Paul Bodenbenner, BSc

Deformable 3D Reconstruction

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematik

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Denis Kalkofen

Institute of Computer Graphics and Vision

Graz, September 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

Deformable 3D Reconstruction provides two different methods for facing the challenging task of a non-rigid 3D reconstruction. This is realized by two separate pipelines, allowing either a template-based or a model-based reconstruction. By running all computational costly tasks on the GPU, Deformable 3D Reconstruction is capable of performing a non-rigid 3D reconstruction in almost real time. Besides the implementation of the actual system, an extensive framework has been evolved, being easy extendable to various kinds of custom 3D reconstruction variants. The actual basis of Deformable 3D Reconstruction is KinectFusion, which already defines a complete pipeline for performing a static 3D reconstruction. KinectFusion provides sophisticated as well as commonly used practices related to the field of 3D reconstruction, which in fact is exploited by Deformable 3D Reconstruction. The required input data for running both new pipelines may be simply generated by using a single consumer RGB-D camera. While such a device produces only poor depth data, the underlying methods of the presented system are able to generate high quality output in form of meshes. Summarized, Deformable 3D Reconstruction enables the reconstruction of deforming objects in a fast and also qualitative way.

Kurzfassung

Deformable 3D Reconstruction bietet zwei verschiedene Methoden zur Behandlung der herausfordernden Aufgabe einer nicht-rigiden 3D Rekonstruktion. Realisiert wird das durch zwei voneinander unabhängigen Pipelines, welche entweder eine template-basierende oder eine model-basierende Rekonstruktion ermöglichen. Da alle rechenintensiven Aufgaben auf der GPU behandelt werden, ist Deformable 3D Reconstruction imstande, eine nicht-rigide 3D Rekonstruktion annähernd in Echtzeit zu bewältigen. Neben der Implementierung des eigentlichen Systems, ist ein aufwändiges Framework entstanden, das leicht zur Verwendung für verschiedenen Arten von 3D Rekonstruktionen herangezogen werden kann. Die eigentliche Basis von Deformable 3D Reconstruction ist KinectFusion, welche bereits eine komplette Pipeline für eine statische 3D Rekonstruktion definiert. KinectFusion bietet durchdachte und geläufige Praktiken im Bereich der 3D Rekonstruktion, dessen sich Deformable 3D Reconstruction bedient. Die von den zwei neuen Pipelines benötigten Input-Daten können einfach mit einer üblichen RGB-D Kamera generiert werden. Während diese Geräte nur ungenaue Tiefendaten erzeugen, entstehen durch die zugrundeliegenden Methoden des zu präsentierenden Systems hochqualitative Ergebnisse in Form von Meshes. Zusammengefasst ermöglicht Deformable 3D Reconstruction die Rekonstruktion von deformierenden Objekten auf eine schnelle und qualitative Art und Weise.

Contents

List of Figures

List of Listings

Acknowledgments

I am grateful for the hard work of my girlfriend Elisabeth Fluch for reviewing this thesis, in especially for correcting the text stylistically and grammatically and making the content understandable.

Additionally my supervisor Denis Kalkofen has to be mentioned for supporting me in any way and providing me his technical expertise for solving some challenging problems.

Furthermore I want to thank Karl Voit for offering an advanced, easy to use and well documented LaTeX template (Voit, 2018), to bring this thesis into a beautiful shape.

Finally, the guideline of Keith Andrews how to write a master's thesis (Andrews, 2012) gave me some important hints for what has to be taken care of to produce a scientific work.

1 Introduction

Putting something on record has always been of great interest for the humanity - beginning with cave paintings, up to the point of photography and now doing the mapping into the virtual world. In the last decades the projection into this world was mostly done in 2D only, but since the relevant technology has been evolving rapidly in recent years, real 3D capturing has become possible too. While the processing of this data has already been explored very well for static and rigid bodies or scenes in previous work, today's research is mostly focused on moving and deforming bodies.

The challenge of doing a 3D reconstruction is in the nature of things. While a 2D image can be easily acquired by taking a single snapshot, the reconstruction of a 3D object requires much more than that. Following questions may arise when dealing with this non trivial task: How can an object be captured from all sides and angles, acquiring all necessary data for a complete reconstruction? How can be determined which part belongs to which snapshot? Does the object move during the recording or does it even change its structure in the meantime? The primary goal of this thesis is to answer these questions by finding solutions based on already made efforts.

An object or scene may be recorded either by using one or multiple cameras. Working with multiple cameras, a recorded object is captured from different sides and angles at the same time. When the cameras are positioned to cover all parts of that object, its complete surface information is available with one shot of every single frame. While this appears to be an ideal situation, some preconditions have to be fulfilled first. To obtain significant results, the challenging task of synchronizing multiple cameras and their extrinsic calibrations has to be solved beforehand. Additionally, such a setup is not portable and therefore restricts the recording to a specific place. In contrast, a single non aligned camera can be easily moved and installed without having the restriction to any special location. In this case, of course, only the information of the camera's currently viewpoint is available. Summarized, both methods have its advantages and disadvantages, but taking additionally the cost factor into account for acquiring multiple cameras, the single camera method becomes very attractive. All in all, this single camera method seems to be more appropriate for this thesis and has therefore been chosen.

Before being able to perform a 3D reconstruction, the underlying input data has to be available. This raw data is in the form of an usual image, but contains depth values instead of color related information. In the past, it was a big deal to get such data, but this has become an easy task now. The needed recording devices which directly output that type of data, in the following referred to as "depth image", reached the consumer level. Such a device provides its captured data in a defined frequency and is mostly equipped to supply color information too. A color image is usually defined by three channels, consisting of a red, green and blue component (abbreviated RGB in the following). The combination of such an RGB image and a depth image is called "RGB-D image". In fact, this data represents the world frame wise only, where each snapshot belongs to one single viewpoint. Before achieving a single result in a meaningful form, several steps have to be processed first.

Generally, for getting a depth image into the 3D virtual world there has to be some knowledge about the internal properties of the used device, also known as "intrinsics". By having either rough data provided by its manufacturer or exact data by doing a calibration of the device by oneself, a depth image can be projected into a 3D representation. The output of such a projection is in the form of an organized point cloud. The basic layout of the data does not change with this transformation and stays in the shape of a regular 2D grid. Therefore the information of the neighborhood is not lost and may be a handy property for a later processing.

Though the previous mentioned projection is only the first basic step towards a 3D reconstruction, a method is needed for merging these single frames together. Usually this is done by comparing consecutive frames with each other. One important part of the concatenation deals with the issue of aligning the point clouds onto each other. This is needed because of an occurring movement of the camera, which is required for capturing an object from various points of views. Basically it makes no difference whether the camera is moving or the recorded object, as long as the complete observed frustum does change accordingly. Additionally to the aligning, the points need to be merged somehow. Otherwise the number of points would increase drastically and would make a further processing unacceptable in the sense of calculation effort. As a side-effect, the merging can also be interpreted as a smoothing process where near points are joined together to become a single point, for example by averaging their positions. After running this pipeline for a while, an extensive point cloud may grow up. Though this representation is not very useful for the end use, the data needs to be transformed into a different shape. A commonly structure used for such data is called "mesh", which consists of multiple triangles connected somehow to each other forming a joined surface. Using this representation, 3D data can be easily displayed, manipulated and also stored.

