TU WIEN Informatics

Verbesserte Integration von Tiefenbilder in 3D Modelle

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Dennis Depner
Matrikelnummer 01632716

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Mag. rer. soc. oec. PhD Stefan Ohrhallinger
Mitwirkung: Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Wien, 27. Dezember 2020

Dennis Depner	Stefan Ohrhallinger

TU WIEN Informatics

Improved Integration of Depth Images in 3D Models

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Dennis Depner
Registration Number 01632716

to the Faculty of Informatics

at the TU Wien

Advisor: Mag. rer. soc. oec. PhD Stefan Ohrhallinger
Assistance: Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Vienna, 27th December, 2020

Dennis Depner	Stefan Ohrhallinger

Erklärung zur Verfassung der Arbeit

Dennis Depner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. Dezember 2020

Dennis Depner

Danksagung

An dieser Stelle möchte ich mich herzlich an die Personen bedanken, die mir geholfen haben diese Bachelorarbeit anzufertigen. Mein größter Dank ist meinem Betreuer Stefan Ohrhallinger gebührt, der mit mir fast jede Woche theoretische Fragen, Ideen und Ansätze besprochen hat. Ebenso hat er mir die nötige Hardware zur Verfügung gestellt, um die Theorie in die Praxis umsetzen zu können.

Acknowledgements

At this point I would like to thank the people who supported me in writing this bachelor thesis. My biggest thanks deserves my supervisor Stefan Ohrhallinger, who discussed theoretical questions, ideas and approaches with me almost every week. He also provided me with the hardware I needed to put the theory into practice.

Kurzfassung

3D Rekonstruktion mithilfe von Tiefenbildern ist in vielen Bereichen der Computergrafik wie z.B. Virtual Reality und Augmented Reality etabliert. Dabei ist es umso wichtiger dass die 3D Rekonstruktion genau und zuverlässig funktioniert um diese auch qualitativ in anderen Bereichen verwenden zu können. Bei dieser Arbeit wird speziell das Verfahren zur 3D Rekonstruktion aus dem Kinect Fusion Algorithmus behandelt. Dieser ermöglicht es aus Tiefenbildern die 3D Oberfläche in Echtzeit zu rekonstruieren. Es ist einer der bisher schnellsten und zuverlässigsten Verfahren zur 3D Rekonstruktion, wobei aber noch Möglichkeiten zur Verbesserung offen stehen. Ein Tiefenbild ist im 3D Raum zurückprojiziert eine Punktwolke, in der jeder 3D Punkt einem Pixel im Tiefenbild zugeordnet ist. Wenn mehrere Scans bzw. Aufnahmen von Tiefenbildern als 3D Modell rekonstruiert werden, ist das Hauptproblem von Kinect Fusion, dass es die Information der Punktwolken vorheriger/älterer rekonstruierter Scans nicht mehr zu den nächsten oder neuen Scans miteinberechnet. Das hat auch zur Folge, dass die Berechnung der Oberfläche an Genauigkeit verliert. Um das zu verbessern, werden die Punktwolken von Tiefenbildern mitgespeichert und zur Berechnung der 3D Oberfläche miteinbezogen. Da der 3D Raum bzw. die Oberfläche wie ein Gitter in gleich große Bereiche eingeteilt oder abgetastet wird, muss nicht zwingend jeder einzelne Punkt von allen Punktwolken abgespeichert werden. Das würde auch einen immensen Speicherverbrauch als weiteres Problem hinzufügen. Durch das 3D Gitter können die Koordinaten der Punkte, die sich im selben Bereich befinden, als Median aggregiert abgespeichert werden, was den Speicherverbrauch drastisch reduziert. Neben diesem Thema werden auch zwei Algorithmen zur Generation von Meshes verglichen, wobei es sich hierbei um den Marching Cubes Algorithmus und eine Adaption davon handelt, die speziell auf Octrees angewendet werden kann. Das Ziel der Arbeit ist zu bestimmen, wie sich die Einberechnung von vorheriger/älterer Tiefenbilder auf die Qualität der 3D Rekonstruktion der Oberfläche auswirkt. Ebenso werden die Unterschiede zwischen dem normalen und den für Octrees adaptierten Marching Cubes Algorithmus analysiert. Die Resultate haben ergeben, dass die Miteinberechnung der aggregierten Informationen vorheriger/älterer Tiefenbilder als Mediane im Vergleich zum gewöhnlichen Kinect Fusion Algorithmus einen positiven Einfluss hinsichtlich der Wasserdichtheit der 3D rekonstruierten Oberfläche hat. In kleineren Bereichen sind weniger Löcher auf der Oberfläche vorhanden. Ebenfalls erscheinen feinere Objekte weicher und weniger kantig rekonstruiert, was aber auch manchmal zu etwas verwascheneren Stellen im Mesh führt. Auch der adaptierte Marching Cubes Algorithmus für Octrees weist

eine starke Verbesserung bezüglich der Wasserdichtheit des Meshes auf. Es sind deutlich weniger Löcher bei den Meshes vorhanden, besonders an Kanten von Objekten. Die oben genannten Änderungen im Kinect Fusion Algorithmus sowie der adaptierte Marching Cubes Algorithmus haben keine sichtbaren Verschlechterungen oder Fehlbildungen am Mesh verursacht.

Abstract

3D reconstruction using depth images is established in many areas of computer graphics such as virtual and augmented reality. That makes it even more important that 3D reconstruction works precisely and reliably in order to use it properly in other applications. In this thesis, the procedure for 3D reconstruction from the Kinect Fusion algorithm is dealt with in particular. This enables the 3D surface to be reconstructed in real time from depth images. It is one of the fastest and most reliable methods for 3D reconstruction to date, but there are still possibilities for improvement. A depth image is back projected in 3D space a point cloud in which each 3D point is assigned to a pixel in the depth image. If several scans or recordings of depth images are reconstructed as a 3D model, the main problem of Kinect Fusion is that it no longer includes the information of the point clouds of previous/older reconstructed scans for new scans. That has the consequence that information and accuracy of the surface get lost. To improve this, the point clouds of previous/older depth images are saved and included in the calculation of the 3D surface. Since the 3D space or surface is divided into areas of equally sized cubes like a grid, it is not necessary to save each individual point from all point clouds. That would also add an immense memory consumption as another problem. Thanks to the 3D grid, the coordinates of the points located in the same area can be stored together as a median vector, which drastically reduces memory consumption. In addition to this, two algorithms for the generation of meshes are compared, which are the Marching Cubes algorithm and an adaptation of it, which can be specifically applied to octrees. The aim of the work is to determine how the inclusion of the aggregated point cloud information of previous / older depth images as median vectors affects the quality of the 3D reconstructed surface. The differences between the normal Marching Cubes algorithm and the adapted one for octrees are also analyzed. The results have shown that the inclusion of the aggregated point cloud information from previous / older depth images as median vectors has a positive influence on the watertightness of the 3D reconstructed surface compared to the usual Kinect Fusion algorithm. In smaller areas, there are fewer holes on the surface. Finer objects also appear softer and reconstructed less edgy, but this sometimes leads to blurred areas in the mesh. The adapted Marching Cubes algorithm for octrees also shows a strong improvement in terms of the watertightness of the mesh. There are significantly fewer holes in the meshes, especially on the edges of objects. The above-mentioned changes in the Kinect Fusion algorithm and the adapted Marching Cubes algorithm have not caused any visible deterioration or malformations on the mesh.

Contents

CHAPTER 1

Introduction

1.1 Overview

The reconstruction of 3D models in the real world plays an important role for various applications in numerous fields like computer graphics, computer vision, medical imaging, virtual reality etc. One example for that is augmented reality, which is heavily dependent on a precise and consistent 3D reconstruction of world objects. Knowledge of position, form, and scale of real world objects is necessary, in order to place new virtual objects in an environment[YCH+13]. Another highly correlated topic to augmented reality is visual SLAM (Simultaneous Localization And Mapping). Visual SLAM technologies enable autonomous sensors to localize their own position by scanning and mapping their environment simultaneously. This can be very useful for the development of autonomous vehicles or robots[YL18]. Another application is visualizing medical data. It is mostly disadvantageous to provide 2D data for a surgery or medical procedure. Medical-purpose 2D image data is often obtained by MRI (Magnetic Resonance Imaging), X-Ray (roentgen radiation) etc. However there are already techniques which can transform those images to 3D objects which gives medical practitioners a significant advantage in terms of observation, precision etc.[CSVV11]. 3D reconstruction also has a big impact on civil engineering. It is concerned with the design, development, construction and maintenance of different infrastructures like city roads, buildings or canals. For those tasks it can be very helpful to reconstruct the environment like the city buildings and its surroundings. Afterwards new structural elements like a new building or road can be virtually placed into the 3D reconstruction in order to simulate the outcome[ML17]. There are also applications for 3D reconstruction which operate on a smaller scale than civil engineering. Sometimes it can be advantageous to reconstruct the interior structure and elements of smaller facilities like a room. New elements like a table or chair can be placed into a 3D reconstructed room for testing its look, appearance and fit. A good reference for that is the application "Modsy Interior Design" provided by "IOS", which makes it possible to

scan a room with an iPhone and place new furniture into it.

Obviously there are a lot more applications for 3D reconstruction than mentioned above, but mostly they all have in common that they need data with 3D information. Depth maps or images are most commonly used for providing 3D data compromised as 2D images. Pixels in those images does not have the purpose to save the color but the depth of an object in a scene[ZLD15]. Depth cameras like the "Kinect v2 camera" are capable of scanning the depth of objects with an infrared sensor. It computes depth by emitting infrared light to the objects and measuring the time of flight until the emitted reflected light comes back to the sensor. This method is also called LIDAR (Light amplification by Stimulated Emission of Radiation detection and ranging)[Bi20]. Another method is called Stereo Depth, e.g. used by the depth camera "Intel RealSense Depth Camera". Here two sensors are obtaining two normal RGB images at the same time from different positions (simulating the eye vision of the human). Finally, the depth image can be calculated with the two images and the distance between the sensors[ABR12].

Reconstructing the depth images to a 3D model mainly consists of two areas, camera tracking and surface generation. Concerning camera tracking it is necessary to know where the camera in space is in order to properly align the depth images to a correct 3D model. Surface generation means the creation or definition of the 3D surface (e.g. as a geometrical function or a mesh) using the depth images. Camera tracking and surface generation can be handled in a Frame-to-Frame (Camera tracking and/or surface generation of one depth image is dependent of the previous depth image), Frame-to-Model (Camera tracking and/or surface generation of one depth image is dependent of the current 3D model generated from previous depth images) or global fashion (Camera tracking and/or surface generation is done with all depth images at once)[NIH+11].

1.2 Motivation

In this work the focus is laid on the Frame-To-Model approach, since all our experiments and evaluation are based on the Kinect Fusion Algorithm[NIH+11]. Kinect Fusion generates a dense 3D model obtaining depth images from a depth sensor (e.g Kinect v2 camera) in real time. It is proven to be one of the fastest and most efficient algorithms for 3D reconstruction using depth images. However, it still leaves some headroom for improvement considering accuracy of the surface generation. In 3D space a depth image is a point cloud in which each 3D point is assigned to a pixel in the depth image. If multiple scans or recordings of depth images are reconstructed as a 3D model, the main problem with Kinect Fusion is that it does not include the information of the point clouds of previous/older reconstructed scans in new scans. In theory this has the consequence that information and accuracy of the surface get lost. An approach for a solution is to simply save and use the point cloud information for the surface generation. Because simply saving all points of every depth image in the 3D model would cause dynamic and massive storage requirements, the coordinate information of multiple points in the same area (the 3D space is divided into multiple equally sized areas/cubes) can be saved as a

median vector. That solution is also the foundation for a currently developed method, which uses the median (and also the statistical variance) of the points in order to create a 3D surface. In the following chapters the 3D reconstruction process as well as the required changes and improvements of Kinect Fusion will be explained. Next to the changes of the Kinect Fusion Algorithm it will also be discussed the differences between two mesh generation algorithms, which are the Marching Cubes Algorithm and an adaptation of Marching Cubes for octrees.

CHAPTER 2

Theory

2.1 Kinect Fusion to reconstruct 3D Models

Kinect Fusion basically processes the depth information of depth images frame after frame to obtain a dense 3D model. To get a better understanding of what actually is happening in the Kinect Fusion Algorithm, we take a closer look at the pipeline. Kinect Fusion consists of four steps: Surface Measurement, Surface Reconstruction Update, Surface Prediction and Surface Pose Estimation, which will be explained in the following[NIH+11].

2.1.1 Kinect Fusion Algorithm Pipeline

1. Surface Measurement

In this step a point cloud consisting of a vertex map V and a normal map N is calculated with the scanned depth image. In order to obtain the vertices for the vertex map V, each of the pixels in the depth image with its corresponding values (u, v) (the image coordinates) and z(the depth measurement) is converted into the 3D world. That is done by first applying the inverse intrinsic matrix of the camera and multiplying it then with z[NIH+11]. In Figure 2.1 is also the conversion of a depth image into a point cloud illustrated.

$$P = z * K^{-1} * \begin{pmatrix} u \\ v \\ 1 \end{pmatrix}$$

$$P := \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

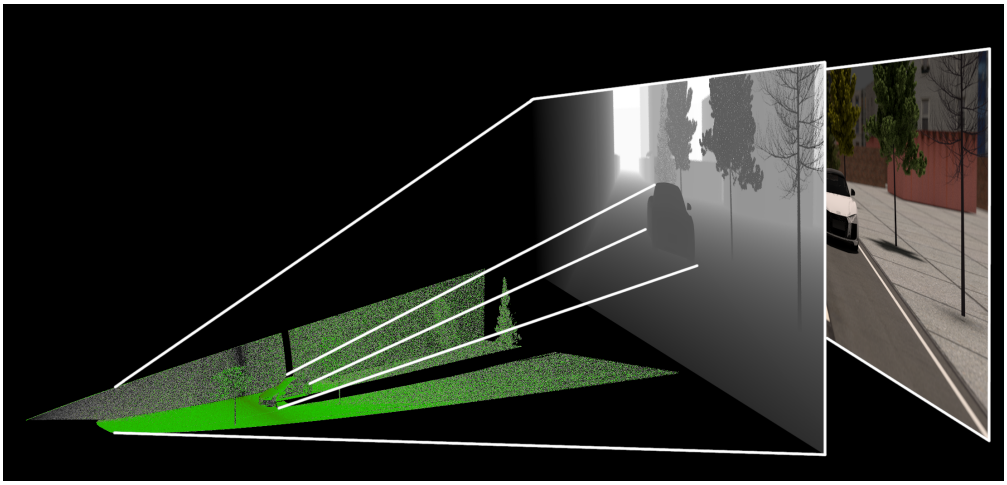


Figure 2.1: Converting a depth image to a point cloud. On the right side is an RGB image and its depth image. On the left side is the converted depth image as a point cloud[Dep].

$$K := \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

f_x ... focal length of the camera to the image plane in x direction
f_y ... focal length of the camera to the image plane in y direction
c_x ... position of the camera on the image plane in x direction
c_y ... position of the camera on the image plane in y direction

It should also be mentioned that Kinect Fusion applies a bilateral filter to the raw depth image before computing V. The purpose of that is to reduce noise of the depth images[NIH+11].

Computing the normal map N means to calculate the normal of each vertex in V. This can be done by simply taking one vertex v and two other vertices a and b, which are neighbours of v in the depth image. Then the cross product of v to a and v to b is the normal vector of v[NIH+11].

Computing the vertex and normal map is essential for being able to estimate the sensor or camera pose later[NIH+11].

2. Surface Reconstruction Update

For extracting a surface of the depth image we need a function to represent it. Therefore we use the $TSDF$ (Truncated Signed Distance Function). Before we explain the $TSDF$, we first take a look at the normal SDF (Signed Distance Function). Simply explained the SDF is a function, which takes as input a 3 dimensional point $p = (x, y, z)$ and

outputs the shortest signed distance of p to a surface within a defined metric (e.g. the point (1,3,7) is 5(cm) away from the surface or $SDF((1,3,7)) = 5$)[OF03]. The sign of the distance is determined whether p lies beyond (positive sign), behind (negative sign) or exactly on the surface (then distance is 0) respectively to the camera position. So the distance is positive, if camera and p are on the same side of the surface, and negative or 0 otherwise. In Kinect Fusion the calculation of the SDF is simplified by only determining the signed distance between the camera position and p only along the z-direction in camera space[NIH+11]. $TSDF$ is actually the same concept except of taking only points in a defined range or truncation band μ into account. The distance inside the truncation band μ needs to be normalized dividing it by μ. If a point's distance from the surface is bigger than μ, it is set to either 1 (far beyond the surface) or -1 (far behind the surface)[NIH+11]. The surface can also be defined as an implicit function $TSDF(p) = 0$, $p \in \mathbb{R}^3$.

Since it is not possible to compute the $TSDF$ continuously, we have to sample it along a 3 dimensional grid with a specific resolution. The grid consists of multiple voxels (imagine a 3D pixel or cube with 8 vertices) and for each of them the signed distance is computed[NIH+11]. In Figure 2.2 is also a 2D example for a $TSDF$.

