R2 linear to see the improvement from our baseline. We test a configuration of hybrid running without running time limitations, as well as a configuration running approximately three times slower than R2 linear.

The results of this test can be seen in fig. 5.16 and table 5.7. It is clear that these results comply with what we have already seen from the Larvik tests. Since these terrains represent two extremes in the range of terrain types used in our application, it seems fair to assume that the hybrid algorithm should perform well on any terrain type that is realistic for us to use.

As discussed in chapter 4 we shall also evaluate these algorithms on the South

Algorithm	**Median rel. err.**	**1st quartile**	**3rd quartile**	**Mean run. time**
R2 linear	2.5×10^{-4}	1.7×10^{-4}	3.5×10^{-4}	137.7 ms
Hybrid low	1.3×10^{-5}	5.7×10^{-6}	2.1×10^{-5}	411.4 ms
Hybrid high	0	0	9.5×10^{-7}	1057.9 ms

Table 5.7: Alta hybrid performance test. The *low* variant of hybrid is constrained to evaluating three times as many LOSs as R2. The *high* is unconstrained.

Korea test set, similar to the one used by Franklin et al. in [FRM94]. This terrain has more spikes than Larvik and Alta, as can be seen in fig. 4.11, thus resembling the white noise terrain somewhat. A detailed description of this terrain can be found in chapter 4. The test is executed using 32 *hard* observers, to best match the one used in the original test. We will evaluate the performance of R2 using both nearest neighbor- and linear estimator, as well as the same *hi* and *lo* versions of the hybrid algorithm.

The results of this test can be seen in fig. 5.17 and table 5.8. As the results show, R2 linear is here only barely more accurate than the original version. The EER of R2 linear and R2 near is less than 89.8% so the improvement is statistically significant. This is, however, a lot weaker than the 25% EER we saw in the Larvik and Alta tests. The primary problem we wanted to solve with the linear estimator, was R2's weak performance on side slopes. As we saw, the linear variant handles this much better. Considering the 3D rendering of the South Korea terrain in fig. 4.11, there are almost no side slopes, only narrow peaks. Therefore it is not a surprise that being good at side slopes has a limited effect on this terrain. It is possible that the spikes in the South Korea terrain are artifacts of the low resolution of this test set. If this is the case, then this test is not relevant for our application, as we will use data sets with high resolution.

Turning to the hybrid variants, we see that these also perform worse on this terrain. Due to the similarities between the South Korea- and white noise terrains, this is explained in our discussion of the white noise terrain. The trend we have seen on Larvik and Alta exists here as well; the hybrid variants greatly outperform the R2 variants in terms of accuracy.

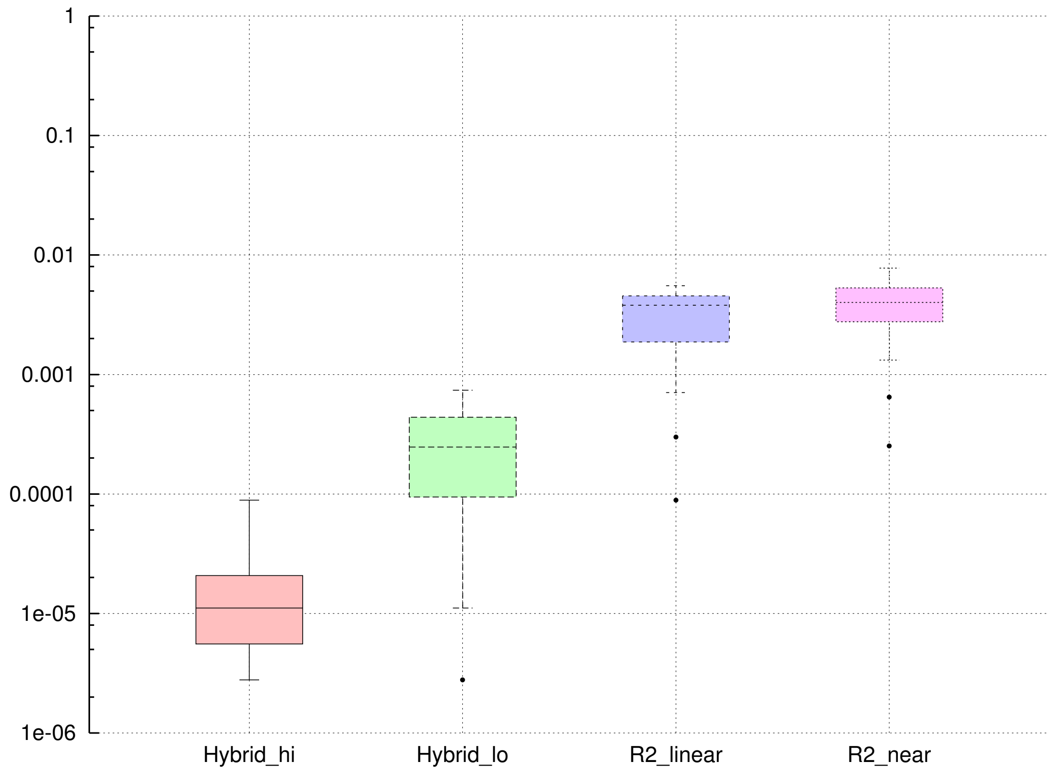


Figure 5.17: South Korea performance test. The *lo* variant of hybrid is constrained to evaluating three times as many LOSs as R2. The *hi* is unconstrained.

Algorithm	**Median rel. err.**	**1st quartile**	**3rd quartile**	**Mean run. time**
R2 near	4.0×10^{-3}	2.8×10^{-3}	5.3×10^{-3}	39.8 ms
R2 linear	3.8×10^{-3}	1.9×10^{-3}	4.6×10^{-3}	47.7 ms
Hybrid low	2.5×10^{-4}	9.4×10^{-5}	4.4×10^{-4}	128.2 ms
Hybrid high	1.1×10^{-5}	5.6×10^{-6}	2.1×10^{-5}	478.6 ms

Table 5.8: South Korea hybrid performance test. The *low* variant of hybrid is constrained to evaluating three times as many LOSs as R2. The *high* is unconstrained.

5.5.4 Limitations and asymptotic behavior

With the hybrid algorithm we have introduced three additions to the general R2 scheme. First we find the set of boundary points of the estimated viewshed, then we sort it by orientation to the observer, and finally we evaluate extra LOSs through some or all of these points. The running time of the algorithm is therefore a function of three parameters; the number of grid points, the number of boundary points and the number of extra LOSs, denoted n, b and k, respectively.

The cardinality of the boundary points is essential to the asymptotic behavior of the algorithm. The experiments conducted in section 5.5.1 suggest that b typically is less than a few percent of n. However, repeating the same experiments on the white noise terrain we see test cases where b is as large as 40% of n. Therefore, we clearly cannot generally claim that $b \ll n$. Thus $b = O(n)$.

As before, we can find the first viewshed in $O(n)$ time. Identifying the boundary points also takes $O(n)$ time, as we have to iterate through each grid point. The sorting step takes $O(b \log b) = O(n \log n)$ time, and the extra LOSs take $O(k\sqrt{n})$ time to evaluate. Finally we re-classify each grid point in $O(n)$ time. Thus, the worst-case running time of the algorithm is $O(n \log n + k\sqrt{n})$.

There is, however, a considerable gap between the worst-case performance and what we will typically see in practice. On typical test cases, it seems we *can* assume $b \ll n$. Additionally, most decent sorting algorithms are much more streamlined than R2, so there is a hidden constant term in the $O(n)$ of R2 that will eclipse the $\log n$-factor for most practical sizes of n. The running time is thus in practice closer to $O(n + k\sqrt{n})$. As long as $k = O(\sqrt{n})$ we therefore expect to get the same asymptotic behavior as R2; $O(n)$.

The primary weakness of the algorithm is viewsheds with large boundaries. As discussed in the beginning of chapter 4, these are also the ones we are interested in. We have seen, however, that this typically translates to b being no more than a few percent of n. For certain applications, this is not the case, but it seems we must have *artificially* large boundaries for this to be a real issue.

5.6 Summary

We started out this chapter investigating the results from chapter 4, which showed that R2 performs weaker than expected on the Alta terrain. Upon closer inspection of some of the test cases we discovered some strange artifacts in the R2 viewsheds. This led us to the hypothesis that R2 struggles with side slopes. The artificial valley test in fig. 5.2 showed that this is indeed the case.

In an attempt to mend these issues we proposed the *generalized R2* algorithm which uses some arbitrary horizon estimator, instead of the nearest neighbor scheme used in the original algorithm. We tested the *weighted*- and *linear* estimators, and saw that both solved the issues with side slopes. As expected this led to a significant improvement in accuracy on the Alta terrain. More surprising was that there was also significant improvement on the flat Larvik terrain, which has few prominent side slopes.

Inspired by the radar-like algorithm with adjustable performance we wanted an algorithm that could increase the performance on demand. Our tests indicated that R2 with the new estimators outperforms radar when using the same LOS-count. This being a feature we wanted to retain, we made a crossover of the two algorithms, combining the uniformly spaced LOS-sampling of the radar-like algorithm, with R2's horizon estimation. The resulting algorithm was the *uniform R2* algorithm, which the tests show outperforms the radar-like algorithm when using the same LOS-count.

