A simple Ladar simulator based on ray tracing

Trym Vegard Haavardsholm

Norwegian Defence Research Establishment (FFI)

01 July 2011

FFI-rapport 2009/01481

1020

P: ISBN 978-82-464-1949-7
E: ISBN 978-82-464-1950-3

Keywords

Ladar

Simulator

Ray tracing

Approved by

Halvor Ajer	Project Manager
Johnny Bardal	Director

English summary

In the study of ladar image processing, it is useful to have a method for producing synthetic data. This report presents a simple ladar simulator implemented in Matlab, which produces synthetic ladar images by simulating the imaging geometry using methods from ray tracing in computer graphics. The report presents background theory, simulation method and usage examples, and is meant to serve as a report for the development as well as a user guide.

Sammendrag

I studien av prosessering av ladarbilder er det nyttig med en metode for å lage syntetiske data. Denne rapporten presenterer en enkel ladarsimulator implementert i Matlab, som lager syntetiske ladarbilder ved å simulere avbildningsgeometrien med metoder fra ray tracing i datagrafikk. Rapporten presenterer bakgrunnsteori, simuleringsmetode og eksempler på bruk, og er ment som både en rapport for utviklingen og brukerveiledning.

Contents

1 Introduction

Ladar is an acronym for laser detection and ranging and is a term commonly used for imaging laser range finders, especially in military contexts. A ladar may typically perform a 2-dimensional (2D) scan of a scene to form a *range image*, where each pixel in the image corresponds to the range to the scene in the direction the laser beam was pointing at a specific scan step. Since these directions are known, the range image may be transformed to three images representing the 3D Cartesian coordinates for the point of reflection in each pixel. This gives us a 3D[1] point cloud of the shape of the scene.

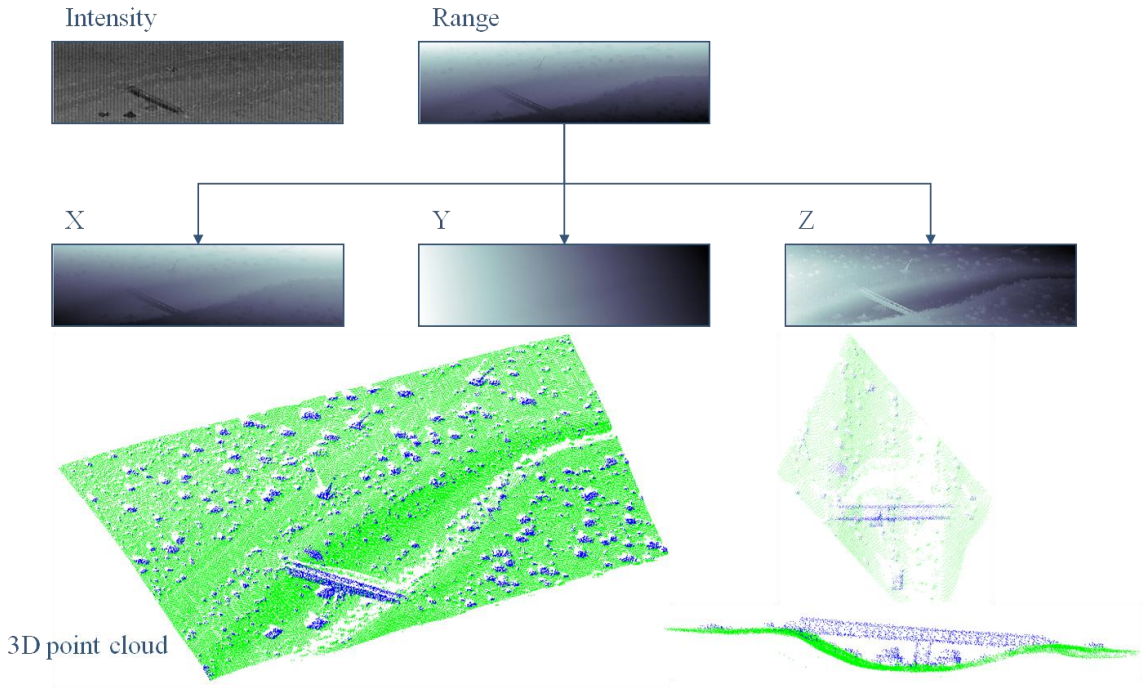

Figure 1.1 Ladar images. A ladar will typically produce a range image and an intensity image of the received energy from the reflected laser light. The range image may be transformed to a 3D point-cloud representation of the scene. The point clouds shown here has been processed so that the points on the ground surface are coloured green, while objects are coloured blue.

There are several applications for ladar images, such as terrain modelling, forestry surveying, and robotic vision. The present work on ladar-image analysis at FFI has focused on shape-based automatic target detection and recognition (ATD/R), and has shown that vehicles may be successfully extracted from ladar images based on information about their size [1], and that vehicles may be separated from similarly sized natural objects by using surface properties [2]. The next step is to classify and recognize the extracted vehicles. One method is to compare the extracted point clouds with a library of 3D models of known vehicles, and this may be performed

[1] Actually, ladar images are often referred to be 2.5D, since only one side of the scene is measured, giving us no information about the 3D shape of the back-sides of objects.

efficiently by constructing synthetic point clouds from the 3D models, and comparing the point clouds with each other.

This report presents a simple ladar simulator implemented in Matlab for creating synthetic point clouds from 3D models by simulating the imaging geometry. The simulator is based on ray tracing and performs well enough to construct images with high resolution from dense 3D models. It may therefore also be used to simulate ladar images for larger synthetic scenes, for example for testing different aspects of ladar processing where real data is not available. Both theory and implementation is covered in this report, and it serves as a documentation of the method as well as a user guide. Although the simulator only simulates the imaging geometry to first order, some thoughts on how to evolve it to include other effects such as the physics involved in the laser pulse generation, atmospheric propagation, and the signal processing are given.

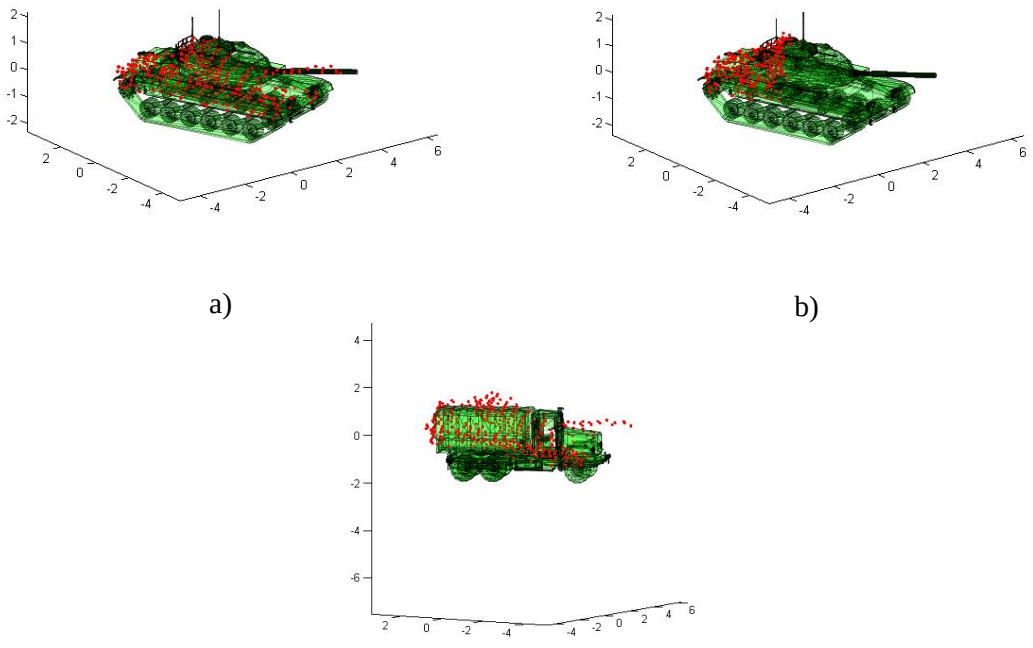

Figure 1.2 Example application of the ladar simulator: Vehicle recognition by comparing point clouds. The ladar simulator is used to generate synthetic ladar point clouds (not shown) from a 3D model library, and the ladar measurements (in red) are compared to these by first aligning the point clouds, and then measuring the difference between them. a) and b) shows two different point clouds aligned with the correct 3D model, and c) shows the result when the measurement from a) is compared with the wrong model.

2 Ray tracing in the ladar simulator

Ray tracing is a computer-graphics technique for generating images by tracing the path of light between each camera pixel and a virtual scene. Optical effects such as reflection, refraction, scattering, depth of field and motion blur can be simulated to achieve a very high degree of visual realism. This comes at a greater computational cost, which makes ray tracing less suited for real-time applications such as games[2]. The last few years has, however, seen an increasing activity on real-time ray tracing. Interactive ray-tracing has been demonstrated in games [3] [4], and real-time ray-tracing engine APIs [5] based on high-performance processing on GPUs are now commercially available.

The ladar simulator is based on the most basic functionality in ray tracing, where rays are cast according to how a ladar would scan a scene, and ray-scene intersections are found to compute the distance to the scene along the rays. This easily leads to a point-cloud representation of the scene that agrees to the simulated imaging geometry. More advanced functionality, like simulation of optical effects such as reflection or refraction or other physical effects such as noise and beam divergence, is not implemented, but is discussed in section 4. This chapter will discuss the 3D models and ray tracing methods used in the ladar simulator.

2.1 3D models

In computer graphics, 3D objects are usually modelled by defining their surface or boundary, rather than their volume. These surfaces are usually approximated as polygonal meshes and most commonly and without loss of generality triangle meshes, which are particularly simple to render. A scene may consist of thousands or millions of triangles, depending on the level of detail and the surface complexity of its objects[3].

When ignoring volume-rendering applications, a notable exception to the use of surface representation is in visualization of fluid-like phenomena such as fire, smoke, explosions and flowing grass or hair, which are not easily represented as polygons. Here the particle system technique is often used, where the phenomena is discretized as a set of particles and often visualized as sets of simple textured polygons or pixels. Such representations can also be used in ray tracing e.g. by intersecting small spherical particles.

[2] 3D computer games are usually based on rasterization and z-buffer techniques, where each 3D polygon is projected onto the image plane (using simplified camera models) and rasterized, and the depth in the z-buffer is used to ensure that pixels on polygons far away does not overwrite pixels on triangles that are closer. This is usually performed in graphics hardware.

[3] Alvy Ray Smith, one of the founding fathers of Pixar and a noted pioneer in computer graphics, stipulated back in the 1990s that reality was about 80 million triangles at 30 frames/s.

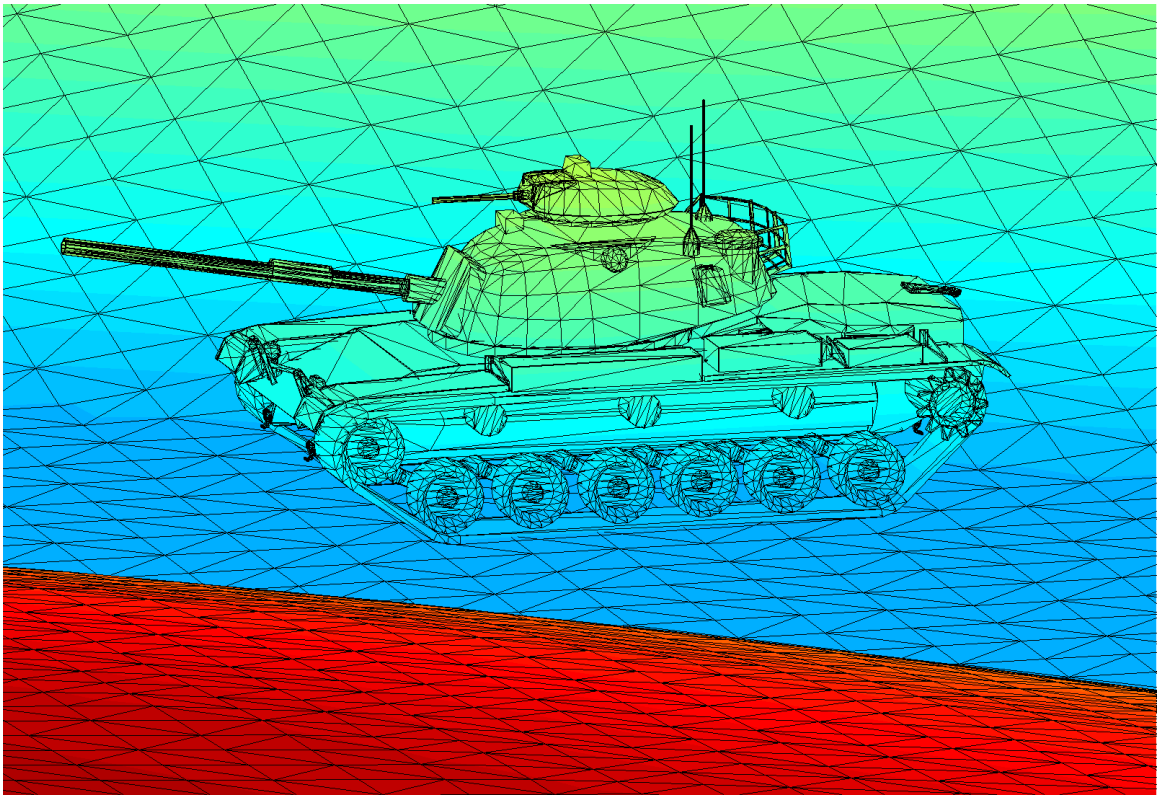

Figure 2.1 A M60 tank and terrain represented as a set of triangles.

