Improved Integration of Depth Images in 3D Models

Dennis Depner*
Supervised by: Stefan Ohrhallinger†

Institute of Visual Computing and Human-Centered Technology
Vienna University of Technology
Vienna / Austria

Abstract

3D reconstruction using depth images is established in many areas of computer graphics, such as virtual and augmented reality. Our research is about the procedure for 3D reconstruction from the Kinect Fusion algorithm. Particularly, the work aims 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. 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 results have shown that including median vectors has a positive influence on the quality of the 3D reconstructed surface compared to the default Kinect Fusion algorithm. In smaller areas, there are fewer holes on the surface. Finer objects also appear softer and are reconstructed less edgy. The adapted Marching Cubes algorithm for octrees also shows a strong improvement in terms of generating fewer holes in the mesh.

Keywords: 3D, Surface, Reconstruction, Scanning, Integration, Depth-Maps, Depth-Images, Kinect Fusion, Marching Cubes, Voxel

1 Introduction

The reconstruction of 3D models in the real world plays an important role in various applications and numerous fields like computer graphics, computer vision, medical imaging, virtual reality, etc. One example of that is augmented reality, which is heavily dependent on a precise and consistent 3D reconstruction of real-world objects. Knowledge of the position, form, and scale of real-world objects is necessary to place new virtual objects in an environment [15]. Another highly related topic to augmented reality is visual SLAM (Simultaneous Localization And Mapping). Visual SLAM technologies enable autonomous sensors to localize their position by scanning and mapping their environment simultaneously. This can be very useful for the development of autonomous vehicles or robots [16].

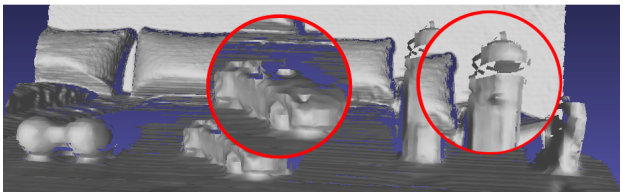

Figure 1: Mesh with default integration / default Marching Cubes

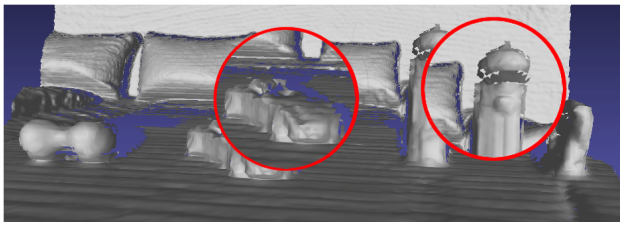

Figure 2: Mesh with median integration / default Marching Cubes

Figure 3: Reference Image

Default integration means default surface generation by Kinect Fusion Algorithm. Median integration means surface generation by a modified version of Kinect Fusion. Comparing Figure 1 with Figure 2, especially the reconstruction of the car body and engine cover seems to be more similar to the real-world object in the median integration. Note the bottle in the right middle and the can on the right, which have slightly fewer holes in the mesh with the median integration.

*e1632716@student.tuwien.ac.at
†ohrhallinger@cg.tuwien.ac.at

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 do not have the purpose to save the color but the depth of an object in a scene [17]. 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) [2].

2 Background

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 is in space 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 on 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) [10].

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 [10]. 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 the accuracy of the surface generation. In 3D space, a depth image is a point cloud in which each 3D point is assigned a pixel in the depth image. If multiple depth images are reconstructed as a 3D model, Kinect Fusion does not include the information of the point clouds of previous/older reconstructed scans in new scans. That has the consequence that information and accuracy of the surface get lost. An approach for a possible improvement is to simply save and use the point cloud information for surface generation. Because simply saving all points of every depth image in the 3D model would cause 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 aggregated as a median vector. In the following chapters, the 3D reconstruction process as well as the changes of Kinect Fusion will be explained. Next to the changes of Kinect Fusion, we will also discuss the differences between two mesh generation algorithms, which are the Marching Cubes Algorithm and an adaptation of Marching Cubes for octrees.

3 Kinect Fusion Pipeline & Data Structures

Kinect Fusion processes the depth information of depth images frame after frame to obtain a dense 3D model. To get a better understanding of what 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 [10].