Our motivating scenario from chapter 2 sparked a particular interest for the boundary of the viewshed. Investigating the error patterns of R2 we saw that most of the error is made on or very close to the boundary of the correct viewshed. With this in mind we developed the hybrid algorithm which estimates this boundary using linear R2, and then trains the estimator further by evaluating extra LOSs to the boundary points. The result was an algorithm with exceptional accuracy and reasonable running times. By letting the number of extra LOSs be a parameter of the algorithm, we could once again adjust the accuracy as a function of running time. By sending the extra LOSs where they are actually needed, this algorithm achieves much higher accuracy than uniform R2 in the same configuration.

Chapter 6

Conclusions and future work

We will now review all the results we have obtained in this thesis and discuss whether they solve the problems we sat out to solve. First we review our proposed improvements, before comparing the corresponding algorithms to their original counterparts. This is done by using the test methods and statistic measures we have developed in chapter 4. Finally we discuss aspects of these methods that are worth studying further.

6.1 Summary of results

In chapter 5 we have gone through a series of steps, each introducing new improvements to the original R2 algorithm proposed by Franklin et al. We now review these steps and compare them to the original algorithm as well as two configurations of the radar algorithm. We evaluate the performance of the algorithms by using both the Alta and Larvik terrains. On both terrains we select 45 observation points according to the recipe in chapter 4, resulting in a total of 90 test cases.

The results of the test are shown in fig. 6.1 and table 6.1. In table 6.2 we have conducted an expected error ratio (EER) analysis, as described in section 4.3. Most of the algorithms in the test should be familiar by now, but a few need some clarification. The *lo*-variants of hybrid and radar are run using three times as many lines of sight (LOSs) as R2. The *hi*-variant of hybrid is unconstrained, but does not evaluate more than ten times as many LOSs as R2 on any of the test cases. The *hi*-variant of radar is run with twelve times as many LOSs as R2. *R2 near* is the original R2 algorithm.

Our improvements started with the generalized R2 algorithm using the weighted- and linear estimators. The EER analysis in table 6.2 shows that these reduce the error of R2 with at least 56% and 78%, respectively. Table 6.1 shows that the weighted variant of R2 is measurably slower than the original. However, this difference is normally considered significant, as it has a t-test p-value of 0.089. The linear variant of R2 is clearly slower than the original, seemingly inflicting a 30% increase in running time.

Comparing the generalized R2 variants to radar, we see that both the weighted- and the linear versions have higher accuracy than radar in the *lo* configuration. This in spite of the fact that radar in this configuration evaluates three times as many LOSs. The *hi* version of radar is better than both R2 variants, but this is hardly a fair comparison, since radar here evaluates twelve times as many LOSs.

The next step we made was analyzing the uniform variant of R2, which allowed us to increase the accuracy by increasing the running time. This led to the hybrid algorithm which works by estimating the boundary of the viewshed using R2 linear, before training the estimator further on an adjustable number of extra LOSs through points on the estimated boundary. As we can see from the plot, these algorithms have accuracy that lie orders of magnitude ahead of the others. Looking at the EER analysis we see that hybrid lo produces less than 0.82% of the error of R2 near. This is an astonishing result given the comparably small increase in running time. Applying the EER method to the running time, we find that hybrid lo increases the running time by less than a factor 4.0. This seems reasonable as it evaluates three times as many LOSs, in addition to the overhead of estimating the viewshed boundary. Compared to the weighted- and linear variants of R2, hybrid lo reduces the error by more than 98% and 95.9%, respectively.

The *hi* variant of hybrid runs without constraints, but as we can see the running times are still acceptable. Using the EER technique on the running time, we see that this variant increases the running time with less than a factor 11.1. In table 6.1 we see that the hybrid algorithm in this configuration classifies the majority of viewsheds correctly. The EER analysis is therefore of limited interest, since no algorithm can do better than 0 error. We do see, however, that hybrid hi makes less than 0.083%, 0.20% and 0.39% of the error of R2 near, -weighted and -linear, respectively.

Following up on the discussions and examples from chapter 4 we know that these results are not necessarily universal for all terrain types. We have also seen examples that indicate that this might not be the case. At the end of chapter 5 we saw that R2 linear and hybrid did not perform as well on

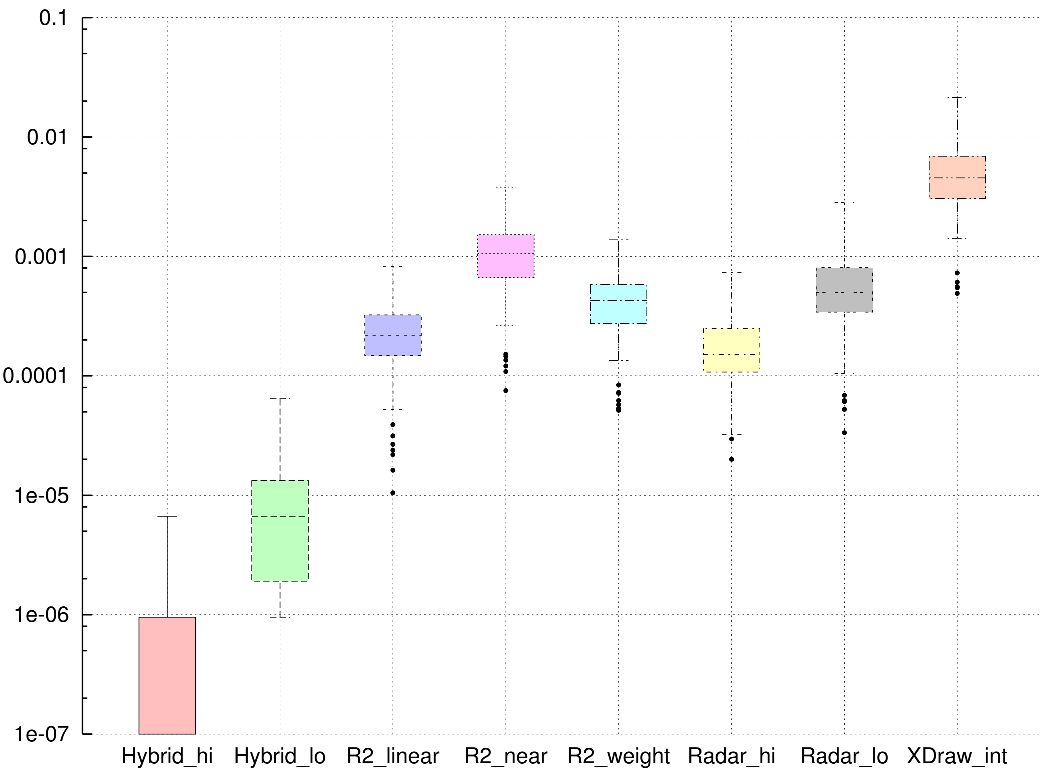


Figure 6.1: The relative error of various algorithms on the combined Alta and Larvik test set. *R2 near* is the original R2 algorithm.

the South Korea data set, as they have done in our other tests. It should, however, be noted that the improvement was still significant by a margin. We can therefore only claim the above results to hold on real world terrain data sets with relatively high resolution and that are reasonably smooth. Our proposed algorithms seem to be significantly better on terrains with lower resolutions as well, but with a smaller gain in accuracy.

6.2 Conclusions

The motivation for this thesis was to investigate viewshed algorithms that are suitable for being used as part of a larger algorithm for planning military operations. We saw in chapter 2 that we need to establish the boundary of the viewshed with reasonable accuracy, as the boundary is essential in attack and observation maneuvers. It is also likely that we have to evaluate a large number of viewsheds, so the algorithms need to be reasonably fast to be

Algorithm	**Median rel. err.**	**1st quartile**	**3rd quartile**	**Mean run. time**
XDraw interpolated	4.5×10^{-3}	3.1×10^{-3}	6.9×10^{-3}	35.4 ms
Radar low	5.0×10^{-4}	3.4×10^{-4}	8.1×10^{-4}	729.1 ms
Radar high	1.5×10^{-4}	1.1×10^{-4}	2.5×10^{-4}	2458.2 ms
R2 near	1.1×10^{-3}	6.7×10^{-4}	1.5×10^{-3}	105.6 ms
R2 weight	4.3×10^{-4}	2.7×10^{-4}	5.8×10^{-4}	107.1 ms
R2 linear	2.2×10^{-4}	1.5×10^{-4}	3.2×10^{-4}	138.5 ms
Hybrid low	6.7×10^{-6}	1.9×10^{-6}	1.3×10^{-5}	408.2 ms
Hybrid high	0	0	9.5×10^{-7}	1025.4 ms

Table 6.1: Alta and Larvik combined performance test. The *low* variants of hybrid and radar are run with three times as many LOSs as R2. The *high* variant of hybrid is unconstrained, but typically does not evaluate more than seven times as many LOSs as R2. The *high* variant of radar is run with twelve times as many LOSs as R2.

