University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Bachelor's thesis

3D surface reconstruction from depth data reflecting empty space

Plzeň 2020 Káčereková Zuzana

Místo této strany bude zadání práce.

Declaration

I hereby declare that this bachelor's thesis is completely my own work and that I used only the cited sources.

Plzeň, 20th July 2020

Káčereková Zuzana

Acknowledgements

I would like to sincerely thank my supervisor, Doc. Ing. L. Váša, Ph.D., for his guidance at every stage of this work, his enthusiasm, time and invaluable advice.

Káčereková Zuzana

Abstract

The main goal of this work is to explore the possibility of using empty space and capture device position data in 3D surface reconstruction from depth data. This thesis describes the process of depth data retrieval and test data synthesis. Furthermore, it describes the principles of depth data registration and subsequent 3D surface reconstruction based on minimizing a defined cost function. It is concluded with result analysis and a discussion of directions for future improvement. Programmer and user documentation are included.

Abstrakt

Cílem této práce je prozkoumat možnost využití dat o prázdném prostoru a poloze snímacího zařízení při rekonstrukci 3D povrchu z hloubkových dat. Bakalářská práce popisuje proces získávání hloubkových dat z reálných zařízení a proces generace syntetických testovacích dat. Dále popisuje princip registrace hloubkových dat a následné rekonstrukce 3D povrchu založené na minimalizaci objektivní funkce. Na závěr je provedena analýza výsledků a návrh směrů pro budoucí zlepšení. Součástí práce je programátorská a uživatelská dokumentace.

Contents

1 Introduction **9**

2 Data retrieval **10**
2.1 Intel Realsense D415 10
2.2 Kinect 2 for Windows 10
2.3 Azure Kinect 10
2.4 Notes on data retrieval precision 11

3 Data synthesis **12**
3.1 Viewpoint generation 12
3.2 Viewpoint ordering 13
3.3 Generating rays 14
3.4 Orienting the cameras and inverse transformation 16
3.5 Optimizing ray intersection 17
3.5.1 Bounding Volume Hierarchy (BVH) 17
3.5.2 Octree 18
3.6 Ray intersection 19
3.6.1 Ray-box intersection 19
3.6.2 Ray-triangle intersection 20
3.7 Error modelling 20
3.7.1 Intersection error 20
3.7.2 Rotational and translational error 20

4 Registration **21**

5 Representing empty space **22**
5.1 Tracing rays in a 3D grid 23

6 Mesh reconstruction **25**

7 Optimization **27**
7.1 Defining the parameters 27
7.2 Defining the cost function 28
7.2.1 Rays going to infinity, which do not intersect the reconstruction 28
7.2.2 Rays going to infinity, which intersect the reconstruction 28

7.2.3 Rays with model intersection, which do not intersect the reconstruction . . . 30
7.2.4 Rays with model intersection, which intersect the reconstruction . . . 31
7.2.5 Missing rays . . . 32

8 Implementation **34**
8.1 Data format . . . 34
8.2 Data collection . . . 35
8.2.1 Intel Realsense . . . 35
8.2.2 Kinect 2 . . . 38
8.2.3 Azure Kinect . . . 39
8.3 Registration . . . 39
8.3.1 Global . . . 39
8.3.2 Local . . . 40
8.3.3 Sparse Iterative Closest Point . . . 40
8.3.4 Conclusions on registration . . . 40
8.4 Synthetic data generation and processing . . . 40
8.4.1 Data generation . . . 41
8.4.2 Registration . . . 43
8.4.3 Reconstruction . . . 46
8.4.4 Optimization . . . 48
8.4.5 Results . . . 51

9 Further directions **57**

10 Conclusion **58**

Bibliography **59**

A User documentation **63**
A.1 IntelRealsenseRecorder . . . 63
A.2 KinectRecorder . . . 63
A.3 AzureRecorder . . . 63
A.4 SphereGenerator . . . 64
A.5 DataGenerator . . . 64
A.6 PointCloudRegistration . . . 64
A.7 PointCloudReconstruction . . . 65

B Programmer documentation **67**
B.1 IntelRealsenseRecorder 67
B.2 KinectRecorder 67
B.3 AzureRecorder 67
B.4 SphereGenerator 68
B.5 DataGenerator 68
B.6 PointCloudRegistration 69
B.7 PointCloudReconstruction 70

C Attachments **72**

1 Introduction

One of the most prominent areas in computer graphics is that which concerns itself with representing 3D objects and surfaces. These objects and surfaces are most commonly represented by polygon meshes. A polygon mesh is a collection of vertices in three-dimensional space, connected with edges, forming polygonal faces.

Polygon meshes have endless applications, most notably in manufacturing or the entertainment industry. Currently, most meshes must be modeled by hand, often requiring artistic or technical skills, and extensive experience with complex 3D modeling software. With such methods, precise replication of real-life objects is time-consuming, or in complex cases, downright impossible.

In order to simplify, or eventually remove, the process of developing a polygon mesh (or other surface representations), methods of capturing real-world data have been developed. This process is called 3D scanning.

A 3D scanning device provides data in the form of distances to objects, measured relative to its position. This data can be converted to a set of points in 3D space. Classical 3D mesh reconstruction methods take this set of points as input. This, however, disregards information about space between the device and these points, which must have been empty.

In this work, data is first retrieved from several types of 3D scanning devices and unified in a process called registration. Then, synthetic depth data is generated along with registration data. Registration outputs a transformation deemed best descriptive of the relative real-world position of two sets of points. An algorithm is then devised to take into consideration the space between the device and the registration result. A triangle mesh of a 3D surface is retrieved and fitted iteratively to the measured data via cost function minimization. Results are analyzed, including a discussion of further directions for improvement.

The reader is first familiarized with the theoretical solution. Later, a description of the implementation of all software developed for the purposes of this thesis is provided.

2 Data retrieval

Nowadays, a great variety of 3D scanning devices is available, each utilizing technologies with a set of strengths and weaknesses. They range from structured-light, time-of-flight and triangulation technologies to photogrammetry and contact-based ones [3]. In this section, devices available for the purposes of this thesis are specified.

2.1 Intel Realsense D415

The Intel Realsense D415 uses Active IR stereo technology, wherein a pair of cameras are used to capture images with a slight offset, which are then correlated and used to generate a depth frame. An infrared light pattern of dots is projected onto the scene to provide additional texture. The relatively narrow field of view of the device results in a greater pixel density and should, therefore, prove more suitable in 3D scanning applications and applications requiring greater precision, especially at a short range (<1 m). The device promises a range of about 0.16 m up to approximately 10 m, with regard to lighting conditions [10][15].

2.2 Kinect 2 for Windows

The Kinect series offers primarily body tracking capabilities. Specifically, Kinect 2 features time-of-flight technology, which uses rays of infrared light and makes precise measurements of their flight time. Given the speed of light, which is known, and the time of flight, surface distance is calculated [30].

2.3 Azure Kinect

The Azure Kinect, released in mid 2019 [29], is a successor to the above mentioned Kinect 2. It features a time-of-flight based depth camera [4] and promises greater precision.

2.4 Notes on data retrieval precision

All of the above-described devices can produce significant errors in measurement. In more professional setups, such errors can be minimized by using tripods in combination with turning platforms, upon which the targeted object is placed. Some devices may use built-in stabilization systems. It is also desirable to minimize reflections and transparency in the scanned objects. To this purpose, scanned objects may be covered in, for example, chalk.

The data produced in the context of this thesis was produced with handheld imaging and may therefore not be representative of the best achievable results.

3 Data synthesis

In order to validate the results of this work, precise data sets are needed to eliminate the possible impacts of 3D scanning device errors, but also to simulate various types of error and study their impact.

During data synthesis, a virtual scanning device is simulated. Such a device must simulate the process of casting rays through a 2D array of pixels, which may then intersect a surface, as illustrated in fig. 3.1. By calculating these intersections, we gain a relatively precise point cloud. Precise registration data can also be retrieved. Registration is discussed in the following section. It is also desirable for the captured frames to be ordered so that consecutive frames have maximal overlap.

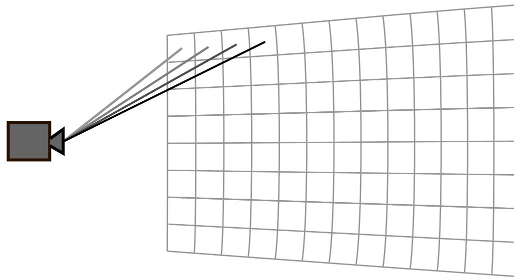

Figure 3.1: An illustration of ray casting.

3.1 Viewpoint generation

As specifying camera positions by hand for each data set would be extremely time consuming, some method of generating these positions is necessary.

An application which could display a desired model and enable movement (both translational and rotational) in 3D space would provide maximum control over positioning the virtual camera. It could provide options to either capture viewpoint data at flight, or button press. However, this still requires some level of time-consuming interaction.

Alternatively, a 3D model could be surrounded by spherically arranged, evenly spaced cameras. Models of spheres with even vertex spacing can be easily generated, for example by subdivision of the faces in an octahedron, as seen in fig. 3.2. Greater coverage of the model can be achieved with more levels of subdivision. While losing some control, this process can be fully automated. However, some parts of the surface might not be captured.

In order to avoid having to create data sets manually, the second option is selected for this work.

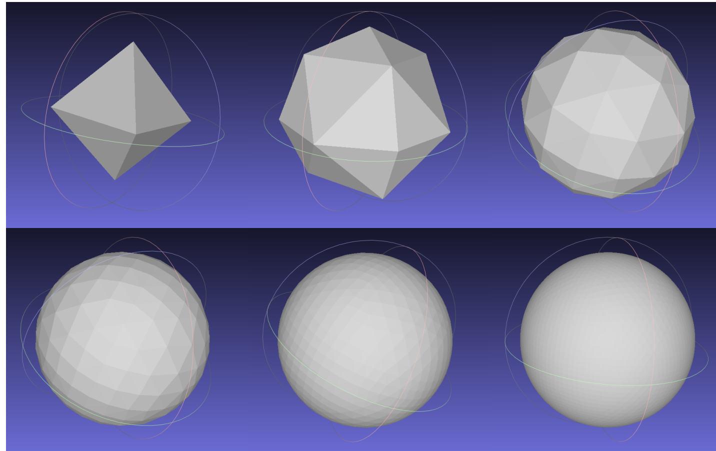

Figure 3.2: An octahedron, subdivided progressively.

3.2 Viewpoint ordering

As stated above, ordering frames to produce maximum overlap in the data is beneficial during the registration stage.

An application capturing camera movement provides this ordering naturally, if the option to capture single frames is neglected.

Such an order does not generally arise from sphere generation methods. However, a sphere model generated by face subdivision maintains vertex connectivity by edges. This configuration of vertices and edges translates seamlessly to a mathematical graph. A breadth-first search (BFS) algorithm can then provide the k-neighborhood of a given vertex in a graph and effectively separate the sphere into levels.

By finding the vertex with a maximal y-axis coordinate and selecting it as the start node for the BFS algorithm, vertices with equal distances from the start can be sorted into individual lists. They can then be sorted by their x-axis coordinates, separating them into a left and right half. Lastly, by sorting one list by increasing z-axis coordinate and the other by decreasing z-axis coordinate, and concatenating them appropriately, a round path along a level of the sphere is retrieved.

An interesting observation is that for graphs which cannot produce a path using a similar, simplified method, this leads to a Hamiltonian path problem, which is NP complete [34].

Positioning cameras at the vertex locations generated by this algorithm and processing frames in this order yields a large overlap in the output data.

3.3 Generating rays

Rays are defined by their origin and direction. The origin is also the position of the camera. Given a distance t, a point along the path of the ray can be described as

$$P = O + t * \vec{d} \tag{3.1}$$

where O is the point of origin and $\vec{d}$ is the normalized direction vector. While simulating a camera, we are only interested in positive t values as they signify a ray intersection in front of the lens.

A camera is defined by the width and the height of the image it produces and its field of view (FOV), that is, the angle under which the world is observed. Typically, near plane and far plane distance are also specified. Together, these values define the *camera frustum* [32], as seen in figure 3.3. Objects inside or intersecting the frustum are visible to the camera, while objects outside of it, in front of the near plane, or beyond the far plane, are not visible.

The width, height and FOV values specify a 2D array of pixels and rays are generated to pass through the centers of these pixels. In this work it is assumed that the camera is pointed in the positive sense of the z-axis and the pixel array is positioned one unit from the origin. No near or far plane is specified.

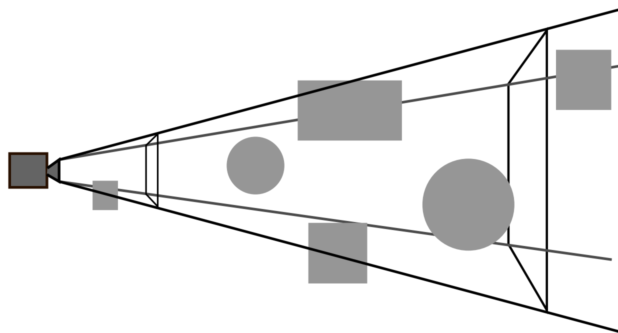

Figure 3.3: Camera view frustum with the near and far plane displayed.

The following formulas are available at *Ray-Tracing: Generating Camera Rays*, an online tutorial by Scratchapixel 2.0 [25].

The pixel coordinates are specified in raster space, but in order to calculate intersections with a desired model, they must be transformed to NDC

(Normalized Device Coordinates) space, further to screen space, and finally to world space. Coordinate spaces are illustrated in figures 3.4 and 3.5.

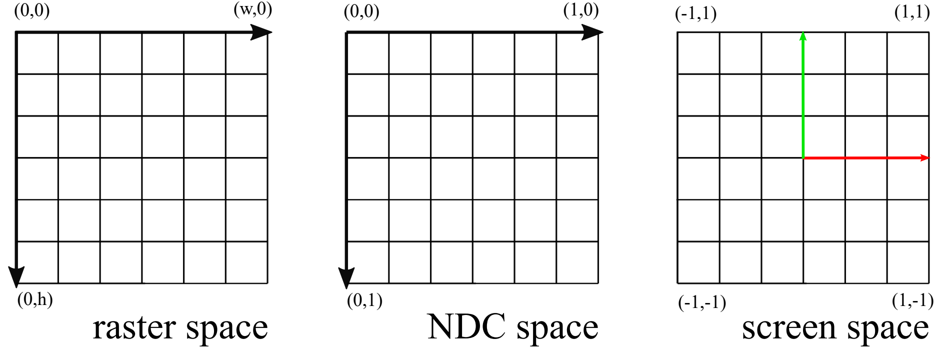

Figure 3.4: Coordinate spaces.