When calculating the signed distance for a voxel, we first need to project the position of a voxel back to the depth image plane, so the corresponding depth measurement can be obtained. Which position of the voxel is taken for the projection and which for saving the distance depends on the implementation. For our implementation the middle of a voxel is used for projection and the minimum vertex of a voxel (the vertex which has the smallest x,y and z coordinates amongst the 8 vertices of a voxel) is used for saving the distance. The projection from world to camera space of a voxel is done by first multiplying its middle position with a matrix defining the estimated camera rotation/translation. After that the position is in camera space and is multiplied by a second matrix defining the camera intrinsics (already mentioned in "Surface Measurement" as K. It is used to transform 3D points from camera space onto the depth image plane). Because there can not be an estimated camera pose for the very first depth image, in the first iteration an initial guess is used (e.g. $(0,0,0)$ as the camera's position in world space or a matrix for the estimated camera pose without any rotation or translation at all). When converting the middle position of a voxel to a point c in camera space by multiplying it with the matrix for the estimated camera pose, Kinect Fusion gets the distance by calculating the difference between the z coordinate of c and the corresponding depth measurement[NIH+11].

The reason for using $TSDF$ is the easy way of fusing it. Because for every depth image, which comes as input for Kinect Fusion, a single $TSDF$ is computed. All $TSDFs$ can be fused to a single $TSDF$ by taking the average of each voxel's computed signed distances. The detailed mathematics will be explained in our "Method" chapter, since this processing stage is the vital part to understand our new method of integrating the depth measurements into the 3D model[NIH+11].

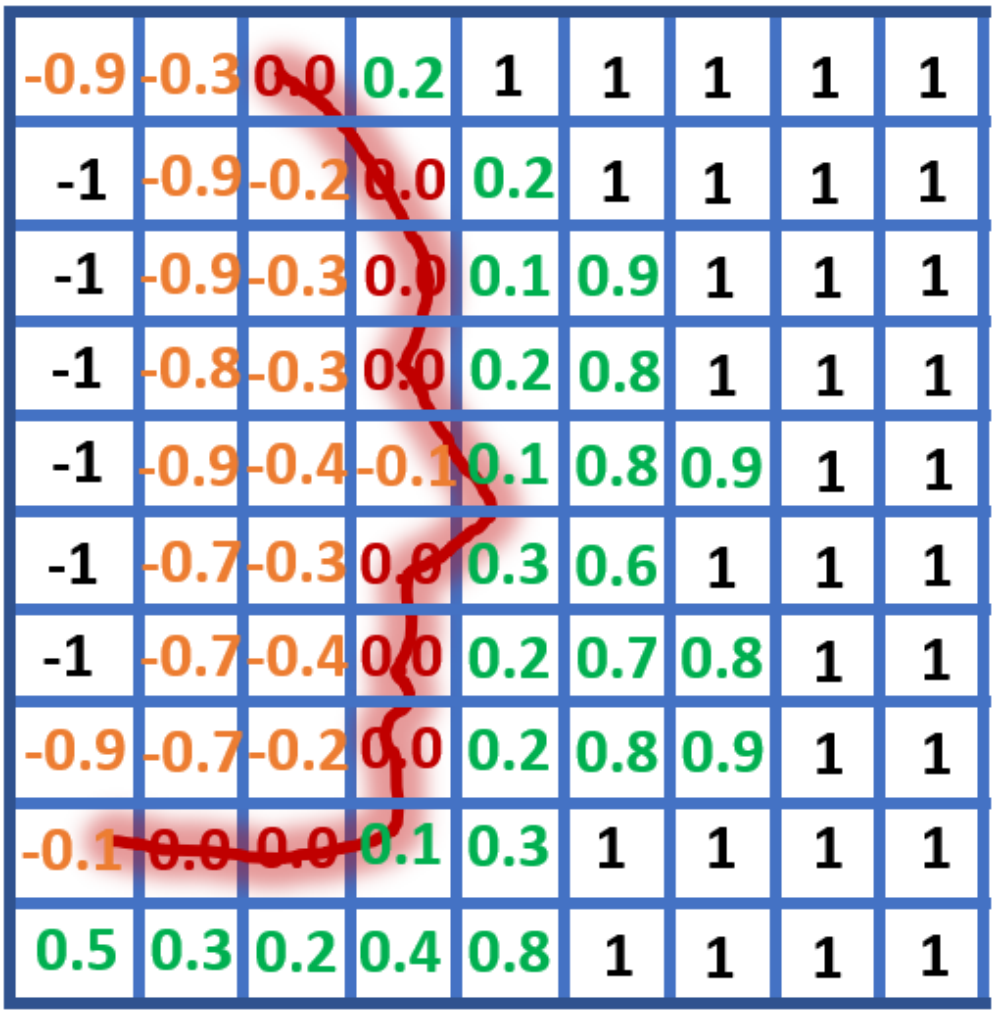


Figure 2.2: 2D $TSDF$ of a 2D surface marked as a red line. For each voxel, here simplified as a 2D square, the shortest signed distance from the square to the red marked surface is computed. In this example the signed distance is depicted inside of each square[Kin].

3. Surface Prediction

In this processing stage the $TSDF$ is raycasted from the depth image along the estimated camera pose in order to create another point cloud consisting of a vertex and normal map. Imagine for every pixel in the depth image, a ray is sent starting from the estimated camera pose, going through the pixel into the 3D grid representing the $TSDF$. Actually the ray does not start from the camera position, but from the pixel's depth, since any smaller depth lies beyond the surface and the $TSDF$ will be 1. While the ray marches through the 3D grid with a step size of the voxel length, it stops when a zero crossing in the grid is found (when the $TSDF$ changes from negative to positive or vice versa). The position for the zero crossing is then saved in the vertex map. The corresponding normal can be calculated by the gradient on the zero crossing, which is computed by the value change of nearby voxels' $TSDF$ values.[NIH+11].

These steps finally generates a vertex and normal map of all fused $TSDFs$ representing the global 3D surface. That vertex and normal map can be used to compare it with the vertex and normal map computed with the raw depth image of the first step "Surface Measurement" to finally estimate the camera position[NIH+11].

4. Sensor Pose Estimation

For estimating the position of the camera to be able to update the $TSDF$, the ICP (Iterative Closest Point) Algorithm is used. ICP takes two point clouds A and B, in our case B is the vertex and normal map of the "Surface Measurement" and A is the vertex and normal map of the "Surface Prediction". In Figure 2.3 the concept of ICP is explained by an example. ICP consists of two parts, first the data association and second the transformation[Low04].

In data association we need to find for every point in B an associated point in A. That is done by an algorithm called Projective Point Plane Data Association. The algorithm searches for every point in B the closest point in A, after projecting them into camera space of the current estimated camera pose[PB13].

In the transformation step after finding all associated point pairs of A and B, ICP searches for a matrix M representing the camera rotation and translation, which minimizes a defined error metric between A and B when applying M to B. Since A is not only seen as a raw point cloud but the predicted global surface, the point-to-plane error metric is used. Therefore the sum of the least squared distances between all point pairs is calculated, while the distance is defined as the smallest distance from point b in B (b is transformed by applying matrix M to B) to the tangent plane of point a in A. In short we have to find a matrix M which minimizes the error distance of this equation.[Low04]

$A = \{(v, n) | v \in \mathbb{R}^3, n \in \mathbb{R}^3\}$
$B = \{(v, n) | v \in \mathbb{R}^3, n \in \mathbb{R}^3\}$
v ... vertex
n ... normal

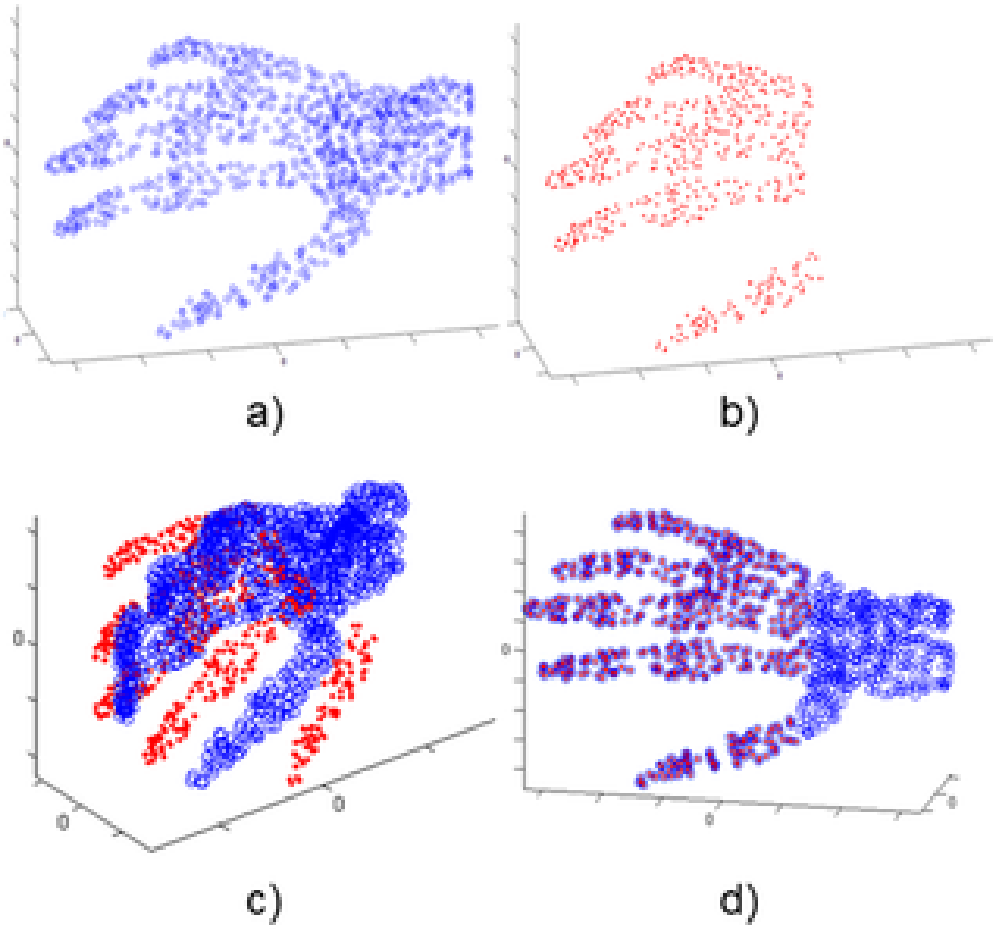


Figure 2.3: Example for the ICP Algorithm. Two different point clouds a) and b) of the same object (a hand) are aligned with ICP. On c) the hands are misaligned, on d) the hands are correctly aligned by ICP[MMPGGMG16].

$$argmin_M \sum_{(v_A,n_A)\in A,(v_B,n_B)\in B}((M * v_B - v_A) * n_A)^2$$

Solving the equation and finding the matrix M of the camera rotation and translation is mathematical a bit more difficult and can be examined for further explanation here [Low04]. To put it simply the matrix M is approximated over several iterations. In the first iteration an initial solution M is set and refined in the next iterations using an output B' created by applying M to B. For B' the data association is done again with A and according to that another M' is heuristically found, which updates or refines M by $M = M' * M$. This procedure iterates until M converges and the decrease of the point-to-plane error between B' and A gets negligible.

2.1.2 Procedure

When applying the above mentioned processing steps on depth images, there is a specific order using them. To the very first depth image step 1 "Surface Measurement" and step 2 "Surface Reconstruction Update" will be applied. Step 3 "Surface Prediction" and 4 "Sensor Pose Estimation" are not necessary, since only one depth image was obtained. After obtaining a new depth image, first step 1 and then step 3 and 4 will be applied before step 2. That is because before updating the surface with step 2 it is necessary to estimate the camera pose of the new depth image with step 3 and 4 according to the current reconstructed surface (the reconstructed surface from previous depth images, e.g. from the first depth image)[Kin].

2.2 Data Structures for the Voxel Grid

Now that we know how Kinect Fusion works, we take a look into some data structures we use to compute the sampled $TSDF$ as a voxel grid. Because for larger scenes the grid becomes very large, we need to use data structures which reduce the voxels to an amount which actually has to be computed. Therefore we are going to look at two solutions, which speed up the computation time.

2.2.1 Grid as Hash Table

The first solution is called Voxel Hashing[MNS13]. Here we use a hash table to store each of the voxels. In a hash table, data is stored in an associated manner, which means every data element has its own unique index or key in an array. That makes insertion, retrieval and searching of data elements very fast (constant time complexity or $O(1)$), since we just need to compute the key for a data element. For computing the key a hash function is used, which takes as input a data value and outputs the corresponding key[Kar20].

In our case we have to imagine a uniform infinite 3D grid consisting of so called voxel blocks. A voxel block is basically a cube consisting of 8 x 8 x 8 or 512 voxels and has a hash key, which can be computed with its position. The position of a voxel block is indexed with integers in (x, y, z) direction, e.g $(0, 0, 0)$ would be the voxel block located on the origin and (1,0,0) would be its neighbouring voxel block in x direction. The hash function takes the indexed position and calculates the sum of its components (x,y,z), while every component is multiplied with large prime numbers. The sum is cut down by an array size n with the modulo operator, to ensure the hash key is smaller than n[MNS13].

$p_1, p_2, p_3 \in \mathbb{N}^3 ... large prime numbers\ x, y, z \in \mathbb{Z}\ H(x, y, z) = (x * p_1 + y * p_2 + z * p_3)\, mod\, n$

Allocation

Since only a small proportion of voxel blocks is likely to be interfering with the measured surface, we take advantage of the $TSDF$, mentioned in "Surface Reconstruction Update".

Because $TSDF$ only takes points into account which lies within a truncation band μ, we only need to allocate voxel blocks, which lies fully or maybe partially in this truncation band. This can be simply done by backprojecting for every pixel of the depth image a line from $d-\mu$ to $d+\mu$ in world pace, while d is the depth measurement for a pixel. That line intersects a specific number of voxel blocks, which will be allocated to corresponding hash keys[MNS13].

Because hash collision is possible, which happens if two or more voxel blocks have the same hash key, we need to use hash buckets. When using hash buckets, it is basically a combination of the hash table and a linked list. Every hash key or entry in the hash table is a header to a linked list or also called bucket with a specific size. When a voxel block a is already allocated at the header with a hash key k and another voxel block b has also k, b is allocated at the next free entry in the bucket. In case the bucket will overflow, the last entry of the bucket is reserved as a pointer to another bucket's free entry. So when reaching the last entry of a bucket, another free entry of another bucket is searched in the entire hash table, in which the voxel block is allocated. The pointer to that allocated voxel block is saved in the last entry of the bucket[MNS13]. In Figure 2.4 the whole concept of Voxel Hashing is illustrated as a 2D example.

During allocation we keep track of the allocated voxel blocks in two lists. The first list saves all allocated voxel blocks. The second list only saves the allocated voxel blocks, which are currently visible (which means only the allocated voxel blocks which are in the current camera frustum). When new depth images are scanned and the camera was rotated or translated, both lists will be updated. In the first list, new allocated voxel blocks are added to the previous ones, since the camera now scans a new area. In the second list, already allocated voxel blocks of previous depth images, which moved out of the camera frustum, are removed. New or allocated voxel blocks, which moved inside the camera frustum, are then added to the second list[MNS13].

Integration

When integrating a depth image or updating the TSDF, we only need to iterate over the list of visible allocated voxel blocks, because only the voxels of those can be backprojected onto the depth image plane. So the number of voxel blocks is reduced to those, which are inside the camera frustum and interfere with the truncation band μ. Integrating a voxel block is simply done by iterating over its 512 voxels and calculate for each voxel the signed distance like already explained in "Surface Reconstruction Update"[MNS13].

Grid as Octree

Another possibility to structure the data of the voxels is an octree. An octree is a tree, in which every node has eight children nodes. In 3D space, the root node is a cube with a defined edge length. That cube can be divided into eight equally sized children cubes/nodes recursively. So every children cube/node can be divided again into eight equally sized children cubes/nodes. In our case every node represents a voxel[SVB99].