	R2 near	**Radar lo**	**R2 weight**	**R2 linear**	**Radar hi**
R2 weight	4.4×10^{-1}	9.0×10^{-1}			
R2 linear	2.2×10^{-1}	4.5×10^{-1}	5.2×10^{-1}		
Hybrid lo	8.2×10^{-3}	1.6×10^{-3}	2.0×10^{-2}	4.1×10^{-2}	5.2×10^{-2}
Hybrid hi	8.3×10^{-4}	1.5×10^{-4}	2.0×10^{-3}	3.9×10^{-3}	5.4×10^{-3}

Table 6.2: EER analysis of the proposed algorithms. The cell in column i and row j contains the EER of algorithms i and j with 99% confidence. Blank cells indicate that the test to show that algorithm i is significantly better than algorithm j failed.

usable.

In chapter 4 we discussed how to evaluate the performance of such algorithms, specifically the importance of using relevant test cases. Using these techniques we established that the R2 algorithm originally proposed by Franklin et al. in [FRM94], gives reasonably good performance both in terms of accuracy and speed for our needs. Adding the weighted- or linear estimator proposed in chapter 5, we get even higher accuracy with no to modest increases in running time.

Should the R2 algorithm be too slow, then the interpolated variant of XDraw should be chosen, as this is much faster, albeit with a significant drop in accuracy. In case we have running time to spare, the hybrid algorithm gives us the flexibility to boost the accuracy, spending the remaining running time.

We have therefore filled the full specter of algorithms in terms of speed and accuracy. Ranging from the fast but inaccurate XDraw, via R2 with the weighted estimator, to the hybrid algorithm, we can obtain good accuracy for any amount of running time. Regardless of what the needs are in the planning algorithm, one of these three candidates should therefore be usable.

6.3 Future work

We have seen that leveraging a priori available knowledge about terrains and viewsheds, specifically that viewshed boundary points seem to be the hardest to classify correctly, can help us greatly improve the efficiency of approximate viewshed algorithms. Although it is beyond the scope of this thesis, it seems natural to investigate whether some terrain features can help us identify points that are likely misclassified. We discussed potential features in chapter 5 that can help us quantify the likelihood for misclassifying each grid point. This can be used in the hybrid algorithm to better prioritize the highlighted points, thus improving the accuracy for a given LOS-count.

The perhaps most natural way to improve the running time of these algorithms, is parallelization. The original R2 algorithm itself is well suited for parallelization since each of the LOSs can be evaluated independently of each other. This can potentially bring down the running time to $O(\sqrt{n})$. Turning to the generalized version of R2 and the way estimators typically are implemented, some care must be taken when operating on the underlying data structures. The potential for improvement is significant.

Acronyms

ECH expanding circular horizon. 24, 25, 39, 44

EER expected error ratio. 48, 71, 85, 96, 106, 107, 109

LOS line of sight. 8, 9, 19–36, 41, 57, 64, 70–72, 77, 78, 81, 87, 88, 90–95, 97, 99, 101–107, 109, 110

RSG regular square grid. 13, 16, 17, 20, 22–25, 34, 39–43, 62, 94

TIN triangulated irregular network. 13, 16, 17, 23–25, 34, 39, 40, 70

Bibliography

[Ata85] Mikhail J Atallah. Some dynamic computational geometry problems. *Computers & Mathematics with Applications*, 11(12):1171–1181, 1985.

[BMCK08] Boaz Ben-Moshe, Paz Carmi, and Matthew J Katz. Approximating the visible region of a point on a terrain. *GeoInformatica*, 12(1):21–36, 2008.

[CS89] Richard Cole and Micha Sharir. Visibility problems for polyhedral terrains. *J. Symb. Comput.*, 7(1):11–30, January 1989.

[FM94] Leila De Floriani and Paola Magillo. Visibility algorithms on triangulated digital terrain models, 1994.

[FRM94] Wm Randolph Franklin, Clark K Ray, and Shashank Mehta. Geometric algorithms for siting of air defense missile batteries. 1994.

[Izr03] David Izraelevitz. A fast algorithm for approximate viewshed computation. *Photogrammetric Engineering & Remote Sensing*, 69(7):767–774, 2003.

[KZ02] Branko Kaučič and Borut Zalik. Comparison of viewshed algorithms on regular spaced points. In *Proceedings of the 18th spring conference on Computer graphics*, pages 177–183. ACM, 2002.

[Ray94] Clark K Ray. Representing visibility for siting problems. Technical report, DTIC Document, 1994.

[XY09] Zhong-Yu Xu and Qi Yao. A novel algorithm for viewshed based on digital elevation model. In *Information Processing, 2009. APCIP 2009. Asia-Pacific Conference on*, volume 2, pages 294–297. IEEE, 2009.

Appendices

Appendix A

Implementations

Listing A.1: VisibilityFinder.hpp

```
//
//  VisibilityFinderBase.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 17/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__VisibilityFinder__
#define __TerrainTest__VisibilityFinder__

#include <iostream>
#include "macros.hpp"
#include "types.hpp"

class VisibilityFinder {
public:
    VisibilityFinder(size_t m, size_t n) : m(m), n(n) {};
    void set_height_data(const double_grid &height_data);
    virtual bool_grid visibility(pos observer, double observer_height, ↩
        double target_height) = 0;

    static bool_grid outer_boundary(const bool_grid &area);
    static bool_grid inner_boundary(const bool_grid &area);
    static bool_grid boundary(const bool_grid &area);
    static bool_grid diff(const bool_grid &lhs, const bool_grid &rhs);
    static bool_grid grid_union(const bool_grid &lhs, const bool_grid &rhs↩
        );
protected:
    double_grid height_data;
    const size_t m, n;

    inline double base_height(const pos observer, const double ↩
        observer_height)
    {
        return height_data[observer.i][observer.j] + observer_height;
    };

    inline static double target_dist(const pos observer, const pos target)
    {
        const int di = target.i - observer.i;
        const int dj = target.j - observer.j;

        return sqrt(di*di + dj*dj);
    }

    inline static bool intersects(double horizon, double slope)
    {
        return horizon > slope - EPS;
    };

    inline double slope(const pos observer, const double base_height, ↩
        const pos target, const double target_height)
    {
        const double h = target_height + height_data[target.i][target.j];

        return (h - base_height)/target_dist(observer, target);
    };
};

#endif /* defined(__TerrainTest__VisibilityFinder__) */
```

Listing A.2: VisibilityFinder.cpp

```
//
//  VisibilityFinderBase.cpp
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 17/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#include <cmath>

#include "macros.hpp"
#include "VisibilityFinder.hpp"

void VisibilityFinder::set_height_data(const double_grid &height_data)
{
    this->height_data = height_data;
}

bool_grid VisibilityFinder::inner_boundary(const bool_grid &area)
{
    size_t m = area.size();
    size_t n = area[0].size();
    bool_grid boundary = vector< vector<bool> >(m, vector<bool>(n));

    for (int i = 1; i < m-1; ++i) {
        for (int j = 1; j < n-1; ++j) {
            if (!area[i][j]) {
                continue;
            }

            boundary[i][j] =
            !area[i-1][j]
            || !area[i][j-1]
            || !area[i][j+1]
            || !area[i+1][j]
            || !area[i-1][j-1]
            || !area[i-1][j+1]
            || !area[i+1][j-1]
            || !area[i+1][j+1];
        }
    }

    return boundary;
};

bool_grid VisibilityFinder::outer_boundary(const bool_grid &area)
{
    size_t m = area.size();
    size_t n = area[0].size();
    bool_grid boundary = vector< vector<bool> >(m, vector<bool>(n));

    for (int i = 1; i < m-1; ++i) {
        for (int j = 1; j < n-1; ++j) {
            if (area[i][j]) {
                continue;
            }

            boundary[i][j] =
            area[i-1][j]
            || area[i][j-1]
            || area[i][j+1]
            || area[i+1][j]
            || area[i-1][j-1]
            || area[i-1][j+1]
            || area[i+1][j-1]
            || area[i+1][j+1];
        }
    }

    return boundary;
};

bool_grid VisibilityFinder::boundary(const bool_grid &area)
{
    size_t m = area.size();
    size_t n = area[0].size();
    bool_grid boundary = vector< vector<bool> >(m, vector<bool>(n));

    for (int i = 1; i < m-1; ++i) {
        for (int j = 1; j < n-1; ++j) {
            if (area[i][j]) {
                boundary[i][j] =
                !area[i-1][j]
                || !area[i][j-1]
                || !area[i][j+1]
                || !area[i+1][j]
                || !area[i-1][j-1]
                || !area[i-1][j+1]
                || !area[i+1][j-1]
                || !area[i+1][j+1];
            } else {
                boundary[i][j] =
                area[i-1][j]
                || area[i][j-1]
                || area[i][j+1]
                || area[i+1][j]
                || area[i-1][j-1]
                || area[i-1][j+1]
                || area[i+1][j-1]
                || area[i+1][j+1];
            }
        }
    }

    return boundary;
};

bool_grid VisibilityFinder::diff(const bool_grid &lhs, const bool_grid &↩
    rhs)
{
    size_t m = lhs.size();
    size_t n = lhs[0].size();

    bool_grid diff(m, vector<bool>(n));

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            diff[i][j] = lhs[i][j] != rhs[i][j];
        }
    }

    return diff;
}

bool_grid VisibilityFinder::grid_union(const bool_grid &lhs, const ↩
    bool_grid &rhs)
{
    size_t m = lhs.size();
    size_t n = lhs[0].size();

    bool_grid uni(m, vector<bool>(n));

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            uni[i][j] = lhs[i][j] || rhs[i][j];
        }
    }

    return uni;
}
```