Figure 2.2 Smoke represented as a system of particles, and rendered with volumetric shadows (sample from the Nvidia GPU Computing SDK).

Complex 3D model, likes those used in Computer-Aided Design (CAD) or games, are typically represented as a set of smaller parts associated with properties such as the colour of the surface material and transparency. Assembly of these parts is described as a set of transformations, and movable joints may be defined. A single triangle representation of a car wheel may for example be copied, translated and rotated to 4 different locations in a car model, and may then be defined to rotate freely on an axel.

In the ladar simulator it is assumed that the whole scene is assembled as a single set of triangles. This set is described with two matrices. All the triangle vertices are stored as rows in the *n*-by-3 matrix **V**:

$$\mathbf{V} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ & \vdots & \\ x_n & y_n & z_n \end{bmatrix}. \tag{2.1}$$

Each row in the *m*-by-3 matrix **F** then describes a triangle by pointing to three triangle vertices using row indices in **V**:

$$\mathbf{F} = \begin{bmatrix} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ & \vdots & \\ A_m & B_m & C_m \end{bmatrix}, \quad A_i, B_i, C_i \in \{1, 2, \ldots, n\}, \tag{2.2}$$

so that $[A \quad B \quad C] = [1 \quad 2 \quad 3]$ is a triangle with the vertices described in the first, second and third rows in the vertex matrix **V**. Using this representation, vertices that are shared between several triangles need only be stored once.

2.2 Ray casting

Ray casting is the process where the originating point and direction of rays in the ray-tracing problem is determined. When defining a camera, for example, one would typically describe each pixel as a ray and cast it according to a camera model. When a ray hits a surface, one would also typically cast new rays according to the surface geometry and surface properties like reflectivity, diffuseness, and transparency. For the ladar simulator, we are interested in casting rays similar to how a ladar would construct a ladar image.

Let a ray be represented as

$$\vec{R}(t) = \vec{O} + t\vec{D}, t \in [0, \infty), \tag{2.3}$$

where $\vec{O}$ is the originating point of the ray and $\vec{D}$ is the unit vector representing the direction of the ray and t is the distance along the ray.

A simple and popular camera model is the pinhole camera model, where the camera aperture is described as a point called the *principal point*. This model is in many situations a quite accurate model for cameras based on lenses as well, where it describes the properties of an ideal lens focusing light through a point. In a physical system, the image plane would lie perpendicular to the optical axis of the camera at a distance *f* behind the principal point, called the *focal length*.

The mapping from the 3D world onto the image plane is a perspective projection through the principal point, followed by a 180° rotation about the optical axis. To avoid having to deal with the rotation in the camera model, it is customary to place a virtual image plane at a distance *f* in front of the principal point, and project the world onto this plane instead. For an image plane with a fixed array of pixels, the field of view (FOV) is controlled by adjusting the focal length *f*.

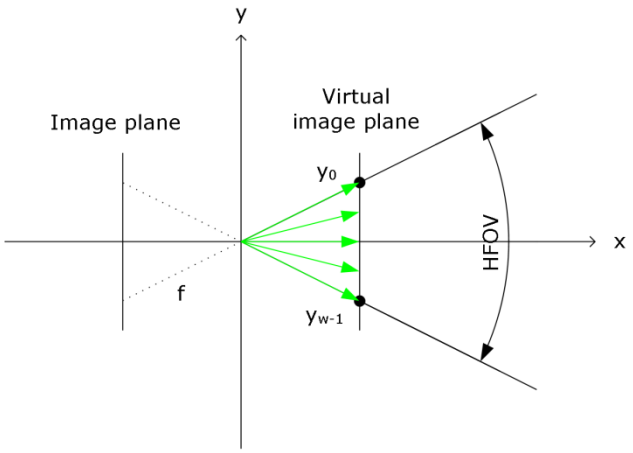

Figure 2.3 The geometry of the pinhole camera model seen from above along the (negative) z-axis, where the x-axis coincides with the optical axis. The rays (in green) are spread uniformly across the image plane.

Let the rays correspond to the view directions of an array of pixels $P_{u,v}$ lying in a regular grid on the virtual image plane, where *u* and *v* are the horizontal and vertical pixel indices, respectively. Then $\vec{O}$ in (2.3) is the principle point for all the rays, and the direction vectors $\vec{D}_{u,v}$ will point from $\vec{O}$ to the pixel $P_{u,v}$ in the virtual image plane. Let the principal point be the origin of a Cartesian coordinate system, and let the x-axis coincide with the optical axis, and the y- and z-axis coincide with the horizontal and vertical directions in the virtual image plane, respectively. Since it is often more intuitive to describe the camera model with an image resolution and a FOV, we will fix the focal length at $f = 1$, and adjust the size of the pixel array in the image plane instead. Given the width *w* and height *h* of the pixel array in number of pixels, and the horizontal and vertical FOV α_H and α_V, the uniform steps in the virtual image plane are

$$\begin{aligned} y_u &= \left(1 - u\frac{2}{w-1}\right)\tan\left(\frac{\alpha_H}{2}\right) \\ z_v &= \left(1 - v\frac{2}{h-1}\right)\tan\left(\frac{\alpha_V}{2}\right). \end{aligned} \tag{2.4}$$

The direction of a ray corresponding to pixel $P_{u,v}$ is then given by

$$\overrightarrow{OP}_{u,v} = [1, y_u, z_v], \tag{2.5}$$

where pixel $P_{0,0}$ is the top left pixel, and pixel $P_{w-1,h-1}$ is the bottom right pixel. By scaling these direction vectors to unit length, we get

$$\vec{D}_{u,v} = \frac{\overrightarrow{OP}_{u,v}}{\left\|\overrightarrow{OP}_{u,v}\right\|}. \tag{2.6}$$

A more appropriate model for the ladar may be to step uniformly in elevation and azimuth angles, mimicking a laser scanning line-by-line or column-by-column. As opposed to the pinhole camera model, this model allows FOVs $\geq 180^\circ$. Let $\vec{O}$ be the scanning laser or mirror in the origin of the coordinate system above, and let θ be the elevation angle and ϕ the azimuth angle. Given a width *w*, a height *h* and horizontal and vertical FOVs α_H and α_V as above, we get the uniform angle steps

$$\begin{aligned} \phi_u &= \left(\frac{1}{2} - \frac{u}{w-1}\right)\alpha_H \\ \theta_v &= \left(\frac{1}{2} - \frac{v}{h-1}\right)\alpha_V. \end{aligned} \tag{2.7}$$

The unit direction vector of a ray for pixel $P_{u,v}$ is then given by

$$\vec{D}_{\phi_u,\theta_v} = [\cos\theta_v \cos\phi_u, \cos\theta_v \sin\phi_u, \sin\theta_v]. \tag{2.8}$$

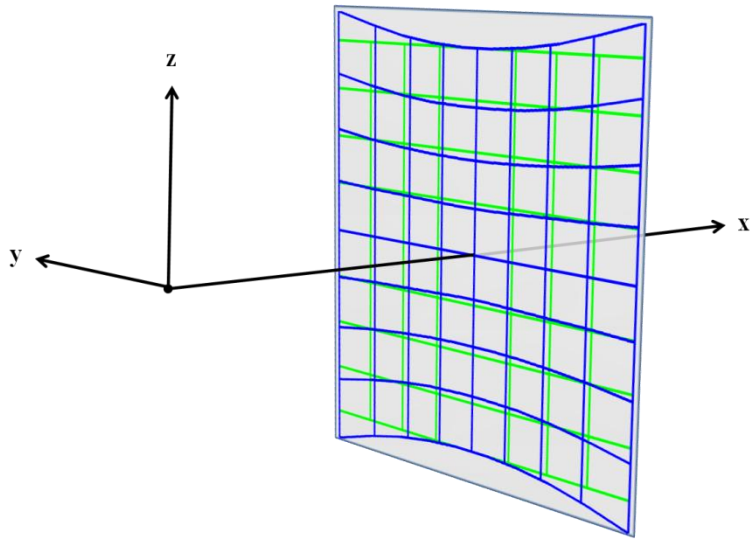

Figure 2.4 The difference in pixel patterns between the pinhole camera model, and the scanning camera model on a plane perpendicular to the optical axis. The green grid shows the pinhole camera model pattern, which is a uniform grid on the plane. The blue grid shows the scanning camera model pattern, which have vertical lines parallel to the pinhole pattern (although differently spaced), and horizontal parabola curves, since the horizontal scanning describes a cone.

2.3 Ray-triangle intersection

A crucial step in ray tracing is to determine if and where a ray intersects an object. There are several different intersection tests that can be performed on different primitives like polygons, spheres and boxes, and the interested reader is encouraged to read [6] for an overview. Since the ladar simulator represents its objects as a set of triangles, we will here discuss the intersection between rays and triangles.

A triangle ABC in $\mathbb{R}^3$ defines a plane Γ given by $\vec{N} \cdot \overrightarrow{AP} = 0$, where $\vec{N} = \overrightarrow{AB} \times \overrightarrow{AC}$ is a normal vector for the plane, and P is an arbitrary point in the plane.

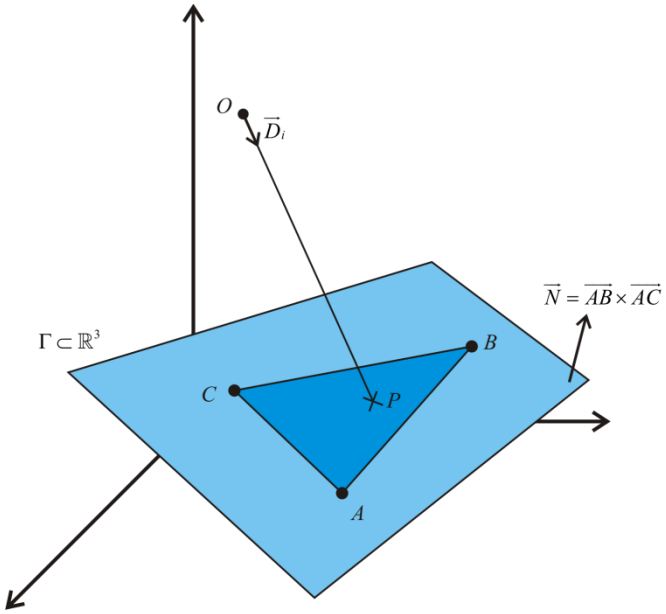

Figure 2.5 The geometry of the ray-triangle intersection problem.

If we let P be the point of intersection between a ray and the plane, it is given by (2.3) with

$$t = -\frac{\vec{N} \cdot (\vec{O} - \vec{A})}{\vec{N} \cdot \vec{D}}, \tag{2.9}$$

where t is the length of the line segment $\overrightarrow{OP}$. If the ray is parallel to the plane, then $\vec{N} \cdot \vec{D} = 0$ and P does not exist because the ray will not intersect the plane.