Normalized device coordinates of pixel centers can be calculated as

$$p'_x = \frac{p_x + 0.5}{w} \tag{3.2}$$

$$p'_y = \frac{p_y + 0.5}{h} \tag{3.3}$$

where p_x and p_y are raster space coordinates (column and row indices, respectively), w is the width of the image in pixels, h is the height of the image in pixels, and p'_x and p'_y are the normalized device coordinates.

We then transform the coordinates to screen space, flipping the values along the Y axis as to reflect its positive upward direction.

$$p''_x = 2 * p'_x - 1 \tag{3.4}$$

$$p''_y = 1 - 2 * p'_y \tag{3.5}$$

Here, p''_x and p''_y are the pixel coordinates in screen space.

Finally, transformation to world space unites the coordinate system for both the model and the camera. At this point we also consider the aspect and scale values for the camera, calculated as follows:

$$a = \frac{w}{h} \tag{3.6}$$

$$s = \frac{FOV}{2} \tag{3.7}$$

where a is the aspect ratio and s it the image scale.

Up to this point, for non-square width to height ratios, the calculated pixels would have been rectangular and their centers, and therefore the rays,

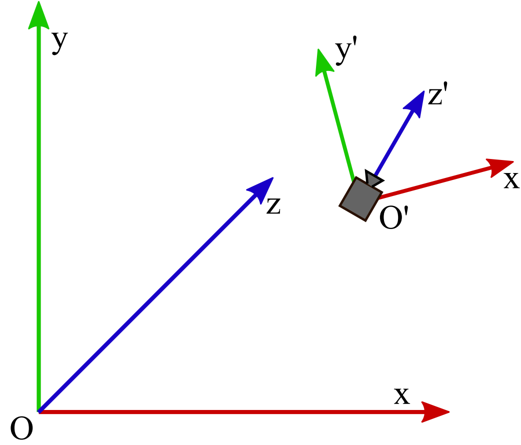

Figure 3.5: World space vs. camera space. O is the world space coordinate system origin. O' is the camera space coordinate system origin, in relation to which synthetic point cloud data is generated.

differently spaced in vertical and horizontal direction. To remedy this, the coordinates are stretched by the image aspect ration along the X axis, leaving the [-1, 1] range.

At last, the world space dimensions of the 2D array are determined, using the camera's field of view and the implicit plane distance of 1. The entire transformation is then

$$p_x''' = (2 * \frac{p_x + 0.5}{w} - 1) * a * s \tag{3.8}$$

$$p_y''' = (1 - 2 * \frac{p_y + 0.5}{h}) * s \tag{3.9}$$

The p_x''' and p_y''' coordinates are said to be in camera space, but in the default position they are identical to world space coordinates. If the camera is moved, the world space coordinates of rays change, but camera space coordinates remain the same, as seen in fig. 3.5.

The points are specified as $P = (P_x, P_y, 1)$, setting the z-coordinate as per the default position of the image plane. They define the directions of individual rays, and given that the camera is in default position, normalizing these values yields the final ray direction.

3.4 Orienting the cameras and inverse transformation

The translation and rotation of a camera can be described using a 4x4 transformation matrix, called the camera-to-world matrix [25]. Transforming

points in camera space using this matrix yields their world space coordinates. The inverse operation is described by the world-to-camera matrix.

Using the vertices of a sphere model, cameras are placed around the model and oriented towards it. This can be achieved by generating a camera in default position. The orientation of the camera can then be defined by the standard basis in R^3 and the camera-to-world transformation can be obtained as

$$\begin{bmatrix} b_1.x & b_2.x & b_3.x & origin.x \\ b_1.y & b_2.y & b_3.y & origin.y \\ b_1.z & b_2.z & b_3.z & origin.z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

from any other orthonormal basis $\{b_1, b_2, b_3\}$ in R^3. More on linear algebra and bases can be found in *Linear Algebra* by Libuše Tesková [28].

The position of the origin (a sphere model vertex), and direction towards the centre of the target mesh, define one of the basis vectors. A second vector for the basis can then be obtained by finding a vector perpendicular to the first vector. Two vectors are perpendicular when their dot product equals zero. The third vector can then be calculated as the cross product of the two existing basis vectors. In order for this basis to be orthonormal, all of the vectors must then be normalized.

This method of synthetic data generation outputs point clouds in their respective camera space coordinates. By attaching the camera-to-world matrix to the output, we maintain information about their original position relative to the model and can then use it for precise point cloud registration.

3.5 Optimizing ray intersection

In a naive case, rays may be tested for intersection with every single polygon in a model. This becomes computationally expensive as the number of polygons in a model grows. Many of these polygons are nowhere near the path of the ray. Ideally, larger areas of space could be tested for ray intersection, excluding many polygons contained within them at once. Two methods of space partitioning are discussed further.

3.5.1 Bounding Volume Hierarchy (BVH)

A BVH [6] is a data structure defining the bounds of a scene by using primitives of which the ray intersection can be calculated faster than for the

scene itself. The better the primitive fits the scene, the fewer rays which miss the model are tagged as intersecting the bounding volume, but the intersection calculation time grows with volume complexity. Mostly, spheres and bounding boxes are used, or sets of bounding planes.

The performance of the structure relies on objects in the scene being grouped hierarchically by proximity, as can be seen in fig. 3.6. In order to achieve this, they may be subdivided by using, for example, an octree. In which case, a bounding volume is calculated for each object and the resulting volumes define the bounding volume of the scene, which could then be subdivided using an octree, which is described in the next section. The tree is traversed top-down and objects are grouped within nodes, with objects intersecting multiple areas being assigned to only one. Once all objects are processed or a specified depth is reached, the structure is traversed upwards, and bounding volumes of the groups are calculated. At each level, the child bounding volumes are united to form a larger one.

Running the ray intersection algorithm then allows for skipping geometry contained in volumes which do not intersect the ray themselves. The process can be further optimised by taking advantage of volume proximity - volumes closer to the ray origin are more likely to contain the true intersection.

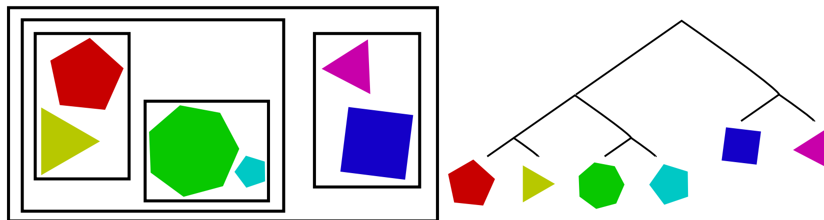

Figure 3.6: BVH example.

3.5.2 Octree

An octree [16] is a cubic data structure, represented as a tree. At each level, it is subdivided into eight sections of identical size, called octants, as per fig. 3.7. The octree provides an efficient method of searching 3D space, allowing non-uniform node subdivision, due to which more detailed division of space can be achieved in denser areas of data sets.

Mesh data, such as the triangular faces of a mesh, can be inserted into an octree. The algorithm traverses the tree from the root node, dividing it until a desired depth is reached, or a user-specified node capacity is exceeded, and inserting references to geometry in the leaf nodes.

This process yields a hierarchy of cubic nodes, which can be tested for ray intersections. As a node can only contain a ray-face intersection if it is itself intersected by the ray, recursive testing of child nodes for ray intersections returns a list of all leaf nodes, which may contain an intersected face. This limits the total number of faces tested for an intersection from the entire data set, to only the faces along the path of the ray. If multiple intersections are found, the closest one is selected.

While a BVH may provide a better hierarchy in which references to geometry are not repeated, the octree was selected for the purposes of this work, as it is easier to implement and can be modified for reduction of mesh detail, which is used in merging real data.

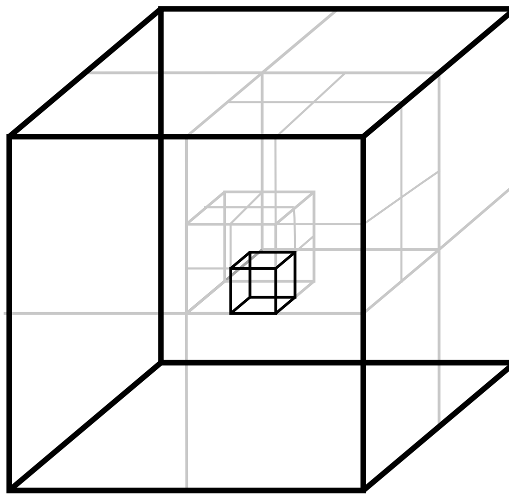

Figure 3.7: Octree subdivision example.

3.6 Ray intersection

Next, the required ray intersection algorithms are discussed.

3.6.1 Ray-box intersection

When using an octree to cut down on the number of intersections calculated, intersections with the octants are first calculated. Running the algorithm from the root of the octree allows us not to further examine the children in octants which have been already determined not to intersect. This process is repeated recursively down the tree structure, until all the leaf nodes intersected by the ray are determined.

Given that these nodes are axis-aligned, we can use a ray-box intersection algorithm for axis-aligned bounding boxes. A bounding box is defined by its extremes - the minimum and maximum coordinates.

In this work, ray-box intersections are calculated using a code snippet by Scratchapixel 2.0 [2], which is distributed under the GNU General Public License version 3 or later.

Once all relevant nodes are retrieved, intersections with scene geometry contained within them are calculated.

3.6.2 Ray-triangle intersection

In this work, ray-triangle intersections are calculated using a code snippet by Scratchapixel 2.0 [26], which is distributed under the GNU General Public License version 3 or later. It is an implementation of the Möller–Trumbore intersection algorithm.

3.7 Error modelling

As real-world data is never captured without error, it is appropriate to test algorithms for their resistance to errors and ability to correct them. To that purpose, methods of introducing error into the synthetic data are included.

3.7.1 Intersection error

One possible source of error comes from the depth measurement, which causes ray intersection to be detected closer or further along the path of a ray. In synthetic data, this is modelled simply by changing the length of the rays randomly, with a specified maximum difference.

3.7.2 Rotational and translational error

This relates to the earlier described process of camera positioning. For real data, the inverse transformation must be calculated during registration, and will be imprecise. This can be simulated by a slight transformation of the inverse matrix.

$$T_e = E * T_o \tag{3.11}$$

T_o is the accurate transformation, E is an error transformation, then T_e is the new transformation with error.

4 Registration

Registration is the process of aligning two point clouds, so that they share a common coordinate system [24]. The position of one of the point clouds remains fixed, while the other is transformed.

A point cloud may first be preprocessed in order to remove outliers. It may also be downsampled to improve registration speed. This can be achieved by using space partitioning to group vertices and taking the average of their positions, the result of which maintains features of the original point cloud, ideally with far fewer points. Using an octree with a specified leaf node size allows control over the resolution of the output.

Implementing a registration algorithm is outside the scope of this work. However, the resulting transformation is subject to optimization at a later stage. Several registration utilities were tested and are later described in the implementation section.

Several types of transformations may be applied, as per table 4.1.

Rigid	Rigid transformation refers to rotation, translation or a combination thereof. It preserves object shape and size.
Affine	In addition to rotation and translation, shearing and scaling may also be applied.
Non-rigid	Non-rigid transformation allows all points to be transformed independently.

Table 4.1: Transformation types.

5 Representing empty space

Registration of depth images and 3D surface reconstruction have been the subject of extensive research and many uses for data about the position of the sensor (and the empty space it defines) have been explored as well. For example, in *Range Image Registration via Probability Field* (2004)[36], a work focusing on point cloud registration, a probability field (p-field) data structure is introduced, which enables the authors to define the probability of a surface passing through a given point in space on a per-ray basis, allowing custom sensor error models (and therefore probability falloff in the space near ray intersection) to be defined. Empty space is also considered in this work, as space before the intersection, once the probability for a given ray reaches zero, can make negative contribution to the overall probability that the surface passes through the area. However, the work was limited to range images taken either from one viewpoint, or viewpoints along the same plane, oriented in the same direction.

Even today, new applications for sensor position data are discovered, such as in *Poisson Surface Reconstruction with Envelope Constraints* (July 2020) [17], where the fact that no objects can lie between the sensor and captured data is used to create a watertight hull using color and depth scans, effectively using the bounds of the object as seen by the camera to define the area where a surface can exist. This is then used to adapt an existing reconstruction method, which otherwise struggles with undersampled areas, and achieve results which adhere much more closely to the real object volume. An octree is used to describe the exterior of the mesh, that is, the space known to be empty.

In this thesis, a volumetric representation is used and a regular grid is constructed around the registered data. This grid is filled with values representing whether a cell lies inside or outside of a surface. Such information can be described by an *indicator function* [27], that is a function of which the value indicates, whether an event has occurred or has not. The grid, assumed to be occupied at start, is then carved by camera rays, revealing the volume of the model.

Volumetric representations of depth data had already been used in early work, such as *A Volumetric Method for Building Complex Models from Range Images* (1996) [11], where *space carving* is defined as the process of carving empty space into the grid. The authors achieve efficient and robust surface reconstruction with inherent hole-filling properties using the marching cubes

algorithm, which is described in chapter 6.

5.1 Tracing rays in a 3D grid

In order to determine, which parts of the grid are being affected by a given ray, it is necessary to implement a line drawing type algorithm in 3D, as testing grid cells for intersections would be extremely wasteful. The direction of a ray does not change as it travels and therefore it can be predicted which cells will be affected by the rate at which it traverses the grid in direction of each axis.

The Bresenham algorithm, first introduced in *Algorithm for computer control of a digital plotter* [9], is a classic method of determining which pixels should be colored when lines are drawn in a discrete environment, such as a screen. The algorithm keeps track of an error value in order to maximize the adherence of the discrete line to the mathematical ideal. Several situations may arise, which define octants that a line to be drawn will fall into based on its slope.

However, due to the many possible situations that can arise, a simpler alternative is implemented in this work.

First, an intersection between the grid and a given ray is found, using the ray-box intersection method described in section 3.6.1. If the ray intersects the grid, both the start and end indices into the grid are calculated.

Then, several arrays are used. First, an array of integer step directions, $idirs$ to be made in the grid indices, calculated as the signum function of each element of the ray direction vector. Then an array of step sizes $ddirs$, which is calculated as

$$ddirs = idirs * r \tag{5.1}$$

where r is the grid resolution.