The process described above produces a mesh as output, which only works properly for static and rigid bodies or scenes. To capture deforming bodies, additional operations need to be involved. In this thesis two different methods are going to be covered to tackle the non-rigid 3D reconstruction issue: The first method tries to perform a continuous static 3D reconstruction at the beginning, by ignoring some possible arising deformation at first. At some specific time the actual reconstruction stops and the deformation related part starts doing its job. This consists of a combination of estimating and integrating any occurring deformation. The other method, the more complicated one, tries to handle the deformation right from the beginning, while the actual reconstruction happens simultaneously.

By comparing both methods which each other, it has to be noticed that the first one has more requirements on the recorded scene, by having the demand of a rigid body in the first stage. In contrast, the other one does not rely on that and is able to extend the body information up until the end of the recording. Therefore it has to take care of more degrees of freedom, resulting in being not that stable like the first one. After now having given just a short summary, a detailed insight into both methods will be given in Chapter 3.

For getting an overview of already made investigations in the field of rigid and non-rigid 3D reconstruction, Chapter 2 will cover some previously made achievements concerning 3D reconstruction.

2 Related Work

The ultimate goal when doing a rigid as well as a non-rigid 3D reconstruction is to get this done in real time. By opening the doors for applications in the fields of Augmented Reality (AR) and Virtual Reality (VR), many new possibilities for interacting with the real world arise.

One milestone in the online reconstruction of static scenes has been achieved with the invention of KinectFusion, proposed by Izadi et al. (2011) and Newcombe et al. (2011). KinectFusion is a method for bringing real world objects into the 3D virtual world by using a single recording device only. For the implementation they used a Microsoft Kinect, which was originally developed for video gaming and was actually invented to interact with their also in-house developed XBox gaming console. Even though the camera itself does provide only poor input data, in form of RGB-D images, they obtained impressive reconstruction results, even in real time (see Figure 2.1). This was achieved by making use of massive parallelism and especially by sourcing the calculation intensive tasks out onto the graphical processing unit (GPU). One such calculation deals with aligning the current input onto the model, which is solved by a fast variant of the Iterative Closest Point (ICP) algorithm (Besl and McKay, 1992). The other computational intensive calculation involves the handling of the data structure used for its internal model. Especially this structure needs to offer a fast way for integrating and fusing the input data into the model and also for exporting it the other way around. Actually they made use of a voxel grid, which represents its properties in an equidistant fashion and yields a fixed size. By using this representation in combination with weighted signed distance functions (SDF), introduced by Curless and Levoy (1996), a cumulative integration is easily achieved. Because the area of the surface is of main interest, voxels, which are positioned farther away, are truncated (abbreviated TSDF in the following). While this representation may behave very speed efficient, the need of much memory is the resulting disadvantage. Basically this may not be a big deal, but by having the requirement for a fast access to the data, there is no other way of storing it directly into the memory of the GPU. Therefore the resolution of this model is restricted by the amount of the available video memory. Probably one of the most advanced solutions to circumvent this problem deals with mapping the fixed sized voxel grid onto a dynamically allocated data structure using a hash function (Nießner et al., 2013). This greatly improves the efficiency of the memory usage by compressing the space with a spatial hashing technique.

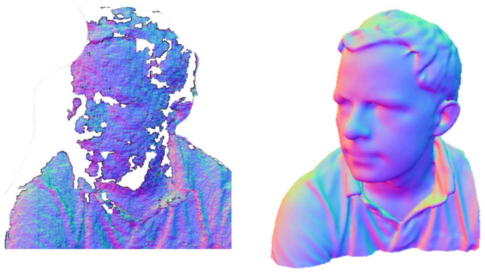

Figure 2.1: KinectFusion - Input Data and Static Reconstruction (Newcombe et al., 2011) (Image taken from Newcombe et al. (2011).)

A complete different way and probably even a more natural way is to deal directly with the form of the acquired input data, using a point-based representation for the data model. Keller et al. (2013) showed a similar pipeline, but compared to KinectFusion, they used an unstructured set of points for the model's representation. So, by having a dynamically growing model, a fixed amount of memory is not needed anymore. While the fusion of the points is based on the same approach presented by Curless and Levoy (1996), the need of the implicit surface representation is avoided. Additionally they present a method for dealing with dynamics in the scene, in which such regions are detected and furthermore ignored for the integration. By acquiring outliers at the ICP stage, the dynamic regions of the input are estimated. This information further influences the integration of the input data into the point-based model. By having a confidence value stored additionally to every single point, wrongly added points, which belong to a dynamic region, are removed also in the second-guess, although they have already been integrated. Compared to the dynamics segmentation, proposed by Izadi et al. (2011), the point-based method does not need an initial scan to distinguish between static and dynamic regions. Another difference is found in the creation of the synthetic map, which is needed for representing the current model, used in the ICP task. While KinectFusion has to render its implicit surface representation by using raycasting, Keller et al. (2013) apply a simple surface-splatting technique on its point-based representation in order to generate such a map.

While the previous methods are only able to reconstruct static scenes, Zollhöfer et al. (2014) introduced a system which combines rigid reconstruction with deformation handling, running in two different stages. This so-called "template-based" solution acquires its geometry in the first phase, and tracks and applies possible occurring deformation on the geometry in the second phase. The latter is shown in form of a deformation sequence in Figure 2.2. For acquiring the input data in form of RGB-D images, a self developed real-time capable stereo setup, consisting of a color (RGB) and an infrared (IR) camera, is introduced. The template is created with the help of an extended version of KinectFusion (Nießner et al., 2013). The resulting triangle mesh is processed to offer a multiple resolution hierarchy, which serves to handle the deformation in the second phase. This non-rigid part deals with minimizing an energy function, containing a data and a regularization term. The first term is responsible for fitting the surface, whereas geometry as well as color properties are covered. The second one is needed to achieve smooth deformations and is applied by using a method known as as-rigid-as-possible (ARAP) (Sorkine and Alexa, 2007). A final step to track some outstanding detail deformations is processed by solving a linear least square system.