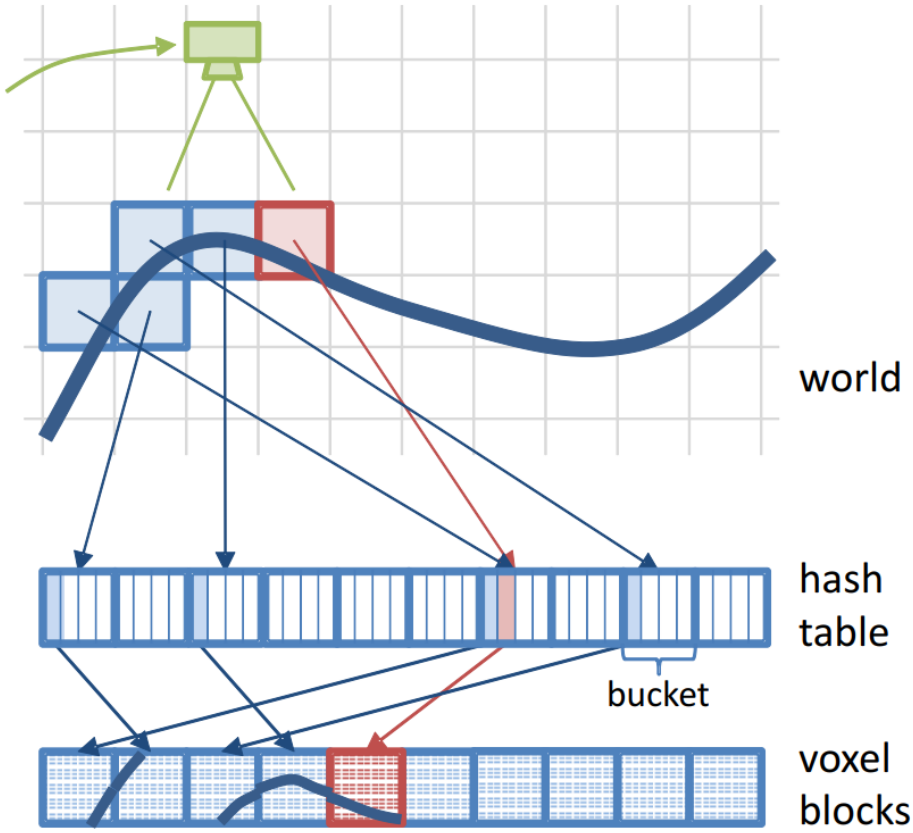


Figure 2.4: Illustration of Voxel Hashing. On top the world is depicted simplified as a 2D infinite grid. The green symbol represents the camera. The blue and red marked squares are allocated voxel blocks, for which a hash key has been computed. In the middle the hash table with buckets containing the entries of the voxel blocks is depicted. On the bottom the pointer from the hash entries to the voxel blocks is depicted[MNS13].

For allocation we only need to build a sparse octree by only using the voxels interfering with the truncation band μ similar to the hash table. However there is a difference compared to the hash table when allocating or inserting voxels into the octree. Before inserting voxels, we first need to find the boundaries or the size of the octree. For every depth image this is simply done by keeping track of the minimum and maximum position amongst all allocated voxels. The octree size or its edge length is the maximum difference between the minimum and maximum position in x, y or z direction. Inserting the voxels is done by looking up recursively in which octree node the voxel lies in space. Visible voxels (inside the current camera frustum) and not visible voxels (outside of the current camera frustum) can be tracked in two separate lists like for the hash table.

For integration we simply iterate over the allocated visible voxels and calculate the signed

distance like explained in "Surface Reconstruction Update".

When comparing the computation time with the hash table, the octree needs significantly more time. Allocating n voxels in the hash table has a time complexity of $O(n)$, since inserting one voxel has only $O(1)$. When allocating n voxels in the octree, we first need to find the voxels with the minimum and maximum position, which already has a time complexity of $O(n)$. For inserting one voxel into the octree, the time complexity is $O(log(m))$ while m is the depth of the octree. The octree's depth m is defined by the longest path to a leaf node (a node without any children nodes[SVB99]). Therefore inserting n voxels has a time complexity of $O(n * log(m))$. Computing the octree size and the allocation of n voxels in the octree have a time complexity of $O(n + n * log(m))$. Integrating n voxels has a time complexity of $O(n * log(m))$, since searching one voxel in the octree takes $O(log(m))$.

Because of the worse computation time, we rather use the hash table for allocation and insertion. However an octree offers some advantages when generating the mesh of the voxel grid, which will be further explained in the next chapters and especially more in detail in the "Method" chapter.

2.2.2 Meshing of Voxel Grid

To obtain a visual surface we generate a mesh of the voxel grid by using the Marching Cubes Algorithm. Here we iterate over all allocated voxels and generate polygons for each of them. Each voxel has eight vertices while each of them has a signed distance marking them as inside (negative distance) or outside (positive distance) of the surface. When processing one voxel, the goal is to separate the inside vertices from the outside vertices with the generated polygons, since that represents the actual surface. That means, that all zero crossings of the voxel edges (an edge has a zero crossing, if on one end the vertex has a negative distance and on the other end the vertex has a positive distance) are connected to polygons so that inside vertices are separated from the outside vertices. All in all there are 2^8 possibilities, in which the eight vertices can be marked as inside or outside. For every possiblity it is already saved, how the zero crossings have to be connected to generate the polygons. All possibilities can be broken down to rotational invariant base cases, which are also illustrated in Figure 2.5. The position of a zero crossing on an edge is found by calculating the linear interpolated position of the two vertices on the edge considering their signed distances to the surface. If the signed distance of one vertex is closer to zero (closer to the surface), than the position of the zero crossing is closer to that vertex and vice versa[LC87]. In Figure 2.6 the whole concept of the Marching Cubes Algorithm is illustrated on a 2D example.

When taking Voxel Hashing into account, we only iterate over all allocated voxels and generate the polygons like mentioned above. However when using an octree, we cannot use the normal Marching Cubes Algorithm. That is because we also iterate over coarser nodes/voxels, which do not necessarily have the same resolution of neighbouring nodes/voxels. For example the edge of a voxel can be neighboured to a smaller or longer

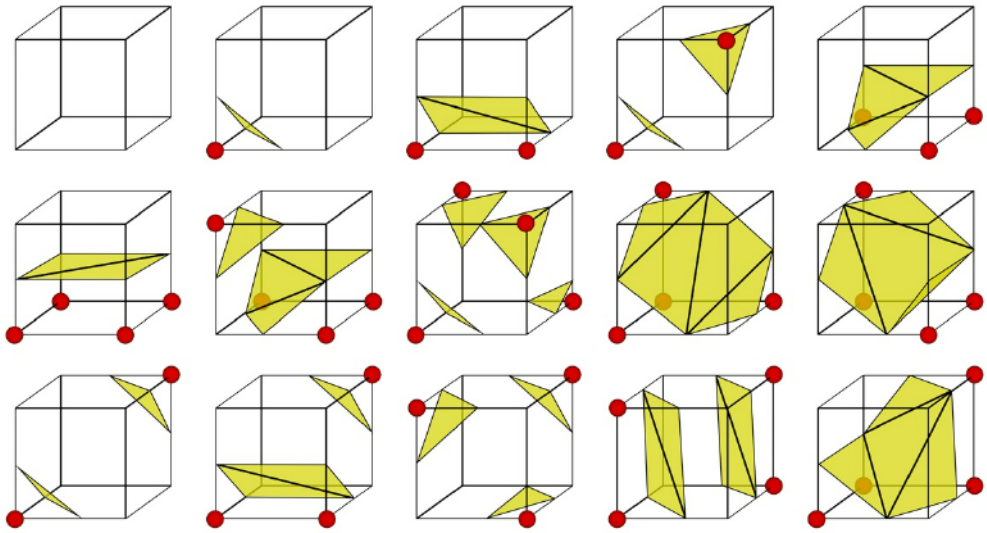


Figure 2.5: Marching Cubes Cases for a voxel. For each voxel considering the surface the inside vertices are marked with a red dot and the outside vertices without one. For each combination of inside and outside vertices a constellation of polygons separating them is generated. Although there are 2^8 possibilities of inside and outside vertices for one voxel, a lot of combinations are rotational invariant and can be broken down into these 15 cases[LHG17].

edge of a voxel, which is not on the same level or depth in the octree. If we would just generate the polygons for every node/voxel in the octree independently, there could appear cracks along such edges, since the zero crossing of a bigger edge must not match with the zero crossings of possibly neighboured smaller edges. Therefore, we have to use a Marching Cubes Algorithm of this paper [MKH07], which is adapted to octrees. Shortly explained, for edges of coarser voxels which are neighboured to smaller edges of finer voxels, the zero crossings of the bigger edges are replaced with those of the neighboured smaller edges. This should eliminate possible cracks. The problem and avoidance of possible cracks is also illustrated in Figure 2.7. Because the Marching Cubes Algorithm adapted to octrees can also mesh coarser leaf nodes, this offers an advantage when it comes to missing measurements or noise in the TSDF of the voxel grid[MKH07]. Further explanation to that will follow more in detail in the "Method" chapter.

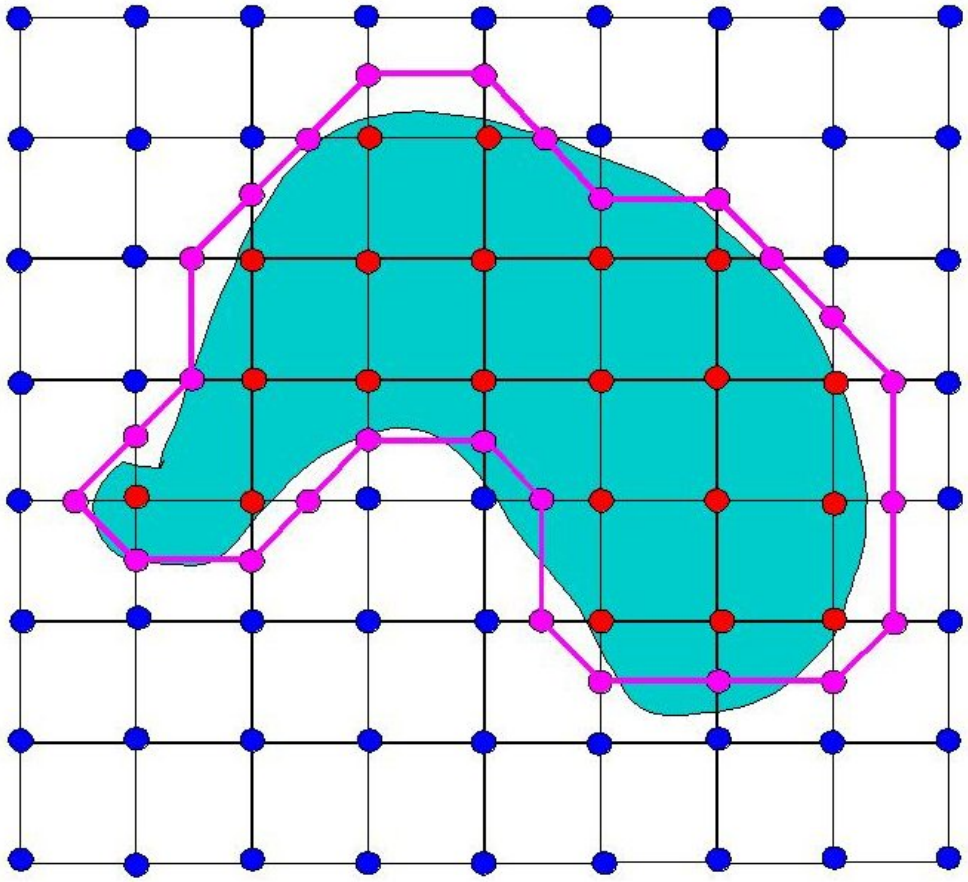


Figure 2.6: Marching Cubes on a 2D grid. A 2D grid with the voxel's vertices (simplified as 2D squares) marked as inside (red) and outside (blue) of the surface. The not interpolated zero crossings are connected to each other and marked as a violet line. The interpolated zero crossings and the surface area are marked with a light blue color[2DM].

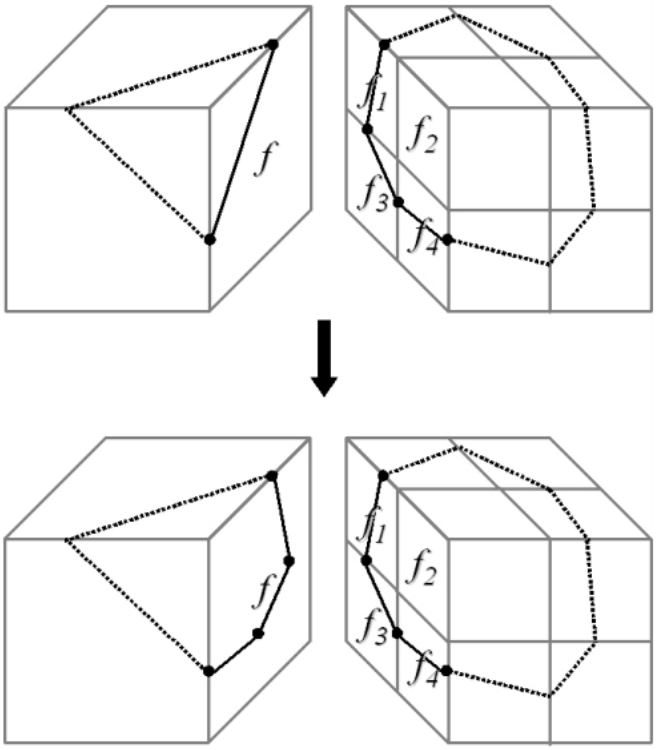


Figure 2.7: Problem with applying normal Marching Cubes to Octrees. On top you can see a coarser voxel node and to its right finer voxel nodes, on each of them normal Marching Cubes is applied independently. It is very clear that the edge of the face f does not match with the faces f_1, f_2, f_3 and f_4, which will cause a crack in the surface. When applying the Marching Cubes adapted to Octrees, the coarser edge of f gets replaced by the finer edges of f_1, f_2, f_3 and f_4 in order to avoid cracks, which is also depicted on the bottom[MKH07].

CHAPTER 3

Method

To improve the integration of the depth information into the global 3D model, we are going to change the processing step "Surface Reconstruction Update" of "Kinect Fusion". When taking a closer look at it, there are several issues to discuss. Firstly Kinect Fusion does not compute for the $TSDF$ the real distance from a voxel to its corresponding depth measurement. It simplifies by transforming the voxel's position into camera space and calculates the signed difference of its z-coordinate to the depth measurement. Since this is an inaccuracy, we transform the corresponding depth measurement to a 3D point in world space and calculate the shortest distance to one of the voxel's eight vertices. Secondly Kinect Fusion only considers the current depth image when computing the local $TSDF$ (local $TSDF$ means the $TSDF$ of one depth image, the global $TSDF$ is the fused $TSDF$ of all depth images) and ignores nearby depth measurments of previous/older scans. When backprojecting a depth measurement to a 3D point in the voxel grid, it falls inside one voxel. However inside this voxel there could also be other depth measurements of previous/older depth images. In order to take also the other depth measurements into account, we save the coordinate information of all depth measurements inside a voxel as a median vector. For computing the TSDF of one voxel, we calculate the shortest distance from its eight vertices to the corresponding median vector. The corresponding median vector can be inside the voxel itself or in another voxel. Saving the median vector for every voxel is also the foundation for a new surface operator currently researched and developed. The surface operator marks vertices as inside or outside of the surface using the median (and also statistical variance) vector of all backprojected depth measurements inside one voxel.

3.1 Preliminaries

In order to explain the above mentioned improvements, some definitions has to be made. The voxel grid is defined as

$G = \{(p, m, d, n_m, w, s) | p \in \mathbb{Z}^3, m \in \mathbb{R}^3, d \in \mathbb{R}, n_m \in \mathbb{N}, w \in \mathbb{N}, s \in \mathbb{R}\}$

consisting of tupels representing each voxel with p (indexed position of the voxel), m (median vector of all backprojected depth measurements inside the voxel), d (global TSDF of the voxel), n_m (number of all backprojected depth measurements inside the voxel), w (number of update cycles of the voxel) and s (size or edge length of the voxel). The k_{th} depth image is defined as

$I_k = \mathbb{N}^2$

$depth : I_k -> \mathbb{R}$
$depth((u, v)) = t$
$(u, v) \in \mathbb{N}^2$
$t \in \mathbb{R}$

u and v are the pixel coordinates and t is the depth measurement. To be able to project the voxels from world space onto the depth image plane and the depth measurements back into world space, a matrix K for the camera intrinsics is defined as already explained in the Kinect Fusion pipeline. For the camera extrinsics a $4x4$ matrix C_{EST} is defined representing the estimated rotation and translation of the camera.

3.2 Old Integration

As already mentioned Kinect Fusion only computes the $TSDF$ for a voxel by calculating the difference between the depth measurement and the z-coordinate of the voxel's position in camera space. Therefore we first need to project the position $p = (x, y, z)$ of a voxel $g \in G$ back onto the depth image plane to get the corresponding depth measurement[NIH+11].

(1) $\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = C_{EST} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$

(2) $\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = 1/z' * K * \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$

(3) $depth((\lfloor u \rfloor, \lfloor v \rfloor)) = t$

In the first step we transform the homogenized position $p = (x, y, z, 1)$ of g into the camera space with C_{EST}. Next we project the transformed position $p' = (x', y', z')$ of g

from camera space onto the depth image plane by multiplying it with K and $1/z'$. In the last step we take the depth measurement t at the pixel coordinates (u, v) (u and v are of course rounded down to integers), which we obtained in the previous step[NIH+11].