The *planes* array contains the plane coordinates along each axis, at which the ray will next intersect a grid cell wall. It is calculated as

$$planes = G_{min} + (coords + (idirs > 0 \; ? \; 1 : 0)) * r \tag{5.2}$$

in which G_{min} is the minimal grid coordinate in each axis and $coords$ is the current grid index array. Lastly, the t array contains the next coordinates at which a plane will be intersected, calculated as follows:

$$t = \frac{(planes - origin)}{dir} \tag{5.3}$$

where dir is and element of the ray direction vector.

Then a while loop is entered, continuing until the current grid indices are equal to the end position. A parameter is maintained to keep track of whether a ray which has entered the grid has left the grid area, at which point the algorithm can also end.

The t value is always positive as a plane in negative direction of the origin will also be approached in the negative direction. Therefore, during the loop, the minimum t value determines in which direction a step should be made. The value is then recalculated for a new plane in the chosen step direction and the loop continues until a break condition is reached.

At each step new coordinates are stored in a list.

6 Mesh reconstruction

Once the 3D grid is carved, the only cells which are filled are the ones which had not been hit by any ray. At this point, we reconstruct a mesh from this data, which will be later subject to optimization.

Marching cubes is a common algorithm used to reconstruct isosurfaces from 3D scalar fields. It has, most prominently, applications in MRI scanning, which produce a 3D value data set [7].

For the purposes of this algorithm and implementation as per *Polygonising a scalar field* [7], the 3D grid values which we have previously considered to be cells are now seen as vertices, interconnected by edges. Such vertices can be classified as either under or above a specified isosurface. These vertices form grid cells, for which a limited number of configurations of vertices above or under the surface exists. Configurations of these vertices then specify mesh faces, of which an example can be seen in fig. 6.1. As the configurations are known and their number is limited, an efficient implementation using lookup tables is possible.

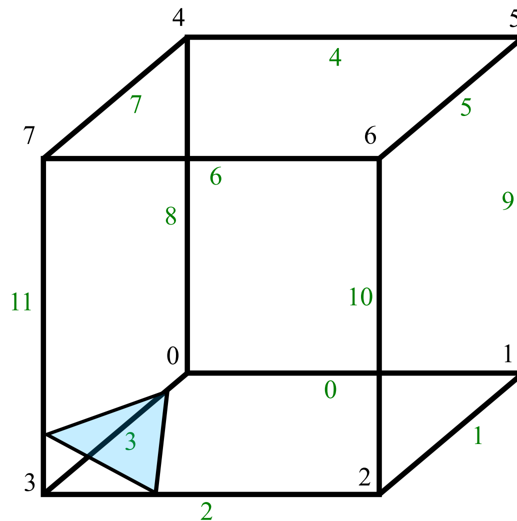

Figure 6.1: Vertex 3 under the surface, and remaining vertices above the surface, defining a triangular face in the reconstruction. The vertices and edges are numbered as per convention in this implementation.

By specifying vertex and edge numbering as per fig. 6.1, we can create an 8-bit indexing scheme into a lookup table of intersected edges, wherein bits corresponding to vertices under the surface are set. This lookup table then returns a 12-bit number, where bits corresponding to the edges through

which a surface passes are marked. In order to find the position at which an edge is intersected, linear interpolation is used. It divides the edge to reflect the value difference between the adjacent vertices, shifting the intersection closer to the vertex which holds a value closer to the isosurface value. It is calculated from the following formula:

$$P = P1 + (i - V1)(P2 - P1)/(V2 - V1) \tag{6.1}$$

where P is the intersection, P_1 and P_2 are the vertex coordinates, V_1 and V_2 are the values contained by the vertices and i is the value at the surface level.

Finally, the same index is used to index the last table, which contains a list of up to 5 faces, specified by the newly formed vertices at edge intersections. More advanced implementations may account for vertex duplication at edges which have already been processed. This version produces a set of triangles rather than a true connected mesh. For use in further applications, duplicate vertices should be unified, either in the reconstruction process, or for example by using an octree, which may, however, suffer from precision issues for pairs of vertices very close to each other. Such an approach should also consider possible minor differences in vertex coordinates due to floating point number representation. Alternatively, a hash table could be used.

In this phase, the grid values are binary - either a ray had passed through a cell, or the cell is full. During the optimization process, the grid values should be adjusted to reflect a wider range of values, resulting in a smoother reconstruction.

7 Optimization

The initial reconstruction outputs a polygon mesh, but this mesh will not fit the data optimally due to errors in registration and depth measurements.

In order to judge the quality of the reconstruction, a value must be assigned to its state. This value should represent how much the reconstruction adheres to the data, a lower value signifying greater adherence. This value is the output of a cost function $C(\vec{p})$, where $\vec{p}$ is a given vector of parameters, like point cloud transformations and the state of the grid.

The value of this function can then be minimized (therefore maximizing adherence to the data) using gradient descent [23]. The optimal solution can be described as

$$\vec{p}_{opt} = argmin(C(\vec{p}), \vec{p}) \tag{7.1}$$

The idea is that if P has n parameters, a function in n-dimensional space exists, which describes the quality of the reconstruction at any given parameter configuration. The gradient of the function then gives the direction of steepest ascent. Using the gradient to modify parameter values, a local minimum of the cost function can be reached relatively quickly, which yields the parameters of a reconstruction that is, in some way, better than the original. However, it is not necessarily optimal, as finding the global minimum is not guaranteed.

This iterative process can be formulated as

$$\vec{p}_{k+1} = \vec{p}_k - \delta \nabla C(\vec{p}_k) \tag{7.2}$$

where $\vec{p}_{k+1}$ is the next iteration, p_k is the current iteration state, $\nabla C(\vec{p}_k)$ is the gradient of the cost function at current iteration state, and δ is step size.

7.1 Defining the parameters

Using gradient descent, both the outcome of point cloud registration and mesh reconstruction can be optimized. These two optimizations can be executed separately, in several rounds.

For registration, the parameters in question are rotation and translation. This yields six parameters if rotation is expressed as rotation around the x, y and z axis.

For reconstruction, grid values can be used as the parameter vector.

7.2 Defining the cost function

The goal of the cost function is to evaluate the adherence of a polygon mesh to a data set consisting of rays. These rays have two relevant characteristics:

- they either have an intersection, or they travelled to infinity,
- and they either intersect the reconstruction, or they miss it.

7.2.1 Rays going to infinity, which do not intersect the reconstruction

Such rays do not contribute to the cost function.

7.2.2 Rays going to infinity, which intersect the reconstruction

For these rays, determining the severity of error means determining how far they are from having missed the reconstruction, as they should have according to the data. Fig. 7.1 illustrates two errors of different severity. The error is then defined by the angle between such a ray, and a ray intersecting the nearest point on the silhouette of the reconstruction.

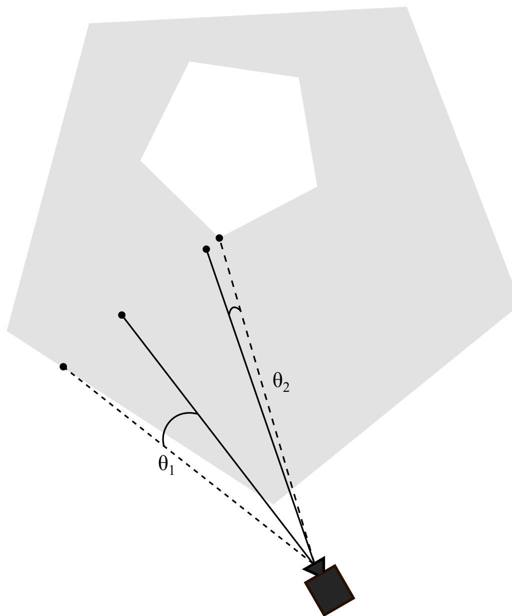

Figure 7.1: Determining severity of error from silhouette intersection. θ_1 represents a more severe error than θ_2.

Determining precisely the nearest point on the silhouette would be a complicated process. Therefore, existing logic is used to trace rays facing the reconstruction at each given viewpoint, creating a 2D array of hit or miss values, effectively defining the silhouette. The coordinates of the current ray are calculated as per

$$h_c = (\frac{\frac{x}{s*a} + 1}{2}) * w - 0.5 \tag{7.3}$$

$$h_r = (\frac{\frac{y}{s} - 1}{-2}) * h - 0.5 \tag{7.4}$$

where h_c is the column index into the hit/miss array, h_r is the row index, s and a refer to the scale and aspect ratio values from earlier mentioned formulas 3.6 and 3.7 and h and w refer to camera pixel array width and height parameters specified in section 3.3, given that the ray direction vector is first scaled so that the z-axis value is equal to one. This is an inverse operation to the ray generation scheme.

Progressively further fields are then searched until the closest miss value is located, as illustrated in fig. 7.2.

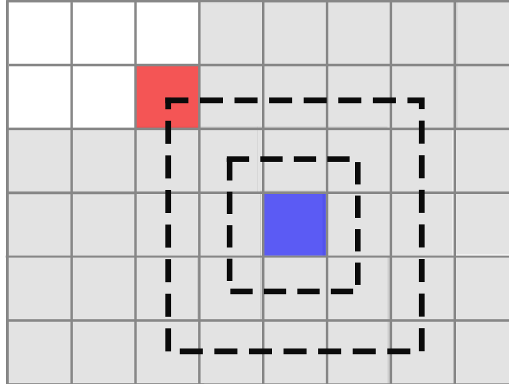

Figure 7.2: Locating the nearest miss value in square paths. The blue area represents the area hit. The red area is the first discovered miss value.

It is to be noted that this is an approximation that may not always yield a correct result. Not only is the search movement not uniform, as the layers searched are rectangular, but some areas of the silhouette which may have caused the ray to pass through the mesh may not be apparent from a lower resolution hit/miss array.

There is also some amount of processing power wasted, as by numbering the depth of nodes within the silhouette, starting at 0 for hit nodes with missed neighbors and progressing inward, a 2D array of values signifying a field's distance from the edge could be constructed. The search would then be completely removed and obtaining the value would be a matter of

calculating this array once for each optimization step and then indexing into it by ray hit coordinates. This would require an implementation of the fast marching method, which is summarized in *A brief description of the fast marching method* [1].

Finally, the angle between the rays is calculated from formula 7.5,

$$cos\theta = \frac{\vec{u} * \vec{v}}{|\vec{u}| * |\vec{v}|} \tag{7.5}$$

making 7.6

$$C_1(\vec{p}) = \sum_{R_1} \theta_i \tag{7.6}$$

the final form of the first part of the cost function, where R_1 is the set of all rays with these properties.

7.2.3 Rays with model intersection, which do not intersect the reconstruction

If a ray hits the scanned surface, it should also intersect the reconstruction, or lie close to the reconstructed surface. A simple approximation of distance to the surface is the nearest vertex, as seen in figure 7.3. Vertices in the reconstruction will be somewhat evenly spaced due the underlying grid and nature of the marching cubes algorithm. The second part of the cost function is therefore calculated as per formula 7.7,

$$C_2(\vec{p}) = \sum_{R_2} (I_i - V_{in}) \tag{7.7}$$

where I_i is the expected intersection and V_{in} is the nearest vertex to it.

Because this calculation will likely be executed for a large amount of rays, there is space for optimization. One data structure suitable for calculating nearest points is the k-dimensional tree (k-d tree, also written as kd-tree) [20], illustrated in fig. 7.4.

A k-d tree can be constructed by recursively dividing a set of points in two by a plane in one of the dimensions, using the median point as the divider. The dimensions are cycled through, in 3-dimensional space starting by using the x coordinate of the median to split the root node, then using the y coordinate at next level, then z and so on.

The complexity of building such a k-d tree depends on the algorithm used to determine the median at each level. Generally, constructing k-d trees is a discipline by itself, with a theoretical O(N log N) lower bound on algorithmic complexity. More on this can be found in *On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)* [33].

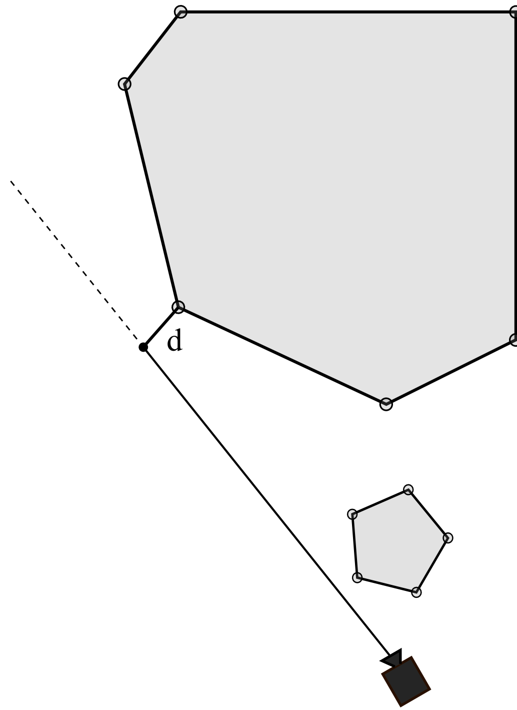

Figure 7.3: Measuring distance d to the nearest vertex in the reconstruction.

However, for each frame used in the reconstruction, a k-d tree only needs to be built once per each cost function calculation, assuming that the underlying model has changed. Finding the nearest vertex to any given point is then a matter of tree traversal and can be done in O(log N) time.

If the k-d tree is used to separate a set of vertices into groups of some maximum size, the nearest neighbor of a ray intersection can most likely be found in the leaf node into which the intersection would fall. This may be false if the nearest neighbor lies further than the nearest node divider, in which case, the neighbor node should be searched (as shown in fig. 7.5). The tree is traversed upwards until the distance-from-divider condition is satisfied by a nearest discovered neighbor.

7.2.4 Rays with model intersection, which intersect the reconstruction

This is the most simple case, in which a ray both intersected the scanned object and the reconstruction. It is unlikely that these intersections are identical, but ideally their distance should be minimal. The cost function is the squared difference of I_s, the scan intersection, and I_r, the reconstruction

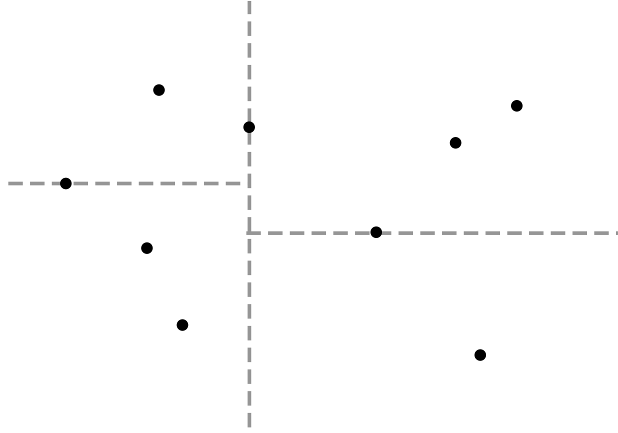

Figure 7.4: A k-d tree example.