3.1 Kinect Fusion Algorithm Pipeline

3.1.1 Surface Measurement

In this step, a point cloud consisting of a vertex map V is calculated with the scanned depth image. 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. Computing the vertex map is essential for being able to estimate the sensor or camera pose later [10].

3.1.2 Surface Reconstruction Update

For extracting a surface of the depth image we need a function to represent it. For this, 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 units away from the surface or $SDF((1,3,7)) = 5$) [12]. The sign of the distance determines whether p lies in front of (positive sign), behind (negative sign) or exactly on the surface (then the distance is 0) with respect to the camera position. In Kinect Fusion, the calculation of the SDF is simplified by only determining the signed distance between the camera position and p along the z-direction in camera space [10]. $TSDF$ is the same concept, except for taking only points in a defined range or truncation band μ into account. The distance inside the truncation band μ is normalized by dividing it by μ. If a point's distance from the surface is bigger than μ, it is set to either 1 (in front of the surface) or -1 (behind the surface) [10]. 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

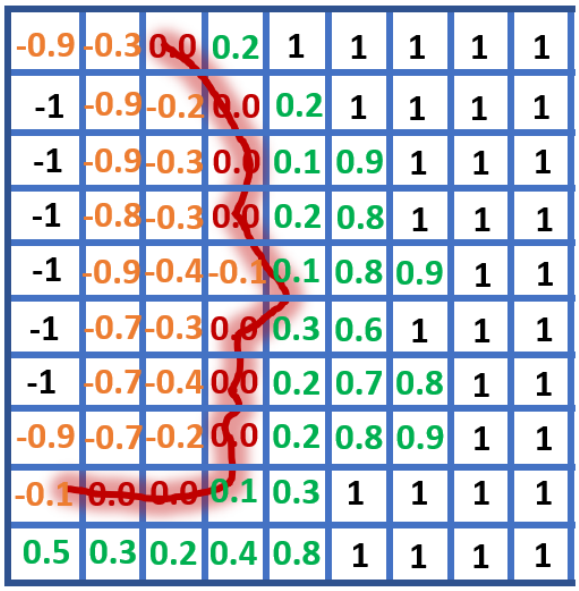

Figure 4: $TSDF$ of a 2D surface is represented as a red line. For each voxel, here as a 2D square, the shortest signed distance from a corner to the red marked surface is computed. In this example, the signed distance is depicted inside each square [3].

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 shortest signed distance to its corresponding depth measurement (obtained by projecting the position of a voxel onto the depth image) is computed [10]. In Figure 4 is also a 2D example for a $TSDF$.

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 [10].

3.1.3 Surface Prediction

This processing stage is also known as "Raycasting". Here, a ray is casted from each pixel of the depth image along the estimated camera pose to read the $TSDF$ and create another point cloud as a vertex map. While a ray marches through the 3D grid with a step size of the voxel length, it stops when a zero crossing in the voxel 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 [10].

These steps finally generate a vertex map of all fused $TSDFs$ representing the global 3D surface. That vertex map can be compared with the vertex map, which was computed in the first step "Surface Measurement" to finally estimate the camera position [10].

3.1.4 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. For our case, B is the vertex map of the "Surface Measurement" and A is the vertex map of the "Surface Prediction". The algorithm tries to find the transformation between A and B by minimizing the distance between the points of A to B with several iterations [9].

3.2 Data Structures for the Voxel Grid

3.2.1 Grid as Hash Table

The first solution is called Voxel Hashing [11]. 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 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 [6].

In our case, we have to imagine a uniform infinite 3D grid consisting of so-called voxel blocks. A voxel block is a cube consisting of e.g. 8 x 8 x 8 or 512 voxels and has a hash key, which can be computed with its position. The voxels inside a voxel block are directly accessible in a list [11].

3.2.2 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. In our case, every node represents a voxel [14].

3.2.3 Meshing of Voxel Grid

To obtain a visual surface, we generate a mesh of the voxel grid by using the Marching Cubes Algorithm (Implementation was done with Bourke's Look Up Tables). 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 approximated surface [8].