Now that we have the depth measurement t and z', we can calculate the local $TSDF$ d_{local} of the voxel g[NIH+11].

$$d_{local} = \begin{cases} 1, & \text{if } t - z' > \mu \\ -1, & \text{if } t - z' < -\mu \\ \frac{t-z'}{\mu}, & \text{otherwise} \end{cases}$$

After that we only need to fuse d_{local} with the $TSDF$s of previous depth images to the global $TSDF$ d, which means calculating the average of all $TSDF$s of that voxel. Calculating the average can be done incrementally with the w property of the voxel g, which tracks the number of updates or computed $TSDF$s. d, which is initially zero, can be calculated like that[NIH+11]:

$$d = \frac{d*w+d_{local}}{w+1}$$

Since we now added the local $TSDF$ d_{local} to the global $TSDF$ d, we only need to increment w by one for the next depth image or update cycle[NIH+11].

$$w = w + 1$$

3.3 New Integration

For the new integration of a depth image, we first update the median vectors of all voxels with the corresponding depth measurements. To do that, we first iterate over all pixels of the depth image and backproject them to 3D points in world space of the voxel grid. For a 3D point we calculate its indexed position in the voxel grid by dividing every coordinate with the voxel size in order to find out in which voxel the 3D point lies. After that the median vector of the voxel will be incrementally updated with the 3D point (See also

Algorithm 3.1).

Algorithm 3.1: Updating median of voxels

// Hashfunction returns a voxel according to an indexed position

$Hash : \mathbb{Z}^3 -> G$

$Hash(pos) = g$

$pos \in \mathbb{Z}^3$

$g \in G$

// Updating all median vectors

for $\forall (u, v) \in I_k$ **do**

// 3D position in camera space of (u, v)

$$p_{cam} = depth(u, v) * K^{-1} * \begin{pmatrix} u \\ v \\ 1 \end{pmatrix}$$

// transformed p_{cam} into world space

$$p_{world} = C_{EST}^{-1} * \begin{pmatrix} p_{cam}.x \\ p_{cam}.y \\ p_{cam}.z \\ 1 \end{pmatrix}$$

// indexed voxel position

$pos = \lfloor \frac{p_{world}}{voxelsize} \rfloor$

// corresponding voxel v for indexed pos

$v = Hash(pos)$

// Updating median vector $v.m$ and the counter $v.n$ of v

$v.m = \frac{(pos + v.n * v.m)}{(v.n+1)}$

$v.n = v.n + 1$

end

After updating the medians the $TSDF$ of the voxels can be calculated. Therefore we iterate over all currently visible voxels (voxels which are inside the frustum of the estimated camera pose), which we define as $G_{vis} \subseteq G$. Every voxel will be matched with its corresponding depth measurement by backprojecting its middle position onto the depth image plane. When obtaining a depth measurement for a voxel v, it is backprojected

into the 3D voxel grid in order to find out in which voxel v_2 it lies. The local $TSDF$ of v can be calculated with the median vector of v_2. Later the global $TSDF$ of the voxel v can be updated with the local $TSDF$ like already explained in the "Old Integration"

(See also Algorithm 3.2).

Algorithm 3.2: Updating TSDF of voxels with median

for $\forall v \in G_{vis}$ **do**

// transform $v.p$ of v from world space into camera space

$$p = C_{EST} * \begin{pmatrix} v.p.x \\ v.p.y \\ v.p.z \\ 1 \end{pmatrix}$$

// transform p from camera space onto the depth image plane

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{z} * K * \begin{pmatrix} p.x \\ p.y \\ p.z \end{pmatrix}$$

// depth measurement t of (u, v)

$$t = depth(\lfloor u \rfloor, \lfloor v \rfloor)$$

// transform depth measurement t back to camera space

$$p_2^{cam} = t * K^{-1} * \begin{pmatrix} \lfloor u \rfloor \\ \lfloor v \rfloor \\ 1 \end{pmatrix}$$

// transform p_2^{cam} into world space

$$p_2^{world} = C_{EST}^{-1} * \begin{pmatrix} p_2^{cam}.x \\ p_2^{cam}.y \\ p_2^{cam}.z \\ 1 \end{pmatrix}$$

$$p_2 = \begin{pmatrix} p_2^{WORLD}.x \\ p_2^{WORLD}.y \\ p_2^{WORLD}.z \end{pmatrix}$$

// indexed voxel position *pos* of backprojected depth measurement

$$pos = \lfloor \frac{p_2}{voxelsize} \rfloor$$

// voxel v_2, in which depth measurement lies

$$v_2 = Hash(pos)$$

// calculate local $TSDF$ with median vector of v_2

$$TSDF_{local} = calculateTSDF(v, v_2.m, t)$$

// update global $TSDF$ and the counter $v.w$

$$v.d = \frac{TSDF_{local} + v.w * v.d}{v.w + 1}$$

$$v.w = v.w + 1$$

end

To calculate the local $TSDF$ of a voxel v we compute its shortest distance among its eight vertices to its corresponding median vector m of v_2. Because we lose the indication whether v lies behind or in front of the surface when calculating only the shortest distance, we simply apply the sign of the difference between the depth measurement t and the z coordinate of $v.p$ in camera space: $t - p.z$ (p is the transformed point $v.p$ in camera

space, see also Algorithm 3.3).

Algorithm 3.3: Calculating TSDF of voxel with median vector

Function `calculateTSDF` ($voxel \in G$, $median \in \mathbb{R}^3$, $depth \in \mathbb{R}$)**:**

`// transform the position` $voxel.p$ `of the voxel from world space into camera space`

$$p = C_{EST} * \begin{pmatrix} voxel.p.x \\ voxel.p.y \\ voxel.p.z \\ 1 \end{pmatrix}$$

`// calculate the difference of p.z and the depth measurement in camera space to indicate whether the voxel lies in front, behind or on the surface. Signum function takes the sign of the difference (1 or -1. 0 if there is no difference)`

$$sign = signum(depth - p.z)$$

`// calculate and save distances from all vertices of the voxel to the median vector`

$$n = 0$$

$$distances[8]$$

for $i = 0; i \leq 1; i = i + 1$ **do**
  for $j = 0; j \leq 1; j = j + 1$ **do**
    for $k = 0; k \leq 1; k = k + 1$ **do**

$$distances[n] = length(median - \begin{pmatrix} voxel.p.x + i * voxelsize \\ voxel.p.y + j * voxelsize \\ voxel.p.z + z * voxelsize \end{pmatrix})$$

$$n = n + 1$$

    end
  end
end

`// take the minimum distance of all calculated distances`

$$distance_{min} = min(distances)$$

`// apply the sign to the minimum distance`

$$distance_{min} = distance_{min} * sign$$

`// return local` $TSDF$ `of the voxel, calculated with the minimum distance`

return

$$TSDF = \begin{cases} 1, & \text{if } distance_{min} > \mu \\ -1, & \text{if } distance_{min} < -\mu \\ \frac{distance_{min}}{\mu}, & \text{otherwise} \end{cases}$$

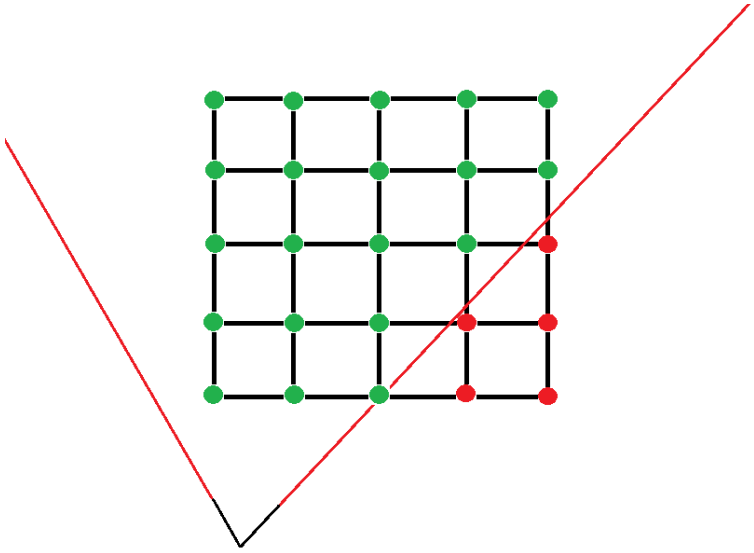


Figure 3.1: Example of noise in 2D grid with voxels simplified as 2D squares. On the grid the vertices are marked as green (distance to surface could be computed) and red dots(distance to surface could not be computed, since it is outside of the camera frustum). The camera is depicted with a black angle and the camera frustum with red lines.

3.3.1 Marching Cubes vs. Adapted Marching Cubes to Octree

We earlier mentioned that we do not want to use only the normal Marching Cubes Algorithm but also the adapted Marching Cubes Algorithm for Octrees [MKH07]. The Marching Cubes for Octrees provides a fairly significant advantage considering noise in the voxel grid. Speaking about noise in the voxel grid means that not every voxel, which was allocated and updated, necessarily have a $TSDF$ at all of its eight vertices. Some vertices might not have any $TSDF$ value due to noise caused by camera movement or general noisy depth images. The following Figures 3.1 and 3.2 should make the problem more clear.

Because the normal Marching Cubes Algorithm can only generate polygons for each voxel independently, it has to ignore those which do not have at every vertex a $TSDF$. However for the Octree, the adapted Marching Cubes Algorithm can also generate polygons for coarser nodes/voxels. Even if the finer voxels of the Octree lack in $TSDF$ information, the coarser voxels containing them might have enough information to generate fairly accurate polygons although it does not has the 100% accuracy of meshing every single voxel independently. That could probably lead to a more watertight, dense and better reconstruction of the surface. The following Figures 3.3 and 3.4 shows the advantage of the Marching Cubes for Octrees more clearly.

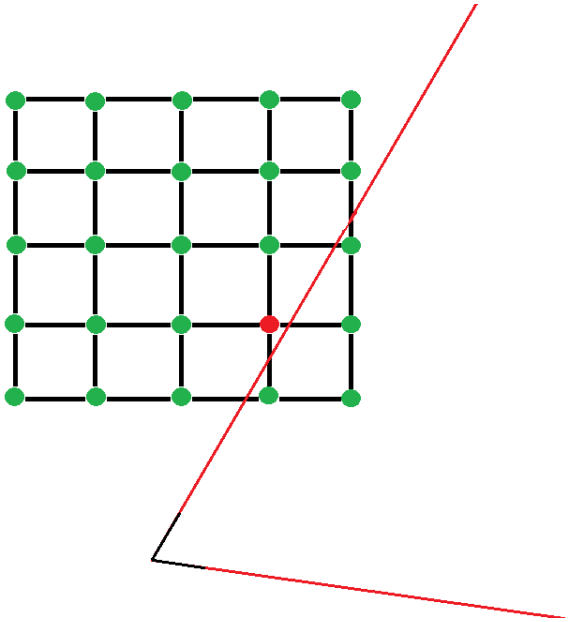


Figure 3.2: Example of noise in 2D grid with voxels simplified as 2D squares. When the camera rotates or translates in the next depth image, it is possible that for some vertices it could still not be computed a distance to the surface because of being outside of the camera frustum or no existing depth measurement at all. This can be caused by camera movement or noisy/grainy depth images.

3.3.2 Conversion - Hashtable to Octree

Since we are updating the $TSDF$ for the voxel grid via the hash table, we need to convert the hash table to an octree to be able to use the adapted Marching Cubes Algorithm. For explaining the conversion the hash table is defined as:

$$Hashtable = \{(vb, p) | vb \subseteq G \wedge |vb| = 512, p \in \mathbb{Z}^3\}$$

It consists of tuples representing the voxel blocks containing a subset vb of G (with 512 voxels) and a voxel coordinate p (which is the indexed position of the voxel block). The octree is defined as

$$Octree = \{(p_0...p_7, d_0...d_7, children) | p_0...p_7 \in [0,1]^3, d_0...d_7 \in CVs, children \subseteq Octree \wedge |children| <= 8\}$$

It consists of tupels representing the octree nodes. Each octree node contains the positions of its eight vertices $p_0...p_7$ with coordinates ranging from 0 to 1, eight TSDF values $d_0...d_7$ which are separately saved in $CVs = \{d | d \in \mathbb{R}\}$ ("CVs" stands for corner values) and children nodes $children$ which are a subset of the $Octree$ itself.

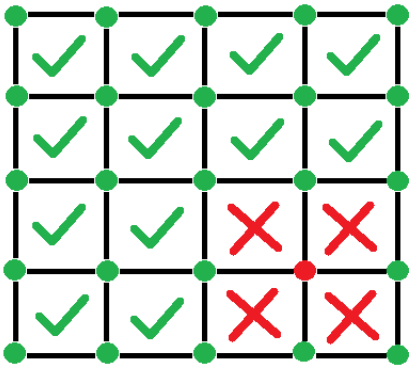


Figure 3.3: Normal Marching Cubes applied to a 2D grid. Same 2D grid as in Figure 3.2. For each voxel, simplified as a 2D square, the Marching Cubes Algorithm is applied independently. However for 4 2D squares a distance for one vertex is missing which makes it not possible to apply Marching Cubes on them. The 2D squares, on which Marching Cubes is appliable, are marked with a check. The 2D squares, on which Marching Cubes is not appliable, are marked with a cross.

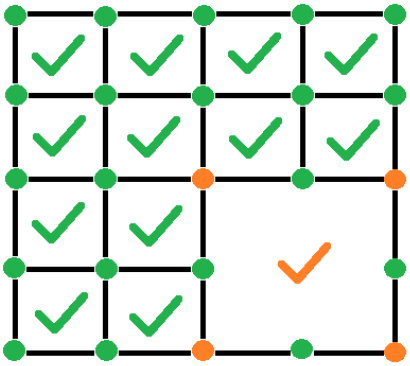


Figure 3.4: Advantage of Marching Cubes for Octrees shown on a quadtree (which is basically an octree in 2D. Every node can have 4 children nodes instead of 8). It is the same 2D grid as in Figure 3.2. For each voxel, the Marching Cubes Algorithm is applied. However the voxels with the vertex without a computed distance can be summarized in the coarser voxel, on which Marching Cubes can be applied. That makes it possible to also mesh areas, in which some vertices might not have any computed distance due to noise.

First we need to find the boundaries or the size of the octree. To do that, we iterate over all voxel blocks of the *Hashtable* and search for the minimum and maximum position. The indexed positions of the voxel blocks always have to be transformed to coordinates indexing the voxels by multiplying them with 8, since one voxel block is 8 voxels wide (e.g. the voxel block at index $(1, 0, 0)$ starts at the indexed voxel coordinate $(8, 0, 0)$). After that we calculate the difference between the minimum and maximum position and take the biggest difference in the x, y or z direction as the width of the octree. It always have to be added 8 to the width since the maximum position is not the real maximum and includes 8 further voxels of the voxel block's width. Since we do not count the number of voxels but the corners of the voxels as the width, the last corner of the last voxel has to be counted as well to the width. Therefore it also needs to be added 1 to the maximum width(See also Algorithm 3.4).

For now the *octreesize* is not necessarily a number equals to $8^i, i \in \mathbb{N}$, therefore we first calculate the *octreedepth* with the *octreesize* by taking the logarithm of $octreesize^3$ to the base 8 and rounding the number up.

$$octreedepth = \lceil log_8(octreesize^3) \rceil$$

Then the correct *octreesize* can be calculated with the *octreedepth* by taking the cube root of the number of voxels (the number of voxels is $8^{octreedepth}$)

$$\sqrt[3]{8^{octreedepth}} = 2^{octreeDepth}$$

After computing the *octreedepth* and *octreesize* we finally convert the hash table by inserting the $TSDF$ values with their position from the voxel grid into the octree. The cardinality of CVs or the total number of corner values can be set to $octreesize^3$. Conversion is done for every voxel block vb by iterating over its 512 voxels with $(x, y, z) \in \{0, 1, ..., 8\}^3$. The position for every voxel is calculated by $vb.p * 8 + (x, y, z)$, which we need to retrieve the voxel data of the hash table. When obtaining the $TSDF$ value for $vb.p * 8 + (x, y, z)$, we still need to convert the voxel position into an octree position, since the corner positions of octree nodes range from 0 to 1. Thus we have to shift the voxel data's position to the origin $(0, 0, 0)$ with (min_x, min_y, min_z) and divide the position by the *octreesize*. For inserting the $TSDF$ value at the octree position, we define a recursive function called "insertPointIntoOctree" which takes as input an octree node (which will be the root node of the octree), the octree position of the $TSDF$, the $TSDF$ value and the *octreeDepth*(See also Algorithm 3.5).