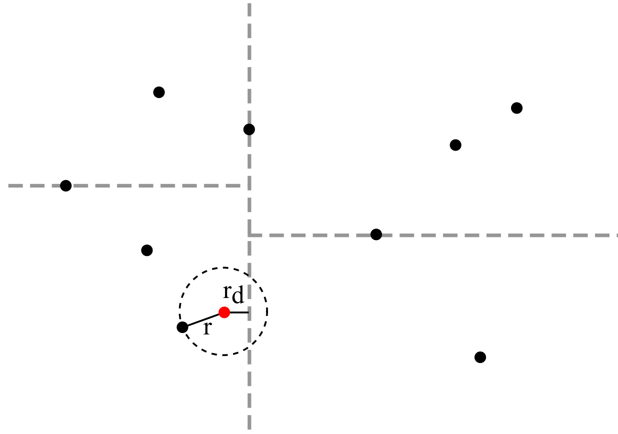

Figure 7.5: Edge case in nearest neighbor search using a k-d tree, where r is the distance to the nearest neighbor candidate and r_d is the distance to the nearest dividing plane.

intersection, summed over the set of these rays.

$$C_3(\vec{p}) = \sum_{R_2}(I_{si} - I_{ri}) \tag{7.8}$$

7.2.5 Missing rays

It is worth noting that some rays may have been omitted by the scanning device. No contribution is acknowledged in the cost function.

Finally, the complete formula is

$$C(\vec{p}) = \alpha * C_1(\vec{p}) + \beta * C_2(\vec{p}) + \gamma * C_3(\vec{p}) \tag{7.9}$$

Where α, β and γ signify the weights with which each set of rays contributes to the total cost.

The cost function can be used both for registration and reconstruction optimization.

8 Implementation

The software created for the purposes of this work has several parts.

The first part deals with data collection and synthesis, consisting of data capture utilities for the 3D scanning devices, and a data synthesizer, a part of which is also a sphere model generation program, developed earlier for KIV/ZPG.

During this work, collection and registration of real data was explored first, eventually leading to a focus on synthetic data in the reconstruction and optimization process.

8.1 Data format

All point cloud and mesh data is represented using a very basic Wavefront OBJ [8] viewer compatible format, as tested using MeshLab, a mesh viewing and editing tool. The applications expect and output only vertex and face data, formatted as per example 8.1. Parameters `c1`, `c2`, `c3` are vertex coordinates in the x, y and z axis respectively. Parameters `v1`, `v2` and `v3` are face vertices in counter-clockwise order.

```
v c1 c2 c3
...
f v1 v2 v3
...
```

Listing 8.1: Data file format.

In some sections, vertices may be formatted as `v c1 c2 c3 1`. These vertices represent directions of rays which travelled beyond a threshold (infinite length rays).

The data file name format is a 9 digits wide number followed by the ".obj" extension. Output log files are named using the same string of digits with "log.txt" appended.

```
tf[0] tf[1] tf[2] tf[3]
tf[4] tf[5] tf[6] tf[7]
tf[8] tf[9] tf[10] tf[11]
```

Listing 8.2: Log file format.

Log files contain the camera-to-world transformation matrix `tf`, not including the last row which will always have the same values. They are formatted as per 8.2.

All values are expected to be space separated.

In settings files, lines beginning with a `#` character are ignored.

8.2 Data collection

In this section, data collection utilities and their perceived accuracy are summarised.

8.2.1 Intel Realsense

Using the librealsense API [14], a capture utility written in C# was created. For a predefined amount of frames, it enables the user to capture data, which it first writes to a binary file and then processes into a series of OBJ files. The raw depth data is retrieved in the form of a 2D array of depth values. It is then transformed into a point cloud as follows:

The z coordinate is set to the depth value, which should represent the distance in millimeters from the plane in which the device lies.

Depth values below the recommended minimum of 0.3 m or above 1.2 m are discarded in order to maintain high precision. Some values may also be missing, as 3D scanning devices may struggle with reflective or translucent surfaces.

The following formulas can be found in *Transforming a depth map into a 3D point cloud* [21].

The x-axis coordinate is calculated as the distance from the center of the device, as per the following formula:

$$\nabla x = \frac{d_i}{tan(\gamma_i)} \tag{8.1}$$

wherein d_i is the depth value and γ_i is the angle between the sensor and the point. Assuming that the array columns divide the field of view evenly, γ_i is calculated from

$$\gamma_i = \alpha + \frac{c_i * \theta_h}{n_c} \tag{8.2}$$

wherein c_i is the column number, n_c is the total number of columns, and θ_h is the horizontal field of view of the camera. The value of the angle to the first column α is then

$$\alpha = \frac{\pi - \theta_h}{2} \tag{8.3}$$

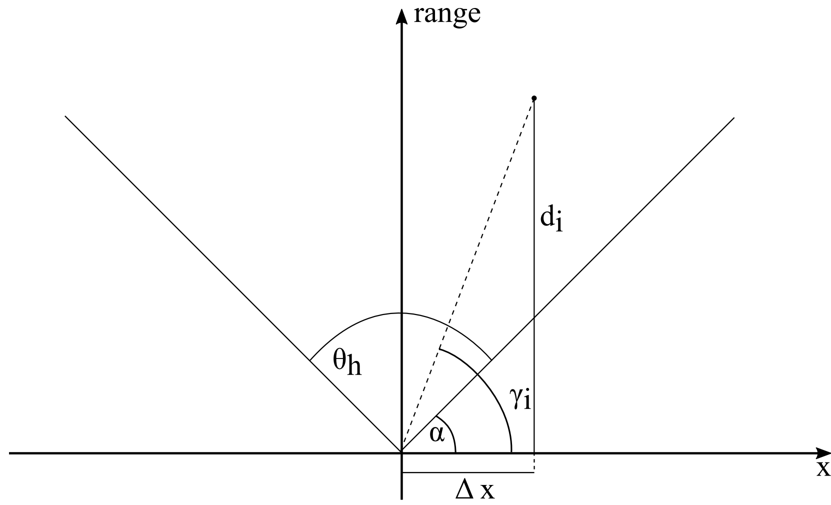

Figure 8.1: X coordinate retrieval.

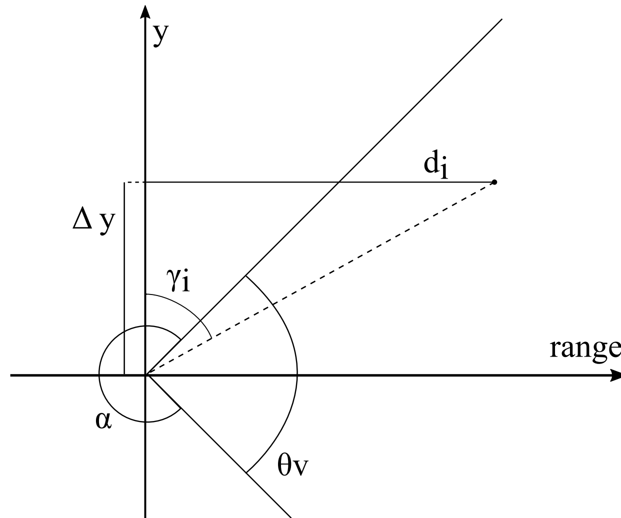

Figure 8.2: Y coordinate retrieval.

Similarly, the y-axis coordinate can be retrieved using the following formulas:

$$\nabla y = d_i * tan(\gamma_i) \tag{8.4}$$

$$\gamma_i = \alpha + \frac{r_i * \theta_v}{n_r} \tag{8.5}$$

$$\alpha = 2\pi - \frac{\theta_v}{2} \tag{8.6}$$

wherein θ_v is the vertical field of view, r_i is the current row number, and n_i is the total number of rows. Depth coordinate retrieval is illustrated in figures 8.1 and 8.2.

However, it must be noted that no specification as to the distribution of rays of the camera was found, and this process might introduce some error to the data, as the assumption that the rays have a constant angle offset is unsupported by any documentation. At the time of implementation (mid 2018), the C# wrapper for the API was largely undocumented and did not offer a clear way to retrieve a point cloud.

The data produced by the Intel sensor seems highly noisy, introducing wave-like patterns to flat areas farther than a metre away from the camera, as seen in figures 8.3 and 8.4. Regardless of errors likely introduced into the raw data, the existence of these patterns can be confirmed using the official Intel software for viewing range images.

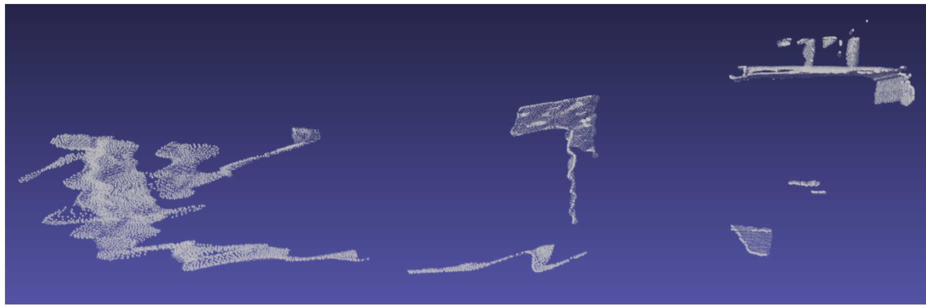

Figure 8.3: Example of wave patterns on flat surfaces captured at distance with Intel sensor.

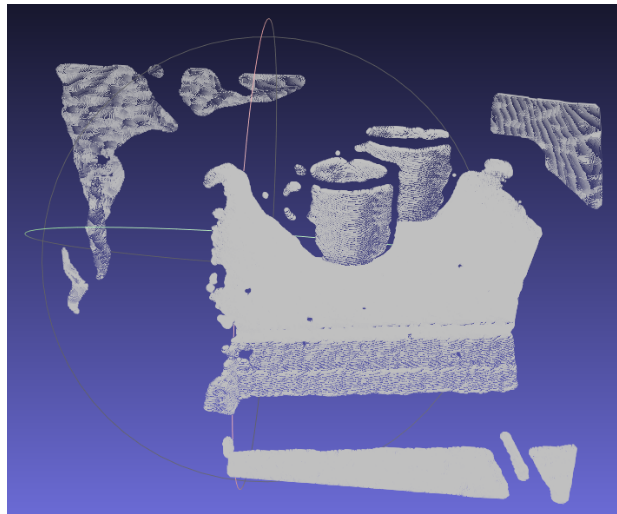

Figure 8.4: Front view of the scene. Two mugs on a small chair are pictured, with the wall of a room appearing in the background, distorted.

8.2.2 Kinect 2

A similar utility was created for the Kinect 2 in C#, using the same method of point cloud generation. The Kinect sensor appeared to provide data much more consistent with surfaces even at the distance of approx. 2-4 m, at which it had been tested. It still, however, struggles with flat surfaces to some extent, especially rounding wall corners. Example data can be seen in figures 8.5 and 8.6.

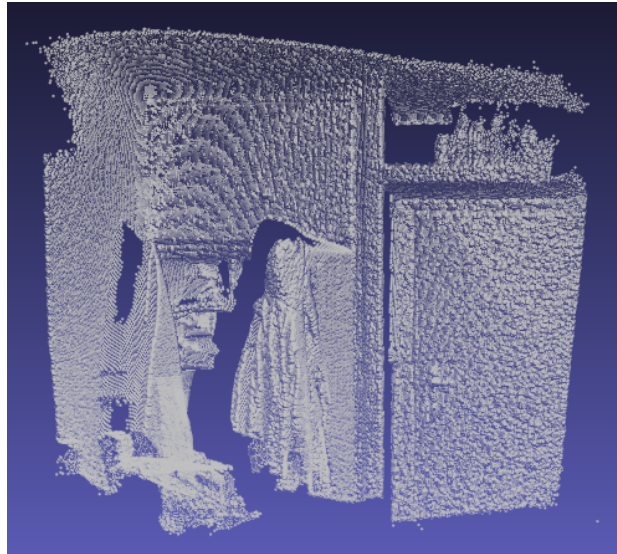

Figure 8.5: A room-scale scene featuring an open closet with jackets and a closed door.

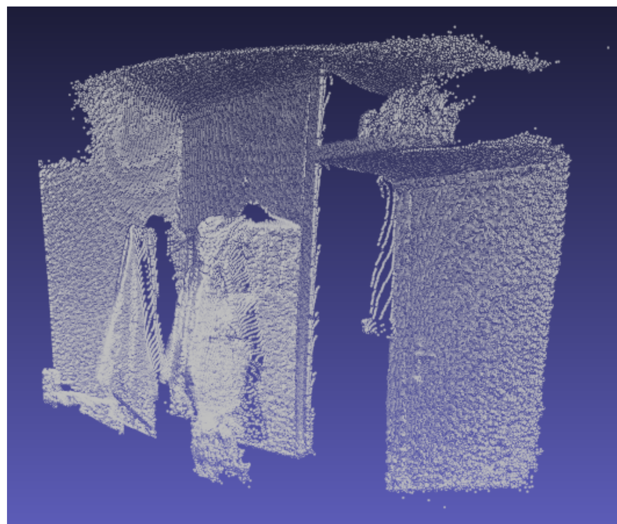

Figure 8.6: Side view of the scene.

8.2.3 Azure Kinect

Finally, tested last was the Azure Kinect at a late stage of the work due to its recent release date. A Microsoft example point cloud generation utility [19] was used to capture data. It appears to produce only marginally better results than its predecessor 8.7.

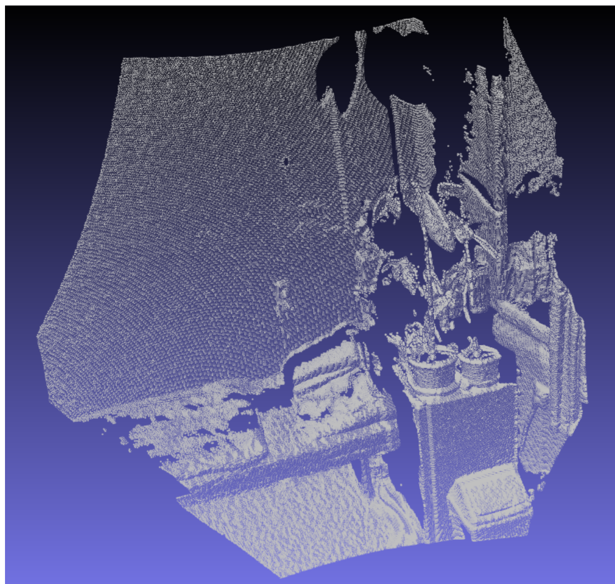

Figure 8.7: A range image captured with the Azure Kinect.