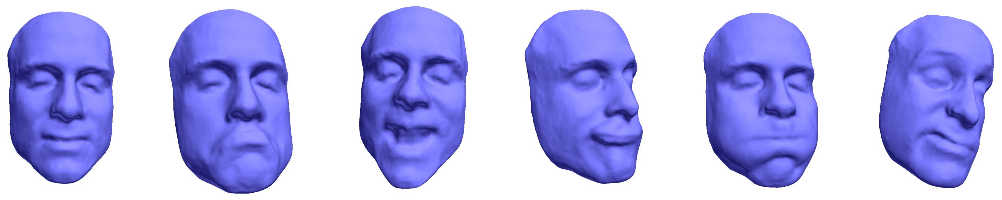

Figure 2.2: Real-time Non-rigid Reconstruction using an RGB-D Camera - Deformation Sequence (Zollhöfer et al., 2014) (Image taken from Zollhöfer et al. (2014).)

Though previous work has shown how to handle deformation, it still relies on having a still standing object at its first stage. Unfortunately this requirement cannot be fulfilled in all situations and therefore the first step has to be avoided somehow. A pioneering work, which tries to overcome this problem, has been proposed by Newcombe, Fox, and Seitz (2015). Their work presents a solution for performing a template-less non-rigid reconstruction running in real time. For achieving this, their initial question was "How can we generalise KinectFusion to reconstruct and track dynamic, non-rigid scenes in real-time?". Their key idea for solving this problem is the separation of the rigid part and the occurring deformation. By estimating a volumetric flow field, representing the transition between a rigid model and the current state of the scene, the actual integration of KinectFusion can be simply adopted. This results in having a procedure of undoing the deformation before the actual integration into the model is performed and afterwards warping the rigid model to represent the currently modeled deformed frame. This enables continuous integration and deformation of the data, as shown by the illustrated results in Figure 2.3. By supposing smoothly deformation on the surface and having the goal to run in real time, they use only a sparse set of estimated transformations. This warp information is represented by a hierarchical deformation graph (Sumner, Schmid, and Pauly, 2007). For getting a dense volumetric warp function out of it, interpolation has to be involved. A single transformation is defined using both, translation and rotation information, which leads to 6 degrees of freedom (DOF). By using that accurate description, Newcombe, Fox, and Seitz (2015) promise to achieve much better tracking and reconstructing results. As already proposed by Zollhöfer et al. (2014), the estimation of the warp field is performed by minimizing an energy function, consisting of a data and regularization term.

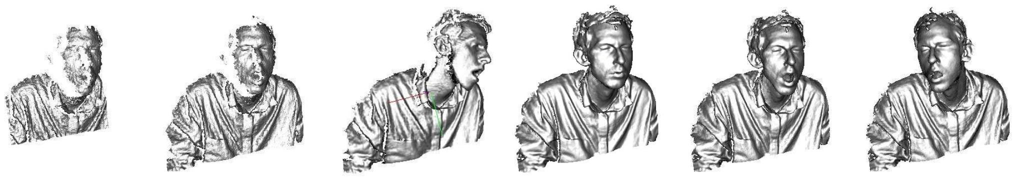
Figure 2.3: DynamicFusion - Continuous Integration and Deformation (Newcombe, Fox, and Seitz, 2015) (Image taken from Newcombe, Fox, and Seitz (2015).)

Shortly after the previously mentioned work was released, Innmann et al. (2016) presented a similar solution targeting the same issue. In fact, this approach enables continuous integration and deformation of the data too (see Figure 2.4). In contrast to the method of Newcombe, Fox, and Seitz (2015) where only the depth data is considered, this approach makes use of the color input as well. While Zollhöfer et al. (2014) have already used RGB values for fitting the data in their energy function, the following method promises to be much more stable. By estimating also Scale Invariant Feature Transform (SIFT) (Lowe, 1999; Lowe, 2004) correspondences in addition to the commonly used geometric based correspondences, they promise to get a far better handling of drift and fast motion. Even though these color related features are estimated in a sparse manner, they get captured on all input images and are used as global anchor points. A second key contribution to the reconstruction pipeline was made by Innmann et al. (2016), by introducing a dense volumetric representation of the warp field. This omits the usually needed interpolation for acquiring the interim values. However, those additions need some extra calculation effort, though the approach promises to stay an online solution, by exploiting their fast hierarchical optimization strategy.

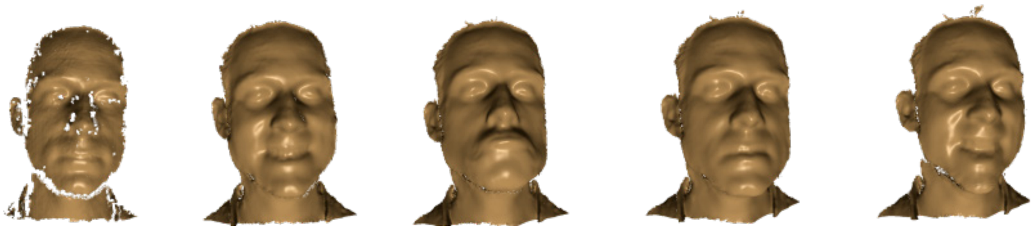

Figure 2.4: VolumeDeform - Continuous Integration and Deformation (Innmann et al., 2016) (Image taken from Innmann et al. (2016).)

While Innmann et al. (2016) have already used sparse features from the color images, Guo et al. (2017) go one step further by taking care of the intrinsic appearance, in the following referred to as “albedo”. By recovering this additional information, next to the geometry related data, a more complete static model can be described by using both for the integration. It is mentioned that the use of intrinsic appearance is an important cue for the correspondence estimation and improves the inter-frame motion detection too. This is because the appearance of a body never changes over time. For representing the warp field a graph-based solution is used, which Newcombe, Fox, and Seitz (2015) have already proposed for their work. The used energy function, which needs to be minimized, consists of four terms. While the first two are defined as data terms, involving the depth and shading properties, the other two are responsible for the regularization part. In addition to the improvements in the motion estimation, the use of the albedo opens the doors for new applications such as relighting and appearance editing.

A totally different approach is shown by Slavcheva et al. (2017) for facing the non-rigid reconstruction problem. While the previously mentioned works rely on finding correspondences for its estimation, this approach totally avoids this error prone task. Like nearly all other solutions storing its geometry related data in a volumetric TSDF grid, the proposal of Slavcheva et al. (2017) does this too but is making much more use of it. The proposed solution totally omits the intermediate steps of generating a mesh out of the model or its view dependent rendered map. For estimating the non-rigid deformation, a pair of two SDF grids have to be compared with each other. The first one contains the accumulated model, while the other one is just produced out of the current input depth image. The result of this procedure is in the form of a dense vector field, yielding the same size as both input grids. In contrast to the work of Newcombe, Fox, and Seitz (2015) and Innmann et al. (2016) where a transformation is defined by 6 DOFs, the vector field is obviously aware of 3 DOFs only. The global camera pose is estimated by using the SDF-2-SDF registration energy (Slavcheva et al., 2016) which registers a pair of voxel grids. This means no correspondences have to be acquired again, because the usually applied ICP algorithm is avoided. The used energy function for estimating the actual non-rigid transformation consists of a data term and two regularization terms. The first one of the regularization terms is taking care of a damped variant of the killing condition, while the other one is maintaining the level set property.