If the ray intersects the plane, we will have to test if the intersection lies within the triangle to determine if t is a valid range or not. Here it is useful to introduce *barycentric coordinates* [6]. Barycentric coordinates parameterize the space that can be formed by a weighted sum of reference points, and is therefore often used to parameterize triangles or the planes of triangles. Any point P in the plane of the triangle ABC can be expressed as $P = uA + vB + wC$, where $u + v + w = 1$. The triplet (u, v, w) corresponds to the barycentric coordinate for that point. A point is inside or on the triangle if and only if $0 \le u, v, w \le 1$, or alternatively if and only if $0 \le v \le 1$, $0 \le w \le 1$, and $v + w \le 1$.

Since $u = 1 - v - w$ and

$$P = (1 - v - w)A + vB + wC = A + v(B - A) + w(C - A), \tag{2.10}$$

we can compute the barycentric coordinates by solving the linear equation

$$\overrightarrow{AP} = v\overrightarrow{AB} + w\overrightarrow{AC}. \tag{2.11}$$

An alternative way to compute the barycentric coordinates is to use ratios of signed triangle areas. We have that

$$\begin{aligned} u &= Ar(PBC) / Ar(ABC) \\ v &= Ar(PCA) / Ar(ABC) \\ w &= Ar(PAB) / Ar(ABC) = 1 - u - v, \end{aligned} \tag{2.12}$$

where the function $Ar(ABC)$ denotes the signed area of triangle ABC, or any function proportional thereof. This means that the parallelogram area we get from the magnitude of the cross product of two triangle edges may be used, if we maintain the correct sign by taking the dot product of this cross product with the normal of ABC.

Since the barycentric coordinates remain invariant under projection, we can simplify the computation of the coordinates by projecting the triangle onto the *xy*, *xz* or the *yz* plane. By dropping the component with the largest absolute value in the triangle normal $\vec{N}$, we ensure that the projected area is the greatest, and thereby avoid degeneracies.

In 2D we may use the pseudo cross product (also called the perp-dot product) to compute the triangle areas. The pseudo cross product is defined as

$$\vec{u}^{\perp} \cdot \vec{v}, \tag{2.13}$$

where $\vec{u}^{\perp} = [-u_2, u_1]$ is the counterclockwise vector perpendicular to $\vec{u}$. The 2D pseudo cross product corresponds to the signed area of the parallelogram determined by $\vec{u}$ and $\vec{v}$, and is positive if $\vec{v}$ is counterclockwise from $\vec{u}$. By examining the cross-product vector $\vec{N} = \overrightarrow{AB} \times \overrightarrow{AC}$, we can notice that each component of $\vec{N}$ is a pseudo cross product, with the middle component being negated. We can therefore find the barycentric coordinates in a 2D coordinate plane using ratios of areas by computing one of

$$\begin{array}{lll} xy: & xz: & yz: \\ u = \dfrac{\overrightarrow{BC}_{xy}^{\perp} \cdot \overrightarrow{BP}_{xy}}{N_z} & u = \dfrac{\overrightarrow{BC}_{xz}^{\perp} \cdot \overrightarrow{BP}_{xz}}{-N_y} & u = \dfrac{\overrightarrow{BC}_{yz}^{\perp} \cdot \overrightarrow{BP}_{yz}}{N_x} \\ v = \dfrac{\overrightarrow{CA}_{xy}^{\perp} \cdot \overrightarrow{CP}_{xy}}{N_z} & v = \dfrac{\overrightarrow{CA}_{xz}^{\perp} \cdot \overrightarrow{CP}_{xz}}{-N_y} & v = \dfrac{\overrightarrow{CA}_{yz}^{\perp} \cdot \overrightarrow{CP}_{yz}}{N_x}. \end{array} \tag{2.14}$$

2.4 Accelerating data structures

For a decent sized 3D model, it is infeasible to perform pairwise ray-triangle intersection tests for all rays on all triangles within an acceptable amount of time. It is therefore common to represent the model in an accelerating data structure, so that fewer tests are performed for each ray. Popular data structures include uniform grids, bounding volume hierarchies (BVHs) and *k*-d trees. All these data structures divide the space into regions, and allow us to quickly traverse parts of the

space where rays will not intersect any object, before (and after) reaching regions where a ray may intersect a constrained amount of triangles. We often need to invest some significant time in building these data structures, however, before we can reap benefits from the acceleration in the ray-triangle intersection tests.

The ladar simulator implements a simple *k*-d tree, and we will here only discuss the specifics for this implementation. There is a lot of literature on this topic, however, where [6] [7] [8] are some examples.

Building a *k*-d tree means recursively splitting the space in two until either the maximum depth is reached, or there is no longer any vertices within a region. The splitting can be performed arbitrarily along one of the axis at each step. The choice of axis and position of the splitting plane can be determined using a cost function. A popular choice of cost function is the Surface Area Heuristic (SAH):

$$C_{SAH} = C_{Traversal} + A \cdot N_{Triangles} \cdot C_{Intersect}, \tag{2.15}$$

where $C_{Traversal}$ is the estimated cost of traversing the tree, A is the surface area of the region bounding box, $N_{Triangles}$ is the number of triangles within the region and $C_{Intersect}$ is the estimated cost of performing an intersection test. The cost of a split at a specific position can be computed by summing the cost for the two resulting regions.

The current implementation in the ladar simulator uses the SAH above in a simplified manner. First, the choice of splitting axis is chosen as *x*, then *y*, then *z*, then *x* again in a cyclic manner. Then the SAH is computed for a split at the first vertex, the median vertex and the last vertex in the region along the axis. It is assumed that the number of triangles on each side of the median vertex position is approximately the same. Then the position with lowest cost is chosen.

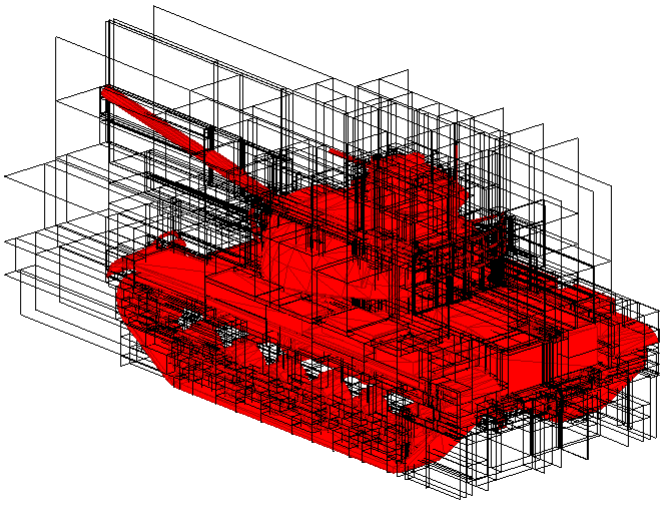

Figure 2.6 Visualization of a k-d tree built for the M60 model. Each plane corresponds to a split plane in the model. The maximum depth in this tree has been set low to make the visualization clearer.

When intersecting a ray with a *k*-d tree, the ray is intersected with the node's splitting plane, and a distance t to the intersection is computed. If t is within the interval $0 \le t < t_{\max}$, the ray straddles the split plane, and both children of the tree are recursively descended. By first descending the side of the ray origin, overlapped nodes are guaranteed to be traversed in order near to far. If t is not within the interval above, only the side containing the ray origin is recursively visited. When reaching a leaf node, the ray-triangle intersections are performed for the triangles within that region.

3 The ladar simulator in practice

This section presents the Matlab functions that implement the ladar simulator and gives some examples on how to use them under different circumstances.

3.1 The ladar-simulator implementation

The ladar simulator functionality is implemented across several Matlab functions using the methods described above in chapter 2. To use the simulator, one would typically write a Matlab script with the following stages:

1. Construct a model object using `buildModel`.
2. Build a *k*-d tree accelerating data structure for the model using `buildKDTree`.
3. Cast rays using `castRaysPinhole` or `castRaysScan`.
4. Intersect rays with *k*-d tree using `rayKDTreeIntersect`.

The Matlab code for the entire implementation is listed in Appendix A, while a summery of the simulator functions is listed below in Table 3.1. Several examples on how to use the simulator is given in section 3.4, with the complete code listed in Appendix B.

`model = buildModel(FV)` Builds a model object from the vertices and faces in `FV`. The model object computes and stores additional information about the 3D model, such as the normal vectors for each triangle and the bounding box.

`root = buildKDTree(model)`
`root = buildKDTree(model, maxD, maxN)`
 	
Constructs the *k*-d tree accelerating data structure for the model by performing recursive calls on `splitKDNode`. The parameters `maxD` and `maxN` specifies the maximum depth and the maximum number of triangles in a leaf node respectively. If the maximum depth is reached, the number of triangles in that leaf node may be exceeded.

`node = splitKDNode(node, d, orient, model, maxD, maxN)` Splits a k-d tree node using the split plane found by `findKDSplitPlane`. `d` is the current depth, and `orient` specifies the current orientation of the split plane: • X-axis: `orient == 1`. • Y-axis: `orient == 2`. • Z-axis: `orient == 3`.

`s = findKDSplitPlane(node, orient, model)`

Determines the position of a *k*-d tree split plane along the axis specified by `orient` according to the method described in section 2.4. Called by `splitKDNode`.

`rays = castRaysPinhole(pos, orient, w, h, hfov, vfov)`

Casts rays according to the pinhole camera model described in section 2.2. The principal point is given by the 1-by-3 vector `pos = [x, y, z]`. The camera orientation is given by the 1-by-3 vector `orient = [roll, pitch, yaw]` given in radians, so that the camera is first yawed about the z-axis, then pitched about the (new) y-axis and then rolled about the x- (or principle-) axis. `w` and `h` is the image width and height in the number of pixels, and `hfov` and `vfov` is the horizontal and vertical field-of-view respectively. The resulting structure array `rays` has the two fields `rays.O` for the ray origins and `rays.D` for the ray direction vectors.

`rays = castRaysScan(pos, orient, w, h, hfov, vfov)`

Casts rays according to the scanning camera model described in section 2.2. The parameters are identical to the ones described above for `castRaysPinhole`.

`[t, fID] = rayKDTreeIntersect(root, rays, model)`

Intersects the rays given by the structure array `rays` with the 3D model given in `model` using the *k*-d tree given by the root node `root`. The output argument `t` returns the resulting ranges for each ray. If a ray did not intersect the model, the corresponding range will be `inf`. The output argument `fID` returns the ID of the triangle where the nearest intersection was found for each ray. The `fID` will be 0 for rays that did not intersect the model. If only the output argument `t` is used in the call to `rayKDTreeIntersect`, a slightly faster intersection implementation that only returns ranges is used.

`[t, fID] = intersectKDNode(node, O, D, tMax, model)`

Intersects the *n* rays given by the origins in the *n*-by-3 matrix `O` and the directions in the *n*-by-3 matrix `D` with the 3D model in `model` using the *k*-d tree node `node` and its children. `tMax` is the maximum range that will be investigated, and can be set to `inf`. This function is called recursively by `rayKDTreeIntersect` when both the ranges `t` and the triangle IDs `fId` are specified as output arguments.

`t = intersectKDNodeRange(node, O, D, tMax, model)` Identical to `intersectKDNode` above, except that it will not return the triangle IDs, and is therefore slightly faster. Recursively called by `rayKDTreeIntersect` when only the range vector `t` is specified as output argument.
`img = raysToImage(t, h, w)` Constructs an image of the values in the *n*-by-1 vector `t`, where each element in this vector is assumed to correspond to rays in the list of rays produced by `castRaysPinhole` or `castRaysScan` with height `h` and width `w`.

Table 3.1 A summary of the ladar-simulator functions. The complete code is listed in Appendix A.

3.2 Data sets

Two data sets are used in this report and have been included with code on the source-code CD. The M60 data set is a 3D model of a M60 main battle tank, represented with about 8000 triangles. The model faces and vertices are stored in the struct "FV" in the file "M60A3.mat". The model is shown in Figure 3.1 below.