For insertion we recursively traverse through the octree starting from the node *currentNode* (which is the root node at the beginning) and search for a node which has a corner with the octree position *octreePos* of the $TSDF$ value. If the current depth level *depth* (we start at 0) is smaller than *octreeDepth*, we continue to search for a node with the

Algorithm 3.4: Compute boundaries of the hashtable for the octree

```
   // minimum and maximum value along x,y and z direction is
      set to −∞ and ∞
 1 min_x, min_y, min_z = −∞
 2 max_x, max_y, max_z = ∞
   // search for the minimum and maximum position amongst all
      vb in Hashtable
 3 for ∀vb ∈ Hashtable do
       // indexed position of the voxel block vb converted to
          indexed voxel coordinate pos
 4     pos = vb.p * 8
 5     if pos.x < min_x then
 6         min_x = pos.x
 7     end
 8
 9     if pos.y < min_y then
10         min_y = pos.y
11     end
12
13     if pos.z < min_z then
14         min_z = pos.z
15     end
16
17     if pos.x > max_x then
18         max_x = pos.x
19     end
20     ;
21     if pos.y > max_y then
22         max_y = pos.y
23     end
24     ;
25     if pos.z > max_z then
26         max_z = pos.z
27     end
28     ;
29 end
   // select the maximum difference in x,y or z direction as
      the width or size of the octree
30 width_x = max_x − min_x
31 width_y = max_y − min_y
32 width_z = max_z − min_z
33 octreesize = max(max(width_x, width_y), width_z) + 8 + 1
```

Algorithm 3.5: Insert all points from hashtable into the octree

```
   // root node of the octree
1  rootNode ∈ Octree
   // insert all points of all voxel blocks into the octree
2  for ∀vb ∈ Hashtable do
       // indexed position of the voxel block converted to
          indexed voxel coordinate pos
3      pos = vb.p * 8
       // iterate over all voxels in vb
4      for x = 0; x ≤ 8; x++ do
5          for y = 0; y ≤ 8; y++ do
6              for z = 0; z ≤ 8; z++ do
                   // indexed position of a voxel in vb
7                  voxelpos = pos + (x, y, z)^T
                   // voxel v ∈ G
8                  v = Hash(voxelpos)
                   // TSDF value of v
9                  TSDF = v.d
                   // converting the indexed voxel position to an
                      octree position
10                 octreePos = (voxelpos − (min_x, min_y, min_z)^T) / octreesize
                   // insert TSDF value with the octree position
                      into the octree
11                 insertPointIntoOctree(rootNode, octreePos, TSDF, 0)
12             end
13         end
14     end
15 end
```

octreePos. If the *octreePos* is inside or along an edge of the *currentNode*, we know that the *octreePos* has to be in one of the children nodes of the *currentNode*. Therefore we pass the *octreePos* into one of the children nodes, which contains the *octreePos*. That is done by recursively calling "insertPointIntoOctree" with the according children node of *currentNode.children*, the *octreePos*, the $TSDF$ value and an increased depth level of $depth + 1$. If the *octreePos* is on a corner of the *currentNode*, we search for the corner which has the same position as *octreePos*. The $TSDF$ value is inserted for that corner of the *currentNode*. If the *octreePos* is outside of the *currentNode*, we can stop searching(See also Algorithm 3.6).

Algorithm 3.6: Insert one point into the octree

```
1  Function insertPointIntoOctree(octreeNode ∈ Octree,
     position ∈ [0,1]^3, TSDF ∈ ℝ, currentDepth ∈ ℕ):
2      if depth > octreeDepth − 1 then
3          return
4      end
5      if position inside or along an edge of octreeNode then
6          for i = 0; i < 8; i++ do
               // pass parameters to the children node which
                  contains octreePos
7              if position not outside of children octreeNode.children[i] then
8                  initialize octreeNode.children[i]
9                  insertPointIntoOctree(octreeNode.children[i], position, TSDF, depth+
                    1)
10                 break
11             end
12         end
13     end
14     else if position on a corner of octreeNode then
           // assign TSDF value to the correct corner of
              currentNode
15         for i = 0; i < 8; i++ do
16             if octreeNode.p_i == position then
17                 octreeNode.d_i = TSDF
18                 break
19             end
20         end
21     end
22     else
23         position outside of octreeNode
24     end
25     return
```

CHAPTER 4

Evaluation

The purpose of the evaluation is to ensure the same reconstruction quality with our new integration method as with the old integration method, so it is reliable for further development of the surface operator using the median and variance vectors for marking vertices as inside or outside. Therefore we want to show the difference between the old and new integration methods as well as the normal and Octree Marching Cubes Algorithm. The setup for the comparison are three different scenes, in which we scan a room with the Kinect v2 sensor. In the first scene the room has only easy geometric shapes, e.g. a couch (See also Figure 4.1). In the second scene, we add more complex geometric shapes, e.g. a table (See also Figure 4.2). In the third scene more complex and geometrically finer shapes are added, which are also harder to reconstruct (See also Figure 4.3).

Considering that we generate the models with different combinations of integration/meshing methods and want to compare all of them with the basic model (the mesh generated with the old integration and normal Marching Cubes Algorithm), we have 3 comparisons for each scene:

1. Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / normal Marching Cubes

2. Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / Marching Cubes for Octrees

3. Mesh with old integration / normal Marching Cubes vs. Mesh with old integra-

Figure 4.1: RGB Image Scene 1. Only a couch in the scene.

Figure 4.2: RGB Image Scene 2. A twisted table to add complexity to the scene.

Figure 4.3: RGB Image Scene 3. Finer objects are now placed on the table to add more complexity

tion / Marching Cubes for Octrees

4.0.1 Hausdorff Distance

When comparing two meshes, a value is required to be able to measure the difference between them. For that reason we use the directed Hausdorff Distance[Ruc96]. It is a simple algorithm which measures the distance between a target point set A and a source point set B. That means the difference from B to A will be computed. To put it simply the directed Hausdorff Distance is very small if a lot of points in B have a small distance to a point in A. Referring that to meshes the point sets are represented by the vertices of the polygons.

For calculation a help function

$$D(x, K) = min\{d(x, k)|k \in K\}$$

is used which calculates the minimum distance from a point x to a point set $K \in \mathbb{R}^3$ with a defined metric

$$d(a, b) = sqrt((a.x - b.x)^2 + (a.y - b.y)^2 + (a.z - b.z)^2),\ a, b \in \mathbb{R}^3$$

The directed Hausdorff Distance from B to A can be calculated with,

$$hausdorff_{max}(A, B) = max\{D(b, A)|b \in B\}$$

which is the maximum value of the point's minimum distances from B to A[Ruc96].

Because taking the maximum distance is heavily affected by outlier points and probably distorts the real difference between A and B, we rely more on other values like the mean or RMS (Root Mean Square)[Bir19]. For the mean the average of the point's minimum distances is calculated instead of the maximum value.

$$hausdorff_{mean}(A, B) = avg\{D(b, A)|b \in B\}$$

The RMS uses a slightly different metric,

$$d(a, b) = (a.x - b.x)^2 + (a.y - b.y)^2 + (a.z - b.z),\ a, b \in \mathbb{R}^3$$

, which is the squared distance between a and b. Then the average over the squared minimum distances of the points from B to A is computed. The final Hausdorff Distance as RMS is the square root of the average.

$$hausdorff_{RMS}(A, B) = \sqrt{avg\{D(b, A)|b \in B\}}$$

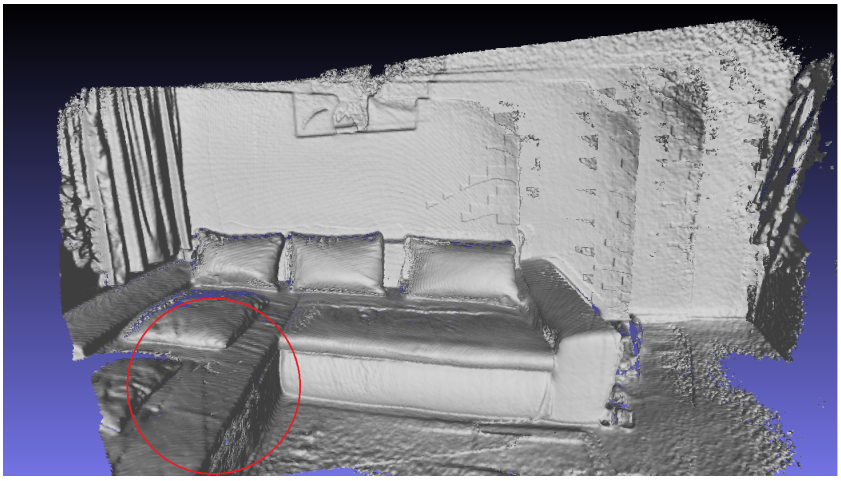
In Figure 4.21 are all directed Hausdorff Distances as RMS depicted (the target point set is always the original mesh and the source point sets are all new meshes generated with the new integration and/or Marching Cubes for Octrees).

4.0.2 Meshing - only with Marching Cubes

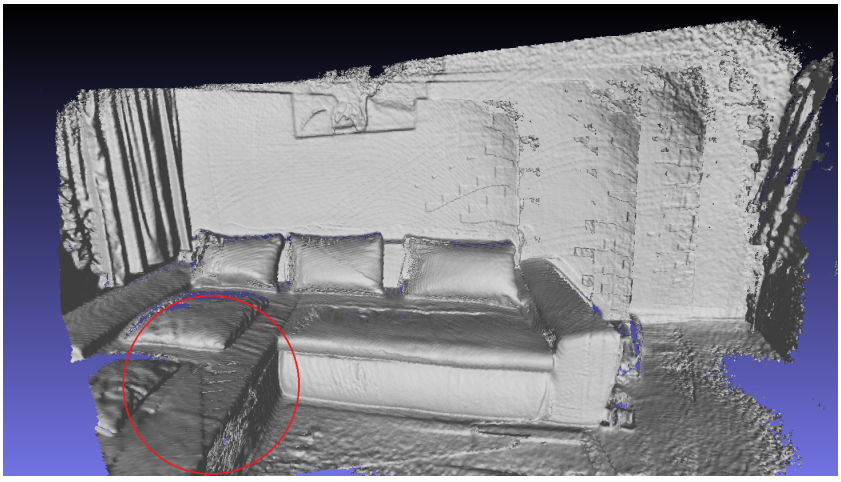
The results for the comparisons between the old and new integration while meshes are only generated with the normal Marching Cubes Algorithm are shown below in Figure 4.4 to 4.7 (First comparison case for every scene, which was previously mentioned in "Evaluation"):

4.0.3 Meshing - with Marching Cubes and Adapted Marching Cubes to Octree

The results for the comparisons of the old and new integration while the original model is meshed with normal Marching Cubes and the other ones are meshed with Marching Cubes for Octrees are shown below in Figure 4.8 to 4.20 (second and third case for every scene, which were previously mentioned in "Evaluation"):

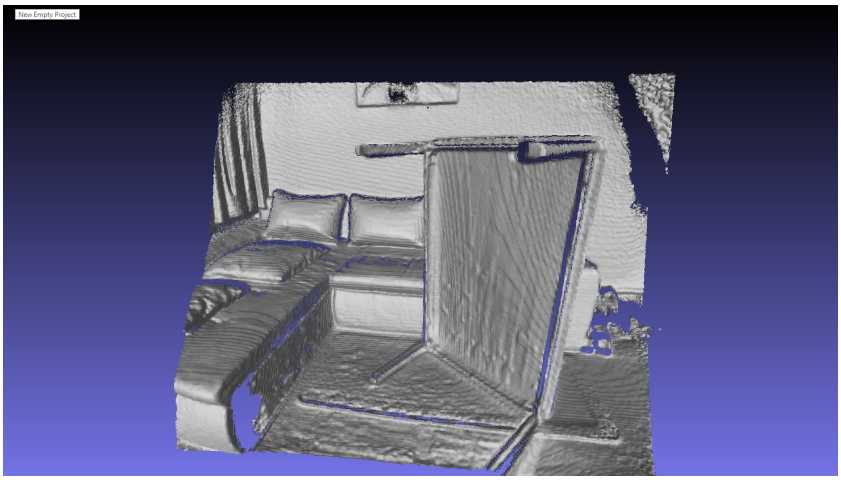


(a) Mesh with old integration / normal Marching Cubes

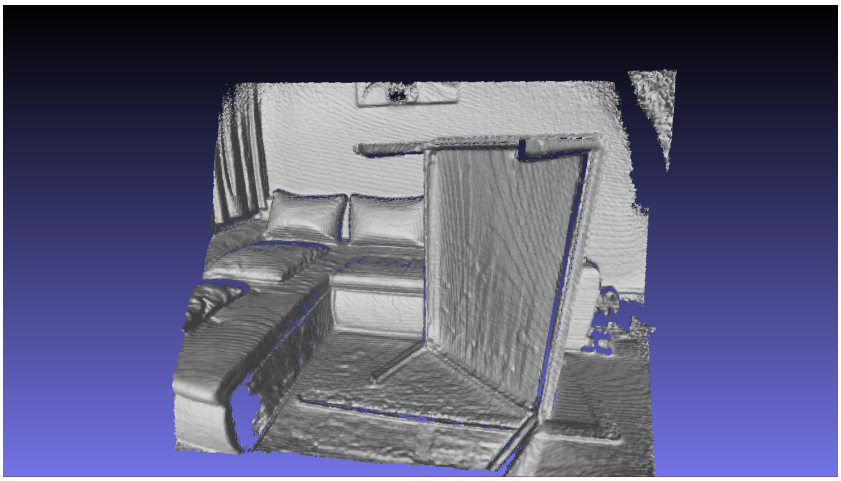


(b) Mesh with new integration / normal Marching Cubes

Figure 4.4: Scene 1 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / normal Marching Cubes. Almost no visible difference between the old and new integration while both meshes are generated with normal Marching Cubes. Minor dissimilarities can be spotted on the left bottom of the image in the red marked area. The RMS of the directed Hausdorff Distance is 5.8 mm.

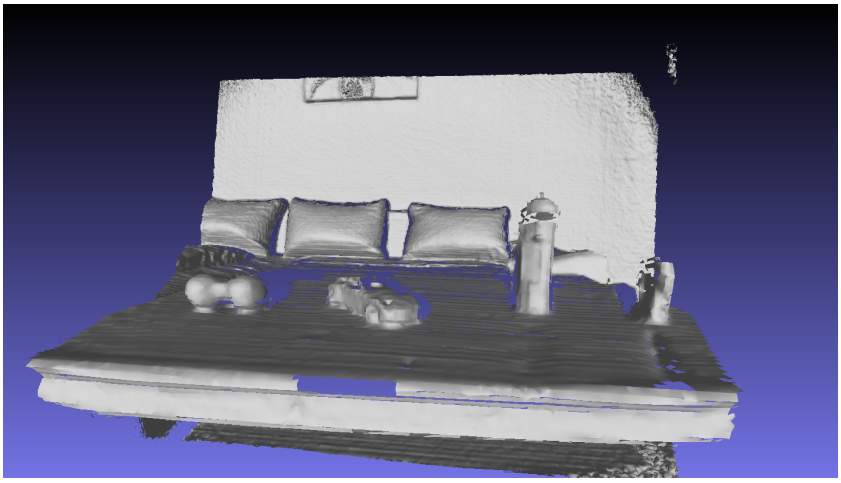


(a) Mesh with old integration / normal Marching Cubes

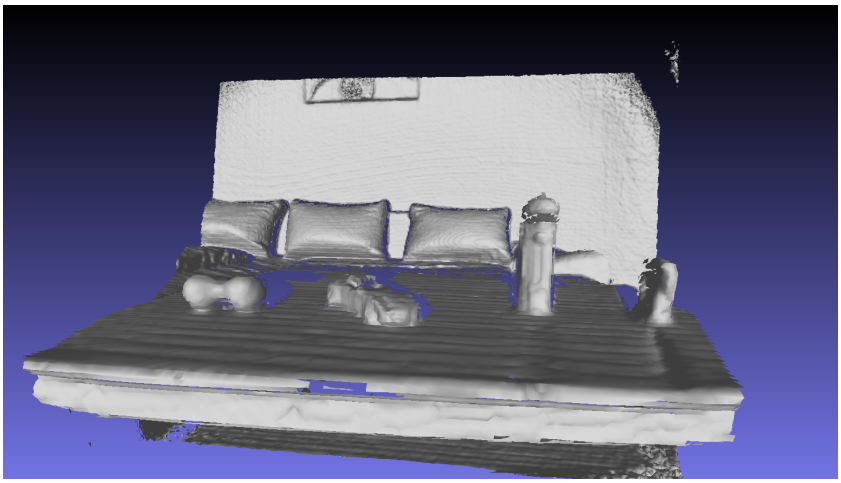


(b) Mesh with new integration / normal Marching Cubes

Figure 4.5: Scene 2 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / normal Marching Cubes. No visible difference between the old and new integration, although the RMS of the directed Hausdorff Distance with 6.02 mm points to small dissimilarities. Differences are probably located in much smaller areas and more scattered around the whole scene compared to Scene 1.