While having discussed only single camera solutions so far, it is time to present some interesting multi view approaches. Collet et al. (2015) propose a multimodal reconstruction system which combines RGB, IR and silhouette information to achieve high quality meshes in real time. The base of their complex setup is a greenscreen stage surrounded by 106 cameras. The to be generated depth images are fused together using a multimodal multi-view stereo algorithm to produce a single point cloud as result. By performing a self extended version of the poisson surface reconstruction (Kazhdan, Bolitho, and Hoppe, 2006), a high resolution mesh is extracted. To be able to provide the resulting content in a streamable and compressed fashion, mesh tracking needs to be performed. This is done by first estimating keyframes which are selected to present the actual content. The in-between frames have to be derived from these keyframes by using their appropriate non-rigid deformation field. For achieving this, the non linear registration algorithm proposed by Li et al. (2009) is applied. Though they mention that any similar algorithm may be adapted, like for example the already discussed method of Zollhöfer et al. (2014). The actual used algorithm needed for estimating these deformation fields minimizes an energy function, containing a data term for fitting the surface and two regularization terms observing the rigidity and smoothness. The last two are based on the rules of the embedded deformation graph (Sumner, Schmid, and Pauly, 2007).

Dou et al. (2016) show another approach of using a multiple camera setup. In there work on the non-rigid reconstruction problem, they combine the power of integrating the data into TSDF grids with the usage of multiple cameras from different viewpoints. For circumvent the problem of requiring much memory for those grids, they use an approach proposed by J. Chen, Bautembach, and Izadi (2013), to transform the regular grid into a hierarchical structure. Similar to the work of (Collet et al., 2015), where key frames are used as anchor points, this work selects some specific TSDF grids for holding the actual content. This means, as soon as there is a big change in the scene (e.g.: topology change), the currently processed TSDF grid will be exchanged by an empty one and the integration starts from scratch again. To demonstrate the ability for handling complex scene changes, some results are shown in Figure 2.5. Compared to the work of Collet et al. (2015) a simpler setup is used, by avoiding the need of a greenscreen stage and minimizing the number of used cameras. By the need of 24 cameras, providing RGB and IR data, a stereo algorithm has to be applied for acquiring the actual depth data. As already found in some previously mentioned approaches, the deformation is also handled by a graph (Sumner, Schmid, and Pauly, 2007). Their used energy function consists of five terms, where, aside from the data and regularization properties, a visual hull and correspondences term can be found.

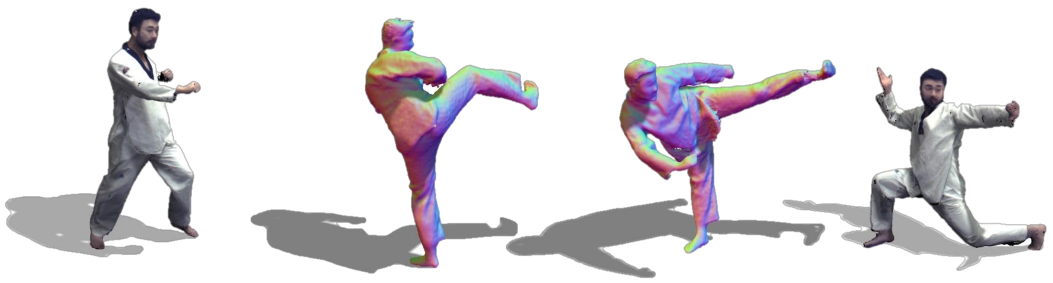

Figure 2.5: Fusion4D - Complex Scene Changes (Dou et al., 2016) (Image taken from Dou et al. (2016).)

An impressive application of how the communication may look like in the future has been demonstrated by Orts-Escolano et al. (2016). Taking the method of Dou et al. (2016) for the reconstruction part, combining it with audio support and bringing the content in a streamable form, a complete scene may be virtually cloned to any location in the world. Because the reconstruction is the computationally most expensive part of their pipeline, Orts-Escolano et al. (2016) applied the framework proposed by Dou et al. (2016) onto a dual GPU pipeline to be able to process even room-sized dynamic scenes in real time. By equipping all communication partners with an AR or VR headset, a new experience of telepresence was achieved.

Although having discussed some interesting approaches related to 3D reconstruction in dynamic scenes, much more efforts have already been made in the field of computer graphics and vision. To get a deeper insight into this topic, Zollhöfer et al. (2018) discuss an impressive list of related work in their survey. Additionally it gives a nice overview of all needed steps to dive into the world of the non-rigid 3D reconstruction.

3 Method

The basis used for the new approach is a reconstruction system called KinectFusion, proposed by Izadi et al. (2011) and Newcombe et al. (2011). As already discussed in Chapter 2, it describes a complete pipeline for doing a 3D reconstruction on rigid objects or scenes. By providing a sequence of depth images, the system is capable of producing a high quality mesh as output. Many parts of its pipeline can be used for the new proposal, therefore starting with KinectFusion is essential. By having access to an open-source implementation of KinectFusion, provided by the Point Cloud Library (PCL) (Rusu and Cousins, 2011), its slightly modified realization will be mainly taken into account. Therefore some small details may deviate from the original KinectFusion system. Though the places where such variations of the used methods have been uniquely noticed, will be marked and furthermore discussed.

After a detailed description of the KinectFusion system in Section 3.1, the actual work, named "Deformable 3D Reconstruction", will be proposed. It consists of two different methods for tracking the non-rigid reconstruction problem. The first one tries to tackle the problem by using a template-based approach in form of two consecutive stages, running two different pipelines for getting the job done (see Section 3.2). In contrast to that, the second one targets the issue by using only one stage and therefore requires only one pipeline to perform a non-rigid 3D reconstruction. This second system is also known as a "model-based" solution, by introducing an additional model for representing the current deformation state and will be discussed in Section 3.3. Because the modifications made on KinectFusion affect both new to be proposed pipelines, these will be discussed only once, on their first occurrence in Section 3.2.

3.1 KinectFusion

The whole pipeline of KinectFusion is shown in Figure 3.1. Blocks, visualized with squared corners, represent essential processing units, whereas the rounded ones show optional tasks. One cycle of the pipeline processes and integrates one image frame of the recording. The transition to a following iteration is marked with a dashed lined arrow. All visualized steps will be explained in detail below.