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 default Marching Cubes Algorithm. That is because we also iterate over coarser nodes/voxels, which do not necessarily have the same resolution of neighboring nodes/voxels. For example, the edge of a voxel can be a

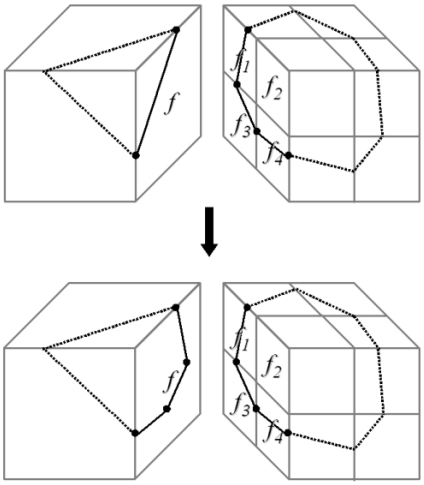

Figure 5: On top, a coarser voxel node and to its right finer voxel nodes are depicted. On each of them default Marching Cubes is independently applied. It is clear that the edge of the face f does not match the edges of the faces f_1, f_3 and f_4, which will cause a crack in the surface. When applying the Marching Cubes adapted to octrees like depicted on the bottom, the coarser edge of f gets replaced by the smaller edges f_1, f_3 and f_4 in order to avoid cracks [7].

neighbor to a smaller or longer edge of a voxel, which is not on the same level or depth in the octree. If we would independently generate the polygons for every node/voxel in the octree, there could appear cracks along such edges, since the zero crossing of a bigger edge may not match with the zero crossings of possibly neighbored smaller edges. Therefore, we have to use a Marching Cubes Algorithm of this paper [7], which is adapted to octrees. Shortly explained, for edges of coarser voxels which are neighbored to smaller edges of finer voxels, the zero crossings of the bigger edges are replaced with those of the neighbored smaller edges. This should eliminate possible cracks. The problem and avoidance of possible cracks is also illustrated in Figure 5. 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 [7].

4 Improved Depth Integration Method

To improve the integration of the depth information into the global 3D model, we change the processing step "Surface Reconstruction Update" of "Kinect Fusion". Taking a closer look at it, Kinect Fusion only considers the current depth image when computing the local $TSDF_{local}$ (with local $TSDF_{local}$ we mean the $TSDF$ of one depth image. The global $TSDF$ is the fused $TSDF$ of all depth images) and ignores nearby depth measurements of previous/older scans. When projecting 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.

4.1 Definitions

In order to explain the above-mentioned improvements, some definitions have to be made. The voxel grid is defined as $V \subset \mathbb{R}^3$ with following functions:

$$p : V \to \mathbb{Z}^3, m : V \to \mathbb{R}^3, d : V \to \mathbb{R}, n_m : V \to \mathbb{N}, w : V \to \mathbb{N}$$

p returns indexed position of the voxel, m returns median vector of all projected depth measurements inside the voxel, d returns global TSDF of the voxel, n_m returns number of all projected depth measurements inside the voxel, w returns number of update cycles of the voxel. The origin of the voxel grid is always assumed to be $(0,0,0) \in \mathbb{R}^3$. The k_{th} depth image is defined as:

$$I_k \subset \mathbb{N}^2, depth : I_k \to \mathbb{R}$$

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. For the camera extrinsics a $4x4$ matrix C_{EST} represents the estimated rotation and translation of the camera.

4.2 Default 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 the camera space. Therefore, we first need to project the position $p = (x,y,z)$ of a voxel $v \in V$ back onto the depth image plane to get the corresponding depth measurement [10].

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = C_{EST} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = 1/z' \cdot K \cdot \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \tag{2}$$

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

In step (1), the homogenized position $p = (x, y, z, 1)$ of v is transformed into the camera space by multiplying with C_{EST}. In step (2), we project the transformed position $p' = (x', y', z')$ of v from camera space onto the depth image plane by multiplying it with K and $1/z'$. In the last step (3), we take the depth measurement t at the pixel coordinates (u, v) (u and v are rounded down to integers), which we obtained in the previous step [10].