8.3 Registration

Two variants of the registration algorithm were tested - a local, and a global variant [31]. Global registration methods allow for a greater difference in the viewpoints from which data is retrieved. That is, they do not assume any close relation between any two point clouds, fitting them together as they deem best. Local registration, on the other hand, works with the assumption that consequent units of data have large overlap and come from very similar viewpoints. This is descriptive, for example, of depth vision applications in robotics, where data is available continuously during the movement of a robot which may need to construct a model of its workspace.

8.3.1 Global

An implementation of *On evaluating consensus in RANSAC surface registration* [13] [Hruda et al.] was initially used for global point cloud matching,

while only Intel Realsense data was available. While functional, the discrepancies in the data would cause some issues and further testing was executed using a localized variant.

8.3.2 Local

A local variant on the previous global implementation by Doc. Ing. Váša, Ph.D. was tested on room-scale data from the Intel Realsense depth sensor and the Kinect 2. It provided satisfactory results for Realsense data, but testing on data from large rooms revealed issues with matching data sets with large, flat surfaces, such as the walls of a room. As there can be very little other geometry present, the utility appears to achieve best overlap of the flat surfaces, disregarding smaller features. A sensor with a greater field of view could, perhaps, capture more features within a single frame and allow for better room-scale reconstruction.

8.3.3 Sparse Iterative Closest Point

An implementation [22] of the Sparse Iterative Closest Point [5] algorithm was also tested, providing comparable results, but requiring overall longer run times.

8.3.4 Conclusions on registration

Due to the severity of errors in input data and time needed to run the registration algorithm for any real-world data set, testing was focused on synthetic data sets, as both the errors in real world data and consequent registration could affect the results of the reconstruction and optimization algorithm, which should be designed with state of the art, high precision devices in mind, or future and cutting edge technology, as a failure to work with imprecise data does not indicate that suggested algorithms could not lead to better results as the technology develops.

8.4 Synthetic data generation and processing

The bulk of the implementation is focused on generating and processing synthetic data. The software developed for these purposes is described in the following sections.

8.4.1 Data generation

In order to generate synthetic data, first a sphere mesh must be generated. It is then used as input to the DataGenerator application, which mimics real world data capture, generating mesh intersection coordinates in camera space, using the vertex positions of the sphere model as viewpoints. Both of these applications were implemented in Java.

SphereGenerator

Originally implemented as a part of KIV/ZPG coursework, this application uses octahedron subdivision as seen in fig. 3.2 to generate sphere models. It starts with a manually specified octahedron and a user specified non-negative number of subdivisions.

The edges of the octahedron are oriented as per figure 8.8 and connected as to form faces. In each subdivision, a new vertex is created in the centre of each edge, projected to a sphere, and edges are reconnected to form new faces as per the scheme on the right. This results in two types of faces. Their type determines the order in which vertices are listed in faces in the output data and their treatment while subdividing the mesh further. This method of surface subdivision is also called dyadic splitting.

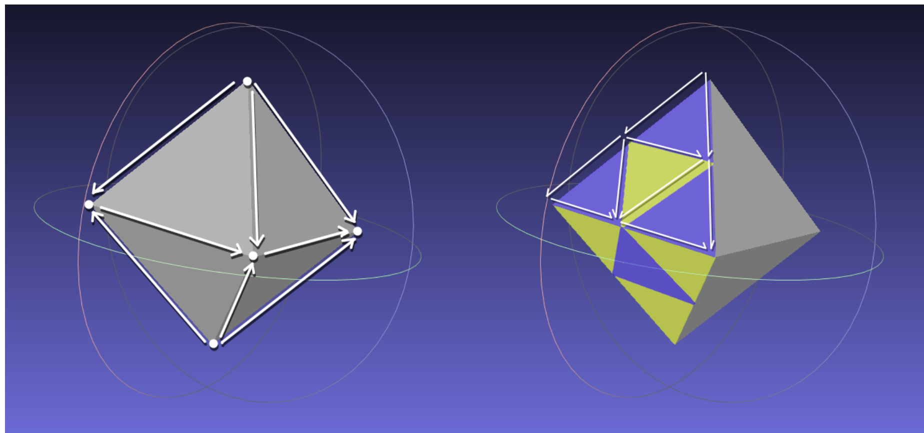

Figure 8.8: Edge direction and face subdivision scheme.

Finally, after the given number of subdivisions, a mesh in OBJ format is exported.

DataGenerator

This application uses an input sphere mesh and a 3D model to generate synthetic point cloud data. The user is free to define the vertical and horizontal

resolution of the virtual camera, its field of view, and parameters controlling octree subdivision. Specifically, the maximum number of faces per each octant and maximum tree depth. This data is specified in a settings file as per example 8.3. The maximum depth parameter has greater priority than the number of faces, therefore, if maximum depth is reached, leaf nodes will contain more faces then specified. A good balance of these parameters is necessary to control both the memory requirements and time spent tracing rays. Deeper trees will consume more memory, but fewer faces per node mean fewer intersections calculated per ray. However, the effectiveness also depends on the number of octant intersections to be calculated.

An octree is not the most efficient data structure available to solve a visibility problem, as in this implementation faces are inserted in all the leaf nodes they intersect, which may cause intersections to be calculated more times than necessary if a ray intersects multiple such nodes. References to faces are also duplicated, costing extra memory. Optimizing for ray tracing applications is an extensive field and some improvements on hierarchical data structures in ray tracing are discussed, for example, in *Accelerating Spatial Data Structures in Ray Tracing through Precomputed Line Space Visibility* [18].

```
# horizontal resolution
512
# vertical resolution
363
# field of view
45
# max octree faces per cell
10
# max octree depth
10
```

Listing 8.3: DataGenerator settings file format and default values.

If no directory named *output* in the .jar file location exists, it is created. Then a directory named as the current date and time is created in the *output* directory. Theoretically, this limits the application to being launched once per second.

Once this application loads the sphere and target models and generates an octree for the target model, it proceeds to generate and position cameras as per sections 3.3 and 3.4. The cameras are positioned along a path on the sphere, allowing for testing using local point cloud registration. The graph data structure and BFS implementation used to determine the path on the

sphere are taken from KIV/PPA2 materials by Doc. Ing. Váša, Ph.D. and coursework.

Finally, the camera rays are traced, and point cloud data is output, sorted by the camera ordering. Rays which did not hit the model are tagged as infinite, as discussed in section 8.1. Log files containing the camera-to-world transformation are output along with the data.

Some runtimes at default settings are listed in table 8.1. Columns *Octree*, *Cameras* and *Rays* refer to generation time. The long total run times can be attributed to time spent writing the data to files.

Sphere	**Target**	**Octree**	**Cameras**	**Rays**	**Tracing**	**Total**
sphere-1	bunny	0.017	0.006	10.825	10.445	87.600
	teapot	0.037	0.006	10.795	5.526	86.729
	dragon	1.314	0.006	7.979	9.929	106.034
	armadillo	0.938	0.006	8.016	9.641	88.274

Table 8.1: Data generation runtimes in seconds measured on i7-7700HQ CPU.

8.4.2 Registration

Registration utility launching and synthetic data registration are functions provided by PointCloudRegistration, a Java application. The program requires a path to input data, a path to the registration utility directory and utility type to be specified by the user. It also contains options for error simulation. The presence or absence of infinite rays in input data must also be specified. Last of all, it provides input data reduction settings. The settings file format is shown in example 8.4.

```
# input directory
path
# registration utility directory
path
# registration utility type (synth / merger / icp)
synth
# introduce errors
false
# max ray length difference
```

```
0
# max camera translation error
0.15
# max camera rotation error (degrees)
3
# has inf rays
true
# reduce
false
# octree node size
0.015
```

Listing 8.4: PointCloudRegistration settings file format and default values.

Synthetic

Synthetic registration is executed simply by reading the appropriate log file and applying the transformation to all vertices in a data file. This is done for each file in the specified input directory. Synthetic registration is the only one which respects error simulation settings.

In figure 8.9, the result of synthetic data generation and subsequent registration (with no errors) can be seen, along with the rays which have missed the model during generation and have been directed behind the camera and marked infinite, to later assist in clearing the grid.

Figure 8.9: Synthetic data after registration.

Error simulation

The reasons for simulating errors in synthetic data are discussed in section 3.7. There, two types of error are presented: intersection error, and rota-

tional/translational error. That is, either error coming from the process of data collection itself, or from the registration process. In modelling errors caused by sensor technology and imprecision, ray length is affected. In this application, this is simulated simply by randomizing a small error which is added to or subtracted from the ray length. In real devices, however, as can be seen with the Intel Realsense camera, such errors may be more structured, forming wave-like patterns, which depend on the distance from a surface. It is worth noting that an algorithm that deals well with one type of error may fail when facing another.

Errors in registration are more straightforward and can be simulated with a simple rotation and translation. In order to do this, six random values and directions are generated, one for rotation and translation in each axis. From these values, a rotation matrix can be calculated [35], as per matrices 8.7, 8.8 and 8.9 and extended to include a translation [12], as per matrix 8.10. A single rotation matrix can be obtained by multiplying the rotation matrices. While the resulting rotation depends on the order of this multiplication, it is inconsequential while introducing a random error.

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\ \alpha & sin\ \alpha \\ 0 & -sin\ \alpha & cos\ \alpha \end{bmatrix} \tag{8.7}$$

$$R_y(\beta) = \begin{bmatrix} cos\ \beta & 0 & -sin\ \beta \\ 0 & 1 & 0 \\ sin\ \beta & 0 & cos\ \beta \end{bmatrix} \tag{8.8}$$

$$R_z(\gamma) = \begin{bmatrix} cos\ \gamma & sin\ \gamma & 0 \\ -sin\ \gamma & cos\ \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.9}$$

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.10}$$

Angles α, β and γ represent x, y and z-axis rotations respectively, while the r_{ij} values represent values from the final rotation matrix, and t_x, t_y and t_z are the translation values.

8.4.3 Reconstruction

The reconstruction and optimization steps are implemented in PointCloudReconstruction, a Java application. Settings relating to the reconstruction are found in `settings.txt` and their default values are listed in example 8.5. The grid resolution controls the size of grid cells in the reconstruction. The `alpha`, `minAlpha`, `beta`, `gamma` and `delta` parameters are optimizer parameters further described in the appended user and programmer documentation. The step size array is expected to contain space separated values. Last of all, a parameter must be set to indicate whether the input contains infinite rays.

```
# input directory
path
# grid resolution
0.05
# optimizer beta parameter
1.0
# optimizer gamma parameter
1.0
# optimizer delta parameter
1.0
# optimizer alpha
2.0
# optimizer minAlpha
0.01
# optimizer step size array
# (tx, ty, tz, rx, ry, rz - degrees)
0.1 0.1 0.1 1 1 1
# optimizer iterations
10
# has inf rays
true
```

Listing 8.5: Reconstruction settings file format and default values.

The main concern with representing 3D space as a regular grid is the memory-intensity of this approach. Most real world data sets would need gigabytes of memory to be represented in high detail, especially when float values are required. Originally, byte type values were considered, but in order to run gradient descent based optimisation, the underlying function must be continuous. Therefore, a 3D array of float values is used. The

resolution of the grid also controls the size of the output model and affects the time and memory requirements of calculating the cost function.

Another concern is with the number of rays available to clear the grid. For too fine resolutions, rays will leave uncarved paths in the reconstruction, as can be seen in figure 8.10. This could be solved, for example, by also clearing paths between neighboring rays.

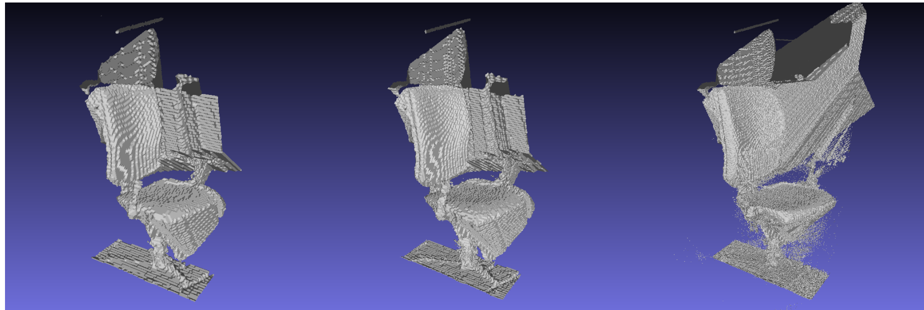

Figure 8.10: Uncarved paths in a reconstruction using a too fine grid. Grid resolutions left to right: 10, 7.5 and 5 units.

Excess geometry can also cause a heap overflow while inserting faces in the octree data structure, especially as it may use several references for some. This can also happen if an optimization step generates too many new mesh fragments.

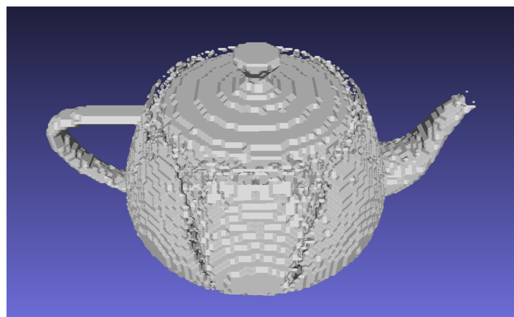

Figure 8.11: Thin surfaces may be carved through during reconstruction due to rays falling into the same cell from two directions.

In 8.11 it can be seen that the reconstruction process may also struggle with reconstructing data where thin surfaces are present. This could likely be solved during grid cell value optimization, by introducing a wider range of values, or by taking into consideration the orientation of rays during reconstruction.

An example of a good reconstruction outcome can be seen in figure 8.12

Figure 8.12: A good reconstruction outcome.

8.4.4 Optimization

In the final part of the implementation, an initial reconstruction is generated and the cost function is calculated. Then, the position of each camera is optimised, running either for a specified number of iterations or until the energy stops decreasing. Grid value optimisation is discussed further below.