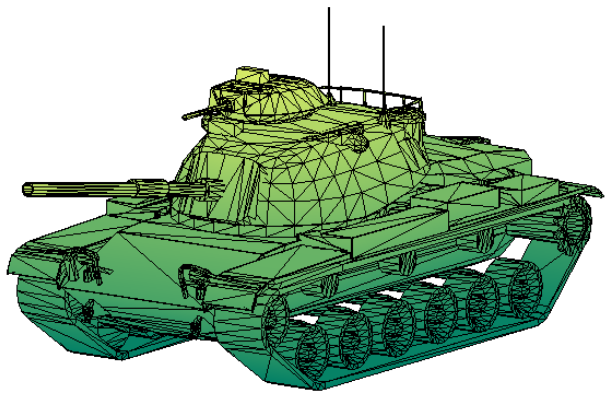

Figure 3.1 The M60 data set.

The Eidsvoll data set is a detailed terrain model[4] of an area near Eidsvoll, Norway, where three M60s have been placed in a small valley. The model covers an area of about 86 x 86 m with a terrain resolution of about 0.86 m, and contains about 104000 triangles. The faces and vertices are stored in the struct "FV" in the file "Eidsvoll_with_3_M60A3.mat". A portion of the model is shown in Figure 3.2 below.

[4] Thanks to Martin Ferstad Aasen for sharing the detailed terrain data!

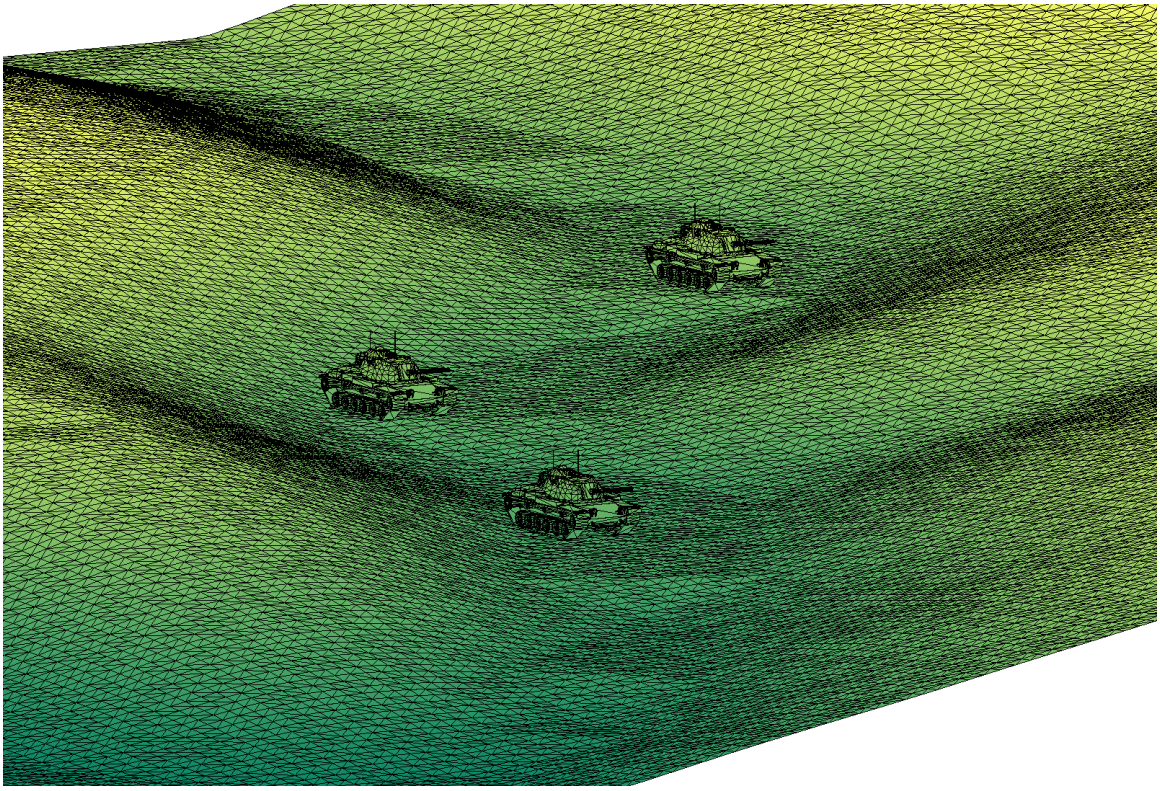

Figure 3.2 The Eidsvoll data set.

3.3 Performance

There is no doubt that the choice to implement this simulator using Matlab has set significant restrictions on performance, especially due to the lack of pointers and the difficulty in exploiting Matlab fully by expressing all computations in an optimal matrix-oriented way. The use of recursive functions is probably not optimal for performance either, but this may be mediated by serializing those functions to loops. The ladar simulator does however run with an acceptable performance, and will probably construct images faster than ladar scans.

To investigate how the ladar-simulator performance scales with the size of the 3D model and the number of pixels, it has been tested on resampled versions of the Eidsvoll data set with different image resolutions. The Eidsvoll data set was resampled gradually using the Matlab function `reducepatch`, which reduces the number of triangles while attempting to preserve the overall shape of the model. The camera was positioned above the 3D terrain, and the FOV adjusted so that the ladar image covers almost the entire scene (see Figure 3.3). The complete code for the performance test is listed in Appendix B.4

Figure 3.4 a) shows the performance for the model building and k-d tree building steps. Both scale approximately linearly with the number of triangles, and the building of the *k*-d tree is by far the most time consuming. The *k*-d tree builder processes the 3D model at more than 10 kTriangles/s, while the model builder is more than 10 times faster.

The ray casting performance for the two camera models is shown in Figure 3.4 b). These scale approximately linearly with the number of rays, with the pinhole camera model being significantly slower than the scanning camera model. The pinhole camera model produces about 1 Mrays/s, while the scanning camera model produces about ten times more.

The performance of the intersection method is shown in Figure 3.4 c). It scales approximately linearly with both the number of rays and the number of triangles, when these numbers are high. As a rule of thumb, one could expect the simulator to process at least 2 Mrays/min for a decent-sized 3D model.

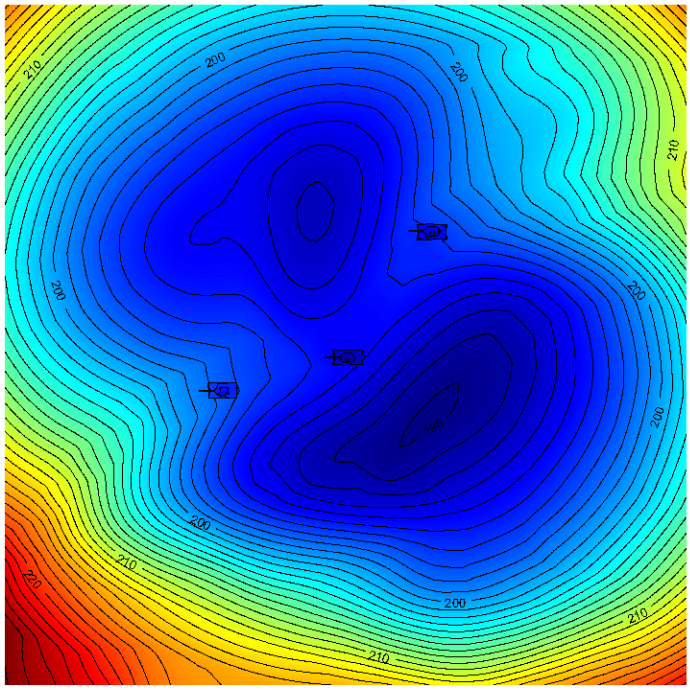

Figure 3.3 Range image illustrating the scene and FOV used in the performance test.

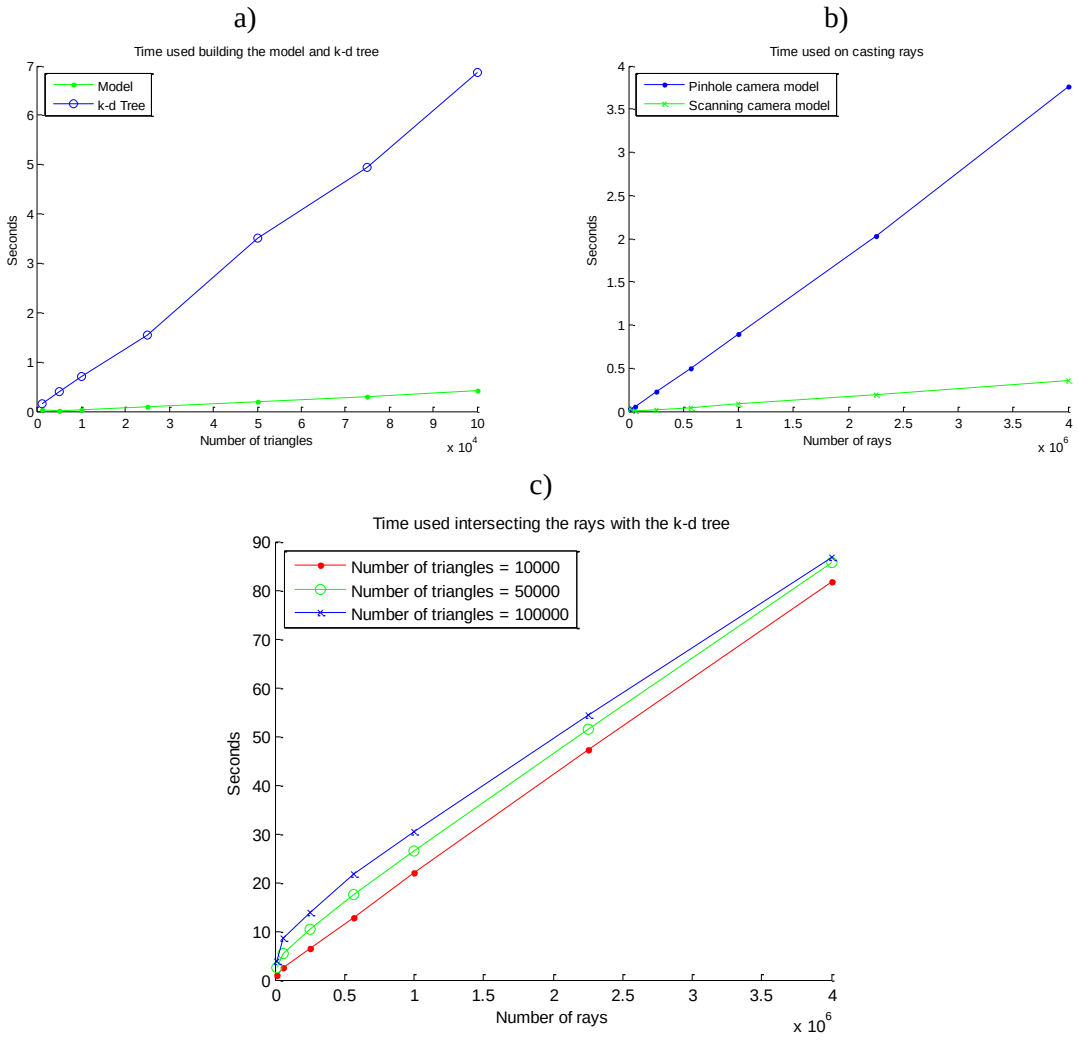

Figure 3.4 *Results from the performance test.*
Plot a) shows the time used building the model and the k-d tree for different sizes of the 3D model. Plot b) shows the time used on casting rays for the two camera models for different number of rays. Plot c) shows the performance of the intersection method for different number of rays and different sizes of the 3D model.

3.4 Examples

3.4.1 A 360° scan

This example simulates a 360° scan of the scene, from a position close to the middle tank. The complete code is listed in Appendix B.3.