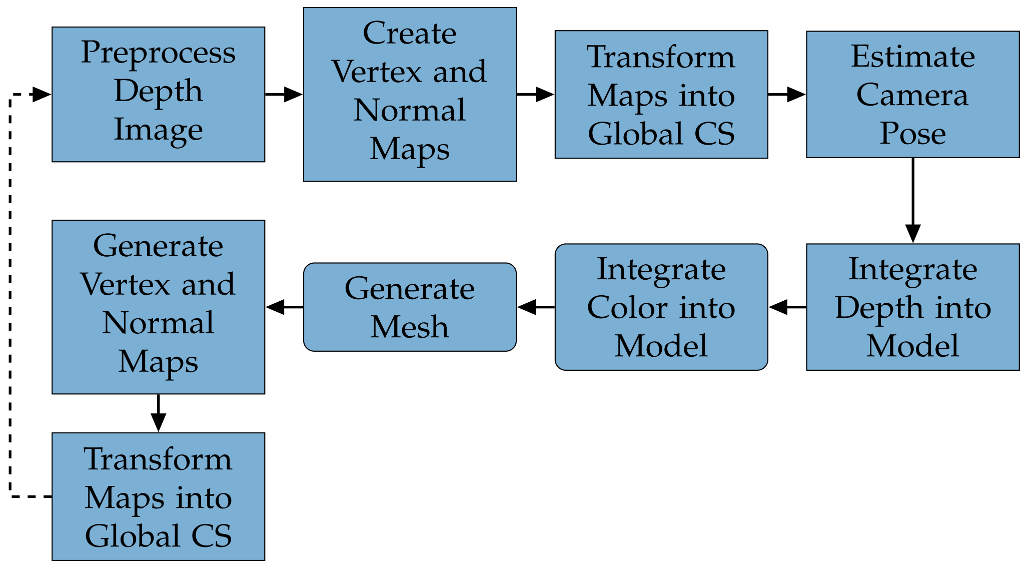
Preprocess Depth Image
Create Vertex and Normal Maps
Transform Maps into Global CS
Estimate Camera Pose
Integrate Depth into Model
Integrate Color into Model
Generate Mesh
Generate Vertex and Normal Maps
Transform Maps into Global CS

Figure 3.1: Pipeline - KinectFusion

3.1.1 Preprocess Depth Image

The first task of KinectFusion deals with the preparation of the input data (see Figure 3.2). This is done by processing a depth image, which is also referred to as “depth map” in the following. It has a fixed resolution, defined by width and height and is structured as an equidistant 2D grid (see Figure 3.3). Every point is accessible by two indices, defined by the row i and the column j and represents a depth value. In contrast to RGB images,

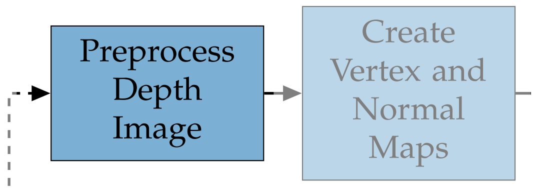

Figure 3.2: Pipeline - KinectFusion - Preprocess Depth Image

in which all points have valid color values, depth images may contain some invalid data. For example, this happens when the sensor of the camera is not able to capture the depth in some areas. For illustration, Figure 3.4 shows an example of a depth image, visualizing the surface of a sphere, in form of depth values. In fact, this is a synthetically generated depth image, where all values are correct. In contrast to that, depth images acquired from real camera sensors contain usually noisy and also invalid data.

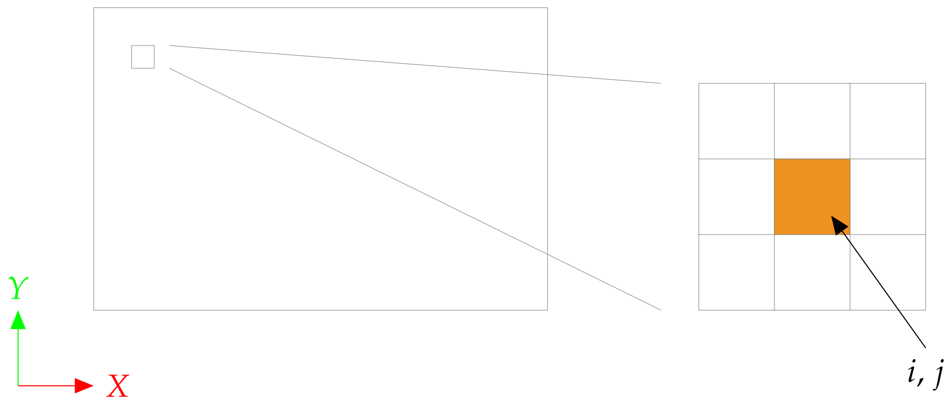

Figure 3.3: Structure of Depth Image, Vertex and Normal Map

The actual processing step applies a bilateral filter (Tomasi and Manduchi, 1998) on the depth image. Commonly this filtering type is used for RGB images, but it is also applicable for depth images, which consist of only one channel. By treating the input as an integral image, the neighbors are known. This is an important information, required for the calculation of the underlying algorithm in the filter. Its output, in form of a modified depth image, will be used in one of the following tasks for estimating the camera

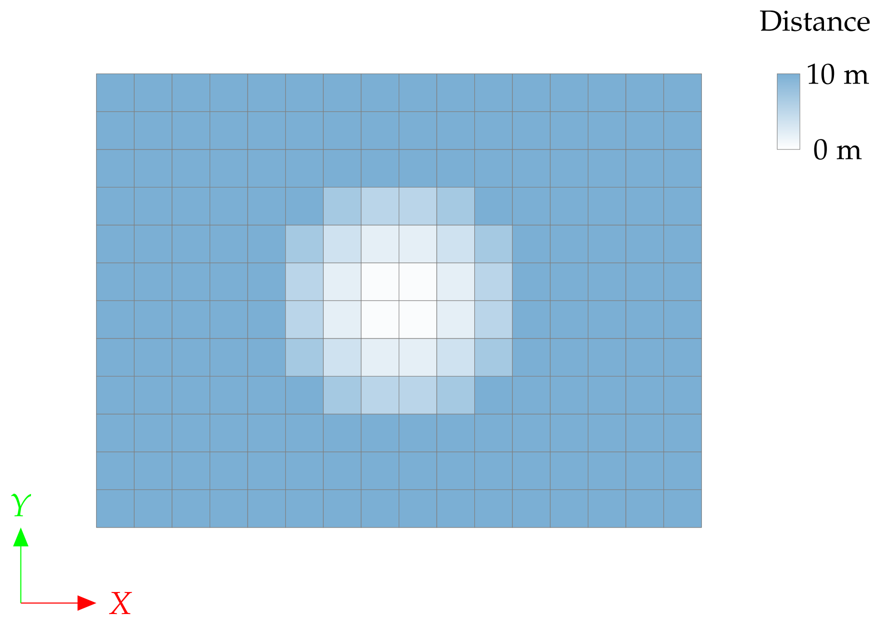

Figure 3.4: Depth Image Showing a Sphere (Resolution = 16x12)

pose. In fact, the only reason for this preprocessing step is to stabilize the camera estimation.