The file `rt_settings.txt` (with default values as seen in example 8.6) contains settings relating to ray tracing during optimization. The field of view and horizontal and vertical resolution relate to the hit/miss array used to approximate the silhouette of the reconstruction. Higher resolutions greatly impact the time necessary to calculate the cost function, but a too low resolution may miss important features. Octree depth and maximum number of faces per node are also set in this file.

```
# horizontal resolution
400
# vertical resolution
200
# field of view
45
# max octree faces per cell
5
# max octree depth
10
```

Listing 8.6: Camera settings file format and default values.

Cost function calculation

The cost function is calculated as a contribution of each ray in every data file. Calculating the cost function requires an octree and a k-d tree to be generated from the current reconstruction data. Then, once per data file, the silhouette of the reconstruction is captured as an image of hit/miss values. Finally, for each ray it must be determined whether it has an intersection with the model, and where. Only then the category of a ray is known and the energy contribution can be calculated, by nearest vertex lookup, distance from expected intersection, or angle to the nearest silhouette edge. The amount and complexity of these operations result in unfavorable run times, as can be seen in table 8.2. This is a function which must be calculated at least 7 times per a registration optimization step, and for all relevant grid values during grid optimization.

In fact, data from 8.2 reveals that a disproportionate amount of time is spent searching for nearest vertices in the reconstruction. This suggests that a more efficient implementation of the k-d tree would go a long way towards speeding up the algorithm. Calculating true distance from the mesh could also be considered. Additionally, the much higher total times of calculation can only be attributed to the additional step of finding ray-reconstruction intersections, and to input reading operations.

Data set [res.]	**Rays 1**		**Rays 2**		**Rays 3**		**Total**
	count	time	count	time	count	time	
armadillo [0.025]	235	0.01	6199	5.30	67609	0.19	9.155
bunny [0.0025]	239	0.01	23584	5.34	103698	0.29	12.375
dragon [0.2]	221	0.01	6393	5.08	59338	0.19	11.244
teapot [0.05]	266	0.01	2876	11.01	52340	0.37	20.558
real [10.0]	0	0.00	540	14.89	437661	1.19	46.01

Table 8.2: Cost function calculation runtimes in seconds measured on i7-7700HQ CPU. Grid resolutions are listed in the brackets. Time 1, 2 and 3 refer to ray types as per sections 7.2.2, 7.2.3, 7.2.4 respectively.

Camera position

Optimizing the transformation of a point cloud means adjusting the transformation matrix by a certain translation and rotation in the x, y and z-axis. For each of these six values, a step size can be specified by the user. Then, a point cloud is transformed for each of these values individually, and the cost function is recalculated for each case. Next, the gradient is calculated.

That is, given a vector describing the current transformation state $\vec{t}$, partial derivatives for each of its elements t_i are approximated. A change vector $\vec{x}$ is added to the transformation state vector and energy is recalculated. Vector $\vec{x}$ has zeroes in each element except element x_i which is set to a small value relative to the input data, representing a step size. This can be seen from formula 8.11.

$$\frac{\partial C}{\partial t_i} \simeq \frac{C(\vec{t}+\vec{x}) - C(\vec{t})}{x_i} \tag{8.11}$$

Partial derivatives are obtained for all translation and rotation elements of the transformation, as per 8.12.

$$\nabla C = [\frac{\partial C}{\partial t_{tx}}, \frac{\partial C}{\partial t_{ty}}, \frac{\partial C}{\partial t_{tz}}, \frac{\partial C}{\partial t_{rx}}, \frac{\partial C}{\partial t_{ry}}, \frac{\partial C}{\partial t_{rz}}] \tag{8.12}$$

A transformation is then constructed by taking a δ multiple of the gradient, which serves to speed up the convergence, and applying it to the data. The cost function is recalculated, after which two situations may arise - either the cost has decreased and the new transformation is applied permanently, and a new iteration is started, or the transformation is discarded, δ is halved, and the last step is recalculated. This process is repeated until the energy decreases, or alpha is smaller than some threshold ϵ.

The transformations in this process can be constructed the same way as the error transformations in section 8.4.2, but it should be noted that while during the initial calculations, rotations are executed individually, their order matters while combining them. In this process, it is assumed that the result will not be far from the direction of steepest cost function descent, as the rotations are relatively small.

Grid values

Due to the extreme time complexity of calculating the cost function, grid value optimisation was not attempted in this work. However, further optimisation is not inconceivable. If a bulk of the calculation did not need to be repeated while transforming one data set, ray contributions could be deducted from the total cost and recalculated. This would significantly reduce

running times. In order to implement such a method, each cell would need to be associated with the rays it had been affected by. Changes in a single cell should not affect the entire model significantly, but contributions of all rays passing through it would need to be recalculated.

Additionally, many different situations could, in theory, arise, and would need to be analysed. Changing certain grid values could introduce new components into the reconstruction, or otherwise affect its shape, causing rays to be blocked from, or to pass through to, distant areas of the reconstruction, which would also be affected. Finding the minimum amount of operations necessary to carry out these changes would be a very complex task.

8.4.5 Results

The implementation was tested on a small synthetic data set, and a larger real world scene.

Synthetic data

In figure 8.15, one of eighteen transformations had been corrupted, as can be seen on the left from the tilted side of the model. The result features both a correction of the silhouette in the affected area and a decrease in energy. The cost function developments can be seen in graphs 8.13, in which refused energies are displayed along with accepted values, and 8.14, where only accepted values are shown.

An obvious drawback is the time needed to generate these results, that is, over 1 hour and 20 minutes even for a very small data set of 18 data files with 512x363 rays each, in a low resolution grid. The small amount of rays and viewpoints used does not even permit a high resolution grid to be used, as this few rays would not be able to clear it. Even for these small synthetic examples, the application uses up to nearly 2 GB of memory.

Calculations with higher resolution settings typically end in a heap overflow while faces are inserted into the octree, which is used during the cost function calculation. The higher number of rays required to clear such grids also causes much slower cost calculation. Together, these properties suggest that the current design of the optimization and reconstruction algorithms cannot also support grid cell value optimization, due to the sheer number of times the cost would need to be calculated. However, they also define the data structures to be optimized and approaches to be considered in any future work.

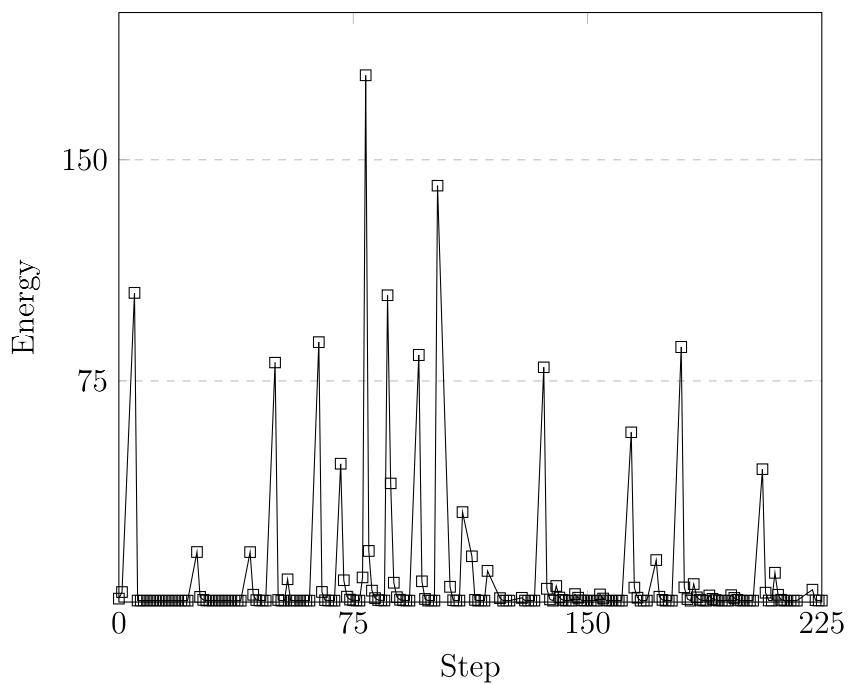

Figure 8.13: Synthetic data energy over run time
(includes states which were refused)

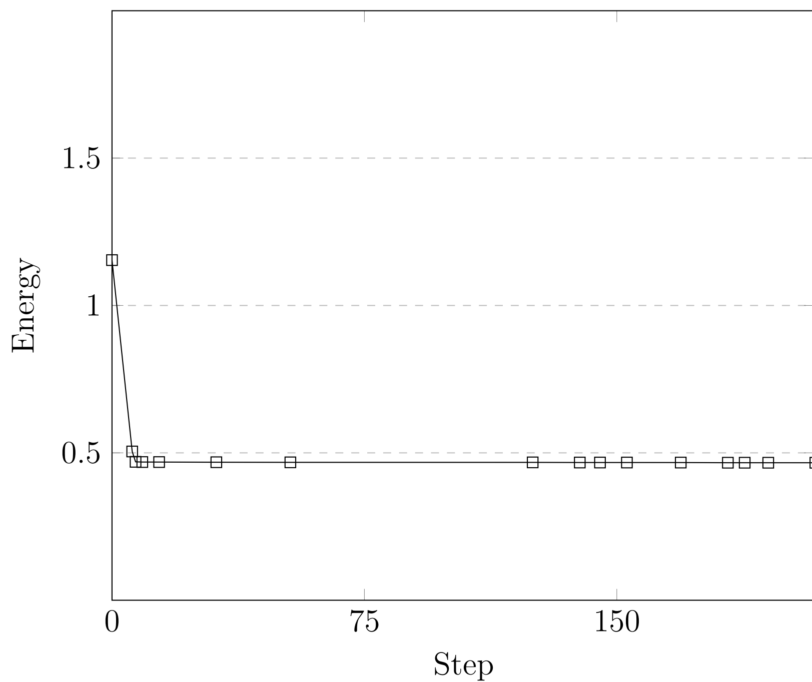

Figure 8.14: Synthetic data energy over run time
(only accepted states)

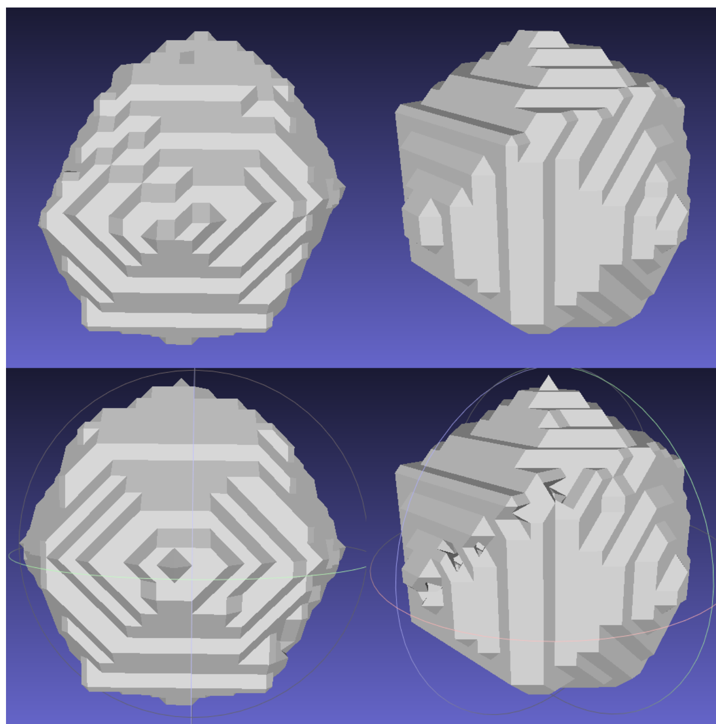

Figure 8.15: Synthetic data optimization with start energy 1.1539 (top) and end energy 0.4664 (bottom). Time: 1h 21min 10s.

Real data

A real data set produced using the Azure Kinect by Doc. Ing. Váša, Ph.D., including manually generated registration data, was used in testing. It can be seen in figure 8.16.

The data has been subject to optimization at three grid resolutions. While the overall energy does somewhat decrease for all of the examples, the visual effects are very mild. This could be attributed to both testing on an extremely small data set and low iteration count. Very conservative optimization parameters were also set, as finding parameters which would result in reasonable transformations is in itself a time consuming task of trial and error. Additionally, the energy of the reconstruction may have already been near a local minimum.

The high detail example in figure 8.19, while still quite small and using only four depth images, can use over 3.5 GB of memory, which may already exceed memory limits for some Java Virtual Machine heap sizes.

Most notably, the very low detail result in figure 8.17 does in fact approach the higher resolution geometry in at least one area, by forming uncarved paths in the same areas as the next higher resolution example seen

in figure 8.18. The energy developments for the low resolution variant can be seen in graphs 8.20 and 8.21.

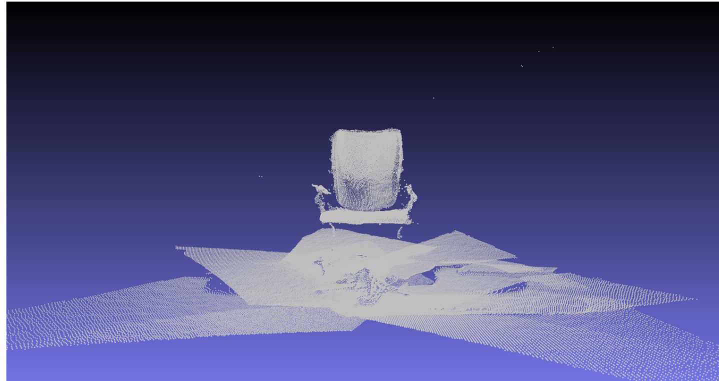

Figure 8.16: Registered real world data (4 depth images).

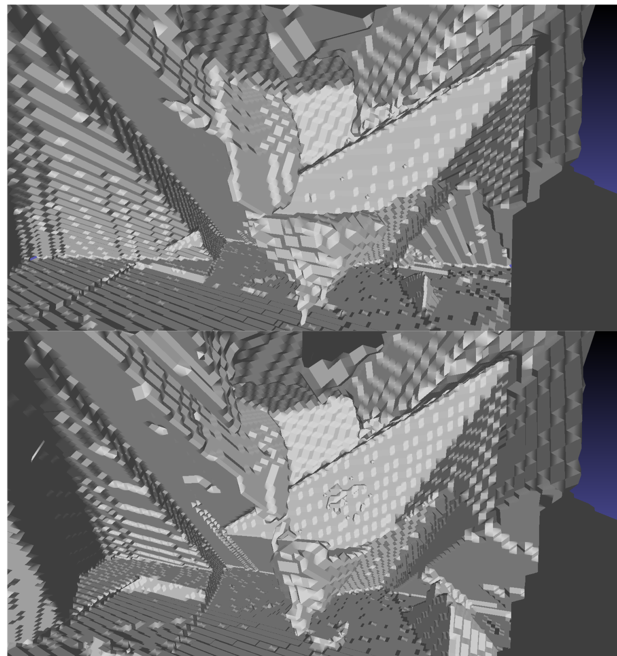

Figure 8.17: Real data optimization with start energy 3588.96 and end energy 3039.22 - very low detail [grid res. 25]. Time: 25 min 1 s.