Listing A.3: LOSVisibilityFinder.hpp

```
//
//  VisibilityFinderBase.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 17/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__LOSVisibilityFinder__
#define __TerrainTest__LOSVisibilityFinder__
```

```
#include <iostream>
#include "macros.hpp"
#include "types.hpp"
#include "VisibilityFinder.hpp"

class LOSVisibilityFinder : public VisibilityFinder {
public:
    LOSVisibilityFinder(size_t m, size_t n) : VisibilityFinder(m, n) {};
protected:
    void eval_los(pos observer, double base_height, double theta, function↩
        <bool(pos, pos, double, double)> cb);
    void eval_los(pos observer, double base_height, pos target, function<↩
        bool(pos, pos, double, double)> cb);
};

#endif /* defined(__TerrainTest__VisibilityFinder__) */
```

Listing A.4: LOSVisibilityFinder.cpp

```
//
//  VisibilityFinderBase.cpp
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 17/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#include <cmath>

#include "macros.hpp"
#include "LOSVisibilityFinder.hpp"

void LOSVisibilityFinder::eval_los(pos observer, double base_height, const↩
     pos target, function<bool(pos, pos, double, double)> cb)
{
    const int di = target.i - observer.i;
    const int dj = target.j - observer.j;

    eval_los(observer, base_height, atan2(di, dj), cb);
}

void LOSVisibilityFinder::eval_los(pos observer, double base_height, const↩
     double theta, function<bool(pos, pos, double, double)> cb)
{
    const double ct = abs(cos(theta));
    const double st = abs(sin(theta));

    const double i_step = 1/st;
    const double j_step = 1/ct;

    const int i_dir = (theta == 0 || theta == M_PI) ? 0 : (theta > 0 ? 1 :↩
         -1);
    const int j_dir = (theta == -M_PI_2 || theta == M_PI_2) ? 0 : (abs(↩
        theta) < M_PI_2 ? 1 : -1);
    const bool dir = i_dir*j_dir >= 0;

    const size_t i_max = (i_dir == 1 ? m - observer.i - 1 : observer.i) - ↩
        abs(i_dir);
    const size_t j_max = (j_dir == 1 ? n - observer.j - 1 : observer.j) - ↩
        abs(j_dir);
    for (int i = 0, j = 0; i <= i_max && j <= j_max;) {
        double i_dist = (i + 1)*i_step;
        double j_dist = (j + 1)*j_step;

        double l, dist;
        pos a, b;
        bool vertical;

        if (j_dist < i_dist) {
            ++j;
            dist = j_dist;
            vertical = false;
            l = st*dist - i;

            a = {.i = i_dir*i + observer.i, .j = j_dir*j + observer.j};
            b = {.i = i_dir*(i+1) + observer.i, .j = j_dir*j + observer.j↩
                };
        } else {
            ++i;
            dist = i_dist;
            vertical = true;
            l = ct*dist - j;

            a = {.i = i_dir*i + observer.i, .j = j_dir*j + observer.j};
            b = {.i = i_dir*i + observer.i, .j = j_dir*(j+1) + observer.j↩
                };
        }

        const double h = (1 - l)*height_data[a.i][a.j] + l*height_data[b.i↩
            ][b.j];
        const double slope = (h - base_height)/dist;
        bool cont;

        if (dir^vertical) {
            cont = cb(a, b, l, slope);
        } else {
            cont = cb(b, a, 1 - l, slope);
        }

        if (!cont) {
            return;
        }
    }
}
```

Listing A.5: macros.hpp

```
//
//  macros.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 16/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef TerrainTest_macros_h
#define TerrainTest_macros_h

#define EPS 1e-7
```

```
#define INFTY numeric_limits<float>().max()/2

#endif
```

Listing A.6: types.cpp

```
//
//  types.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 16/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef TerrainTest_types_h
#define TerrainTest_types_h

#include <vector>

using namespace std;

struct pos {
    int i, j;

    bool operator==(const pos &rhs) const
    {
        return i == rhs.i && j == rhs.j;
    }
};

struct vec2 {
    double x, y;

    bool operator==(const vec2 &rhs) const
    {
        return x == rhs.x && y == rhs.y;
    }
};

typedef vector< vector<bool> > bool_grid;
typedef vector< vector<float> > float_grid;
typedef vector< vector<double> > double_grid;

#endif
```

Listing A.7: R3VisibilityFinder.hpp

```
//
//  R3VisibilityFinder.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 16/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__R3VisibilityFinder__
#define __TerrainTest__R3VisibilityFinder__
```

```
#include <cmath>

#include "LOSVisibilityFinder.hpp"
#include "types.hpp"

class R3VisibilityFinder : public LOSVisibilityFinder {
public:
    R3VisibilityFinder(size_t m, size_t n) : LOSVisibilityFinder(m, n) {};

    bool_grid visibility(pos observer, double observer_height, double ↩
        target_height)
    {
        bool_grid visible = vector< vector<bool> >(m, vector<bool>(n));
        const double base_height = VisibilityFinder::base_height(observer,↩
             observer_height);

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                visible[i][j] = eval_target(observer, base_height, {.i = i↩
                    , .j = j}, target_height);
            }
        }

        visible[observer.i][observer.j] = true;

        return visible;
    };
private:
    inline bool eval_target(const pos &observer, const double base_height,↩
         const pos &target, const double target_height)
    {
        const double target_slope = slope(observer, base_height, target, ↩
            target_height);
        bool visible = false;

        eval_los(observer, base_height, target, [&](const pos a, const pos↩
             b, const double l, const double slope) mutable -> bool
        {
            if (a == target || b == target) {
                visible = true;

                return false;
            }

            return !intersects(slope, target_slope);
        });

        return visible;
    };
};

#endif /* defined(__TerrainTest__R3VisibilityFinder__) */
```

Listing A.8: R2VisibilityFinder.hpp

```
//
//  R3VisibilityFinder.h
//  TerrainTest
//
```

```
//  Created by Martin Vonheim Larsen on 16/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__R2VisibilityFinder__
#define __TerrainTest__R2VisibilityFinder__

#include "LOSVisibilityFinder.hpp"
#include "R2Estimator.hpp"
#include "macros.hpp"
#include "types.hpp"

template<class Estimator>
class R2VisibilityFinder : public LOSVisibilityFinder {
public:
    R2VisibilityFinder(size_t m, size_t n) : LOSVisibilityFinder(m, n), ↩
        estimator(m, n), slopes(m, vector<double>(n)) {};

    bool_grid visibility(pos observer, double observer_height, double ↩
        target_height)
    {
        estimator.reset();

        const double base_height = height_data[observer.i][observer.j] + ↩
            observer_height;
        precalc_slopes(observer, base_height, target_height);
        eval_boundary_targets(observer, base_height);

        return visible();
    };
protected:
    Estimator estimator;
    double_grid slopes;

    void eval_boundary_targets(const pos observer, const double ↩
        base_height)
    {
        for (int i = 0; i < m; ++i) {
            eval_target(observer, base_height, (pos){.i = i, .j = 0});
            eval_target(observer, base_height, (pos){.i = i, .j = (int)n↩
                -1});
        }

        for (int j = 0; j < n; ++j) {
            eval_target(observer, base_height, (pos){.i = 0, .j = j});
            eval_target(observer, base_height, (pos){.i = (int)m-1, .j = j↩
                });
        }
    };

    inline virtual void eval_target(const pos observer, const double ↩
        base_height, const pos target)
    {
        double horizon = -INFTY;

        eval_los(observer, base_height, target, [=](const pos lhs, const ↩
            pos rhs, const double l, const double slope) mutable -> bool
        {
            estimator.train(lhs, l, true, horizon);
            estimator.train(rhs, 1 - l, false, horizon);

            horizon = max(horizon, slope);
            return true;
        });
    };

    inline virtual void eval_target(const pos observer, const double ↩
        base_height, const double theta)
    {
        double horizon = -INFTY;

        eval_los(observer, base_height, theta, [=](const pos lhs, const ↩
            pos rhs, const double l, const double slope) mutable -> bool
        {
            estimator.train(lhs, l, true, horizon);
            estimator.train(rhs, 1 - l, false, horizon);

            horizon = max(horizon, slope);

            return true;
        });
    };

    void precalc_slopes(const pos observer, const double base_height, ↩
        const double target_height)
    {
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                slopes[i][j] = slope(observer, base_height, {.i = i, .j = ↩
                    j}, target_height);
            }
        }

        slopes[observer.i][observer.j] = INFTY;
    };

    bool_grid visible()
    {
        bool_grid v = vector< vector<bool> >(m, vector<bool>(n));

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                v[i][j] = visible({.i = i, .j = j});
            }
        }

        return v;
    };

    inline bool visible(const pos p)
    {
        return !intersects(estimator.estimate(p), slopes[p.i][p.j]);
    };
};

#endif /* defined(__TerrainTest__R2VisibilityFinder__) */
```