3.1.2 Create Vertex and Normal Maps

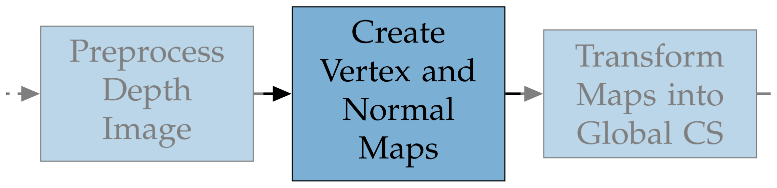

Figure 3.5: Pipeline - KinectFusion - Create Vertex and Normal Maps

Now (see Figure 3.5), the manipulated input data is ready for being projected into the 3D space. The outcome of this operation is in form of an organized point cloud. This type of point cloud is structured like a map, containing a 3D point in every index pair *i* and *j*. Consequently this type of data structure is called "vertex map". While having the same structure and resolution as the input data (see Figure 3.3), it contains three values instead of only one, stored in each position.

For calculating a 3D point V out of a depth value d, besides the information of the currently processed indices pair i, j, also the intrinsic of the camera has to be known. This characteristic can be described by four values f_x, f_y, c_x and c_y. The actual calculation is done by multiplying every depth value by a projection matrix (see Equation 3.1). Missing depth values, encoded in the form of zero values, result in invalid 3D points. For not loosing the organized shape property of the point cloud, also these values are kept stored in the resulting vertex map. Depth values, which exceed a predefined threshold, in especially which are too far away from the camera, are marked as invalid points, instead of being projected. Additionally to the positions, also the normals are calculated in this task. They are stored in the same structure as the point-related data, forming a so-called "normal map". In contrast to the work of Izadi et al. (2011) and Newcombe et al. (2011), where the normals are estimated by calculating a cross product involving the $i+1$ and $j+1$ neighbors, the estimation of the normals is done by using the covariance matrix, as found in the implementation of PCL. This leads to a smoother normal field, but may let the normals show in the wrong directions in some areas.

$$V = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} \frac{1}{f_x} & 0 & 0 \\ 0 & \frac{1}{f_y} & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} i - c_x \\ j - c_y \\ 1 \end{bmatrix} \cdot d \tag{3.1}$$

As another preparatory step for the camera pose estimation (see Section 3.1.4), the vertex and normal maps have to be created in three different sizes, referred to as "vertex and normal map pyramid" in the following. Therefore, besides the already acquired map pair, two additionally pairs have to be produced. This is done by first scaling down the previously bilaterally filtered depth map by applying the factor two. By averaging and joining the values of the depth map, defining patches in the size of 2x2, a half sized depth map is created. By also dividing each single intrinsic value by the factor two, a valid, half sized vertex map can be produced through out the previously mentioned projection. By repeating these steps once more, in sum, two additional vertex maps are created. Finally, their related normal maps are calculated, as already done for the original sized vertex map.

3.1.3 Transform Maps into Global Coordinate System (1)

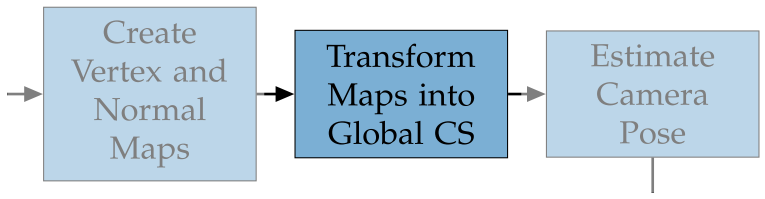

Figure 3.6: Pipeline - KinectFusion - Transform Maps into Global Coordinate System (1)

At this step, the vertices are transformed into the global coordinate system (CS) (see Figure 3.6). This is done by a multiplication and following addition on the previously calculated content of the vertex maps, using the rotation and translation components of the camera matrix (see Equation 3.2). Initially, this matrix is defined to represent a user defined position and viewing angle, which will be important for the next tasks. The general objective of this step is to put the vertex maps into a common 3D space, independently of the current position and angle of the camera. The structure of the vertex maps remains the same and furthermore the indices of the points are not changed during this transformation. In contrast to the vertices, the normals are only multiplied with the rotation matrix. This is because they are vectors instead of positions (see Equation 3.3). Both calculation types are performed on vertex and normal maps of all three available sizes.

$$V_g = \begin{bmatrix} v_{xg} \\ v_{yg} \\ v_{zg} \end{bmatrix} = R \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} + t \tag{3.2}$$

$$N_g = \begin{bmatrix} n_{xg} \\ n_{yg} \\ n_{zg} \end{bmatrix} = R \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \tag{3.3}$$

3.1.4 Estimate Camera Pose

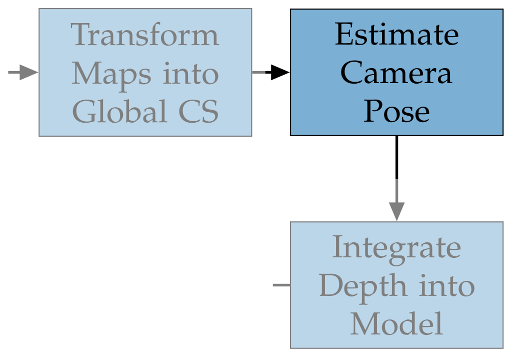

Figure 3.7: Pipeline - KinectFusion - Estimate Camera Pose

Now the Iterative Closest Point (ICP) algorithm (Besl and McKay, 1992) takes place - used for acquiring the camera pose, by aligning the current maps onto the previous ones (see Figure 3.7). If it is the first iteration of the pipeline, there will be no previous maps available and therefore this step will be skipped in this case.

By only being interested in integrate new data, processed in the following section, the movement of the camera is a matter of moment. If the result of this task, in form of a camera matrix, is too similar compared to the previous one, the current iteration is aborted at this step and the whole pipeline starts from the beginning. This additional check is also an extension to the original KinectFusion method, in which such a test is not mentioned at all.

As shown in Figure 3.8, the aligning of the vertex and normal maps is divided into four different subtasks. To get a more stable and faster result, the ICP is performed by using a vertex and normal map pyramid. This is the reason for the creation of the differently sized map pairs, performed in the previous sections. Starting with the smallest map pair, each size is processed multiple times, more precisely, four, five and ten times, to converge to a meaningful result. In total, this leads to 19 iterations. As obvious, the transition to the next iteration is visualized by a dashed lined arrow.