First, the dataset is loaded, and the model and *k*-d tree is built:

```
% Load dataset.
load('Eidsvoll_with_3_M60A3.mat');

% Build model and k-d tree.
model = buildModel(FV);
root = buildKDTree(model);
```

Then 3600 x 200 rays are cast from a position near the middle tank with a horizontal FOV of 360° and a vertical FOV of 20°:

```
% RayCast a 360 degree scan.
w = 3600;
h = 200;

rays = castRaysScan([-20 0 180], [0 0 0], w, h, 2*pi, pi/9);
```

The rays are intersected with the *k*-d tree, and ranges are computed:

```
% Intersect the rays with the k-d tree.
t = rayKDTreeIntersect(root, rays, model);
```

The range image is constructed using `raysToImg`, and the point cloud is produced by following each ray the computed distance to intersection:

```
% Construct the range image.
rImg = raysToImg(t,h,w);

% Construct the point cloud.
x = rays.O(:,1) + t.*rays.D(:,1);
y = rays.O(:,2) + t.*rays.D(:,2);
z = rays.O(:,3) + t.*rays.D(:,3);
```

Finally the result is visualized (see Figure 3.5). The point cloud is laid on top of the 3D model, and the range image is used as a texture map on the surface determined by the rays at a 10-meter distance from the origin.

```
% Plot the model surface.
figure;
hold on;
p1 = patch(FV);
heights = FV.vertices(:,3);
heights = heights-min(heights);
heights = heights/max(heights);
cmap = summer(128);
cdata = cmap(round(1+(heights*127)),:);
set(p1,'FaceColor','interp',...
'FaceVertexCData',cdata,...
'EdgeColor','flat');

% Plot the point cloud
plot3(x,y,z,'k.', 'MarkerSize',1);

% Use the range image as texture on scan surface 10 meter from O.
surf = rays.O + 10*rays.D;
x = raysToImg(surf(:,1),h,w);
y = raysToImg(surf(:,2),h,w);
z = raysToImg(surf(:,3),h,w);
warp(x,y,z,rImg,jet(64));
set(gca, 'YDir','normal');
caxis([0 64])
colorbar;
axis equal;
```

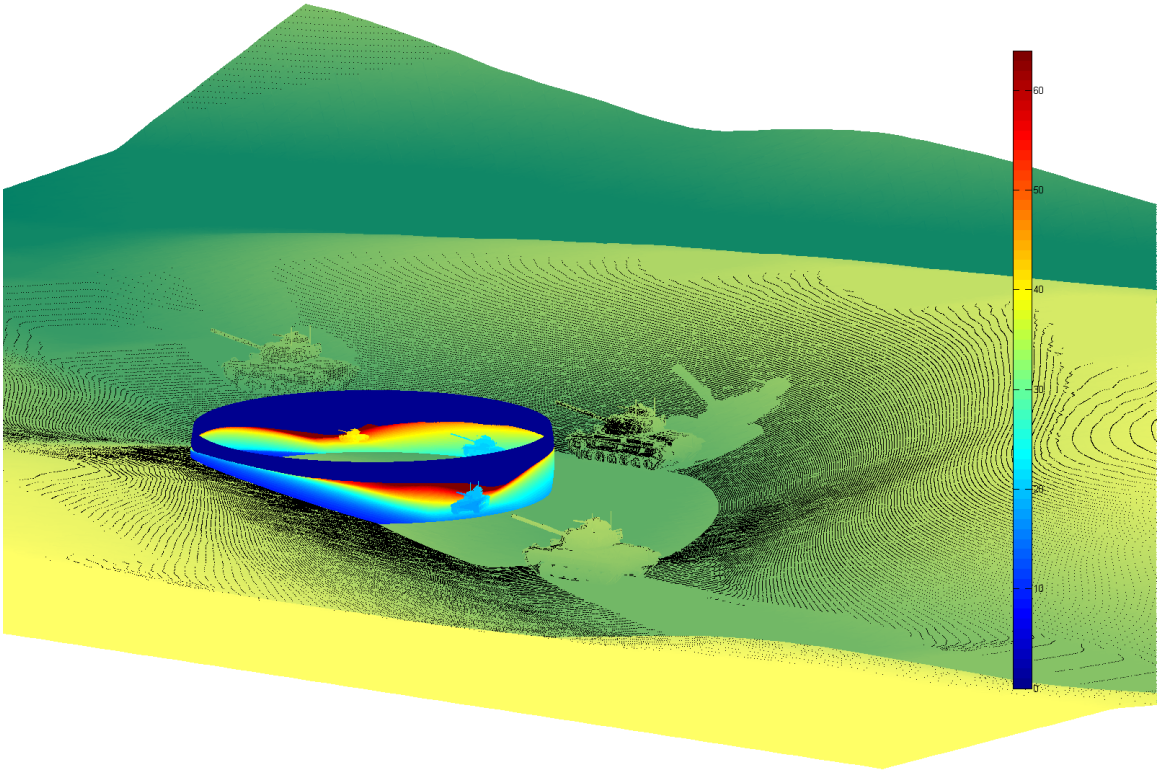

Figure 3.5 The range image and point cloud resulting from a 360° scan.

3.4.2 Using face IDs: Occlusion example

This example uses the face IDs from the ray tracing to remove points that hit a plane, while keeping the points that hit the tank behind it. The example thereby illustrates how to simulate occlusion. The complete code is listed in Appendix B.2.

First, the M-60 tank dataset is loaded, and an occlusion plane is defined and added to the triangle set:

```
% Load dataset.
load('M60A3.mat');

% Construct a plane.
planeV = [ ...
    0 0 0; ...
    0 1 0; ...
    0 1 1; ...
    0 0 1];
planeF = [1 3 4; 1 2 3];

% Translate and scale the plane.
planeP = [6 -5 0];
```

```
planeS = [1 4 4];
planeV = repmat(planeP,size(planeV,1),1)+ planeV*diag(planeS);

% Add the plane to occlude the tank and remember the plane face IDs.
planeIDs = [1 2] + size(FV.faces,1);
FV.faces = [FV.faces; planeF + size(FV.vertices,1)];
FV.vertices = [FV.vertices; planeV];
```

The face IDs for the occlusion plane is stored. The model and *k*-d tree are built, and rays are cast using the pinhole camera model:

```
% Build model and k-d tree.
model = buildModel(FV);
root = buildKDTree(model);

% RayCast using a pinhole camera model.
w = 500;
h = 200;

rays = castRaysPinhole([100 0 2], [0 0 -pi], w, h, 0.11, 0.05);
```

Then the rays are intersected with the k-d tree, and the face IDs for each intersection is collected as well as the ranges:

```
% Intersect the rays with the k-d tree.
[r,fid] = rayKDTreeIntersect(root, rays, model);
```

To get the face IDs, `rayKDTreeIntersect` uses another version of the intersection implementation, which may be slightly slower than the version computing the ranges only. Choosing which version to use is handled automatically in `rayKDTreeIntersect` by checking the number of output arguments in the function call.

After the intersection computations are completed, the range image is plotted:

```
% Compute the range image.
rImg = raysToImg(r,h,w);

% Plot the range image.
figure;
imagesc(rImg,[min(r(:))-2 max(rImg(:))]);
colorbar;
colormap(jet(256));
axis image
axis off
```

Figure 3.6 A range image showing the occlusion in front of the tank.

The result is shown in Figure 3.6. Here, the results from the intersections with both the occlusion plane and the tank are shown. If we want to show only the rays that hit the tank, we can use the face IDs (see Figure 3.7):

```
% Construct the point cloud for those rays that hit the tank.
tankOnly = ~ismember(fid, planeIDs);
x = rays.O(tankOnly,1) + r(tankOnly).*rays.D(tankOnly,1);
y = rays.O(tankOnly,2) + r(tankOnly).*rays.D(tankOnly,2);
z = rays.O(tankOnly,3) + r(tankOnly).*rays.D(tankOnly,3);

% Plot the model surface.
figure;
hold on;
p1 = patch(FV);
heights = FV.vertices(:,3);
heights = heights-min(heights);
```

```
heights = heights/max(heights);
cmap = summer(128);
cdata = cmap(round(1+(heights*127)),:);
set(p1,'FaceColor','interp',...
'FaceVertexCData',cdata,...
'EdgeColor','k');

% Plot the point cloud
plot3(x,y,z,'b.', 'MarkerSize',5);
axis equal;
axis off;
```

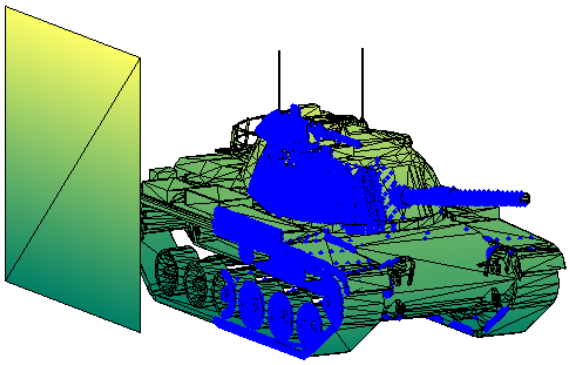

Figure 3.7 The point cloud (shown as blue dots) after removing points on the occlusion.

3.4.3 Using custom scan patterns: Push broom scan example

This example shows how to implement custom scan patterns by simulating a push broom scanning ladar on a platform with sinusoidal roll-movements. The complete code is listed in Appendix B.1.

First, the dataset is loaded, and the model and *k*-d tree are built:

```
% Load dataset.
load('Eidsvoll_with_3_M60A3.mat');

% Build model and k-d tree.
model = buildModel(FV);
root = buildKDTree(model);
```

The push broom scanning is simulated by moving the camera, and casting lines of rays at regular intervals using a 1D pinhole camera model. Each ray casting result is aggregated to a single large ray structure:

```
% RayCast a line scan.
w = 1;
h = 500;
roll = 0;
yaw = pi/4;
t = linspace(0,1,h);
pStart = [-50 -50 370];
pEnd = [50 50 370];
pos = repmat(pStart,h,1) + t'*(pEnd-pStart);
pitch = -pi/2 + (pi/180)*sin(t*4*pi); % Makes the platform "roll"

rays.O = zeros(h^2, 3);
rays.D = zeros(h^2, 3);
for i=1:h
    tmp = castRaysPinhole(...
        pos(i,:), ...
        [roll pitch(i) yaw], ...
        w, h, (pi/9)/h, pi/9);

    rays.O((i-1)*h+1:i*h,:) = tmp.O;
    rays.D((i-1)*h+1:i*h,:) = tmp.D;
end
```

The rays are intersected with the k-d tree, and the results are visualized (see Figure 3.8 and Figure 3.9):

```
% Intersect the rays with the k-d tree.
r = rayKDTreeIntersect(root, rays, model);

% Construct the range image.
rImg = raysToImg(r,h,h);

% Construct the point cloud.
px = rays.O(:,1) + r.*rays.D(:,1);
py = rays.O(:,2) + r.*rays.D(:,2);
pz = rays.O(:,3) + r.*rays.D(:,3);

% Plot the model surface.
figure;
hold on;
p1 = patch(FV);
heights = FV.vertices(:,3);
heights = heights-min(heights);
heights = heights/max(heights);
cmap = summer(128);
cdata = cmap(round(1+(heights*127)),:);
set(p1,'FaceColor','interp',...
'FaceVertexCData',cdata,...
'EdgeColor','flat');

% Plot the point cloud
plot3(px,py,pz,'k.', 'MarkerSize',1);
axis equal; axis off;

% Plot the range image.
figure;
imagesc(rImg);
colorbar;
colormap(jet(256));
axis image; axis off;
```

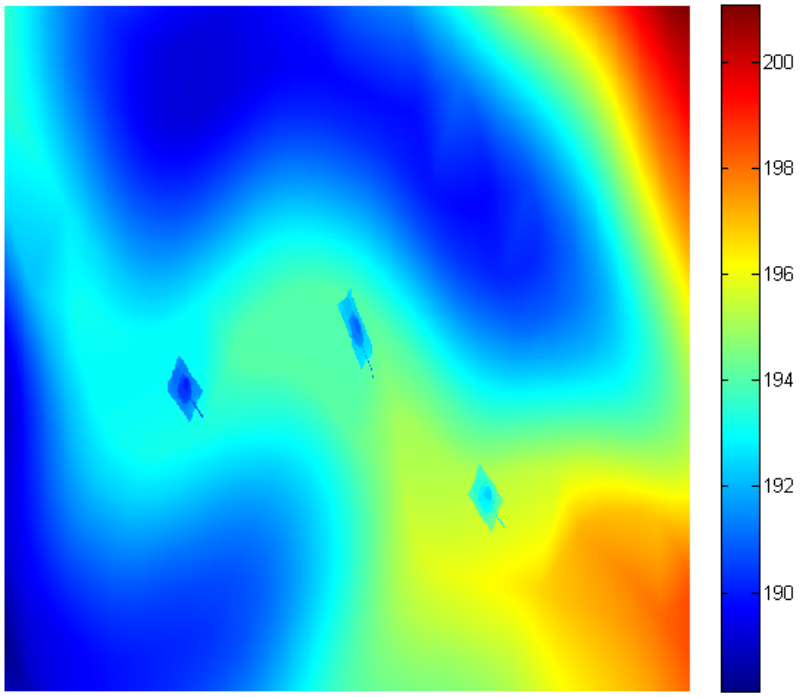

Figure 3.8 *In this example, the scan has been performed by aggregating 1D scans while moving the ladar perpendicular to the scanning from right to left in the image above, and rolling slightly. The range image shows significant distortions because of the roll movements during scanning.*