Listing A.9: R2Estimator.hpp

```
//
//  R2EstimationPolicy.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 16/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__R2EstimationPolicy__
#define __TerrainTest__R2EstimationPolicy__

#include "types.hpp"

class R2Estimator {
public:
    R2Estimator(size_t m, size_t n) : m(m), n(n) {};

    /** Trains the estimator with the horizon of a grid line
        intersection of some LOS in the neighborhood of a grid
        point `p`.

        `dist` is the distance from the grid line intersection to `p`
        `is_left` indicates whether the LOS is left or right of `p`
        `horizon` is the horizon at the grid line intersection
     */
    virtual void train(const pos p, const double dist, const bool is_left,↩
         const double horizon) = 0;

    /** Estimates the horizon at some grid point `p` */
    virtual double estimate(const pos p) = 0;
protected:
    size_t m, n;
};

#endif /* defined(__TerrainTest__R2EstimationPolicy__) */
```

Listing A.10: R2BasicEstimator.hpp

```
//
//  R2EstimationPolicy.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 16/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__R2BasicEstimator__
#define __TerrainTest__R2BasicEstimator__

#include "types.hpp"
#include "R2Estimator.hpp"

class R2BasicEstimator : public R2Estimator {
public:
    R2BasicEstimator(size_t m, size_t n)
    : R2Estimator(m, n), init(0), estimated_horizon(m, vector<double>(n)) ↩
        {};

    void reset()
    {
        for (int i = 0; i < m; ++i) {
            fill(estimated_horizon[i].begin(), estimated_horizon[i].end(),↩
                init);
        }
    };

    inline double estimate(const pos p)
    {
        return estimated_horizon[p.i][p.j];
    };
protected:
    R2BasicEstimator(size_t m, size_t n, double init)
    : R2Estimator(m, n), init(init), estimated_horizon(m, vector<double>(n↩
        , init)) {};

    double_grid estimated_horizon;
    double init;
};

#endif /* defined(__TerrainTest__R2EstimationPolicy__) */
```

Listing A.11: R2NearestNeighborEstimator.hpp

```
//
//  R2NearestNeighborEstimator.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 16/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__R2NearestNeighborEstimator__
#define __TerrainTest__R2NearestNeighborEstimator__

#include "R2BasicEstimator.hpp"
#include "macros.hpp"

class R2NearestNeighborEstimator : public R2BasicEstimator {
public:
    R2NearestNeighborEstimator(size_t m, size_t n)
    : R2BasicEstimator(m, n), min_dist(m, vector<double>(n, INFTY)) {};

    void reset()
    {
        R2BasicEstimator::reset();

        for (int i = 0; i < m; ++i) {
            fill(min_dist[i].begin(), min_dist[i].end(), INFTY);
        }
    };

    inline void train(const pos p, const double dist, const bool is_left, ↩
        const double horizon)
    {
        if (min_dist[p.i][p.j] <= dist) {
            return;
        }

        min_dist[p.i][p.j] = dist;
        estimated_horizon[p.i][p.j] = horizon;
    };
protected:
    double_grid min_dist;
};

#endif /* defined(__TerrainTest__R2NearestNeighborEstimator__) */
```

Listing A.12: R2PWeightedEstimator.hpp

```
//
//  R2PWeightedEstimator.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 27/03/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__R2PWeightedEstimator__
#define __TerrainTest__R2PWeightedEstimator__

#include "R2Estimator.hpp"
#include "macros.hpp"

class R2PWeightedEstimator : public R2Estimator {
public:
    R2PWeightedEstimator(size_t m, size_t n)
    : R2Estimator(m, n), numerator(m, vector<double>(n)), denominator(m, ↩
        vector<double>(n)) {}

    void reset()
    {
        for (int i = 0; i < m; ++i) {
            fill(numerator[i].begin(), numerator[i].end(), 0);
            fill(denominator[i].begin(), denominator[i].end(), 0);
        }
    };

    inline void train(const pos p, const double dist, const bool is_left, ↩
        const double horizon)
    {
        const double weight = dist < EPS ? 1/EPS : pow(1 - dist, P);

        numerator[p.i][p.j] += weight*horizon;
        denominator[p.i][p.j] += weight;
    };

    inline double estimate(const pos p)
    {
        const double num = numerator[p.i][p.j];
        const double den = denominator[p.i][p.j];

        if (num == 0 && den == 0) {
            return -INFTY;
        } else if (den == 0) {
            return INFTY;
        }

        return num/den;
    };

    double P = 2;
protected:
    double_grid numerator, denominator;
};

#endif /* defined(__TerrainTest__R2PWeightedEstimator__) */
```

Listing A.13: R2LinearEstimator.hpp

```
//
//  R2WeightedEstimationPolicy.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 16/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__R2LinearEstimator__
#define __TerrainTest__R2LinearEstimator__

#include "R2Estimator.hpp"
#include "macros.hpp"

class R2LinearEstimator : public R2Estimator {
public:
    R2LinearEstimator(size_t m, size_t n)
    : R2Estimator(m, n), l_dist(m, vector<double>(n)), l_hor(m, vector<↩
        double>(n)), r_dist(m, vector<double>(n)), r_hor(m, vector<double↩
        >(n)) {};

    void reset()
    {
        for (int i = 0; i < m; ++i) {
            fill(l_dist[i].begin(), l_dist[i].end(), INFTY);
            fill(r_dist[i].begin(), r_dist[i].end(), INFTY);
        }
    };

    inline void train(const pos p, const double dist, const bool is_left, ↩
        const double horizon)
    {
        if (is_left && dist < l_dist[p.i][p.j]) {
            l_dist[p.i][p.j] = dist;
            l_hor[p.i][p.j] = horizon;
        } else if (!is_left && dist < r_dist[p.i][p.j]) {
            r_dist[p.i][p.j] = dist;
            r_hor[p.i][p.j] = horizon;
        }
    };

    inline double estimate(const pos p)
    {
        const double l = l_dist[p.i][p.j]/(l_dist[p.i][p.j] + r_dist[p.i][↩
            p.j]);

        return (1 - l)*l_hor[p.i][p.j] + l*r_hor[p.i][p.j];
    }
protected:
    double_grid l_dist, r_dist, l_hor, r_hor;
};

#endif /* defined(__TerrainTest__R2WeightedEstimationPolicy__) */
```

Listing A.14: R2MaxEstimator.hpp

```
//
//  R2MaxEstimationPolicy.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 16/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__R2MaxEstimationPolicy__
#define __TerrainTest__R2MaxEstimationPolicy__

#include "R2BasicEstimator.hpp"
#include "macros.hpp"

class R2MaxEstimator : public R2BasicEstimator {
public:
    R2MaxEstimator(size_t m, size_t n) : R2BasicEstimator(m, n, -INFTY) ↩
        {};
    inline void train(const pos p, const double dist, const bool is_left, ↩
        const double horizon)
    {
        estimated_horizon[p.i][p.j] = max(estimated_horizon[p.i][p.j], ↩
            horizon);
    };
};

#endif /* defined(__TerrainTest__R2MaxEstimationPolicy__) */
```

Listing A.15: R2MinEstimator.hpp

```
//
//  R2MinEstimationPolicy.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 16/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__R2MinEstimationPolicy__
#define __TerrainTest__R2MinEstimationPolicy__

#include "R2BasicEstimator.hpp"
#include "macros.hpp"

class R2MinEstimator : public R2BasicEstimator {
public:
    R2MinEstimator(size_t m, size_t n) : R2BasicEstimator(m, n, INFTY) {};

    inline void train(const pos p, const double dist, const bool is_left, ↩
        const double horizon)
    {
        estimated_horizon[p.i][p.j] = min(estimated_horizon[p.i][p.j], ↩
            horizon);
    };
};

#endif /* defined(__TerrainTest__R2MinEstimationPolicy__) */
```