Now that the depth measurement t and z' is obtained, the local $TSDF_{local}$ of the voxel v can be calculated like in step (4) [10].

$$TSDF_{local} = \begin{cases} 1, & \text{if } t - z' > \mu \\ -1, & \text{if } t - z' < -\mu \\ \frac{t-z'}{\mu}, & \text{otherwise} \end{cases} \tag{4}$$

After that, we only need to fuse $TSDF_{local}$ with the global $TSDF$ $d(v)$, which means calculating the average. This can be done incrementally with the w function, which tracks a voxels' number of updates or computed local $TSDF$s. The global $TSDF$ $d(v)$, which is initially zero, can be calculated for a voxel $v \in V$ as shown in step (5) [10].

$$d(v) = \frac{d(v) \cdot w(v) + TSDF_{local}}{w(v) + 1} \tag{5}$$

4.3 Median Integration

By using the point cloud information of previous scans aggregated as median vectors, we expect to see a higher accuracy in updating the surface and generating the surface as a mesh.
For the median integration of a depth image, the median vectors of all voxels are updated with their corresponding depth measurements. To do that, we first iterate over all pixels of the depth image and project them to 3D points in the world space of the voxel grid. After that, the median vector of the voxel will be incrementally updated with the 3D point (See also Algorithm 1)

After updating the medians, the global $TSDF$ $d(v)$ of each voxel can be calculated. Every voxel will be matched with its corresponding depth measurement by projecting its middle position onto the depth image plane. When obtaining a depth measurement after projecting a voxel $v \in V$ into depth image space, it is again projected back into the 3D voxel grid to find out in which voxel $v_2 \in V$ it lies. When projecting the obtained depth measurement into the voxel grid, it does not necessarily lie in v since the measured depth could be smaller or larger than the depth of v. That is why we defined another voxel v_2. The local $TSDF_{local}$ of v can be calculated with the median vector of v_2. Later the global $TSDF$ $d(v)$ of v can be updated with the local $TSDF_{local}$ like already explained in the "Default Integration" (See also Algorithm 2).

To calculate the local $TSDF_{local}$ of a voxel v we compute its shortest distance among its eight vertices to its corresponding median vector $m(v_2)$, see also Algorithm 3.

Algorithm 1: Updating median of voxels

```
// Hashfunction returns a voxel
   according to an indexed
   position
Hash : ℤ³ → V
Hash(pos) = v
pos ∈ ℤ³ , v ∈ V
// Updating all median vectors
for ∀(u,v) ∈ I_k do
    // 3D position in camera space
       of (u,v)
    p_cam ← depth(u,v) · K⁻¹ · (u, v, 1)ᵀ
    // transformed p_cam into world
       space
    p_world ← C_EST⁻¹ · (p_cam.x, p_cam.y, p_cam.z, 1)ᵀ
    // indexed voxel position
    pos ← ⌊p_world / voxelsize⌋
    // corresponding voxel v for
       indexed pos
    v ← Hash(pos)
    // Updating median vector m(v)
       and the counter n(v) of v
    m(v) ← (pos + n(v)·m(v)) / (n(v)+1)
    n(v) ← n(v) + 1
end
```

5 Implementation

Implementation of the changes in Kinect Fusion was done in the open-source framework "InfiniTAM v3" [5]. It is a framework written in C++ and optionally using Nvidia CUDA (a parallel computing platform for programming with an Nvidia GPU [4]). The framework has the whole Kinect Fusion Pipeline and a Hashing Voxel Data Structure already implemented. Since our purpose is to test the changes in quality for the default integration in Kinect Fusion vs. our median integration and the default Marching Cubes vs. the Marching Cubes adapted to Octrees, it was sufficient to do the implementation only running on the CPU without using Nvidia CUDA. Our implementation was running on a machine with a CPU

"Intel Core i7 4770k" and an Nvidia GTX 1060 graphics card.