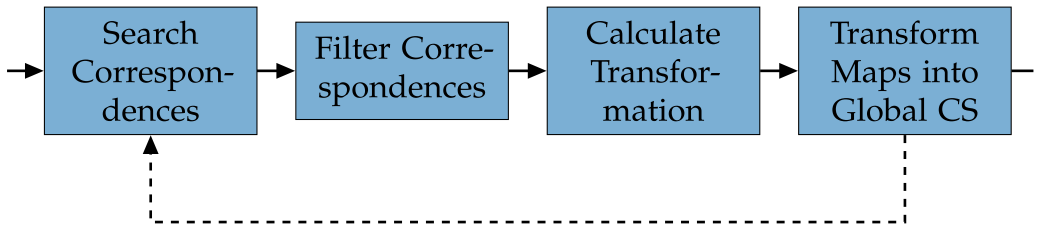

Figure 3.8: Pipeline - KinectFusion - Estimate Camera Pose - Subtasks

Search Correspondences

The correspondences are determined by performing a projective data association (Rusinkiewicz and Levoy, 2001). In its first step, the points of the globally aligned previous vertex and normal maps are transformed into the current coordinate system. After that, these points are projected into the image plane. The resulting i and j indices already define the correspondences to the current map pair. In fact, this procedure is exactly the inverse calculation as done in the Sections 3.1.2 and 3.1.3. Vertices, which have neither a valid position, nor a related normal information in the corresponding normal map, are already ignored at this stage. Therefore those points will not obtain a correspondence. In contrast to those, all remaining vertices acquire a correspondence, never the less of their resulting quality.

Filter Correspondences

For getting rid of bad correspondences, also called outliers, point pairs which have a too high deviation in the quantities of vertex position and normal angle, are removed. This is estimated by calculating two specific values d_1 and d_2, as shown in Equations 3.4 and 3.5. The calculation of the normal deviation differs compared to the original KinectFusion method, shown in Equation 3.6. By applying these calculations on every found correspondence pair, their euclidean distances and angle differences are specified. If one of those values exceed a defined threshold e_i, the appropriate correspondence will be removed (see Equation 3.7).

$$d_1 = \|V_g - V_{g,prev}\|^2 \tag{3.4}$$

$$d_2 = \|N_g \times N_{g,prev}\|^2 \tag{3.5}$$

$$d_3 = \text{abs}(N_g \cdot N_{g,prev}) \tag{3.6}$$

$$d_i \overset{!}{<} e_i \tag{3.7}$$

Calculate Transformation

The used distance metric for the performed optimization is called “Point-To-Plane” (Y. Chen and Medioni, 1991). As the name already implies, its goal is to minimize the sum of all distances, ranging from a point to a plane, between the found correspondences (see Equation 3.8). The result is in form of a rotation matrix R_{ICP} and a translation vector t_{ICP}, forming a single transformation matrix. Though this matrix describes only the transformation between the current and the previous map. Therefore its rotation and translation components are accumulated onto the global camera matrix, finally evolving into the new camera pose.

$$E_1 = \min(\sum_{i=1}^{n} \|(R_{ICP} \cdot V_i + t_{ICP} - V_{g,corr(i),prev})^T \cdot N_{g,corr(i),prev}\|^2) \tag{3.8}$$

Transform Maps into Global Coordinate System

By having now a new transformation matrix available, acquired in the previous step, the vertex and normal maps are updated to represent the new state. This is done by transforming the current vertex and normal maps from the current coordinate system into the global one, by applying the updated rotation matrix and translation vector. The maps are now ready for the next iteration.

3.1.5 Integrate Depth into Model

By using the updated camera matrix and its intrinsics, the raw depth image is going to be integrated into the used storage model (see Figure 3.9). This

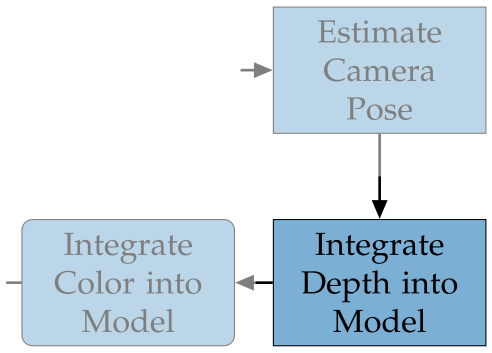

Figure 3.9: Pipeline - KinectFusion - Integrate Depth into Model

model is described by Truncated Signed Distance Functions (TSDF) (Curless and Levoy, 1996; Izadi et al., 2011). It is defined as a cubic, equidistant voxel grid holding two specific values on every voxel (see Figure 3.10). Any value pair can be accessed by three indices i, j and k. The first value is the actual TSDF value and represents the distance to the surface. It is normalized in the interval of -1 and 1, in which the value of 0 represents the surface. By definition, the maximum negative value -1 describes empty space (air), whereas the positive maximum value 1 represents a still unexplored area. Besides that, the second value defines the weight for determining the quality of the first value. The dimension of the voxel grid is defined by the quantity of length and a dimensionless resolution, having the same value in all its three dimensions (e.g.: $3\ \mathrm{m}^3$ and 512^3 voxels, respectively). Because the voxels are equally spaced, the fixed distance between each of them can be easily computed, by dividing the length through the dimension. This cell size is needed to convert the indices into the global voxel position (see Equation 3.9). Because of integrating the data into the voxel's center, the addition of 0.5 has to be taken care of. Figure 3.11 shows a voxel grid including its centers, represented by spheres. Besides the definition of the cube's properties, also the camera pose is of main interest during the data integration.

$$V_g = \left(\begin{bmatrix} i \\ j \\ k \end{bmatrix} + 0.5 \right) \cdot \text{cell size} \tag{3.9}$$

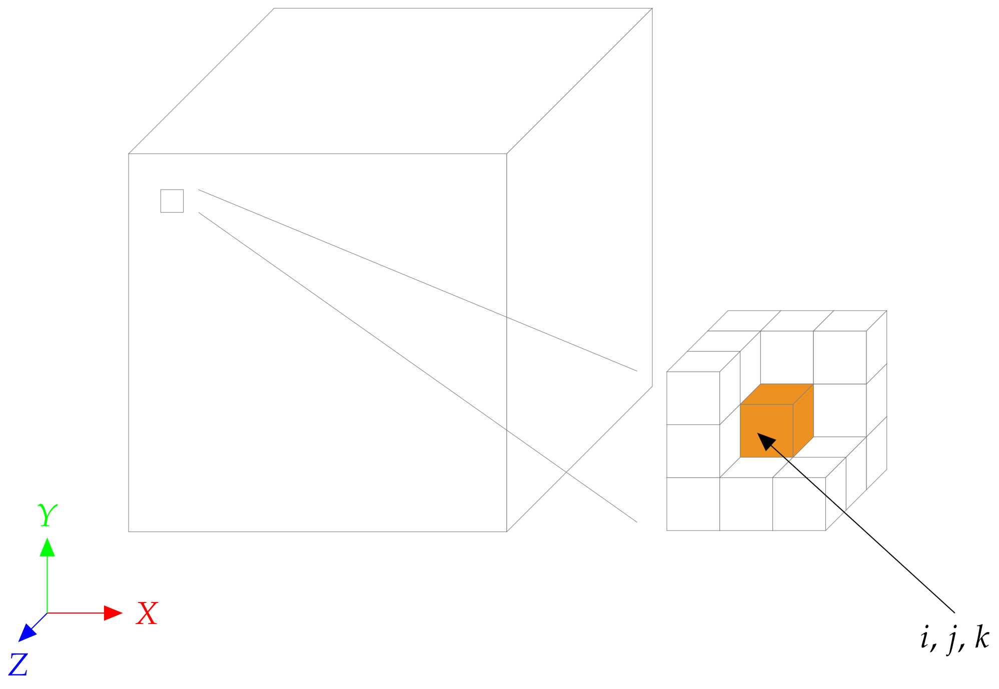

Figure 3.10: Cubic Voxel Grid

To gain a better understanding of how a TSDF volume may look like, refer to Figure 3.12. The illustrated model contains a flat surface, which is visualized by its TSDF values, but without their corresponding weights. The whole voxels have been colored, although the position of its representing value is located in the voxel's center. The surface, marked by the color blue, is either orientated in the positive or negative X-Direction. Because of using the same color for the negative as well as positive maximum values, -1 and 1 respectively, the front and the back cannot be distinguished anymore, from each other.