Listing A.16: XDrawVisibilityFinder.hpp

```
//
//  XDrawVisiblityFinder.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 20/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__XDrawVisiblityFinder__
#define __TerrainTest__XDrawVisiblityFinder__

#include <vector>

#include "VisibilityFinder.hpp"
#include "macros.hpp"
#include "types.hpp"

using namespace std;

template<class Estimator>
class XDrawVisiblityFinder : public VisibilityFinder {
public:
    XDrawVisiblityFinder(size_t m, size_t n) : VisibilityFinder(m, n), ↩
        theta(m, vector<double>(n)) {};

    bool_grid visibility(pos observer, double observer_height, double ↩
        target_height)
    {
        calc_theta(observer, observer_height);


        double base_height = (height_data[observer.i][observer.j] + ↩
            observer_height);
        bool_grid visible = vector< vector<bool> >(m, vector<bool>(n));

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (i == observer.i && j == observer.j) {
                    visible[i][j] = true;
                    continue;
                }

                int di = observer.i - i;
                int dj = observer.j - j;

                double dl = sqrt(di*di + dj*dj);
                double dh = height_data[i][j] + target_height - ↩
                    base_height;
                double target_theta = dh/dl;
                double diff = target_theta - theta[i][j];

                visible[i][j] = diff > EPS;
            }
        }

        return visible;
    }
protected:
    double_grid theta;
    Estimator estimator;
```

```
void calc_theta(pos observer, double observer_height)
{
    theta[observer.i][observer.j] = -INFTY;

    calc_axis_theta(observer, observer_height);
    calc_diag_theta(observer, observer_height);
    calc_internal_theta(observer, observer_height);
};

void calc_axis_theta(pos observer, double observer_height)
{
    double base_height = (height_data[observer.i][observer.j] + ↩
        observer_height);

    // north
    for (int i = observer.i-1; i >= 0; --i) {
        double dl = observer.i - i;
        double dh = height_data[i][observer.j] - base_height;

        theta[i][observer.j] = max(dh/dl, theta[i+1][observer.j]);
    }

    // south
    for (int i = observer.i+1; i < height_data.size(); ++i) {
        double dl = i - observer.i;
        double dh = height_data[i][observer.j] - base_height;

        theta[i][observer.j] = max(dh/dl, theta[i-1][observer.j]);
    }

    // west
    for (int j = observer.j-1; j >= 0; --j) {
        double dl = observer.j - j;
        double dh = height_data[observer.i][j] - base_height;

        theta[observer.i][j] = max(dh/dl, theta[observer.i][j+1]);
    }

    // east
    for (int j = observer.j+1; j < height_data[0].size(); ++j) {
        double dl = j - observer.j;
        double dh = height_data[observer.i][j] - base_height;

        theta[observer.i][j] = max(dh/dl, theta[observer.i][j-1]);
    }
};

void calc_diag_theta(pos observer, double observer_height)
{
    double base_height = (height_data[observer.i][observer.j] + ↩
        observer_height);
    int north = 0;
    int west = 0;
    int south = (int)height_data.size() - 1;
    int east = (int)height_data[0].size() - 1;

    // north-west
    for (int k = 1;; ++k) {
        int i = observer.i - k;
        int j = observer.j - k;
        if (i < north || j < west) {
            break;
        }

        double dl = sqrt(2*k*k);
        double dh = height_data[i][j] - base_height;

        theta[i][j] = max(dh/dl, theta[i+1][j+1]);
    }

    // north-east
    for (int k = 1;; ++k) {
        int i = observer.i - k;
        int j = observer.j + k;

        if (i < north || j > east) {
            break;
        }

        double dl = sqrt(2*k*k);
        double dh = height_data[i][j] - base_height;

        theta[i][j] = max(dh/dl, theta[i+1][j-1]);
    }

    // south-east
    for (int k = 1;; ++k) {
        int i = observer.i + k;
        int j = observer.j + k;

        if (i > south || j > east) {
            break;
        }

        double dl = sqrt(2*k*k);
        double dh = height_data[i][j] - base_height;

        theta[i][j] = max(dh/dl, theta[i-1][j-1]);
    }

    // south-west
    for (int k = 1;; ++k) {
        int i = observer.i + k;
        int j = observer.j - k;

        if (i > south || j < west) {
            break;
        }

        double dl = sqrt(2*k*k);
        double dh = height_data[i][j] - base_height;

        theta[i][j] = max(dh/dl, theta[i-1][j+1]);
    }
};

void calc_internal_theta(pos observer, double observer_height)
{
    double base_height = (height_data[observer.i][observer.j] + ↩
        observer_height);
    int max_i = (int)height_data.size();
    int max_j = (int)height_data[0].size();
```

```
// north-north-west
for (int di = -2; observer.i + di >= 0; --di) {
    for (int dj = -1; dj > di && observer.j + dj >= 0; --dj) {
        int i = observer.i + di;
        int j = observer.j + dj;

        double dl = sqrt(di*di + dj*dj);
        double dh = height_data[i][j] - base_height;

        theta[i][j] = max(dh/dl, estimator.estimate_horizon(theta[↩
            i+1][j], theta[i+1][j+1], di, dj));
    }
}

// north-north-east
for (int di = -2; observer.i + di >= 0; --di) {
    for (int dj = 1; dj < -di && observer.j + dj < max_j; ++dj) {
        int i = observer.i + di;
        int j = observer.j + dj;

        double dl = sqrt(di*di + dj*dj);
        double dh = height_data[i][j] - base_height;

        theta[i][j] = max(dh/dl, estimator.estimate_horizon(theta[↩
            i+1][j], theta[i+1][j-1], di, dj));
    }
}

// south-south-west
for (int di = 2; observer.i + di < max_i; ++di) {
    for (int dj = -1; -dj < di && observer.j + dj >= 0; --dj) {
        int i = observer.i + di;
        int j = observer.j + dj;

        double dl = sqrt(di*di + dj*dj);
        double dh = height_data[i][j] - base_height;

        theta[i][j] = max(dh/dl, estimator.estimate_horizon(theta[↩
            i-1][j], theta[i-1][j+1], di, dj));
    }
}

// south-south-east
for (int di = 2; observer.i + di < max_i; ++di) {
    for (int dj = 1; dj < di && observer.j + dj < max_j; ++dj) {
        int i = observer.i + di;
        int j = observer.j + dj;

        double dl = sqrt(di*di + dj*dj);
        double dh = height_data[i][j] - base_height;

        theta[i][j] = max(dh/dl, estimator.estimate_horizon(theta[↩
            i-1][j], theta[i-1][j-1], di, dj));
    }
}

// north-west-west
for (int dj = -2; observer.j + dj >= 0; --dj) {
    for (int di = -1; di > dj && observer.i + di >= 0; --di) {
        int i = observer.i + di;
        int j = observer.j + dj;
```

```
                double dl = sqrt(di*di + dj*dj);
                double dh = height_data[i][j] - base_height;

                theta[i][j] = max(dh/dl, estimator.estimate_horizon(theta[↩
                    i][j+1], theta[i+1][j+1], di, dj));
            }
        }

        // north-east-east
        for (int dj = 2; observer.j + dj < max_j; ++dj) {
            for (int di = -1; -di < dj && observer.i + di >= 0; --di) {
                int i = observer.i + di;
                int j = observer.j + dj;

                double dl = sqrt(di*di + dj*dj);
                double dh = height_data[i][j] - base_height;

                theta[i][j] = max(dh/dl, estimator.estimate_horizon(theta[↩
                    i][j-1], theta[i+1][j-1], di, dj));
            }
        }

        // south-west-west
        for (int dj = -2; observer.j + dj >= 0; --dj) {
            for (int di = 1; di < -dj && observer.i + di < max_i; ++di) {
                int i = observer.i + di;
                int j = observer.j + dj;

                double dl = sqrt(di*di + dj*dj);
                double dh = height_data[i][j] - base_height;

                theta[i][j] = max(dh/dl, estimator.estimate_horizon(theta[↩
                    i][j+1], theta[i-1][j+1], di, dj));
            }
        }

        // south-east-east
        for (int dj = 2; observer.j + dj < max_j; ++dj) {
            for (int di = 1; di < dj && observer.i + di < max_i; ++di) {
                int i = observer.i + di;
                int j = observer.j + dj;

                double dl = sqrt(di*di + dj*dj);
                double dh = height_data[i][j] - base_height;

                theta[i][j] = max(dh/dl, estimator.estimate_horizon(theta[↩
                    i][j-1], theta[i-1][j-1], di, dj));
            }
        }
    };
};

#endif /* defined(__TerrainTest__XDrawVisiblityFinder__) */
```

Listing A.17: XDrawInterpolatedEstimator.hpp

```
//
//  XDrawInterpolatedEstimator.h
//  TerrainTest
```

```
//
//  Created by Martin Vonheim Larsen on 20/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__XDrawInterpolatedEstimator__
#define __TerrainTest__XDrawInterpolatedEstimator__

#include <cmath>

class XDrawInterpolatedEstimator {
public:
    inline double estimate_horizon(const double near_horizon, const double↩
         far_horizon, const int di, const int dj)
    {
        const double l = abs(abs(di) < abs(dj) ? (double)di/dj : (double)↩
            dj/di);
        return (1 - l)*near_horizon + l*far_horizon;
    };
};

#endif /* defined(__TerrainTest__XDrawInterpolatedEstimator__) */
```

Listing A.18: XDrawMaxEstimator.hpp

```
//
//  XDrawMaxEstimator.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 20/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__XDrawMaxEstimator__
#define __TerrainTest__XDrawMaxEstimator__

#include <algorithm>

using namespace std;

class XDrawMaxEstimator {
public:
    inline double estimate_horizon(const double lhs_horizon, const double ↩
        rhs_horizon, const int di, const int dj)
    {
        return max(lhs_horizon, rhs_horizon);
    };
};

#endif /* defined(__TerrainTest__XDrawMaxEstimator__) */
```