Algorithm 2: Updating TSDF of voxels with median

for $v \in V$ **do**
  // transform $p(v)$ from world space into camera space
  $p_1^{cam} \leftarrow C_{EST} \cdot \begin{pmatrix} p(v).x \\ p(v).y \\ p(v).z \\ 1 \end{pmatrix}$
  // project p_1^{cam} from camera space to the depth image plane
  $\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \leftarrow \frac{1}{z} \cdot K \cdot \begin{pmatrix} p_1^{cam}.x \\ p_1^{cam}.y \\ p_1^{cam}.z \end{pmatrix}$
  // depth measurement t of (u,v)
  $t \leftarrow depth(\lfloor u \rfloor, \lfloor v \rfloor)$
  // project depth measurement t back to camera space
  $p_2^{cam} \leftarrow t \cdot K^{-1} \cdot \begin{pmatrix} \lfloor u \rfloor \\ \lfloor v \rfloor \\ 1 \end{pmatrix}$
  // project p_2^{cam} into world space
  $p_2^{world} \leftarrow C_{EST}^{-1} \cdot \begin{pmatrix} p_2^{cam}.x \\ p_2^{cam}.y \\ p_2^{cam}.z \\ 1 \end{pmatrix}$
  $p_2 \leftarrow \begin{pmatrix} p_2^{WORLD}.x \\ p_2^{WORLD}.y \\ p_2^{WORLD}.z \end{pmatrix}$
  // indexed voxel position pos of projected depth measurement
  $pos \leftarrow \lfloor \frac{p_2}{voxelsize} \rfloor$
  // voxel v_2, in which t lies
  $v_2 \leftarrow Hash(pos)$
  // calculate local $TSDF_{local}$ with median vector of v_2
  $TSDF_{local} \leftarrow calculateTSDF(v, m(v_2), t)$
  // update global $TSDF$ $d(v)$
  $d(v) \leftarrow \frac{TSDF_{local} + w(v) \cdot d(v)}{w(v)+1}$
end

Algorithm 3: Calculating TSDF of voxel with median vector

Function calculateTSDF($voxel \in V$, $median \in \mathbb{R}^3$, $depth \in \mathbb{R}$):
  // transform $p(voxel)$ from world to camera space
  $p \leftarrow C_{EST} \cdot \begin{pmatrix} p(voxel).x \\ p(voxel).y \\ p(voxel).z \\ 1 \end{pmatrix}$
  // indicate whether the voxel lies in front, behind or on the surface
  $sign \leftarrow signum(depth - p.z)$
  // calculate minimum distance and apply $sign$
  $n = 0$
  $distances[8]$
  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**
        $distances[n] \leftarrow length(median - \begin{pmatrix} p(voxel).x + i \cdot voxelsize \\ p(voxel).y + j \cdot voxelsize \\ p(voxel).z + z \cdot voxelsize \end{pmatrix})$
        $n \leftarrow n+1$
      end
    end
  end
  $distance_{min} \leftarrow min(distances)$
  $distance_{min} \leftarrow distance_{min} \cdot sign$
  // return local $TSDF$
  return
  $TSDF \leftarrow \begin{cases} 1, & \text{if } distance_{min} > \mu \\ -1, & \text{if } distance_{min} < -\mu \\ \frac{distance_{min}}{\mu}, & \text{otherwise} \end{cases}$

6 Results

To measure the difference between the meshes reconstructed with the default and median integration methods as well as the default and Octree Marching Cubes Algorithm, we use the RMS (Root Mean Square) of the directed Hausdorff Distance. It is an algorithm that computes the RMS-distance from a source point set A and a target point set B [13]. The setup for the comparison is 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 with rather flat surfaces. In the second scene, we add more complex geometric shapes, e.g. a flipped table. In the third scene, geometrically finer shapes are added, which are also harder to reconstruct. The overall results of the measurements in all scenes are shown and explained in Figure 8.

The source point set for computing the Hausdorff distance is always the mesh generated with the default integration and the default Marching Cubes Algorithm. The target point set is the mesh generated with the median integration and (optionally) with Marching Cubes for octrees. The comparison of the meshes shows that the median integration method has no bad influence in terms of quality compared to the default integration. It rather highlights improvements considering the quality of the meshes. When generating meshes with the median integration, there are fewer holes in some areas of the surface, while finer objects also appear to be a bit smoother and more similar to real-world objects. Figures 1, 2, 6 and 7 are examples of the third scene in which those improvements can be observed. The results for the directed Hausdorff Distance as RMS also reflect with its rather low values (only a few millimeters) that there is only a slight difference between the default and median integration. Comparing the default Marching Cubes with the advanced Marching Cubes for Octrees, the differences between the meshes become slightly larger, which is also represented by the higher RMS of the directed Hausdorff Distance. Marching Cubes for Octrees provides the best improvement considering the fewer holes in the mesh. Combining Marching Cubes for Octrees with the median integration slightly amplifies that effect.