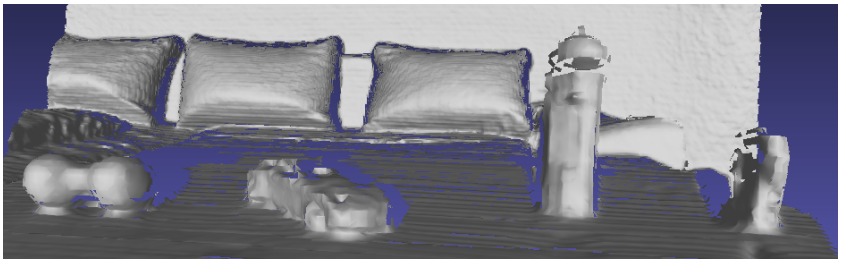


(a) Mesh with old integration / normal Marching Cubes

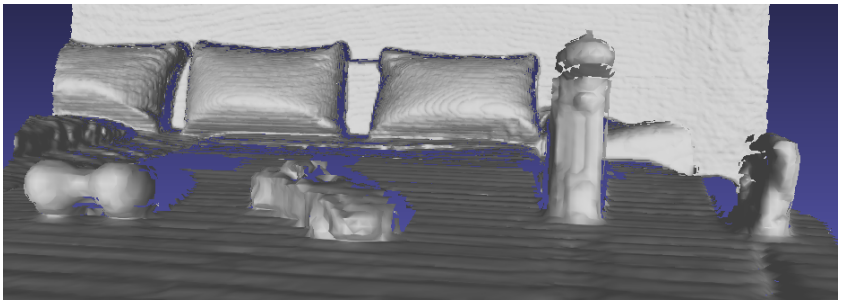


(b) Mesh with new integration / normal Marching Cubes

Figure 4.6: Scene 3 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / normal Marching Cubes. Clear visible differences between the old and new integration considering the finer objects on the table (fascia roll on the left, car in the left middle, bottle on the right middle and a can on the right. It is slightly noticeable that the surface of those objects for the new integration are less edgy, fuller and more watertight. Especially the engine cover of the car seems to be more close to the real car than with the old integration. However the wheels appears to be more washed-out. In this scene the bigger differences are also represented in the RMS of the Hausdorff Distance with 6.57 mm.)



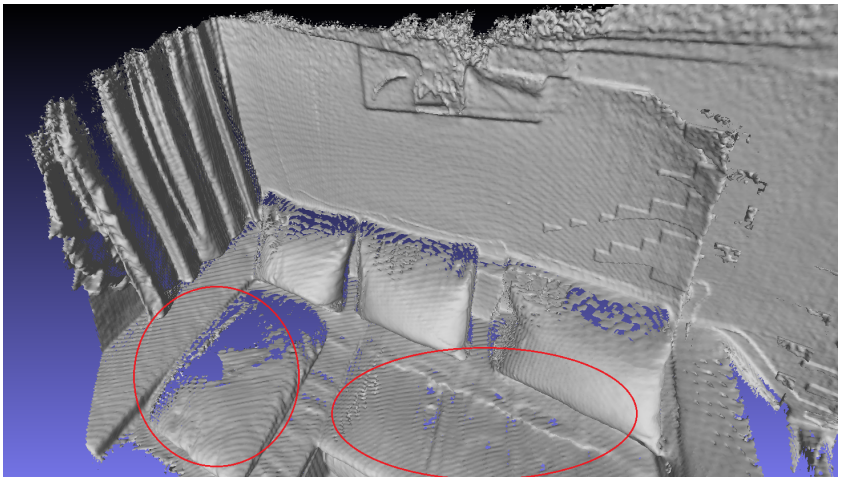
(a) Mesh with old integration / normal Marching Cubes



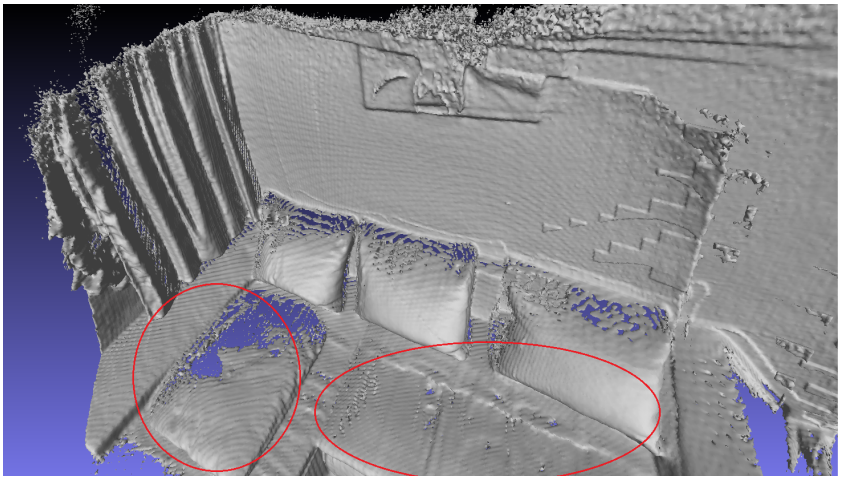
(b) Mesh with new integration / normal Marching Cubes

(c) Mesh with new integration / normal Marching Cubes

Figure 4.7: Scene 3 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / normal Marching Cubes. A Close-Up of Figure 4.6 to compare the objects on the table. Especially the reconstruction of the car body and engine cover seems to be more similar to the real world object in the new integration. Also note the bottle in the right middle and the can on the right, which have a more watertight reconstruction in the new integration.

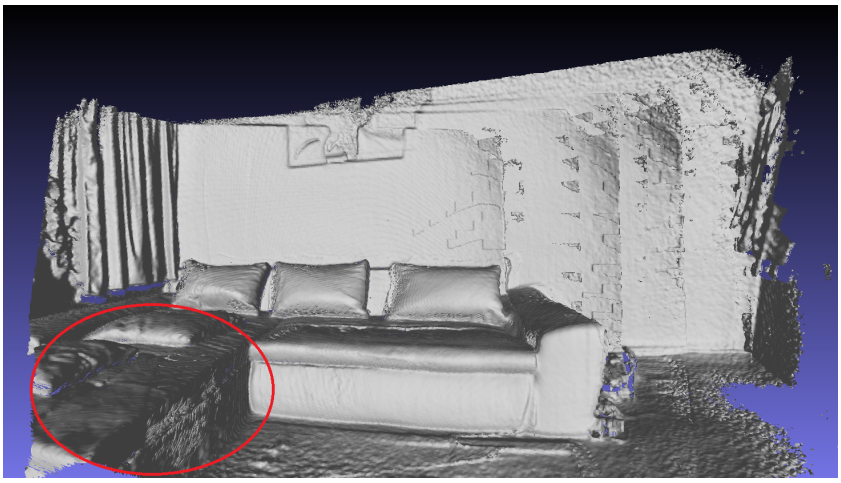


(a) Mesh with old integration / normal Marching Cubes



(b) Mesh with old integration / Marching Cubes for Octrees

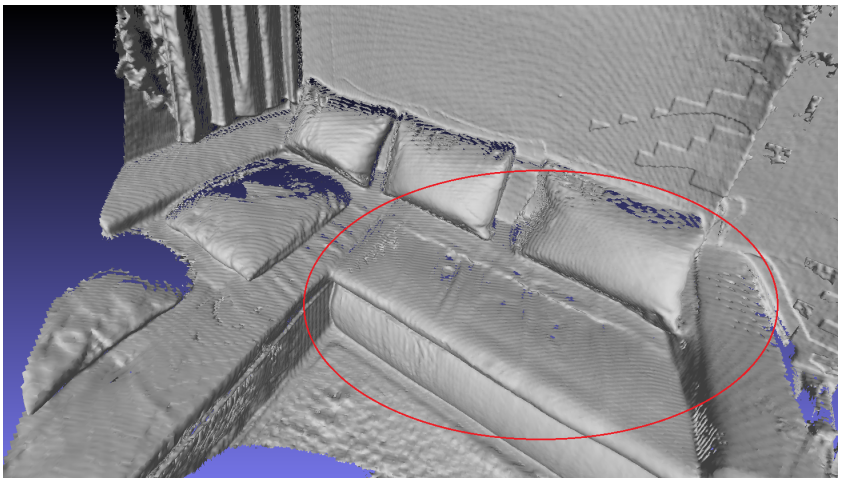
Figure 4.8: Scene 1 - Mesh with old integration / normal Marching Cubes vs. Mesh with old integration / Marching Cubes for Octrees. The advantage of the Marching Cubes for Octrees is already well perceptible. The smaller and bigger holes in the red marked areas close up and make the mesh more watertight. The RMS of the directed Hausdorff Distance is 42.16 mm.



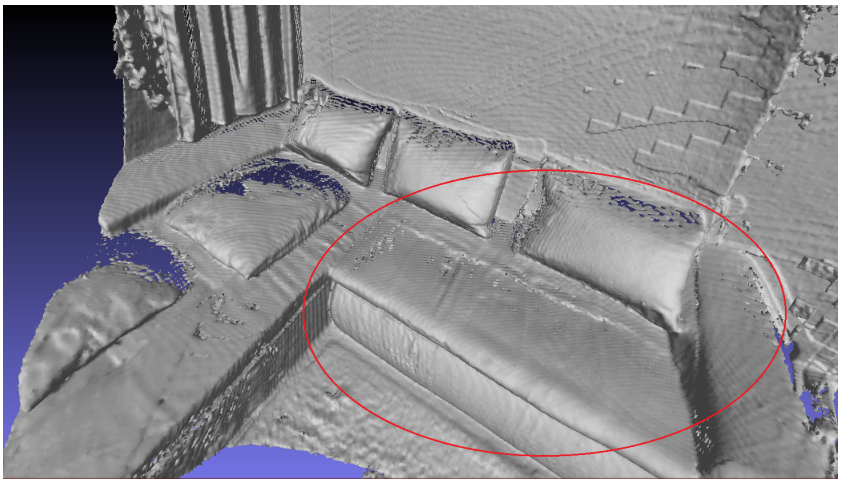
(a) Mesh with old integration / normal Marching Cubes

(b) Mesh with old integration / Marching Cubes for Octrees

Figure 4.9: Scene 1 - Mesh with old integration / normal Marching Cubes vs. Mesh with old integration / Marching Cubes for Octrees. Looking from another perspective there is also a significant improvement on the left side of the couch in the red marked area. There are significantly less holes with Marching Cubes for Octrees. The RMS of the directed Hausdorff Distance with 42.16 mm represents those differences as well.



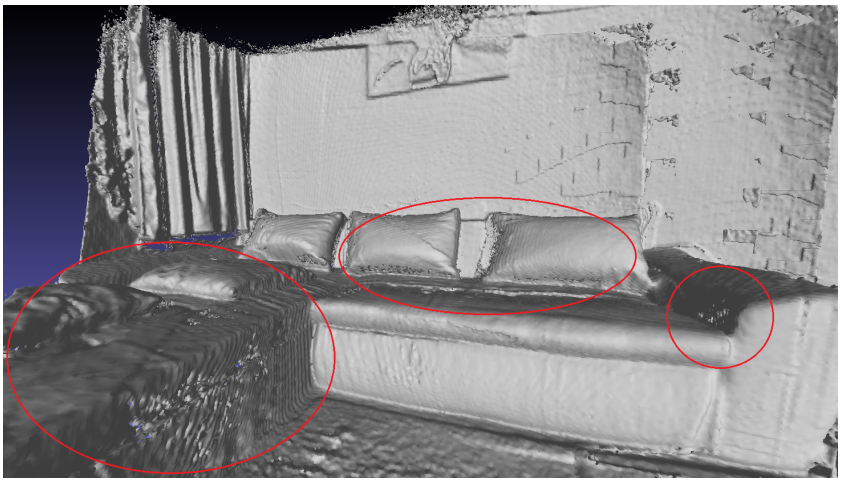
(a) Mesh with old integration / normal Marching Cubes



(b) Mesh with new integration / Marching Cubes for Octrees

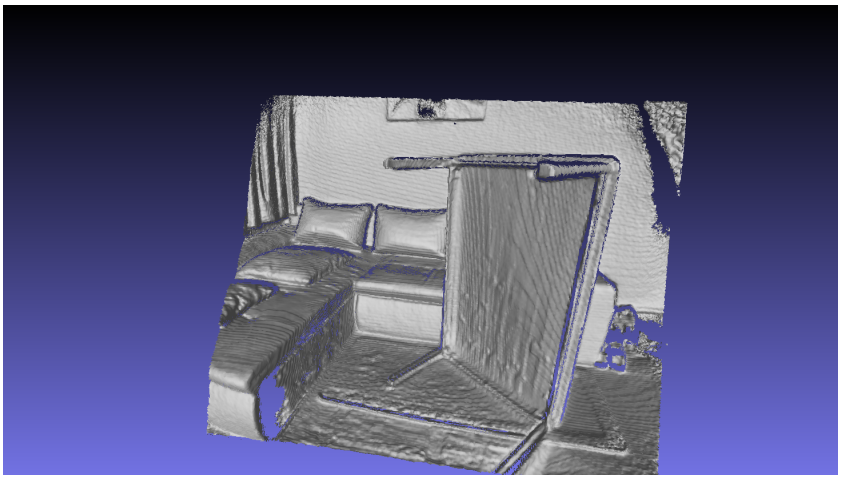
Figure 4.10: Scene 1 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / Marching Cubes for Octrees. When not only using Marching Cubes for Octrees but also the new integration, there are even less holes visible on the couch in the red marked area. Compared to the last case in which both meshes have the old integration, the RMS of the directed Hausdorff Distance is now slightly higher with 42.67 mm.

(a) Mesh with old integration / normal Marching Cubes

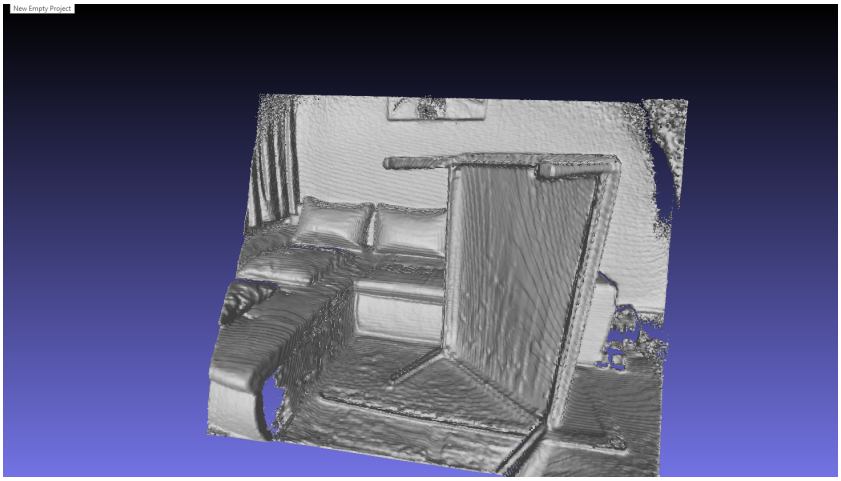


(b) Mesh with new integration / Marching Cubes for Octrees

Figure 4.11: Scene 1 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / Marching Cubes for Octrees. The same mentioned before applies from this perspective. There are also less holes along the edge of the couch in the red marked area. The RMS of the directed Hausdorff Distance is 42.67 mm.

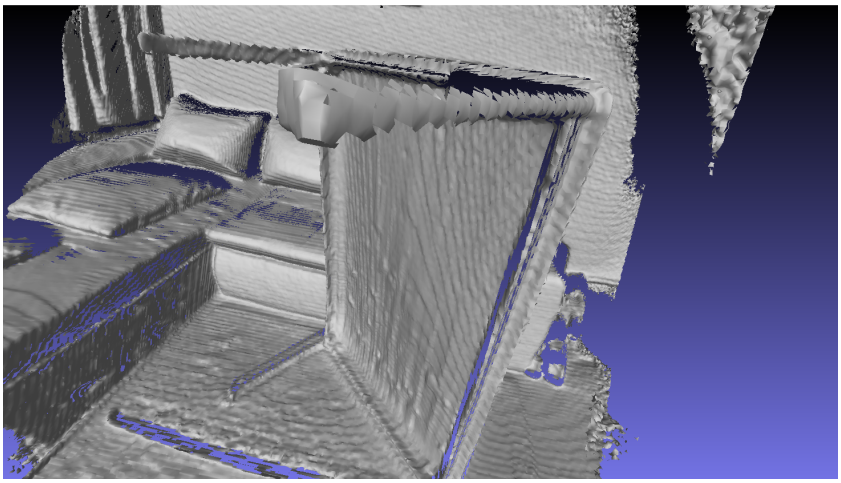


(a) Mesh with old integration / normal Marching Cubes

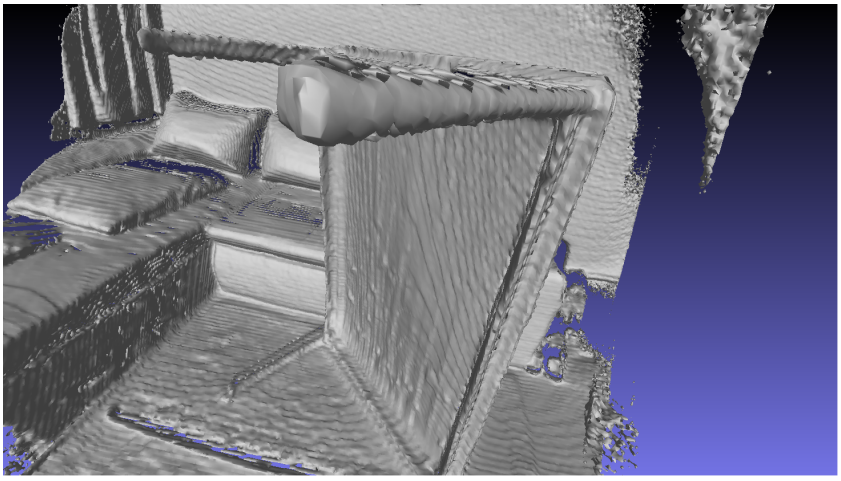


(b) Mesh with old integration / Marching Cubes for Octrees

Figure 4.12: Scene 2 - Mesh with old integration / normal Marching Cubes vs. Mesh with old integration / Marching Cubes for Octrees. Only applying Marching Cubes for Octrees already provides an obviously less holey mesh especially along the edges of the table and on multiple areas of the couch in the background. The RMS of the directed Hausdorff Distance is 19.31 mm.