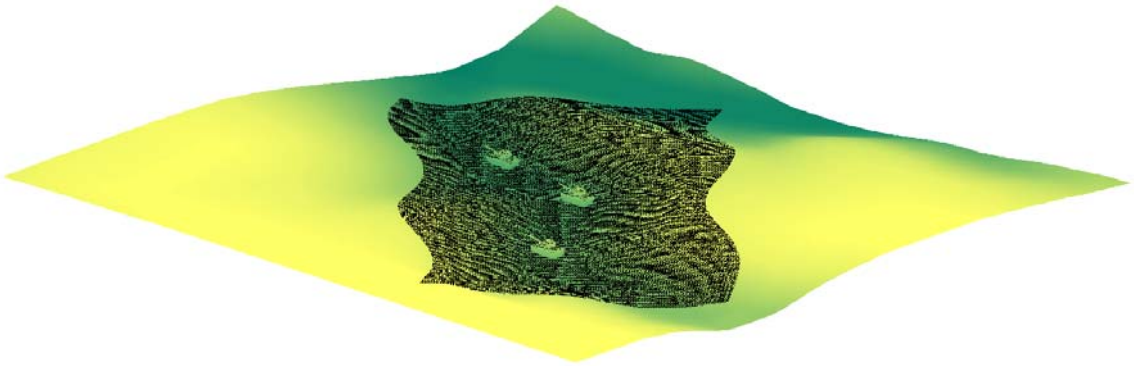

Figure 3.9 *The point cloud illustrates the footprint of the scan.*

4 Ideas for further development

The ladar simulator presented here only simulates the geometry of ladar imaging, and does not include the physics involved or the signal processing. Such processes may however also be incorporated, and some ideas will be presented in this section.

Effects in the atmosphere, such as scattering and attenuation, may be simulated using data from other simulation software such as Modtran [9]. Surface reflectance properties for the different objects could be included in the 3D model. Beam divergence could also be simulated by casting several rays for each laser beam with slightly different directions. By incorporating these effects, one could also simulate the received signal in each ladar pixel, and thereby the signal processing as well.

Implementing the ideas above would probably make a simulator based on matlab perform very slowly. There are however many ray-tracing libraries that could be used in C++, for example. This includes Nvidia Optix [5], which is a free high-level ray-tracing engine for C++ based on processing on Graphics Processing Units (GPUs).

Finally, it could be possible to include the ladar simulator in simulations and games like those performed in FFIs BattleLab. With access to an updated high-resolution 3D representation of the scene and objects, as well as camera position and orientation, it is feasible to perform such ladar simulations in real-time. If the scene is static (and not too big), the matlab implementation presented here could probably be used.

References

[1] Hans Christian Palm, Trym Haavardsholm and Halvor Ajer,
"Detection of military objects in LADAR images", FFI Rapport 2007/02472, 2008.

[2] Hans Christian Palm, "Detection of clutter in LADAR imagery",
FFI Notat 2011/01215, 2011.

[3] Daniel Pohl, "Experimental Cloud-based Ray Tracing Using Intel© MIC Architecture for Highly Parallel Visual Processing",
http://software.intel.com/en-us/articles/cloud-based-ray-tracing/,
Intel© Software Network, 2011.

[4] Daniel Pohl, "Quake Wars Gets Ray Traced",
http://software.intel.com/en-us/articles/quake-wars-gets-ray-traced/,
Intel© Software Network, 2009.

[5] Nvida OptiX, http://developer.nvidia.com/optix.

[6] Christer Ericson, *Real-Time Collision Detection*, Morgan Kaufmann, 2005.

[7] Vlastimil Havran, "Heuristic Ray Shooting Algorithms", Dissertation, Faculty of Electrical Engineering, Czech Technical University, Prague, 2000.

[8] Ingo Wald, "Realtime Ray Tracing and Interactive Global Illumination", Dissertation, Saarland University, Saarbrücken, 2004.

[9] A.Berk, G.P.Anderson, P.K.Acharaya, L.S.Bernstein, E.P.Shettle, S.M.Adler-Golden, J.Lee, and L.Muratov, "MODTRAN™-5 Version 2 Revision 11 USER'S MANUAL", 2005.

Appendix A The ladar simulator code

This appendix presents the Matlab code for all the functions that make up the ladar simulator. See chapter 3 for an explanation of how to use the code.

A.1 buildKDTree.m

```
function root = buildKDTree(model, maxD, maxN)
% BUILDKDTREE Build a k-d tree representation of a 3D model.
%    ROOT = BUILDKDTREE(MODEL) returns the root node of the k-d tree
%    representation. MODEL must be on the same form as that returned
%    by BUILDMODEL.
%
%    ROOT = BUILDKDTREE(MODEL, MAXD, MAXN) sets the maximum depth of
%    the tree to MAXD, and the maximum number of triangles in a leaf
%    node to MAXN.
%
%    See also BUILDMODEL.

if nargin == 1
    maxD = 25;
    maxN = 30;
elseif nargin ~= 3
    error('Wrong number of arguments!');
end

% Construct root.
root.faceNums = 1:size(model.faces,1);
root.bb = [min(model.vertices); max(model.vertices)];
root.dist = model.bb(2,:) - model.bb(1,:);

% Perform recursive splits.
root = splitKDNode(root, 1, 1, model, maxD, maxN);
```

A.2 buildModel.m

```
function model = buildModel(model)
% BUILDMODEL builds a model structure.
%    MODEL = BUILDMODEL(FV) builds a model structure from a
%    structure FV, which contains the fields 'vertices' and 'faces'.

if nargin ~= 1
    error('Wrong number of arguments!');
end

% Compute bounding box and lenghts of bounding box edges.
model.bb = [min(model.vertices); max(model.vertices)];
model.dist = model.bb(2,:) - model.bb(1,:);

% For simpler code.
v = model.vertices;
f = model.faces;

% Compute the normal vector for each triangle.
model.N = cross(v(f(:,2),:)-v(f(:,1),:),v(f(:,3),:)-v(f(:,1),:));

% Find the coordinate plane where the triangle has greatest area.
[~,dimInd] = sort(abs(model.N),2);
model.dimInd = sort(dimInd(:,[1 2]),2);

% Store the vertices in this plane, as well as the triangle area.
largestInd = dimInd(:,3);
n = size(f,1);
model.vx = zeros(n,3);
model.vy = zeros(n,3);
model.ood = zeros(n,1);
for i=1:n
    model.vx(i,:) = v(f(i,:), model.dimInd(i,1));
    model.vy(i,:) = v(f(i,:), model.dimInd(i,2));

    model.ood(i) = 1/model.N(i,largestInd(i));
    if largestInd(i) == 2
         model.ood(i) = - model.ood(i);
    end
end
```

A.3 castRaysPinhole.m

```
function rays = castRaysPinhole(pos,orient,w,h,hfov,vfov)
% CASTRAYSPINHOLE Cast rays according to a pinhole camera model.
%    RAYS = CASTRAYSPINHOLE(POS,ORIENT,W,H,HFOV,VFOV) cast rays
%    according to a pinhole camera model, where the principal point
%    is given by the 1-by-3 vector POS, W is the image width in number
%    of pixels, H is the image height in number of pixels, HFOV is the
%    horizontal field-of-view (FOV) in radians and VFOV is the vertical
%    FOV in radians.
%
%    The 1-by-3 vector ORIENT describes the roll-pitch-yaw orientation
%    of the camera. With ORIENT = zeros(1,3), the optical axis is along
%    the x-axis, and the z-axis points up.
%
%    RAYS is a structure with the fields 'O' for the ray origins, and
%    'D' for the ray directions.


% Copy origin to O matrix.
rays.O = repmat(pos, w*h, 1);

% Compute directions in camera frame.
x0 = tan(0.5*hfov);
y0 = tan(0.5*vfov);
[x,y] = meshgrid(linspace(x0, -x0, w), ...
                 linspace(y0, -y0, h));
D = [ones(w*h, 1), x(:), y(:)];

% Normalize to unit.
for i=1:size(D,1)
    D(i,:) = D(i,:) ./ norm(D(i,:));
end

% Rotate camera in world.
roll = orient(1);
pitch = orient(2);
yaw = orient(3);

Rx = [1 0 0; 0 cos(roll) -sin(roll); 0 sin(roll) cos(roll)];
Ry = [cos(pitch) 0 sin(pitch); 0 1 0; -sin(pitch) 0 cos(pitch)];
Rz = [cos(yaw) -sin(yaw) 0; sin(yaw) cos(yaw) 0; 0 0 1];

R = Rx*Ry*Rz;
rays.D = D*R;
```

A.4 castRaysScan.m

```
function rays = castRaysScan(pos,orient,w,h,hfov,vfov)
% CASTRAYSSCAN Cast rays according to a scanning camera model.
%    RAYS = CASTRAYSSCAN(POS,ORIENT,W,H,HFOV,VFOV) cast rays
%    according to a camera model for vertical and horizontal scans,
%    where the origin for the two scan axes is given by the 1-by-3
%    vector POS, W is the image width in number of pixels, H is the
%    image height in number of pixels, HFOV is the horizontal
%    field-of-view (FOV) in radians and VFOV is the vertical FOV in
%    radians.
%
%    The 1-by-3 vector ORIENT describes the roll-pitch-yaw orientation
%    of the camera. With ORIENT = zeros(1,3), straight ahead is along
%    the x-axis, and the z-axis points up.
%
%    RAYS is a structure with the fields 'O' for the ray origins, and
%    'D' for the ray directions.

% Copy origin to O matrix.
rays.O = repmat(pos, w*h, 1);

% Compute directions in camera frame.
[phi,theta] = meshgrid(linspace(0.5*hfov, -0.5*hfov, w), ...
                       linspace(0.5*vfov, -0.5*vfov, h));

phi = phi(:);
theta = theta(:);

D = [cos(theta).*cos(phi), cos(theta).*sin(phi), sin(theta)];

% Rotate camera in world.
roll = orient(1);
pitch = orient(2);
yaw = orient(3);

Rx = [1 0 0; 0 cos(roll) -sin(roll); 0 sin(roll) cos(roll)];
Ry = [cos(pitch) 0 sin(pitch); 0 1 0; -sin(pitch) 0 cos(pitch)];
Rz = [cos(yaw) -sin(yaw) 0; sin(yaw) cos(yaw) 0; 0 0 1];

R = Rx*Ry*Rz;
rays.D = D*R;
```

A.5 findKDSplitPlane.m

```
function s = findKDSplitPlane(node, orient, model)
% FINDKDSPLITPLANE Determines the position of a k-d tree split plane
%    S = FINDKDSPLITPLANE(NODE,ORIENT,MODEL) determines the split
%    plane in the node NODE along the axis given by ORIENT. With
%    ORIENT == 1 is the x-axis, ORIENT == 2 the y-axis, and ORIENT == 3
%    the z-axis. MODEL is a model built by BUILDMODEL.
%
%    S is a structure with the fields 'pos' for the positon of the
%    split plane, and 'orient' for the split axis.
%
%    See also BUILDMODEL, SPLITKDNODES

bestCost = inf;
cTrav = 1;
cIntersect = 10;

f = model.faces(node.faceNums,:);
vInd = unique(f);
numFaces = size(f,1);
rightNum = [numFaces 0 0.5*numFaces];
leftNum = [0 numFaces 0.5*numFaces];

pos = sort(model.vertices(vInd,orient));
oMed = pos(ceil(0.5*length(pos)));

candSplit = [pos(1) pos(end) oMed];

for i = 1:length(candSplit)
    tmpS.orient = orient;
    tmpS.pos = candSplit(i);

    rightBb = node.bb;
    rightBb(1,orient) = tmpS.pos;
    rightDist = node.dist;
    rightDist(orient) = rightBb(2,orient) - rightBb(1,orient);

    leftBb = node.bb;
    leftBb(2,orient) = tmpS.pos;
    leftDist = node.dist;
    leftDist(orient) = leftBb(2,orient) - leftBb(1,orient);

    rightArea = 2 * rightDist(1)*rightDist(2) ...
        + rightDist(2) * rightDist(3) ...
        + rightDist(1) * rightDist(3);
    leftArea = 2 * leftDist(1)*leftDist(2) ...
        + leftDist(2) * leftDist(3) ...
        + leftDist(1) * leftDist(3);

    cost = 2*cTrav + rightArea * rightNum(i) * cIntersect ...
        + leftArea * leftNum(i) * cIntersect;

    if cost < bestCost
        bestCost = cost;
        s = tmpS;
    end
end
```