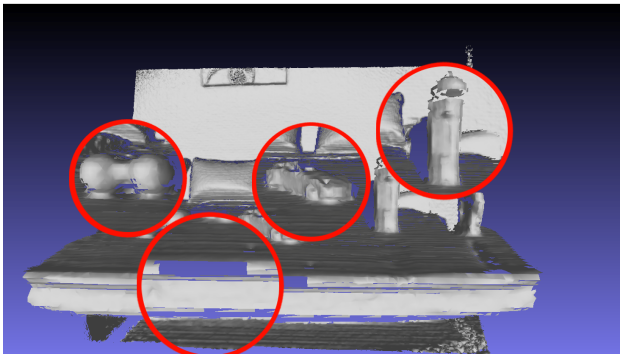

Figure 6: Mesh with default integration / default Marching Cubes

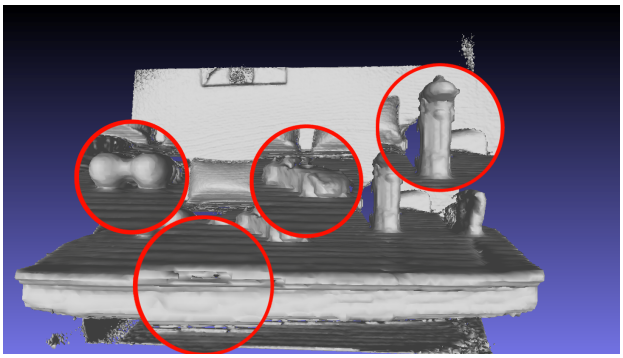

Figure 7: Mesh with median integration / Marching Cubes for Octrees

The Hausdorff Distance is 34.51 mm and is measured from mesh in Figure 6 to mesh in Figure 7. Using also the median integration with Marching Cubes for Octrees provides even fewer holes in the mesh compared to using the default integration.

7 Conclusion and Future Work

All in all, the median integration provides slight improvements in terms of generating fewer holes and letting the mesh appear to be smoother. The low values of the directed Hausdorff distance also ensure that the median integration does not cause any malformation or deterioration compared to the original Kinect Fusion Algorithm. Since our modifications only require slight changes in one update step of the Kinect Fusion Algorithm, it should be easily adaptable to other applications too. The Marching Cubes Algorithm for octrees has even more significant improvements since it further reduces holes in the mesh.

The median integration can be used for further studies, which requires mesh generation with Kinect Fusion. One of our current works is concerned with detecting changes in two different scenes captured and reconstructed with the Kinect Fusion Algorithm. In the future, the difference of meshes will also be measured to ground truth data with the BlenSor plugin [1]. The BlenSor Plugin simulates depth data of an already generated 3D model and imitates a depth sensor. We expect the more precise median integration to deliver more faithful results in change detection.

References

[1] Blensor: Free open source simulation package for light detection and ranging (lidar/ladar) and kinect sensors. https://www.blensor.org/. Accessed: 2022-02-16.

[2] Xin Bi. Lidar technology. In *Environmental Perception Technology for Unmanned Systems*, pages 67–103. Springer, 2021.

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

[4] Mark Harris. Many-core gpu computing with nvidia cuda. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 1–1, 2008.

[5] O. Kahler, V. A. Prisacariu, C. Y. Ren, X. Sun, P. H. S Torr, and D. W. Murray. Very High Frame Rate Volumetric Integration of Depth Images on Mobile Device. *IEEE Transactions on Visualization and Computer Graphics*, 22(11), 2015.

[6] Elshad Karimov. Hash table. In *Data Structures and Algorithms in Swift*, pages 55–60. Springer, 2020.