Listing A.19: XDrawMinEstimator.hpp

```
//
//  XDrawMinEstimator.h
//  TerrainTest
//
```

```
//  Created by Martin Vonheim Larsen on 20/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__XDrawMinEstimator__
#define __TerrainTest__XDrawMinEstimator__

#include <algorithm>

using namespace std;

class XDrawMinEstimator {
public:
    inline double estimate_horizon(const double lhs_horizon, const double ↩
        rhs_horizon, const int di, const int dj)
    {
        return min(lhs_horizon, rhs_horizon);
    };
};

#endif /* defined(__TerrainTest__XDrawMinEstimator__) */
```

Listing A.20: XDrawMeanEstimator.hpp

```
//
//  XDrawMeanEstimator.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 20/02/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__XDrawMeanEstimator__
#define __TerrainTest__XDrawMeanEstimator__

#include <algorithm>

using namespace std;

class XDrawMeanEstimator {
public:
    inline double estimate_horizon(const double lhs_horizon, const double ↩
        rhs_horizon, const int di, const int dj)
    {
        return (lhs_horizon + rhs_horizon)/2;
    };
};

#endif /* defined(__TerrainTest__XDrawMeanEstimator__) */
```

Listing A.21: RadarVisibilityFinder.hpp

```
//
//  RadarVisibilityFinder.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 21/03/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__RadarVisibilityFinder__
#define __TerrainTest__RadarVisibilityFinder__

#include <cmath>
#include <deque>

#include "macros.hpp"
#include "VisibilityFinder.hpp"

class RadarVisiblityFinder : public VisibilityFinder {
public:
    RadarVisiblityFinder(size_t m, size_t n, double K) : VisibilityFinder(↩
        m, n), num_sectors((size_t)floor(K*(2*m + 2*n - 4))), sector_size↩
        (2*M_PI/(num_sectors-1)) {};

    bool_grid visibility(pos observer, double observer_height, double ↩
        target_height)
    {
        vector<vector<pos>> sector_points = get_sector_points(observer);

        bool_grid visible = vector<vector<bool>>(m, vector<bool>(n));

        vector<double> prev_rl = eval_LOS(observer, observer_height, ↩
            target_height, 0);

        for (int sector = 0; sector < num_sectors; ++sector) {
            vector<double> rl = eval_LOS(observer, observer_height, ↩
                target_height, (sector + 1)*sector_size);
            interpolate(visible, prev_rl, rl, sector_points[sector], ↩
                observer);

            prev_rl = rl;
        }

        return visible;
    };
protected:
    const size_t num_sectors;
    const double sector_size;

    inline vector<double> eval_LOS(pos observer, double observer_height, ↩
        double target_height, double theta)
    {
        vector<double> run_length;
        theta -= M_PI;

        double ct = cos(theta);
        double st = sin(theta);
        double tt = tan(theta);
        double ast = abs(st);
        double act = abs(ct);
        double att = abs(tt);

        int dir_x = (theta > -M_PI_2 && theta < M_PI_2) ? 1 : -1;
        int dir_y = theta < 0 ? -1 : 1;

        bool visible = true;
        double el = -INFTY;
        vec2 p = {.x=0, .y=0};
        double dist = 0;
        double tel = -INFTY;

        for (;;) {
            double step_x = ceil(abs(p.x) + EPS) - abs(p.x);
            double step_y = ceil(abs(p.y) + EPS) - abs(p.y);
            double dist_x = step_x/act;
            double dist_y = step_y/ast;

            if (dist_x <= dist_y) {
                step_y = step_x*att;
            } else {
                step_x = step_y/att;
            }

            vec2 pos = {.x=observer.j + p.x + dir_x*step_x, .y=observer.i ↩
                + p.y + dir_y*step_y};

            if (!(0 < pos.x + EPS && 0 < pos.y + EPS && pos.x - EPS < n - ↩
                1 && pos.y - EPS < m - 1)) {
                break;
            }

            double new_dist = mag(pos, observer);
            double new_tel = eta(observer, observer_height, pos, ↩
                target_height);

            if ((new_tel > el + EPS) ^ visible) { // visibility status has↩
                 changed
                double x = new_dist;

                run_length.push_back(x);
                visible = !visible;
            }

            p = {.x=p.x + dir_x*step_x, .y=p.y + dir_y*step_y};
            dist = new_dist;
            tel = new_tel;
            el = max(el, eta(observer, observer_height, pos, 0));
        }

        return run_length;
    };

    inline double mag(const vec2 p, const pos observer)
    {
        const double dx = p.x - observer.j;
        const double dy = p.y - observer.i;

        return sqrt(dx*dx + dy*dy);
    };

    inline double eta(pos observer, double observer_height, vec2 target, ↩
        double target_height)
    {
        vec2 p = target;
        double h = 0;

        if (abs(p.x - round(p.x)) < EPS && abs(p.y - round(p.y)) < EPS) {
            int i = (int)round(p.y);
            int j = (int)round(p.x);
```

```
            h = height_data[i][j];
        } else if (abs(p.x - round(p.x)) < EPS) {
            int i = max(0, (int)floor(p.y));
            int j = (int)round(p.x);

            h = (p.y - i)*height_data[i+1][j] + (i + 1 - p.y)*height_data[↩
                i][j];
        } else if (abs(p.y - round(p.y)) < EPS) {
            int i = (int)round(p.y);
            int j = max(0, (int)floor(p.x));

            h = (p.x - j)*height_data[i][j+1] + (j + 1 - p.x)*height_data[↩
                i][j];
        } else {
            cout << "foo" << endl;
            // p is not on any grid line
            // elevation is undefined
            // this is an error
        }

        vec2 v = {.x = observer.j - p.x, .y = observer.i - p.y};
        double d = sqrt(v.x*v.x + v.y*v.y);
        return ((target_height + h) - (observer_height + height_data[↩
            observer.i][observer.j])) / d;
};

inline void interpolate(bool_grid &visible, const vector<double> &lhs,↩
     const vector<double> &rhs, const vector<pos> &points, const pos ↩
    observer)
{
    bool lhs_visible = true;
    bool rhs_visible = true;
    int lhs_idx = -1;
    int rhs_idx = -1;

    for (pos p : points) {
        vec2 p_ = get_sector_pos(p, observer);

        // loop forward to relevant section at lhs and rhs
        for (; lhs_idx + 1 < lhs.size() && lhs[lhs_idx + 1] < p_.x; ++↩
            lhs_idx, lhs_visible = !lhs_visible);
        for (; rhs_idx + 1 < rhs.size() && rhs[rhs_idx + 1] < p_.x; ++↩
            rhs_idx, rhs_visible = !rhs_visible);

        double lhs_lo = lhs_idx > -1 ? lhs[lhs_idx] : 0;
        double lhs_hi = lhs_idx + 1 < lhs.size() ? lhs[lhs_idx + 1] : ↩
            INFINITY;
        double rhs_lo = rhs_idx > -1 ? rhs[rhs_idx] : 0;
        double rhs_hi = rhs_idx + 1 < rhs.size() ? rhs[rhs_idx + 1] : ↩
            INFINITY;

        visible[p.i][p.j] = point_visible(lhs_lo, lhs_hi, lhs_visible,↩
             rhs_lo, rhs_hi, rhs_visible, p_);
    }
}

inline const bool point_visible(double up_lo, double up_hi, bool ↩
    up_visible, double down_lo, double down_hi, bool down_visible, ↩
    vec2 p)
{
    if (up_visible == down_visible) {
        return up_visible;
    }

    if (up_lo > down_lo) {
        // flip up/down
        return point_visible(down_lo, down_hi, down_visible, up_lo, ↩
            up_hi, up_visible, {.x=p.x, .y=1 - p.y});
    }

    double lo = down_lo;
    double hi = min(up_hi, down_hi);
    double m = (lo + hi)/2;

    if (p.x <= m) {
        // left of center, which is where up dominates the side ↩
            triangle

        if ((hi - lo)*p.y < p.x - lo) {
            return up_visible;
        } else {
            return down_visible;
        }
    } else {
        // right of center
        if (up_hi > down_hi) {
            // up dominates the side triangle

            if ((hi - lo)*(1 - p.y) < p.x - lo) {
                return up_visible;
            } else {
                return down_visible;
            }
        } else {
            // down dominates the side triangle

            if ((hi - lo)*p.y < p.x - lo) {
                return down_visible;
            } else {
                return up_visible;
            }
        }
    }
};

vector<vector<pos>> get_sector_points(const pos observer)
{
    vector<vector<pos>> sector_points(num_sectors, vector<pos>());

    bool_grid visited = vector<vector<bool>>(m, vector<bool>(n, false)↩
        );
    visited[observer.i][observer.j] = true;

    deque<pos> q;
    q.push_back(observer);

    pos dps[] = {{.i=1, .j=0}, {.i=-1, .j=0}, {.i=0, .j=1}, {.i=0, .j↩
        =-1}};

    while (!q.empty()) {
        pos p = q.front();
        q.pop_front();
```

```
            sector_points[get_sector_idx(p, observer)].push_back(p);

            for (pos dp : dps) {
                pos np = {.i=p.i + dp.i, .j=p.j + dp.j};

                if (np.i < 0 || np.i >= m || np.j < 0 || np.j >= n || ↩
                    visited[np.i][np.j]) {
                    continue;
                }

                visited[np.i][np.j] = true;
                q.push_back(np);
            }
        }

        return sector_points;
    };

    inline double get_theta(const pos p, const pos observer)
    {
        return atan2(p.i - observer.i, p.j - observer.j) + M_PI;
    };

    inline int get_sector_idx(const pos p, const pos observer)
    {
        return ((int)floor(get_theta(p, observer)/sector_size));
    };

    inline vec2 get_sector_pos(const pos p, const pos observer)
    {
        int di = p.i - observer.i;
        int dj = p.j - observer.j;
        double h = get_theta(p, observer)/sector_size;

        return {.x=sqrt(di*di + dj*dj), .y=h - floor(h)};
    };
};

#endif /* defined(__TerrainTest__RadarVisibilityFinder__) */
```