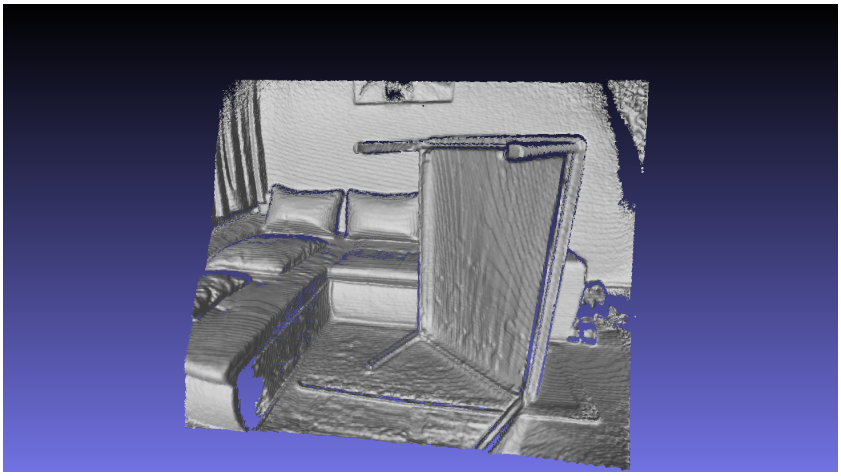


(a) Mesh with old integration / normal Marching Cubes

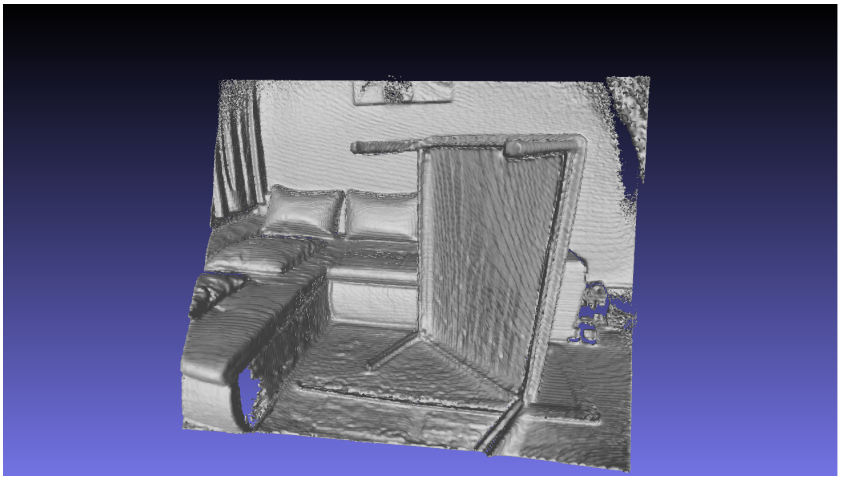


(b) Mesh with old integration / Marching Cubes for Octrees

Figure 4.13: Scene 2 - Mesh with old integration / normal Marching Cubes vs. Mesh with old integration / Marching Cubes for Octrees. From a closer perspective It is very clear to see that the legs of the table and also the couch in the background have a more watertight and full reconstruction. However compared to Scene 1 the RMS of the directed Hausdorff Distance is lower with 19.31 mm.

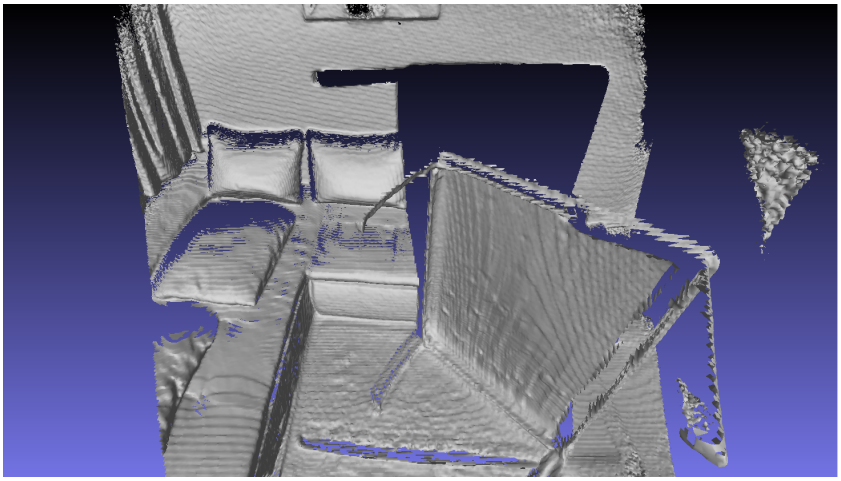


(a) Mesh with old integration / normal Marching Cubes

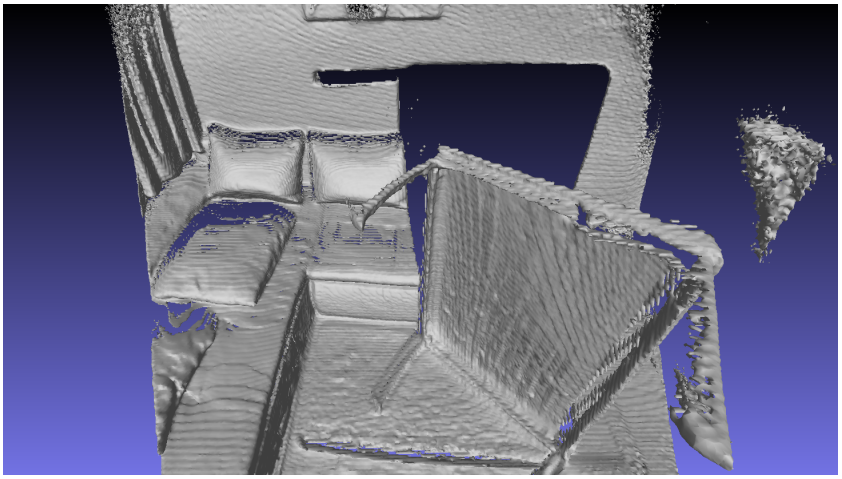


(b) Mesh with new integration / Marching Cubes for Octrees

Figure 4.14: Scene 2 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / Marching Cubes for Octrees. Using also the new integration the table does not change much in quality compared to only using Marching Cubes for Octrees. The RMS of the directed Hausdorff Distance is 19.76 mm.

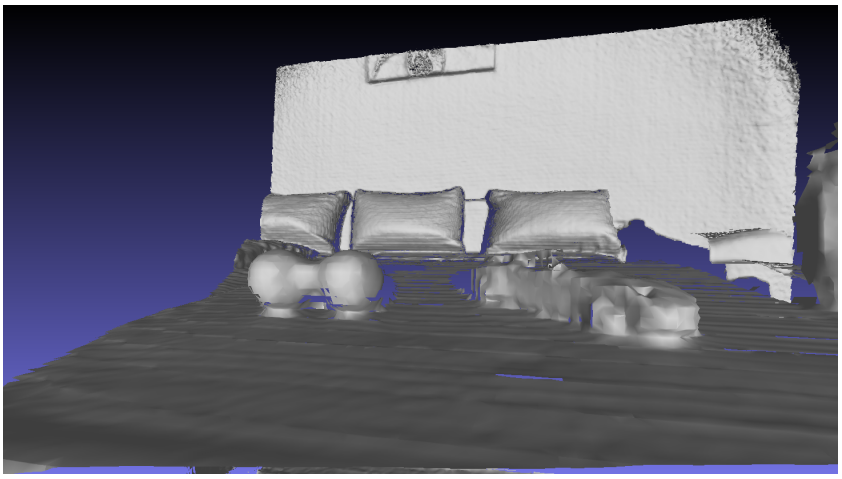


(a) Mesh with old integration / normal Marching Cubes

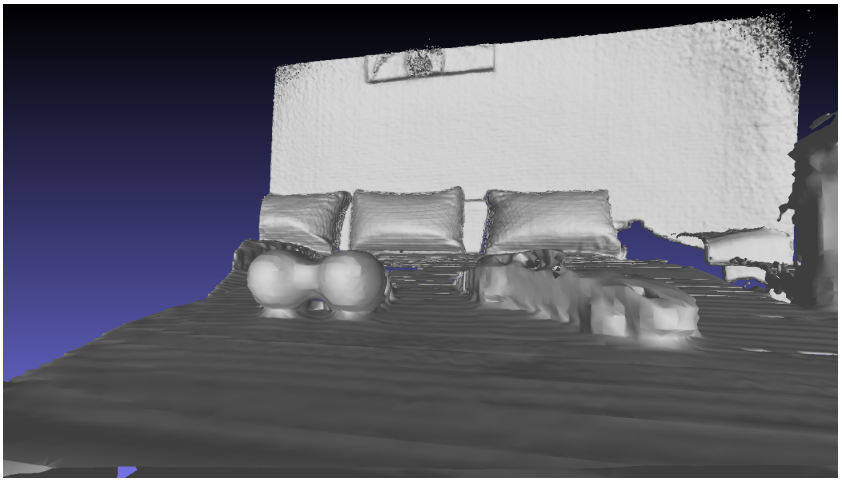


(b) Mesh with new integration / Marching Cubes for Octrees

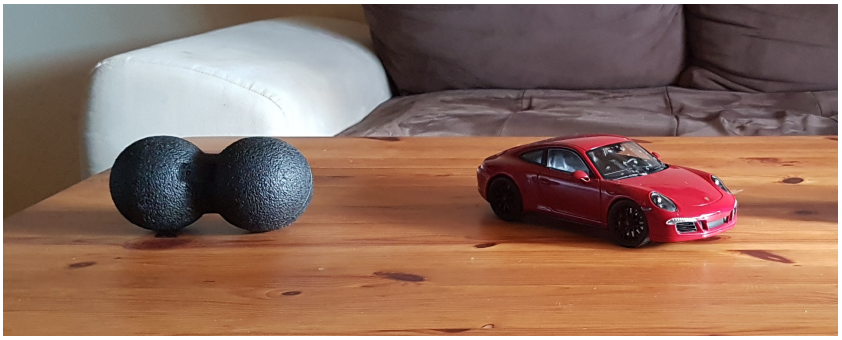
Figure 4.15: Scene 2 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / Marching Cubes for Octrees. Changing the perspective also shows that the original mesh has big holes along the edges of the table and on multiple areas of the couch contrary to the mesh with the new integration and Marching Cubes for Octrees. The RMS of the directed Hausdorff Distance is 19.76 mm.



(a) Mesh with old integration / normal Marching Cubes

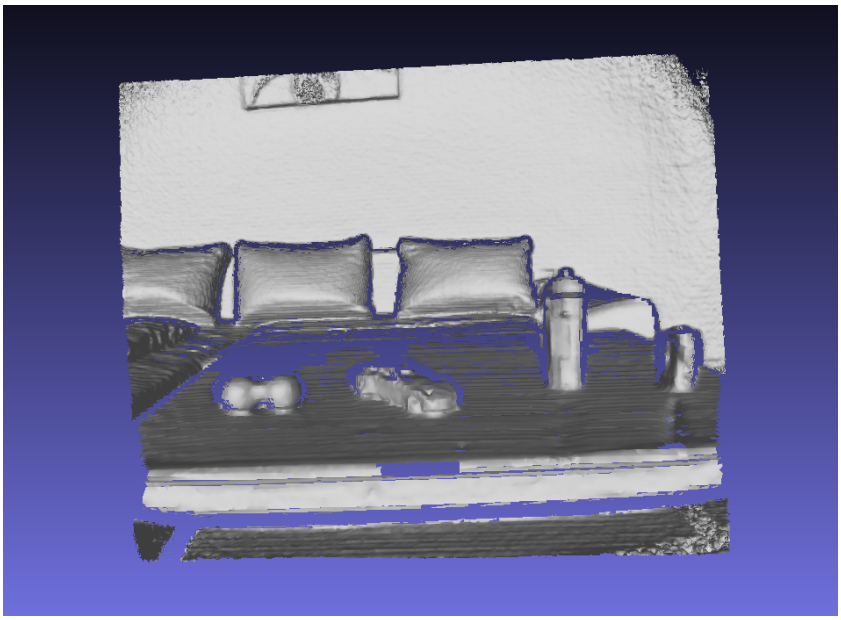


(b) Mesh with old integration / Marching Cubes for Octrees

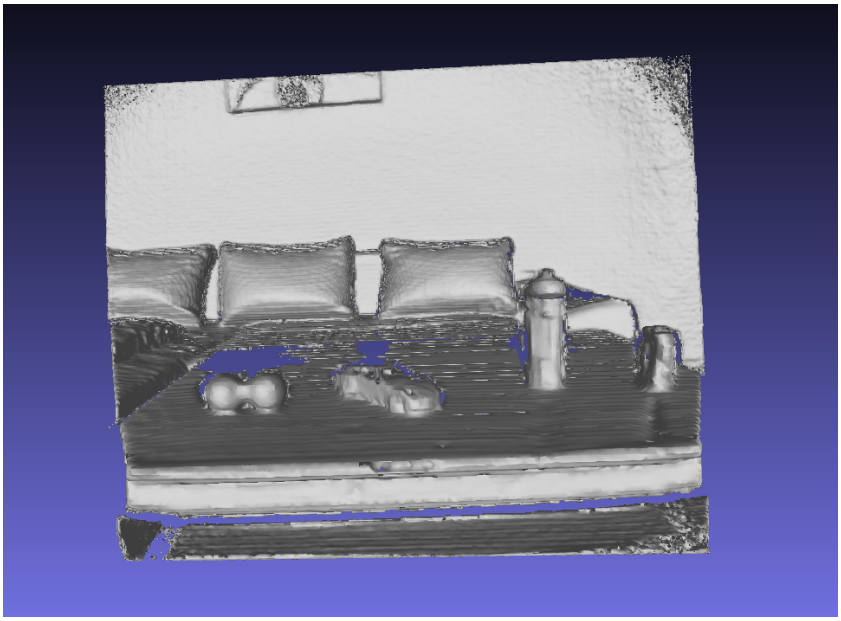


(c) Mesh with old integration / Marching Cubes for Octrees

Figure 4.16: Scene 3 - Mesh with old integration / normal Marching Cubes vs. Mesh with old integration / Marching Cubes for Octrees. The table as well as the car and the fascia roll have a more watertight surface only using Marching Cubes for Octrees. The RMS of the directed Hausdorff Distance is 35.63 mm.

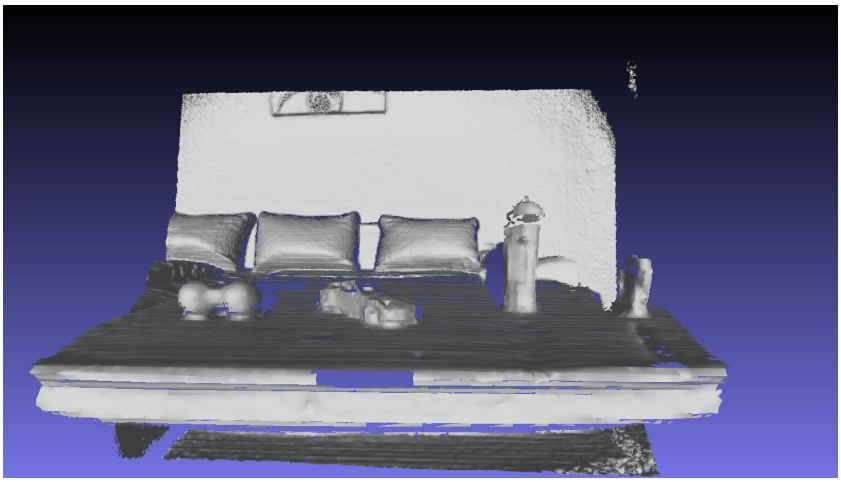


(a) Mesh with old integration / normal Marching Cubes

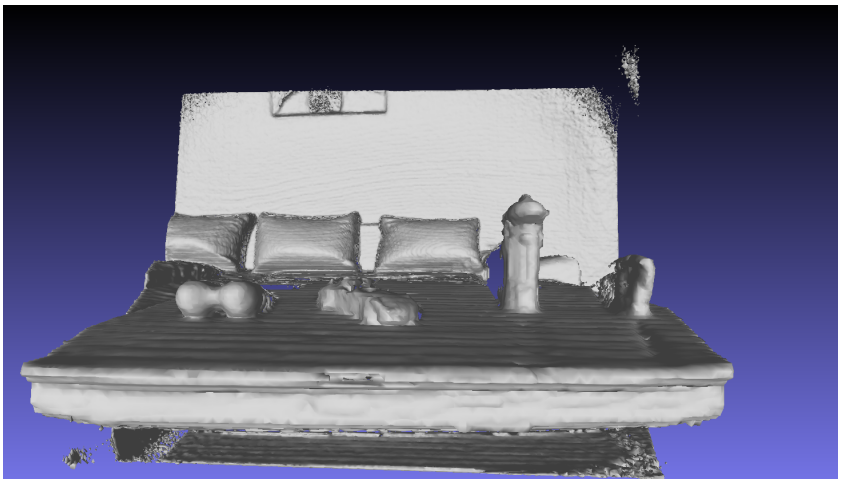


(b) Mesh with old integration / Marching Cubes for Octrees

Figure 4.17: Scene 3 - Mesh with old integration / normal Marching Cubes vs. Mesh with old integration / Marching Cubes for Octrees. Showing the whole scene with four small objects standing on a table (fascia roll on the left, car in the left middle, bottle in the right middle and a can on the right), Marching Cubes for Octrees generates a mesh with significant less holes around especially around the edges of objects. The RMS of the directed Hausdorff Distance is 35.63 mm.