A.6 intersectKDNode.m

```
function [hit,hitF] = intersectKDNode(node, O, D, tMax, model)
% INTERSECTKDNODE Intersects a set of rays with a k-d tree node.
%    [T,FID] = INTERSECTKDNODE(NODE,O,D,TMAX,MODEL) traverses the node
%    recursively, and intersects the rays, given by the origins in the
%    N-by-3 matrix O and directions in the N-by-3 matrix D, with the
%    k-d tree representation of the 3D model in MODEL. TMAX is a
%    N-by-1 vector representing the maximum range for each ray.
%
%    T is a N-by-1 vector containing the distances along each ray to the
%    point of intersection. Rays that do not intersect returns INF. F is
%    a N-by-1 vector containing the face ID (the row in the MODEL's
%    face matrix) for the triangle each ray hit.
%
%    See also RAYKDTREEINTERSECT, INTERSECTKDNODERANGE

hit = inf(size(D,1),1);
hitF = zeros(size(D,1),1);

if isempty(node)
    return;
end

if node.isLeaf
    n = length(node.faceNums);

    for i=1:n
        f = node.faceNums(i);

        N = model.N(f,:);
        Q = model.vertices(model.faces(f,1),:);

        denom = N(1).*D(:,1) + N(2).*D(:,2) + N(3).*D(:,3);

        t = -(N(1).*(O(:,1)-Q(1)) ...
            + N(2).*(O(:,2)-Q(2)) ...
            + N(3).*(O(:,3)-Q(3)))./denom;

        candT = find(t < hit);

        if ~isempty(candT)
            px = O(candT, model.dimInd(f,1)) + t(candT) .* ...
                D(candT, model.dimInd(f,1));
            py = O(candT, model.dimInd(f,2)) + t(candT) .* ...
                D(candT, model.dimInd(f,2));

            Vx = model.vx(f,:);
            Vy = model.vy(f,:);

            u = model.ood(f) .* ((Vx(3)-Vx(2)).*(py-Vy(2)) - (px-Vx(2)).*(Vy(3)-Vy(2)));
            v = model.ood(f) .* ((Vx(1)-Vx(3)).*(py-Vy(3)) - (px-Vx(3)).*(Vy(1)-Vy(3)));
            w = 1-u-v;

            within = v >= 0 & w >= 0 & (v+w) <= 1;

            hit(candT(within)) = t(candT(within));
            hitF(candT(within)) = f;
        end
    end
else
    % Compute distance to splitting plane.
    t = (node.split.pos - O(:,node.split.orient)) ./ D(:, node.split.orient);
    tol = 1e-2;

    leftRays = O(:,node.split.orient) < node.split.pos | ...
              (O(:,node.split.orient) == node.split.pos & ...
               D(:,node.split.orient) > 0);
    rightRays = ~leftRays;
    nearOnly = t >= tMax | t < 0 | D(:,node.split.orient) == 0;
```

```
    if any(nearOnly & rightRays)
        % Whole interval is on near node.
        curr = nearOnly & rightRays;
        [hit(curr),hitF(curr)] = intersectKDNode(...
            node.right, O(curr,:), D(curr,:), tMax(curr), model);
        rightRays = rightRays & ~nearOnly;
    end

    if any(nearOnly & leftRays)
        % Whole interval is on near node.
        curr = nearOnly & leftRays;
        [hit(curr),hitF(curr)] = intersectKDNode(...
            node.left, O(curr,:), D(curr,:), tMax(curr), model);
        leftRays = leftRays & ~nearOnly;
    end

    if any(rightRays)
        % The ray intersects the plane.
        % Test near node.
        [hit(rightRays),hitF(rightRays)] = intersectKDNode(...
            node.right, O(rightRays,:), D(rightRays,:), t(rightRays), model);

        rightRays = isinf(hit) & rightRays;
        if any(rightRays)
            t(rightRays) = t(rightRays) - tol;
            updOrig = O(rightRays,:);
            updOrig(:,1) = O(rightRays,1) + t(rightRays).*D(rightRays,1);
            updOrig(:,2) = O(rightRays,2) + t(rightRays).*D(rightRays,2);
            updOrig(:,3) = O(rightRays,3) + t(rightRays).*D(rightRays,3);

            % Test far node.
            [hit(rightRays),hitF(rightRays)] = intersectKDNode(...
                node.left, updOrig, D(rightRays,:), tMax(rightRays)-t(rightRays), model);
            hit(rightRays) = t(rightRays) + hit(rightRays);
        end
    end

    if any(leftRays)
        % The ray intersects the plane.
        % Test near node.
        [hit(leftRays),hitF(leftRays)] = intersectKDNode(...
            node.left, O(leftRays,:), D(leftRays,:), t(leftRays), model);

        leftRays = isinf(hit) & leftRays;
        if any(leftRays)
            % Test far node.
            t(leftRays) = t(leftRays) - tol;
            updOrig = O(leftRays,:);
            updOrig(:,1) = O(leftRays,1) + t(leftRays).*D(leftRays,1);
            updOrig(:,2) = O(leftRays,2) + t(leftRays).*D(leftRays,2);
            updOrig(:,3) = O(leftRays,3) + t(leftRays).*D(leftRays,3);

            [hit(leftRays),hitF(leftRays)] = intersectKDNode(...
                node.right, updOrig, D(leftRays,:), tMax(leftRays)-t(leftRays), model);
            hit(leftRays) = t(leftRays) + hit(leftRays);
        end
    end
end
```

A.7 intersectKDNodeRange.m

```
function hit = intersectKDNodeRange(node, O, D, tMax, model)
% INTERSECTKDNODERANGE Intersects a set of rays with a k-d tree node.
%    T = INTERSECTKDNODE(NODE,O,D,TMAX,MODEL) traverses the node
%    recursively, and intersects the rays, given by the origins in the
%    N-by-3 matrix O and directions in the N-by-3 matrix D, with the
%    k-d tree representation of the 3D model in MODEL. TMAX is a
%    N-by-1 vector representing the maximum range for each ray.
%
%    T is a N-by-1 vector containing the distances along each ray to the
%    point of intersection. Rays that do not intersect returns INF.
%
%    The only difference between this function and INTERSECTKDNODE, is
%    that this version performs a little faster, since it does not need
%    to return the face IDs for the triangle each ray hit.
%
%    See also RAYKDTREEINTERSECT, INTERSECTKDNODE

hit = inf(size(D,1),1);

if isempty(node)
    return;
end

if node.isLeaf
    n = length(node.faceNums);

    for i=1:n
        f = node.faceNums(i);

        N = model.N(f,:);
        Q = model.vertices(model.faces(f,1),:);

        denom = N(1).*D(:,1) + N(2).*D(:,2) + N(3).*D(:,3);

        t = -(N(1).*(O(:,1)-Q(1)) ...
            + N(2).*(O(:,2)-Q(2)) ...
            + N(3).*(O(:,3)-Q(3)))./denom;

        candT = find(t < hit);

        if ~isempty(candT)
            px = O(candT, model.dimInd(f,1)) + t(candT) .* ...
                D(candT, model.dimInd(f,1));
            py = O(candT, model.dimInd(f,2)) + t(candT) .* ...
                D(candT, model.dimInd(f,2));

            Vx = model.vx(f,:);
            Vy = model.vy(f,:);

            u = model.ood(f) .* ((Vx(3)-Vx(2)).*(py-Vy(2)) - (px-Vx(2)).*(Vy(3)-Vy(2)));
            v = model.ood(f) .* ((Vx(1)-Vx(3)).*(py-Vy(3)) - (px-Vx(3)).*(Vy(1)-Vy(3)));
            w = 1-u-v;

            within = v >= 0 & w >= 0 & (v+w) <= 1;

            hit(candT(within)) = t(candT(within));
        end
    end
else
    % Compute distance to splitting plane.
    t = (node.split.pos - O(:,node.split.orient)) ./ D(:, node.split.orient);
    tol = 1e-2;

    leftRays = O(:,node.split.orient) < node.split.pos | ...
              (O(:,node.split.orient) == node.split.pos & ...
               D(:,node.split.orient) > 0);
    rightRays = ~leftRays;
    nearOnly = t >= tMax | t < 0 | D(:,node.split.orient) == 0;

    if any(nearOnly & rightRays)
        % Whole interval is on near node.
        curr = nearOnly & rightRays;
```

```
        hit(curr) = intersectKDNodeRange(...
            node.right, O(curr,:), D(curr,:), tMax(curr), model);
        rightRays = rightRays & ~nearOnly;
    end

    if any(nearOnly & leftRays)
        % Whole interval is on near node.
        curr = nearOnly & leftRays;
        hit(curr) = intersectKDNodeRange(...
            node.left, O(curr,:), D(curr,:), tMax(curr), model);
        leftRays = leftRays & ~nearOnly;
    end

    if any(rightRays)
        % The ray intersects the plane.
        % Test near node.
        hit(rightRays) = intersectKDNodeRange(...
            node.right, O(rightRays,:), D(rightRays,:), t(rightRays), model);

        rightRays = isinf(hit) & rightRays;
        if any(rightRays)
            t(rightRays) = t(rightRays) - tol;
            updOrig = O(rightRays,:);
            updOrig(:,1) = O(rightRays,1) + t(rightRays).*D(rightRays,1);
            updOrig(:,2) = O(rightRays,2) + t(rightRays).*D(rightRays,2);
            updOrig(:,3) = O(rightRays,3) + t(rightRays).*D(rightRays,3);

            % Test far node.
            hit(rightRays) =  t(rightRays) + intersectKDNodeRange(...
                node.left, updOrig, D(rightRays,:), tMax(rightRays)-t(rightRays), model);
        end
    end

    if any(leftRays)
        % The ray intersects the plane.
        % Test near node.
        hit(leftRays) = intersectKDNodeRange(...
            node.left, O(leftRays,:), D(leftRays,:), t(leftRays), model);

        leftRays = isinf(hit) & leftRays;
        if any(leftRays)
            % Test far node.
            t(leftRays) = t(leftRays) - tol;
            updOrig = O(leftRays,:);
            updOrig(:,1) = O(leftRays,1) + t(leftRays).*D(leftRays,1);
            updOrig(:,2) = O(leftRays,2) + t(leftRays).*D(leftRays,2);
            updOrig(:,3) = O(leftRays,3) + t(leftRays).*D(leftRays,3);

            hit(leftRays) = t(leftRays) + intersectKDNodeRange(...
                node.right, updOrig, D(leftRays,:), tMax(leftRays)-t(leftRays), model);
        end
    end
end
```

A.8 rayKDTreeIntersect.m

```
function [t,fID] = rayKDTreeIntersect(root, rays, model)
% RAYKDTREEINTERSECT Intersects a set of rays with a k-d tree.
%    [T,FID] = RAYKDTREEINTERSECT(ROOT,RAYS,MODEL) recursively intersects
%    the k-d tree given by the root node ROOT with the rays given
%    by the ray structure RAYS, containing the fields 'O' for the ray
%    origins and 'D' for the ray directions. The k-d tree represents
%    the 3D model given by MODEL, which is built using the function
%    BUILDMODEL.
%
%    See also BUILDMODEL, BUILDKDTREE, CASTRAYSPINHOLE, CASTRAYSSCAN

nRays = size(rays.O,1);
tMax = inf(nRays,1);

if nargout <= 1
    t = intersectKDNodeRange(root, rays.O, rays.D, tMax, model);
else
    [t,fID] = intersectKDNode(root, rays.O, rays.D, tMax, model);
end
```