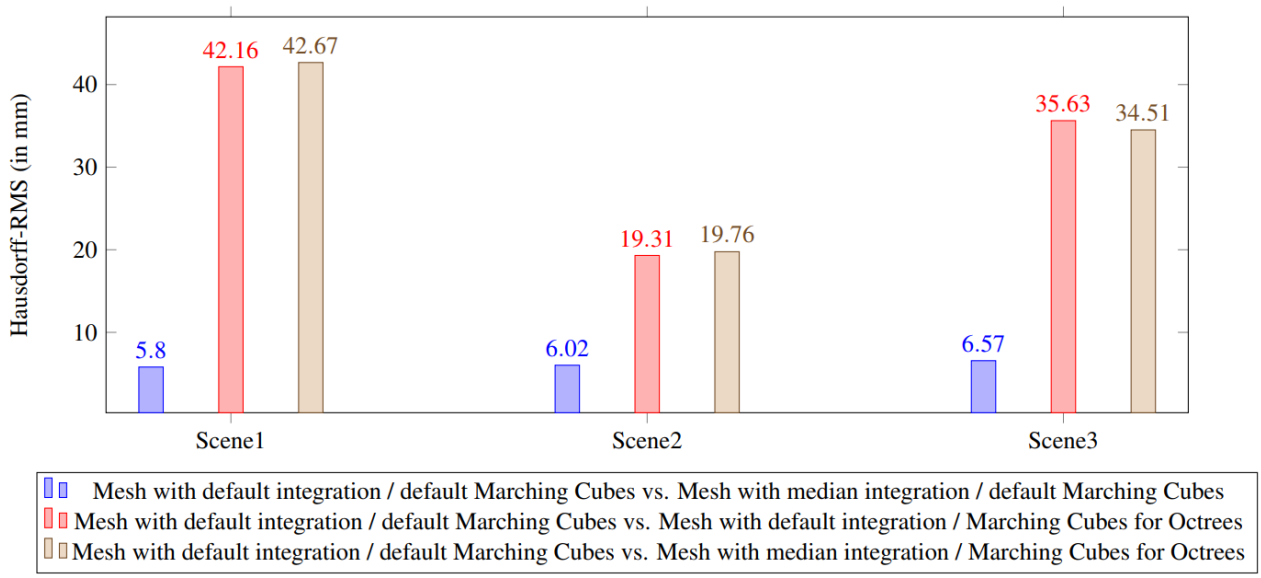

Figure 8: Directed Hausdorff Distance as RMS from the mesh with default integration/default Marching Cubes to all new meshes for every scene. It can be observed, that only using the median integration makes the smallest difference in terms of Hausdorff Distance. However, as complexity in the scene gets higher (Scene 1 low, Scene 2 middle, Scene 3 high), the Hausdorff Distance also gets higher, which indicates that more complex scenes might be influenced more by the median integration. When only comparing the default Marching Cubes Algorithm with the adapted Marching Cubes for octrees (without using the median integration), Hausdorff Distances turn out to be higher. The reason for that is obvious since Marching Cubes for octrees can also generate polygons for bigger nodes and is better at compensating for missing values/noise. This can be observed very well in Figure 6 and 7 at the table. Using Marching Cubes for Octrees in combination with the median integration slightly raises the Hausdorff Distance, but in the third scene it is slightly lower.

[7] Michael Kazhdan, Allison Klein, Ketan Dalal, and Hugues Hoppe. Unconstrained isosurface extraction on arbitrary octrees. In *Symposium on Geometry Processing*, volume 7, 2007.

[8] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.

[9] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration. *Chapel Hill, University of North Carolina*, 4(10):1–3, 2004.

[10] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE international symposium on mixed and augmented reality*, pages 127–136. IEEE, 2011.

[11] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (ToG)*, 32(6):1–11, 2013.

[12] Stanley Osher and Ronald Fedkiw. Signed distance functions. In *Level set methods and dynamic implicit surfaces*, pages 17–22. Springer, 2003.

[13] William Rucklidge. The hausdorff distance. In *Efficient Visual Recognition Using the Hausdorff Distance*, pages 27–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[14] Carlos Saona-Vazquez, Isabel Navazo, and Pere Brunet. The visibility octree: a data structure for 3d navigation. *Computers & Graphics*, 23(5):635–643, 1999.

[15] 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, 2013.

[16] Yun-Jia Yeh and Huei-Yung Lin. 3d reconstruction and visual slam of indoor scenes for augmented reality application. In *2018 IEEE 14th International Conference on Control and Automation (ICCA)*, pages 94–99. IEEE, 2018.

[17] Ping Zhang, Jincong Luo, and Guanglong Du. Depth image application in analysis of automatic 3d reconstruction. In *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*, pages 409–414. IEEE, 2015.