Listing A.22: R2UniformVisibilityFinder.hpp

```
//
//  R2UniformVisibilityFinder.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 16/05/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__R2UniformVisibilityFinder__
#define __TerrainTest__R2UniformVisibilityFinder__

#include "R2VisibilityFinder.hpp"
#include "macros.hpp"
#include "types.hpp"

template<class Estimator>
class R2UniformVisibilityFinder : public R2VisibilityFinder<Estimator> {
public:
    R2UniformVisibilityFinder(size_t m, size_t n, double K) : ↩
        R2VisibilityFinder<Estimator>(m, n), num_sectors((size_t)floor(K↩
        *(2*m + 2*n - 4))) {};

    bool_grid visibility(pos observer, double observer_height, double ↩
        target_height)
    {
        estimator.reset();

        const double base_height = VisibilityFinder::base_height(observer,↩
             observer_height);
        precalc_slopes(observer, base_height, target_height);

        const double sector_size = 2*M_PI/num_sectors;

        for (int sector = 0; sector < num_sectors; ++sector) {
            eval_target(observer, base_height, sector*sector_size - M_PI);
        }

        return visible();
    };
protected:
    using R2VisibilityFinder<Estimator>::estimator;

    const size_t num_sectors;

    using R2VisibilityFinder<Estimator>::eval_target;
    using R2VisibilityFinder<Estimator>::precalc_slopes;
    using R2VisibilityFinder<Estimator>::visible;
};

#endif /* defined(__TerrainTest__R2UniformVisibilityFinder__) */
```

Listing A.23: HybridVisibilityFinder.hpp

```
//
//  HybridFastVisibilityFinder.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 23/03/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__HybridVisibilityFinder__
#define __TerrainTest__HybridVisibilityFinder__

#include "R2VisibilityFinder.hpp"
#include "R2MaxEstimator.hpp"
#include "R2MinEstimator.hpp"
#include "macros.hpp"
#include "types.hpp"

template<class Estimator>
class HybridVisibilityFinder : public R2VisibilityFinder<Estimator> {
public:
    HybridVisibilityFinder(size_t m, size_t n, double K) : ↩
        R2VisibilityFinder<Estimator>(m, n), max_estimator(m, n), ↩
        min_estimator(m, n), K(K) {};

    bool_grid visibility(pos observer, double observer_height, double ↩
        target_height)
    {
        reset();

        const double base_height = VisibilityFinder::base_height(observer,↩
             observer_height);
        precalc_slopes(observer, base_height, target_height);
        eval_boundary_targets(observer, base_height);

        vector<double> angles;
        angles.reserve(100000);

        for (int i = 0; i < m; ++i) {
            if (i == observer.i) {
                continue;
            }

            for (int j = 0; j < n; ++j) {
                const pos p = {.i=i, .j=j};

                if (max_visible(p) != min_visible(p)) {
                    angles.push_back(atan2(i - observer.i, j - observer.j)↩
                        );
                }
            }
        }

        const size_t r2_size = (size_t)floor(K*(2*m + 2*n - 4));
        const size_t n_angles = angles.size();

        if (n_angles < 1.5*r2_size) {
            for (double theta : angles) {
                eval_target(observer, base_height, theta);
            }
        } else {
            sort(angles.begin(), angles.end());

            for (int k = 0; k < r2_size; ++k) {
                const size_t i = (k*n_angles)/r2_size;
                const double theta = angles[i];

                eval_target(observer, base_height, theta);
            }
        }

        return visible();
    };
protected:
    using R2VisibilityFinder<Estimator>::estimator;
    using R2VisibilityFinder<Estimator>::slopes;
    using VisibilityFinder::m;
    using VisibilityFinder::n;

    const double K;
    R2MaxEstimator max_estimator;
    R2MinEstimator min_estimator;

    using R2VisibilityFinder<Estimator>::eval_boundary_targets;
    using R2VisibilityFinder<Estimator>::eval_target;
    using R2VisibilityFinder<Estimator>::precalc_slopes;
    using R2VisibilityFinder<Estimator>::visible;
    using LOSVisibilityFinder::eval_los;
    using VisibilityFinder::intersects;

    void reset()
    {
        estimator.reset();

        max_estimator.reset();
        min_estimator.reset();
    };

    inline void eval_target(const pos observer, const double base_height, ↩
        const pos target)
    {
        double horizon = -INFTY;

        eval_los(observer, base_height, target, [=](const pos lhs, const ↩
            pos rhs, const double l, const double slope) mutable -> bool
        {
            estimator.train(lhs, l, true, horizon);
            estimator.train(rhs, 1 - l, false, horizon);

            max_estimator.train(lhs, l, true, horizon);
            max_estimator.train(rhs, 1 - l, false, horizon);
            min_estimator.train(lhs, l, true, horizon);
            min_estimator.train(rhs, 1 - l, false, horizon);

            horizon = max(horizon, slope);

            return true;
        });
    };

    inline bool max_visible(const pos p)
    {
        return !intersects(max_estimator.estimate(p), slopes[p.i][p.j]);
    };

    inline bool min_visible(const pos p)
    {
        return !intersects(min_estimator.estimate(p), slopes[p.i][p.j]);
    };
};

#endif /* defined(__TerrainTest__HybridFastVisibilityFinder__) */
```

Listing A.24: HybridBoundVisibilityFinder.hpp

```
//
//  HybridFastVisibilityFinder.h
//  TerrainTest
//
//  Created by Martin Vonheim Larsen on 23/03/15.
//  Copyright (c) 2015 Martin Vonheim Larsen. All rights reserved.
//

#ifndef __TerrainTest__HybridBoundVisibilityFinder__
#define __TerrainTest__HybridBoundVisibilityFinder__

#include "R2VisibilityFinder.hpp"
#include "R2MaxEstimator.hpp"
```

```
#include "R2MinEstimator.hpp"
#include "macros.hpp"
#include "types.hpp"

template<class Estimator>
class HybridBoundVisibilityFinder : public R2VisibilityFinder<Estimator> {
public:
    HybridBoundVisibilityFinder(size_t m, size_t n, double K) : ↩
        R2VisibilityFinder<Estimator>(m, n), K(K) {};

    bool_grid visibility(pos observer, double observer_height, double ↩
        target_height)
    {
        estimator.reset();

        const double base_height = VisibilityFinder::base_height(observer,↩
             observer_height);
        precalc_slopes(observer, base_height, target_height);
        eval_boundary_targets(observer, base_height);

        if (K > 0) {
            bool_grid bound = VisibilityFinder::boundary(visible());

            vector<double> angles;
            angles.reserve(100000);

            for (int i = 0; i < m; ++i) {
                if (i == observer.i) {
                    continue;
                }

                for (int j = 0; j < n; ++j) {
                    if (bound[i][j]) {
                        angles.push_back(atan2(i - observer.i, j - ↩
                            observer.j));
                    }
                }
            }

            const size_t r2_size = (size_t)floor(K*(2*m + 2*n - 4));
            const size_t n_angles = angles.size();

            if (n_angles < 1.5*r2_size) {
                for (double theta : angles) {
                    eval_target(observer, base_height, theta);
                }
            } else {
                sort(angles.begin(), angles.end());

                for (int k = 0; k < r2_size; ++k) {
                    const size_t i = (k*n_angles)/r2_size;
                    const double theta = angles[i];

                    eval_target(observer, base_height, theta);
                }
            }
        }

        return visible();
    };
protected:
    using R2VisibilityFinder<Estimator>::estimator;
    using VisibilityFinder::m;
    using VisibilityFinder::n;

    const double K;

    using R2VisibilityFinder<Estimator>::eval_boundary_targets;
    using R2VisibilityFinder<Estimator>::eval_target;
    using R2VisibilityFinder<Estimator>::precalc_slopes;
    using R2VisibilityFinder<Estimator>::visible;
};

#endif /* defined(__TerrainTest__HybridFastVisibilityFinder__) */
```