A.9 raysToImg.m

```
function img = raysToImg(t,h,w)
% RAYSTOIMG Constructs an image from ray tracing results.
%    IMG = RAYSTOIMG(T,H,W) returns an image, given the
%    data in the N-by-1 vector T, and the image
%    height H and width W.
%
%    This function assumes that the rays have been constructed by
%    the functions CASTRAYSPINHOLE or CASTRAYSSCAN, or a function
%    that cast the rays in the same order as these.
%
%    See also CASTRAYSPINHOLE, CASTRAYSSCAN, RAYKDTREEINTERSECT

img = reshape(t,h,w);
img(isinf(img)) = 0;
```

A.10 splitKDNode.m

```
function node = splitKDNode(node, d, orient, model, maxD, maxN)
% SPLITKDNODES Recursively splits the k-d tree nodes.
%    NODE = SPLITKDNODES(NODE,D,ORIENT,MODEL,MAXD,MAXN) splits
%    the node NODE using the function FINDKDSPLITPLANE. D is the
%    current depth, ORIENT is the current split orientations
%    (ORIENT == 1 is the x-axis, ORIENT == 2 the y-axis,
%    and ORIENT == 3 is the z-axis), MODEL is the 3D model the k-d tree
%    represents, MAXD is the maximum depth of the tree, and MAXN
%    is the maximum number of triangles in a leaf node.
%
%    See also BUILDMODEL, BUILDKDTREE, FINDKDSPLITPLANE.

% Number of triangles in node.
n = length(node.faceNums);

% Check if leaf node.
if n <= maxN || d >= maxD
    node.isLeaf = true;
    return;
end

% Find split plane position and orientation.
s = findKDSplitPlane(node, orient, model);

% Partition the triangles.
f = model.faces(node.faceNums,:);
nearV = model.vertices(:,s.orient) <= s.pos;
farV = ~nearV;
nearF = all(nearV(f),2);
farF = all(farV(f),2);
intersectF = ~(nearF | farF);
nearSplit = nearF | intersectF;
farSplit = farF | intersectF;

% Right child
right.faceNums = node.faceNums(farSplit);
right.bb = node.bb;
right.bb(1,s.orient) = s.pos;
right.dist = node.dist;
right.dist(s.orient) = ...
    right.bb(2,s.orient) - right.bb(1,s.orient);

% Left child.
left.faceNums = node.faceNums(nearSplit);
left.bb = node.bb;
left.bb(2,s.orient) = s.pos;
left.dist = node.dist;
left.dist(s.orient) = ...
    left.bb(2,s.orient) - left.bb(1,s.orient);

% Store split.
node.isLeaf = false;
node.split = s;

% Split children.
if orient == 3
    orient = 1;
else
    orient = orient + 1;
end

node.right = splitKDNode(right, d+1, orient, model, maxD, maxN);
node.left = splitKDNode(left, d+1, orient, model, maxD, maxN);
```

Appendix B Example and test scripts

This appendix presents different Matlab scripts that have been used to produce data in this report.

B.1 MovingScanExample.m

```
% Load dataset.
load('Eidsvoll_with_3_M60A3.mat');

% Build model and k-d tree.
model = buildModel(FV);
root = buildKDTree(model);

% RayCast a line scan.
w = 1;
h = 500;
roll = 0;
yaw = pi/4;
t = linspace(0,1,h);
pStart = [-50 -50 370];
pEnd = [50 50 370];
pos = repmat(pStart,h,1) + t'*(pEnd-pStart);
pitch = -pi/2 + (pi/180)*sin(t*4*pi); % Makes the platform "roll"

rays.O = zeros(h^2, 3);
rays.D = zeros(h^2, 3);
for i=1:h
    tmp = castRaysPinhole(...
        pos(i,:), ...
        [roll pitch(i) yaw], ...
        w, h, (pi/9)/h, pi/9);

    rays.O((i-1)*h+1:i*h,:) = tmp.O;
    rays.D((i-1)*h+1:i*h,:) = tmp.D;
end

% Intersect the rays with the k-d tree.
r = rayKDTreeIntersect(root, rays, model);

% Construct the range image.
rImg = raysToImg(r,h,h);

% Construct the point cloud.
px = rays.O(:,1) + r.*rays.D(:,1);
py = rays.O(:,2) + r.*rays.D(:,2);
pz = rays.O(:,3) + r.*rays.D(:,3);

% Plot the model surface.
figure;
hold on;
p1 = patch(FV);
heights = FV.vertices(:,3);
heights = heights-min(heights);
heights = heights/max(heights);
cmap = summer(128);
cdata = cmap(round(1+(heights*127)),:);
set(p1,'FaceColor','interp',...
'FaceVertexCData',cdata,...
'EdgeColor','flat');

% Plot the point cloud
plot3(px,py,pz,'k.', 'MarkerSize',1);
axis equal; axis off;

% Plot the range image.
figure;
imagesc(rImg);
colorbar;
colormap(jet(256));
axis image; axis off;
```

B.2 OcclusionExample.m

```
% Load dataset.
load('M60A3.mat');

% Construct a plane.
planeV = [ ...
    0 0 0; ...
    0 1 0; ...
    0 1 1; ...
    0 0 1];
planeF = [1 3 4; 1 2 3];

% Translate and scale the plane.
planeP = [6 -5 0];
planeS = [1 4 4];
planeV = repmat(planeP,size(planeV,1),1)+ planeV*diag(planeS);

% Add the plane to occlude the tank and remember the plane face IDs.
planeIDs = [1 2] + size(FV.faces,1);
FV.faces = [FV.faces; planeF + size(FV.vertices,1)];
FV.vertices = [FV.vertices; planeV];

% Build model and k-d tree.
model = buildModel(FV);
root = buildKDTree(model);

% RayCast using a pinhole camera model.
w = 500;
h = 200;

rays = castRaysPinhole([100 0 2], [0 0 -pi], w, h, 0.11, 0.05);

% Intersect the rays with the k-d tree.
[r,fid] = rayKDTreeIntersect(root, rays, model);

% Compute the range image.
rImg = raysToImg(r,h,w);

% Plot the range image.
figure;
imagesc(rImg,[min(r(:))-2 max(rImg(:))]);
colorbar;
colormap(jet(256));
axis image
axis off

% Construct the point cloud for those rays that hit the tank.
tankOnly = ~ismember(fid, planeIDs);
x = rays.O(tankOnly,1) + r(tankOnly).*rays.D(tankOnly,1);
y = rays.O(tankOnly,2) + r(tankOnly).*rays.D(tankOnly,2);
z = rays.O(tankOnly,3) + r(tankOnly).*rays.D(tankOnly,3);

% Plot the model surface.
figure;
hold on;
p1 = patch(FV);
heights = FV.vertices(:,3);
heights = heights-min(heights);
heights = heights/max(heights);
cmap = summer(128);
cdata = cmap(round(1+(heights*127)),:);
set(p1,'FaceColor','interp',...
'FaceVertexCData',cdata,...
'EdgeColor','k');

% Plot the point cloud
plot3(x,y,z,'b.', 'MarkerSize',5);
axis equal;
axis off;
```

B.3 PanoramaScanExample.m

```
% Load dataset.
load('Eidsvoll_with_3_M60A3.mat');

% Build model and k-d tree.
model = buildModel(FV);
root = buildKDTree(model);

% RayCast a 360 degree scan.
w = 3600;
h = 200;

rays = castRaysScan([-20 0 180], [0 0 0], w, h, 2*pi, pi/9);

% Intersect the rays with the k-d tree.
t = rayKDTreeIntersect(root, rays, model);

% Construct the range image.
rImg = raysToImg(t,h,w);

% Construct the point cloud.
x = rays.O(:,1) + t.*rays.D(:,1);
y = rays.O(:,2) + t.*rays.D(:,2);
z = rays.O(:,3) + t.*rays.D(:,3);

% Plot the model surface.
figure;
hold on;
p1 = patch(FV);
heights = FV.vertices(:,3);
heights = heights-min(heights);
heights = heights/max(heights);
cmap = summer(128);
cdata = cmap(round(1+(heights*127)),:);
set(p1,'FaceColor','interp',...
'FaceVertexCData',cdata,...
'EdgeColor','flat');

% Plot the point cloud
plot3(x,y,z,'k.', 'MarkerSize',1);

% Use the range image as texture on scan surface 10 meters from O.
surf = rays.O + 10*rays.D;
x = raysToImg(surf(:,1),h,w);
y = raysToImg(surf(:,2),h,w);
z = raysToImg(surf(:,3),h,w);
warp(x,y,z,rImg,jet(64));
set(gca, 'YDir','normal');
caxis([0 64])
colorbar;
axis equal;
axis off;
```

B.4 PerformanceTest.m

```
% Load dataset.
load('Eidsvoll_with_3_M60A3.mat');

% Test performance for differenct number of rays and triangles.
r = [1000 5000 10000 25000 50000 75000 100000];
rayDims = [100 250 500 750 1000 1500 2000];

tModelBuild = zeros(size(r));
tTreeBuild = zeros(size(r));
tCast = zeros(length(r),length(rayDims));
tCastScan = zeros(length(r),length(rayDims));
tIntersect = zeros(length(r),length(rayDims));
t = cell(length(r),length(rayDims));

for i=1:length(r)
    % Reduce the model.
    fvr = reducepatch(FV,r(i));

    % Build model and k-d tree.
    tic;
    model = buildModel(fvr);
    tModelBuild(i) = toc;
    tic;
    root = buildKDTree(model);
    tTreeBuild(i) = toc;

    for j=1:length(rayDims)
        % RayCast pinhole rays covering most of the scene.
        w = rayDims(j);
        h = rayDims(j);
        roll = 0;
        pitch = -pi/2;
        yaw = 0;
        tic;
        rays = castRaysPinhole([0 0 370], [roll pitch yaw], w, h, 0.80, 0.80);
        tCast(i,j) = toc;

        % RayCast scan rays covering most of the scene (not used).
        w = rayDims(j);
        h = rayDims(j);
        roll = 0;
        pitch = -pi/2;
        yaw = 0;
        tic;
        tmp = castRaysScan([0 0 370], [roll pitch yaw], w, h, 0.80, 0.80);
        tCastScan(i,j) = toc;

        % Intersect the pinhole rays with the k-d tree.
        tic;
        t{i,j} = rayKDTreeIntersect(root, rays, model);
        tIntersect(i,j) = toc;
    end
end

% Save results.
nRays = rayDims.^2;
nTriangles = r;
save('performanceResults.mat','t','tModelBuild','tTreeBuild', ...
    'tCast','tCastScan','tIntersect','nRays','nTriangles');

% Plot results
figure;
hold on;
plot(nTriangles,tModelBuild,'-g.');
plot(nTriangles,tTreeBuild,'-bo');
hold off;
legend('Model', ...
    'k-d Tree', ...
    'Location','NorthWest');
title('Time used building the model and k-d tree');
xlabel('Number of triangles');
ylabel('Seconds');
```

```
figure;
hold on;
plot(nRays,tCast(end,:),'-b.');
plot(nRays,tCastScan(end,:),'-gx');
hold off;
legend('Pinhole camera model', ...
    'Scanning camera model', ...
    'Location','NorthWest');
title('Time used on casting rays');
xlabel('Number of rays');
ylabel('Seconds');

figure;
hold on;
plot(nRays,tIntersect(3,:),'-r.');
plot(nRays,tIntersect(5,:),'-go');
plot(nRays,tIntersect(7,:),'-bx');
hold off;
legend( ...
    sprintf('Number of triangles = %d',nTriangles(3)), ...
    sprintf('Number of triangles = %d',nTriangles(5)), ...
    sprintf('Number of triangles = %d',nTriangles(7)), ...
    'Location','NorthWest');
title('Time used intersecting the rays with the k-d tree');
xlabel('Number of rays');
ylabel('Seconds');

% Construct and show the range image.
i = 7;
j = 5;
rImg = raysToImg(t{i,j},rayDims(j),rayDims(j));

rMin = floor(min(rImg(:))/10)*10;
rMax = ceil(max(rImg(:))/10)*10;
v = rMin:1:rMax;

figure;
hold on;
imagesc(rImg);
[C,handle] = contour(rImg, v, 'k');
clabel(C,handle,rMin:10:rMax);
axis off;
axis image;

colormap(jet(256));
```