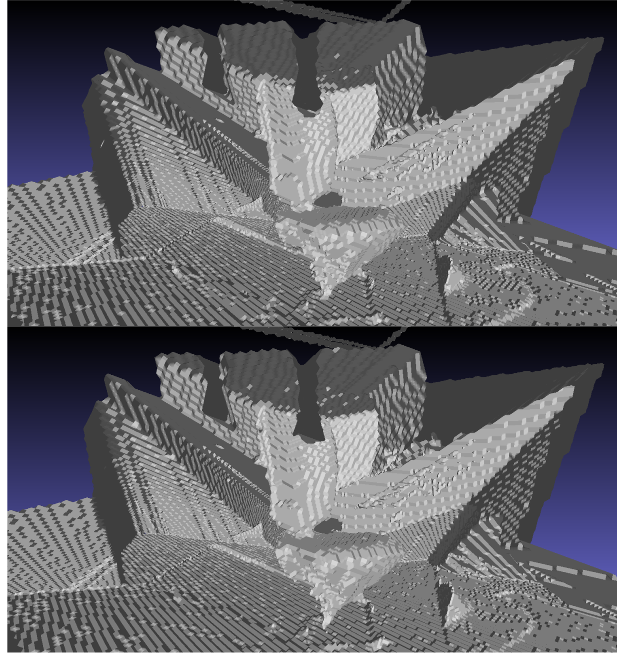

Figure 8.18: Real data optimization with start energy 1676.60 and end energy 1636.87 - low detail [grid res. 15]. Time: 19 min 50 s.

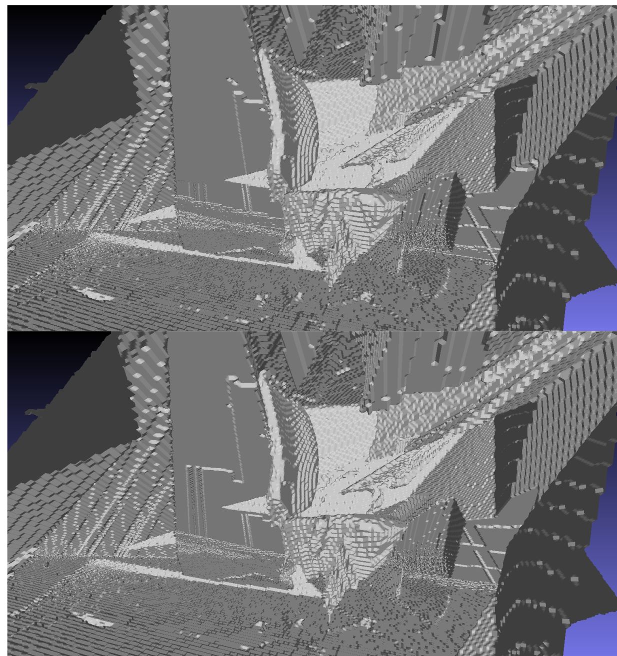

Figure 8.19: Real data optimization with start energy 2317.73 and end energy 2316.33 - higher detail [grid res. 10]. Time: 32 min 16 s.

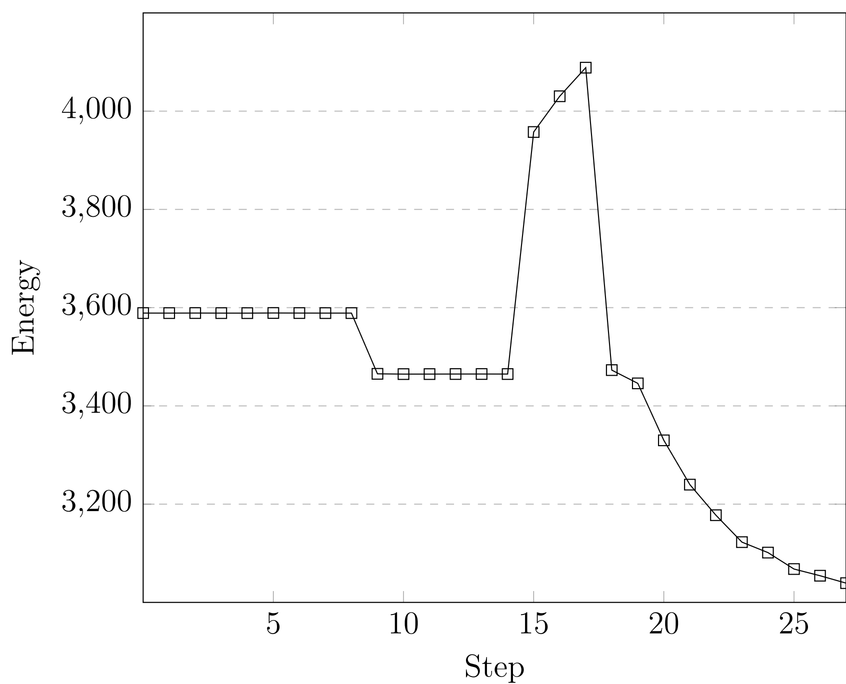

Figure 8.20: Low resolution real data energy over run time (includes states which were refused)

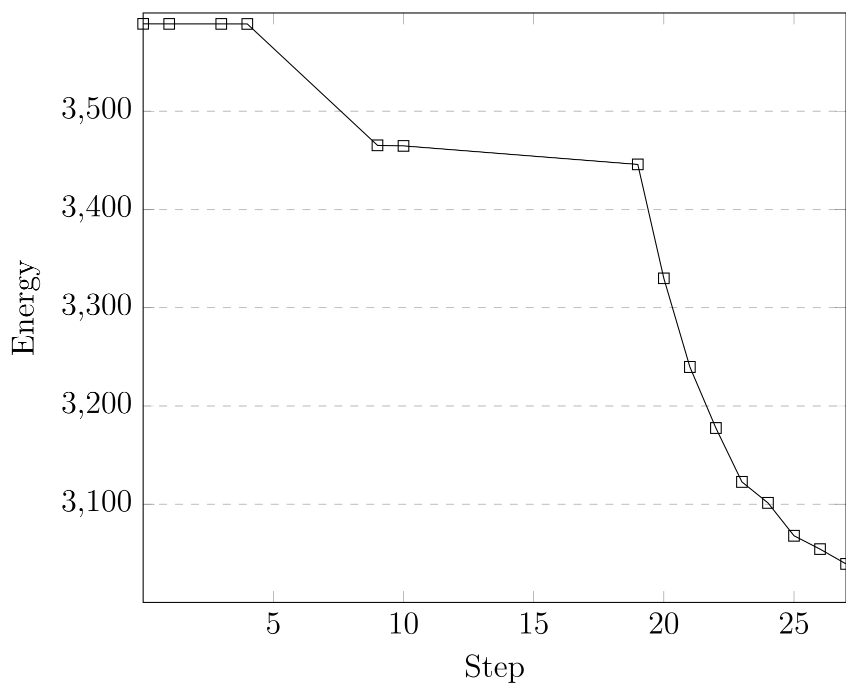

Figure 8.21: Low resolution real data energy over run time (only accepted states)

9 Further directions

In future applications, three types of issues would need to be resolved - managing memory, improving time complexity and enabling grid optimization.

Memory issues mostly arose from the use of an octree as a somewhat naive ray tracing optimization. The duplication of references and inefficient grouping of geometry, while improving time complexity, led to inefficient memory usage and became a bottleneck in testing the optimization process with realistic data sets. Therefore, a more efficient acceleration structure should be used, such as a BVH.

Some issues with memory management could be resolved simply by using devices with more memory available. Then, point cloud data up to a certain size could be kept in the memory and not loaded repeatedly.

An analysis of the cost function calculation had revealed that much of the total time is spent looking for the nearest point in the reconstruction to a ray intersection . Implementing a more efficient k-d tree as discussed in [33] would provide some improvement to this approach. Calculating true distance to the reconstruction would, however, also improve the accuracy of this step and should be considered.

The angle-to-silhouette calculation should also be reconsidered, as it becomes expensive for higher resolutions and loses much of its accuracy if a low resolution is selected.

In order to make optimizing the reconstruction itself possible, the cost function must be possible to calculate on per-ray basis, so that contributions could be deducted and recalculated only for affected rays, rather then the entire data set. The underlying model would therefore have to provide an efficient method of determining which parts of the reconstruction are being affected when a grid cell value changes. Defining a way to prioritize cells for optimization instead of testing the entire grid could also help focus changes where they matter the most and speed up the entire iteration process.

Finally, others have also considered the orientation of rays during reconstruction, as thin surfaces will face issues with being erased when observed from opposing viewpoints. This work would also benefit from their approach.

10 Conclusion

The goal of this thesis has been to explore the possibility of using empty space and device position data in the process of 3D surface reconstruction. This goal was fulfilled in several steps.

First, methods of capturing real world point cloud data were tested, using the Intel Realsense D415, Kinect 2 for Windows and Azure Kinect devices. Additionally, an application for point cloud data synthesis was developed, which provides precise registration data and is also compatible with local registration methods. A way to introduce user specified amounts of error into the synthetic registration outcome was also included.

Next, methods of representing data about space occupancy were designed, supporting 3D surface extraction, with optimization by gradient descent in mind. Then, the designed data structures and algorithms were implemented.

Testing the applications has had limited success due to time and memory complexity of the optimization steps. However, registration optimization has had the expected effect on small data sets. Problematic areas of the implementation were analyzed and further work required in order to achieve reconstruction optimization was discussed.

Bibliography

[1] *A brief description of the fast marching method* [online]. 2013. [cit. 2020/07/14]. Fast marching. Available at: http://ahay.org/RSF/book/sep/fmeiko/paper_html/node2.html.

[2] *A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.)* [online]. Scratchapixel 2.0. [cit. 2020/07/15]. Ray-box intersection. Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection.

[3] Arrighi, P.-A. *3D scanning technologies and the 3D scanning process* [online]. Aniwaa, 2020. [cit. 2020/07/13]. 3D Scanning. Available at: https://www.aniwaa.com/guide/3d-scanners/3d-scanning-technologies-and-the-3d-scanning-process/.

[4] *Azure Kinect DK depth camera* [online]. Microsoft, 2019. [cit. 2020/07/13]. Azure Kinect DK documentation. Available at: https://docs.microsoft.com/en-us/azure/kinect-dk/depth-camera.

[5] Bouaziz, S. – Tagliasacchi, A. – Pauly, M. Sparse Iterative Closest Point. *Computer Graphics Forum*. 2013, 32, s. 113–123.

[6] *Introduction to Acceleration Structures* [online]. Scratchapixel 2.0. [cit. 2020/07/13]. BVH data structure. Available at: https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part1.

[7] Bourke, P. *Polygonising a scalar field* [online]. Paul Bourke, 1994. [cit. 2020/07/15]. Available at: http://paulbourke.net/geometry/polygonise/.

[8] Bourke, P. *Object Files (.obj)* [online]. [cit. 2020/07/15]. Available at: http://paulbourke.net/dataformats/obj/.

[9] Bresenham, J. E. Algorithm for computer control of a digital plotter. *IBM Systems Journal*. 1965, 4, 1, s. 25–30.

[10] Carfagni, M. et al. Metrological and Critical Characterization of the Intel D415 Stereo Depth Camera. *Sensors*. 01 2019, 19, s. 489. doi: 10.3390/s19030489.

[11] Curless, B. – Levoy, M. A Volumetric Method for Building Complex Models from Range Images. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’96, s. 303–312, New York, NY, USA, 1996. Association for Computing Machinery. doi: 10.1145/237170.237269. Available at: https://doi.org/10.1145/237170.237269. ISBN 0897917464.

[12] *Geometric Transformations* [online]. [cit. 2020/07/17]. Available at: https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/geometry/geo-tran.html.

[13] Hruda, L. – Dvořák, J. – Váša, L. On evaluating consensus in RANSAC surface registration. *Computer Graphics Forum*. 08 2019, 38, s. 175–186. doi: https://doi.org/10.1111/cgf.13798.

[14] *IntelRealSense / librealsense* [online]. GitHub. [cit. 2020/07/17]. Available at: https://github.com/IntelRealSense/librealsense.

[15] *Intel® RealSenseTM Camera 400 Series (DS5) Product Family Datasheet* [online]. Intel, 2019. [cit. 2020/07/13]. Intel RealSense Datasheet. Available at: https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/Intel-RealSense-D400-Series-Datasheet.pdf.

[16] James, M. *Quadtrees and Octrees* [online]. IProgrammer, 2018. [cit. 2020/07/13]. Octree data structure. Available at: https://www.i-programmer.info/programming/theory/1679-quadtrees-and-octrees.html?start=1.

[17] Kazhdan, M. et al. Poisson Surface Reconstruction with Envelope Constraints. *Computer Graphics Forum (Proc. Symposium on Geometry Processing)*. July 2020, 39, 5.

[18] Keul, K. – Müller, S. – Lemke, P. Accelerating Spatial Data Structures in Ray Tracing through Precomputed Line Space Visibility. 06 2016.

[19] Microsoft. *Azure Kinect Fastpointcloud Example* [online]. GitHub. [cit. 2020/07/16]. Available at: https://github.com/microsoft/Azure-Kinect-Sensor-SDK/tree/develop/examples/fastpointcloud.

[20] Nguyen, T. *Nearest Neighbor Search* [online]. [cit. 2020/07/15]. Available at: http://andrewd.ces.clemson.edu/courses/cpsc805/references/nearest_search.pdf.

[21] omaralejandrorodriguez. *Transforming a depth map into a 3D point cloud* [online]. 2017. [cit. 2020/07/17]. Available at:

https://elcharolin.wordpress.com/2017/09/06/transforming-a-depth-map-into-a-3d-point-cloud/.

[22] PALANGLOIS. *Sparse Iterative Closest Point Algorithm* [online]. 2018. [cit. 2020/07/14]. SICP registration utility. Available at: https://github.com/palanglois/icpSparse.