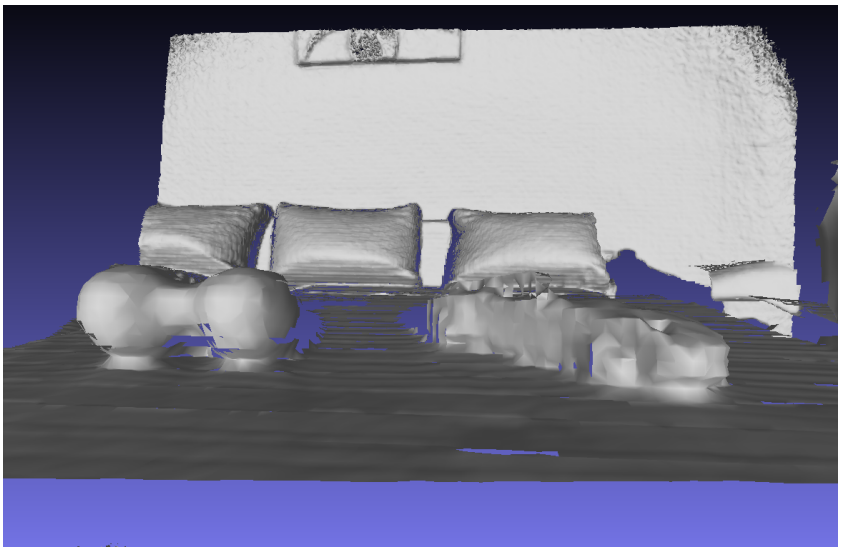


(a) Mesh with old integration / normal Marching Cubes

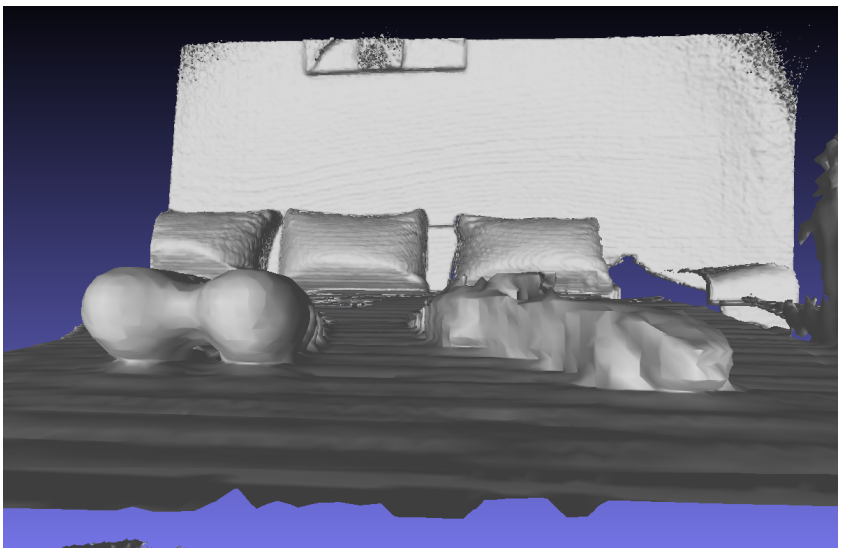


(b) Mesh with new integration / Marching Cubes for Octrees

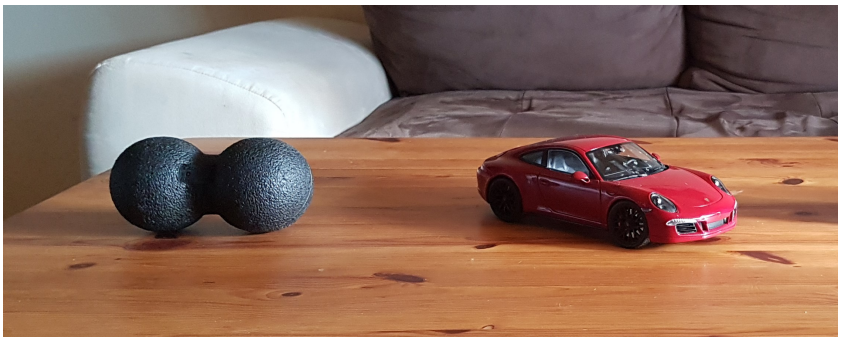
Figure 4.18: Scene 3 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / Marching Cubes for Octrees. Using also the new integration along with Marching Cubes for Octrees provides even less holes in the mesh compared to using the old integration. The RMS of the directed Hausdorff Distance has not changed much with 34.51 mm.



(a) Mesh with old integration / normal Marching Cubes



(b) Mesh with new integration / Marching Cubes for Octrees



(c) Mesh with old integration / Marching Cubes for Octrees

Figure 4.19: Scene 3 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / Marching Cubes for Octrees. A closer perspective to the fascia roll and the car. The fascia roll on the left and the car on the right have a much more watertight reconstruction with the new integration and Marching Cubes for Octrees. However there are also more washed out areas like the wheels on the car with the new integration and Marching Cubes for Octrees. The RMS of the directed Hausdorff Distance is 34.51 mm.

(a) Mesh with old integration / normal Marching Cubes

(b) Mesh with new integration / Marching Cubes for Octrees

(c) RGB image

Figure 4.20: Scene 3 - Mesh with old integration / normal Marching Cubes vs. Mesh with new integration / Marching Cubes for Octrees. A direct comparison between the original mesh of the bottle and the mesh generated with new integration and Marching Cubes for Octrees. The bottle generated with the new integration and Marching Cubes for Octrees has a much better reconstruction in terms of watertightness and detail (especially on the top part of the bottle)

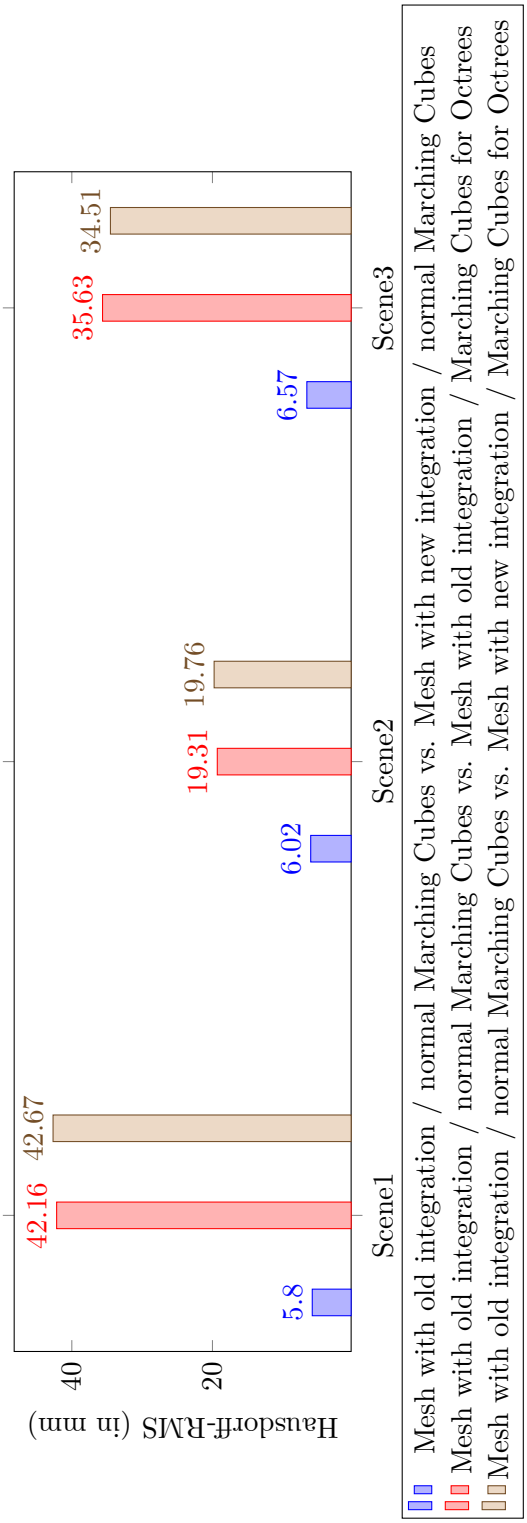


Figure 4.21: Directed Hausdorff Distance as RMS between the original method with old integration/normal Marching Cubes and all new methods for every scene.

4.1 Comparison of Time and Memory Consumption

4.1.1 Time Consumption

The differences of time consumption between the old and new integration as well as the Marching Cubes Algorithm and the adapted Marching Cubes Algorithm are listed and described below in Table 4.1 and 4.2:

Scene	Integration	Normal Marching Cubes	Marching Cubes for Octrees
1	old	1.29s	54.78s
	new	1.13s	54.86s
2	old	0.56s	19.28s
	new	0.65s	19.52s
3	old	0.53s	12.67s
	new	0.53s	12.71s

Table 4.1: Time in seconds to generate a mesh with the new/old integration and normal Marching Cubes/Marching Cubes for Octrees. Marching Cubes for Octrees takes much longer than normal Marching Cubes to generate a mesh. There is little to no difference between the old and new integration.

Old Integration	New Integration
350ms	470ms

Table 4.2: Average time in milliseconds to finish one update cycle with the old/new integration for computing the surface. Please note that the computation time for integration is heavily dependent on what scenery is scanned by the camera and all numbers are specifically evaluated with our scenes.

4.1.2 Memory Consumption

The differences of memory consumption between the old and new integration as well as the Marching Cubes Algorithm and the adapted Marching Cubes Algorithm are listed and described below in Table 4.3. The memory consumption is evaluated by the amount of vertices and faces, which needs to be saved for a mesh. There is only a slight difference between the old and new integration in memory consumption considering the voxel grid. For the new integration a voxel has an additional median vector (which are three floating numbers - $3 * 4$ bytes) and a counter (e.g. as an integer - 4 bytes) which tracks the number of inside lying depth measurements. That makes a difference of 16 bytes for every single voxel.

Scene	Integration	Normal Marching Cubes	Marching Cubes for Octrees
1	old	vertices: 726.348	vertices: 1.487.126
		faces: 1.322.134	faces: 2.865.562
	new	vertices: 778.160	vertices: 1.503.044
		faces: 1.370.599	faces: 2.893.052
2	old	vertices: 315.520	vertices: 751.040
		faces: 566.688	faces: 1.457.352
	new	vertices: 356.354	vertices: 764.716
		faces: 605.301	faces: 1.482.932
3	old	vertices: 172.389	vertices: 451.248
		faces: 305.205	faces: 875.688
	new	vertices: 200.346	vertices: 464.940
		faces: 334.104	faces: 901.137

Table 4.3: Memory consumption evaluated by the amount of vertices and faces generated for the old/new integration and normal Marching Cubes/Marching Cubes for Octrees. Marching Cubes for Octrees creates significantly more vertices and faces than the normal Marching Cubes Algorithm. The new integration only generates slightly more vertices and faces than the old integration.

CHAPTER 5

Conclusion

All in all the results of the evaluation are promising considering the quality outcome and the future work for the new surface operator mentioned in previous chapters. The comparison of the meshes shows that the new integration method has no bad influence in terms of quality compared to the old integration. It rather highlights some improvements considering watertightness and smoothness of the meshes. When generating meshes with the new integration there are slightly less holes in some areas of the surface. Furthermore finer objects (like the engine cover of the car in Scene 3) appear to be a little bit smoother and more similar to the real world objects. To the contrary, the new integration also makes some details more washed out (like the wheels of the car in Scene 3). The results for the directed Hausdorff Distance as RMS also reflects with its low values (only a few millimeters) that there is only a slight difference between the old and new integration. Comparing the normal Marching Cubes with the advanced Marching Cubes for Octrees, the differences between the meshes become noticeably larger, which is also represented by the higher RMS of the directed Hausdorff Distance lying between 10 to 50 mm. Marching Cubes for Octrees provides the best improvement considering the watertightness of the mesh. Combining Marching Cubes for Octrees with the new integration even amplifies that effect. Taking all these aspects into account, the new integration as well as the combination with the adapted Marching Cubes for Octrees has no quality deterioration at all and provides a promising foundation for the new surface operator.

List of Figures

List of Tables

List of Algorithms

Bibliography

[2DM] Voxel to mesh conversion: Marching cube algorithm. https://medium.com/zeg-ai/voxel-to-mesh-conversion-marching-cube-algorithm-43dbb0801359. Accessed: 2021-12-01.

[ABR12] Rajeswari A Arumugam, B. Bhuvaneshwari, and Gowripriyaa Rajkumar. Depth measurement and 3d reconstruction of stereo images. 05 2012.

[Bi20] Xin Bi. *LiDAR Technology*, pages 67–103. 10 2020.

[Bir19] John Bird. *Root mean square values*, pages 357–359. 10 2019.

[CSVV11] Ligia Chiorean, Teodora Szasz, Mircea Vaida, and Alin Voina. 3d reconstruction and volume computing in medical imaging. *Acta Technica Napocensis Electronics and Telecommunications*, 52, 09 2011.

[Dep] From depth map to point cloud. https://medium.com/yodayoda/from-depth-map-to-point-cloud-7473721d3f. Accessed: 2021-12-01.

[Kar20] Elshad Karimov. *Hash Table*, pages 55–60. 03 2020.

[Kin] Understanding real time 3d reconstruction and kinectfusion. https://itnext.io/understanding-real-time-3d-reconstruction-and-kinectfusion-33d61d1cd402. Accessed: 2021-12-01.

[LC87] William Lorensen and Harvey Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21:163–, 08 1987.

[LHG17] Mark Laprairie, Howard Hamilton, and Andrew Geiger. Icvg : Practical constructive volume geometry for indirect visualization. *International Journal of Computer Graphics and Animation*, 7:1–19, 10 2017.

[Low04] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration. 2004.

[MKH07] Ketan Dalal Michael Kazhdan, Allison Klein and Hugues Hoppe. Unconstrained isosurface extraction on arbitrary octrees. *SGP '07: Proceedings of the fifth Eurographics symposium on Geometry processing*, 2007.

[ML17] Zhiliang Ma and Shilong Liu. Image-based 3d reconstruction techniques and their application in civil engineering. 04 2017.

[MMPGGMG16] Higinio Mora, Jerónimo Mora-Pascual, Alberto Garcia-Garcia, and Pablo Martinez-Gonzalez. Computational analysis of distance operators for the iterative closest point algorithm. *PLOS ONE*, 11:e0164694, 10 2016.

[MNS13] Shahram Izadi Matthias Nießner, Michael Zollhofer and Marc Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics*, 2013.

[NIH+11] Richard Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. pages 127–136, 10 2011.

[OF03] Stanley Osher and Ronald Fedkiw. *Signed Distance Functions*, pages 17–22. 01 2003.

[PB13] Brian Peasley and Stan Birchfield. Replacing projective data association with lucas-kanade for kinectfusion. pages 638–645, 05 2013.

[Ruc96] William Rucklidge, editor. *The Hausdorff distance*, pages 27–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[SVB99] Carlos Saona-Vázquez and Pere Brunet. The visibility octree: A data structure for 3d navigation. *Computers and Graphics*, 23:635–643, 10 1999.

[YCH+13] Ming-Der Yang, Chih-Fan Chao, Kai-Siang Huang, Liang-You Lu, and Yi-Ping Chen. Image-based 3d scene reconstruction and exploration in augmented reality. *Automation in Construction*, 33:48–60, 08 2013.

[YL18] Yun-Jia Yeh and Huei-Yung Lin. 3d reconstruction and visual slam of indoor scenes for augmented reality application. pages 94–99, 06 2018.

[ZLD15] Ping Zhang, Jincong Luo, and Guanglong Du. Depth image application in analysis of automatic 3d reconstruction. pages 409–414, 06 2015.