[23] PANDEY, P. *Understanding the Mathematics behind Gradient Descent* [online]. Medium, 2019. [cit. 2020/07/15]. Available at: https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e.

[24] *Point Cloud Registration Overview* [online]. The MathWorks. [cit. 2020/07/13]. Registration Methods. Available at: https://www.mathworks.com/help/vision/ug/point-cloud-registration-workflow.html.

[25] *Ray-Tracing: Generating Camera Rays* [online]. Scratchapixel 2.0. [cit. 2020/07/13]. Generating camera rays. Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays.

[26] *Ray Tracing: Rendering a Triangle* [online]. Scratchapixel 2.0. [cit. 2020/07/15]. Ray-triangle intersection. Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection.

[27] TABOGA, M. *Indicator functions* [online]. Statlect, 2010. [cit. 2020/07/16]. Available at: https://www.statlect.com/fundamentals-of-probability/indicator-functions.

[28] TESKOVÁ, L. *Lineární algebra.* University of West Bohemia, Faculty of Applied Sciences, 2010. ISBN 978-80-7043-966-1.

[29] THORP-LANCASTER, D. *Azure Kinect developer kit hits general availability, preorders begin shipping* [online]. Future US, 2019. [cit. 2020/07/13]. Azure Kinect release date. Available at: https://www.windowscentral.com/azure-kinect-developer-kit-hits-general-availability.

[30] TILLMAN, M. *What is a ToF camera? The Time-of-flight sensor explained* [online]. Pocket-lint, 2020. [cit. 2020/07/13]. Time-of-flight technology. Available at: https://www.pocket-lint.com/phones/news/147024-what-is-a-time-of-flight-camera-and-which-phones-have-it.

[31] Varga, D. et al. *On Fast Point Cloud Matching with Key Points and Parameter Tuning*, s. 498–511. 02 2020. doi: 10.1007/978-3-030-41404-7_35. ISBN 978-3-030-41403-0.

[32] *View Frustum Culling* [online]. Lighthouse3D. [cit. 2020/07/16]. Available at: `http://www.lighthouse3d.com/tutorials/view-frustum-culling/`.

[33] Wald, I. – Havran, V. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, s. 61–69, September 2006.

[34] Weisstein, E. W. *Hamiltonian Path* [online]. MathWorld–A Wolfram Web Resource. [cit. 2020/07/14]. Available at: `https://mathworld.wolfram.com/HamiltonianPath.html`.

[35] Weisstein, E. W. *Rotation Matrix* [online]. From MathWorld–A Wolfram Web Resource. [cit. 2020/07/17]. Available at: `https://mathworld.wolfram.com/RotationMatrix.html`.

[36] Zhang, H. – Hall-Holt, O. – Kaufman, A. Range image registration via probability field. In *Proceedings Computer Graphics International, 2004.*, s. 546–552, 2004.

A User documentation

A.1 IntelRealsenseRecorder

The IntelRealsenseRecorder is a data capture utility for the D415 camera. The application is launched using the command

```
IntelRealsenseRecorder.exe [frame count]
```

with one optional parameter specifying the number of frames to be recorded. The default value is 100 frames, but numbers from 1 to 99,999 are accepted.

The application proceeds to record the given number of frames to a binary file. Upon completion, it outputs elapsed time and begins to transform the data to point cloud data files, which it then exports to the *output* directory in OBJ format.

The program requires *Intel.Realsense.dll* and *realsense2.dll* libraries to be present in the same directory as the executable.

A.2 KinectRecorder

The KinectRecorder is a data capture utility for the Kinect 2 for Windows. It is launched using the command

```
KinectRecorder.exe [time]
```

with an optional parameter specifying the recording time. The default value is 5000 ms, but times from 1 to 60,000 ms are accepted (with no guarantee of capturing at least one frame).

Upon start, the application will attempt to open the depth stream. When successful, it outputs the message "Recording...". If no Kinect 2 device is connected, it will stay in the "Opening..." phase.

Upon opening the depth stream, the application records depth images to a binary file, until the time runs out. When complete, it begins to transform the data to point cloud data files, which it then exports to the *output* directory in OBJ format.

A.3 AzureRecorder

The AzureRecorder is an MIT licensed Microsoft example point cloud capture utility [19] for the Azure Kinect. It is launched with no parameters and

will produce a single depth image in OBJ format.

A.4 SphereGenerator

Launch SphereGenerator.jar as

```
java -jar SphereGenerator.jar <non-negative integer>
```

with one non-negative integer parameter, signifying the number of subdivisions to be applied. No upper limit is given, but values over 10 are not recommended.

A.5 DataGenerator

Launch DataGenerator.jar as

```
java -jar DataGenerator.jar <sphere path> <mesh path>
```

with the sphere path being a path to a viewpoint sphere model and the mesh path being a path to the observed target model. On first run, and output directory will be created. All subsequent launches will write data to directories under the output directory, named as the time at launch. PointCloudRegistration.jar expects its input path to be a path to one of these directories for synthetic registration.

A.6 PointCloudRegistration

Launch PointCloudRegistration.jar as

```
java -jar PointCloudRegistration.jar
```

with no parameters. All required settings can be found in *settings.txt* in the same directory.

The following must be set:

- input directory path,
- registration utility directory (for merger - local or global, and for icpSparse, utilities),
- utility type - *synth*, *merger* or *icp*,
- error generation toggle (true/false),

- maximum ray length error (by default set to zero),
- maximum camera translation error,
- maximum camera rotation error,
- infinite ray toggle (true if infinite rays are present in the input data, false otherwise),
- reduce toggle (to reduce input data),
- and octree leaf node size, which controls the input reduction (effectively a limit of one vertex per each cell of the given size).

A.7 PointCloudReconstruction

Launch PointCloudReconstruction.jar as

```
java -jar PointCloudReconstruction.jar
```

with no parameters. All required settings can be found in the settings.txt and rt_settings.txt files in the same directory.

The following must be set in the *settings.txt* file:

- input directory path,
- grid resolution for the marching cubes algorithm,
- optimizer beta, gamma and delta parameters, which represent ray group B, C and D energy multipliers as per ray types below,
- optimizer alpha and minAlpha parameters, which represents the gradient multiplier,
- optimizer step sizes (translation and rotation steps),
- number of optimizer iterations
- and a toggle for infinite ray presence (true if present, false otherwise).

Rays are grouped as follows:

- type D - not infinite, with intersections
- type C - infinite, with intersections

- type B - not infinite, no intersections
- type A - infinite, no intersections

The *rt_settings.txt* file expects the following parameters:

- vertical and horizontal resolution of the hit/miss images to be created during cost function calculation,
- camera field of view
- and octree faces-per-node and depth settings.

Appropriate presets are included in the example files.

B Programmer documentation

B.1 IntelRealsenseRecorder

The application consists of a single class, *Program.cs*.

Upon launch, input parameters are parsed using the *ProcessArguments()* method. One optional parameter is expected - the number of frames to be captured. If it does not lie in the permitted interval of <1, 99999>, a default value of 100 is used.

Next, the camera is accessed and frames are saved to a binary file by the *WriteDepthFrames()* method. The program requires *Intel.Realsense.dll* and *realsense2.dll* libraries in order to access these functions. It uses a mid-2018 version of the librealsense API. During development, the program has been known to detect a present device even while it is unplugged - this may be an old bug, however, recent versions of the API have changed and cannot be used with this code. This error is accounted for in the code and should not cause issues.

In *Depth2Obj()* the binary data file is parsed and point cloud data is calculated as per the formulas discussed in section 8.2.1. It is then output in a simple OBJ format, as shown in section 8.1.

B.2 KinectRecorder

The application consists of a single class, *Program.cs*.

Upon launch, input parameters are parsed using the *ProcessArguments()* method. One optional parameter is expected - the time in milliseconds to spend capturing depth data. If it does not lie in the permitted interval of <1, 60000>, a default value of 5000 is used.

Within the main method, a depth stream is opened, and the incoming frame data is written to a binary file using the *ProcessFrame()* method. Then, in *Depth2Obj()* the binary data file is processed the same way as in the IntelRealsenseRecorder.

B.3 AzureRecorder

The enclosed code mostly consists of the Microsoft example code [19] in C++, with an added OBJ format output method, in order to maintain

compatibility with the rest of this work.

B.4 SphereGenerator

SphereGenerator is a simple Java program consisting of *Edge*, *Face*, *Mesh*, *Vertex* and *Main* classes.

The *Edge* class provides a method for splitting the edge and shifting the new point to a given radius distance from the origin. The *Vertex* and *Face* classes simply contain coordinates and indices respectively of geometry. The *Face* class has a type parameter, specifying triangle type as per section 8.4.1 on sphere generation.

The *Mesh* class contains lists of faces, edges and vertices.

The *Main* class provides the algorithm itself, splitting the edges in each turn, adding them to new lists, and replacing the lists in a *Mesh* object.

Once the calculation is finished, an OBJ file of a sphere is exported.

B.5 DataGenerator

DataGenerator is a Java program. It uses JAMA, a linear algebra package for Java, and consists of the following packages:

- algorithms
- commons
- datastruct
- geometry
- raycasting

Algorithms is a package containing a Möller-Trumbore algorithm implementation (class *MollerTrumbore*) as per Scratchapixel 2.0 [26], and a sphere ordering algorithm (class *SphereOrder*).

The *commons* package contains the *Main* class and a *Constants* class.

The *datastruct* package contains data structures used in this application. That is, a *Graph* implementation as per KIV/PPA2 providing the k-neighborhood search algorithm, *Octree*, the ray tracing acceleration structure, and a *Queue*.

In *geometry*, basic objects relating to mesh geometry are defined in the *BoundingBox*, *Face*, *Mesh*, *PointCloud* and *Vertex* classes.

Under *raycasting*, the class *Settings* reads, writes and provides the camera settings to other classes. The *Ray* and a *CoordinateSystem* classes are also defined here, the latter providing the camera-to-world matrix to output along with the point cloud data. Finally, the *Camera* class provides a method for generating cameras at viewpoints specified by the sphere file, and the *trace()* method used is to cast the rays and calculate their intersections with the target model, using the Möller-Trumbore algorithm. It also contains a method, which transforms these intersections to camera space using the inverse of the camera-to-world matrix and outputs them as a *PointCloud*.

B.6 PointCloudRegistration

PointCloudRegistration is a Java application for launching registration utilities and synthetic data registration. It also features error simulation and data reduction functions.

The application consists of the following packages:

- commons
- geometry
- mergers

The *commons* package contains the *Constants* class, the *IOUtil* class for reading and writing OBJ files, the *Main* class, *MathUtil* for vertex transformation and matrix multiplication and the *Settings* class which provides user-defined values to the rest of the application and takes care of reading and writing settings files.

The geometry contains a *Vertex* class, and the *Octree* class, which is used for data reduction.

In mergers, an abstract *RegUtil* class is defined, which features a *run()* method. The *IcpSparse*, *Match* and *SyntheticUtil* classes then extend the *RegUtil* class and provide implementations of its run method, which uses their respective utilities to register multi-file data sets.

Only *IcpSparse* and *Match* classes accept reduction parameters. In order to reduce the input data, the option must be set to "true" in the settings file. Octree node size must also be set appropriately.

The *Match* class launches the global and local utilities. *Match* and *IcpSparse* both use the same method of file registration:

Given files A, B and C, first, A and B will be registered. Then, B and C will. This should provide, ideally, enough information to fit all files to the

first file. In this case, by first fitting C to B and then fitting C and B both to A. This is achieved by multiplying the retrieved transformation matrices and only then transforming the data itself.

B.7 PointCloudReconstruction

PointCloudReconstruction is a Java application which uses the JAMA linear algebra library. It consists of the following packages:

- algorithms
- commons
- datastruct
- geometry
- optimization
- raycasting
- reconstruction

The *algorithms* package features the Möller-Trumbore algorithm and a *NearestMiss* algorithm which searches the hit/miss image during cost function calculation.

The *commons* package, similarly to the previous application, contains the *Constants*, *IOUtil*, *Main*, *MathUtil* and *Settings* classes which provide the same functionality.

The *datastruct* package contains the *KDTree* and *Octree* classes. This variant on an octree is used in ray tracing, unlike the previous application in which a different octree implementation is used to reduce input data resolution.

The *geometry* package contains the *BoundingBox*, *Face*, *Mesh*, *PointCloud* and *Vertex* classes.

The *optimization* package contains a single *Optimizer* class which runs the optimization algorithm. It contains the cost function calculation, named *getEnergy()* and functions *getEnergyB()*, *getEnergyC()* and *getEnergyD()*, which each calculate the contribution of a different type of rays.

Rays are grouped as follows:

- type D - not infinite, with intersections

- type C - infinite, with intersections
- type B - not infinite, no intersections
- type A - infinite, no intersections

The parameter naming conventions have been chosen as per this grouping, therefore parameter beta is the B group energy multiplier, gamma corresponds to the C group and delta to the D group. This leaves parameter alpha as the step size in the direction of the steepest descent of the optimized function.

The *raytracing* package features the *Camera* and *Ray* classes from the DataGenerator application. An additional class, *RTSettings*, deals with the ray tracing settings.

Lastly, the *reconstruction* package features classes such as *MarchingCubes*, *Line3D*, *Grid*, *GridData*, *MyArrayList* and *Tables*. The Line3D class uses mostly static variables, along with a static list (*MyArrayList*), as it features some of the most heavily used methods.

C Attachments

The following files are attached to this thesis:

- Data
  - spheres
  - models
  - azure_short
  - intel_short
  - kinect_short
- Examples
  - example 1
  - example 2
  - example 3
  - example 4
- Projects
  - AzureRecorder
  - IntelRealsenseRecorder
  - KinectRecorder
  - SphereGenerator
  - DataGenerator
  - PointCloudRegistration
  - PointCloudReconstruction
- Results
  - result 1.zip, result 1 log.txt
  - result 2.zip, result 2 log.txt
  - result 3.zip, result 3 log.txt
  - result 4.zip, result 4 log.txt

In the Data folder, examples of files extracted from the recording devices are enclosed, along with sphere models and simple 3D meshes, which can be used to test the DataGenerator and PointCloudRegistration applications.

In the Examples folder, settings files and data necessary to validate the output of this work are present.

The Projects folder contains Visual Studio 2017 and Eclipse 2020-06 projects of the developed software. For C# projects, the runnable .exe files are present in a "Build" directory. Java projects contain a "jar" directory with runnable .jar files.

The Results folder contains the results presented in this work. The first .obj file in each result directory is the initial reconstruction, while the last .obj file is the optimization output.