To integrate the depth data, every voxel is considered on its own. This means that the iteration is performed over all voxels, but without having knowledge about the actual geometry. As illustrated in Figure 3.13, only voxels which are occluded by the projected points of the to be integrated depth image are updated. Furthermore it has to be noted, that the visualized image is oriented exactly as like the camera, by also having the same center in their common plane. Now, the goal is to integrate the depth image into

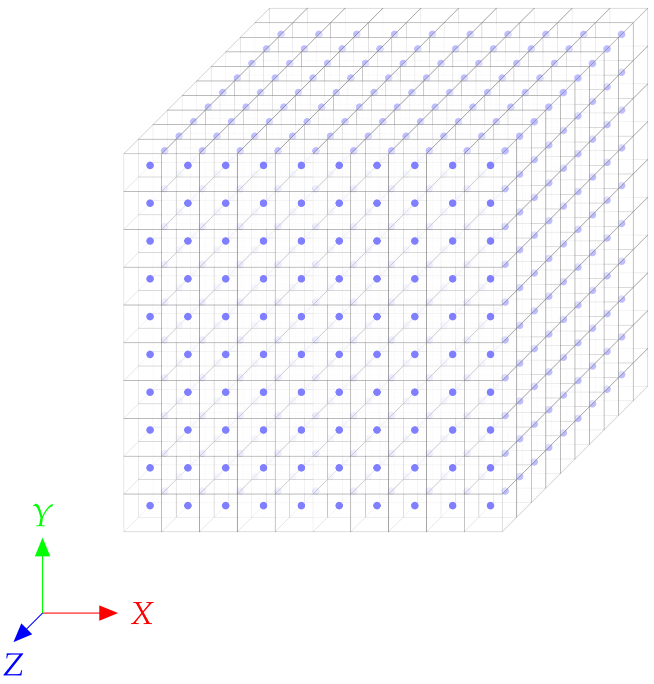

Figure 3.11: Cubic Voxel Grid Showing its Centroids (Resolution = 10^3)

the TSDF model, by a projection as shown in the illustration. The used coordinate system for the storage model is the global coordinate system. Therefore by using the voxel's three indices i, j and k, the global position can be easily determined (see Equation 3.9). With the knowledge of the current camera matrix, the voxel is being transformed into the current coordinate system. Using the intrinsics of the camera, the projection into the image plane is performed. This is exactly the opposite calculation as done in Sections 3.1.2 and 3.1.3. The result of the projection consists of two values, which are rounded to become integer values, forming two indices i and j, for accessing the actual depth out of the depth map. If these indices do not exceed the range of the image, its appropriate depth value can be accessed and used for further processing. In the other case, either if one of these indices is negative or higher than the resolution of the image, the integration of the voxel is aborted. Also if there is no depth value available, the integration is skipped too. Otherwise the value of the Signed Distance Function (SDF) is going to be calculated (see Equation 3.10), by using the

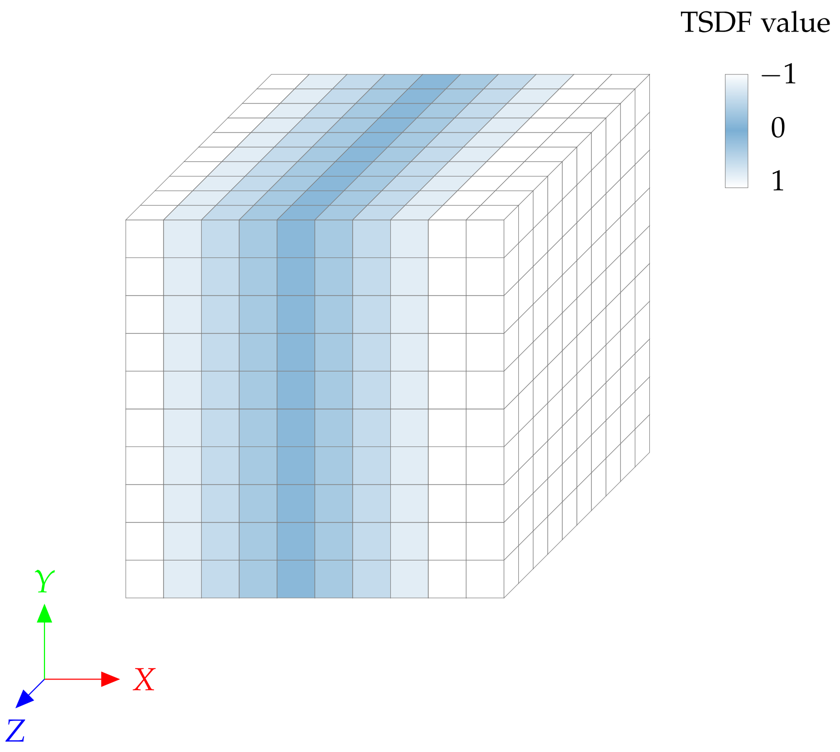

Figure 3.12: Cubic Voxel Grid Containing a Flat Surface (Resolution = 10^3)

appropriate value of the depth image. Based on the resulting SDF value sdf, a decision will be made, if the to be integrated data is of interest. This is the case if the processed depth is near enough to the surface and is estimated by comparing the SDF value with a maximum allowed distance d_{trunc}. If the SDF value is lower than the negative d_{trunc} constant, then the integration is stopped at this place (see Equation 3.11). This means, voxels located behind the surface and being farther away than d_{trunc}, will not be updated. Otherwise, the TSDF value $tsdf$ is calculated, by firstly normalizing the SDF value and secondly forcing it into the range of -1 and 1. The normalization is processed by dividing the value by the constant d_{trunc} (see Equation 3.12). After that, the average TSDF value $tdsf_{average}$ and its associated weight w are calculated, as shown in Equations 3.13 and 3.14. Obviously, the weights are simply incremented by 1, as long as the maximum value w_{max} has not been reached. Finally, both values are stored at the certain position of the currently processed voxel. By doing this procedure for every single voxel, a huge number of iterations have to be performed.

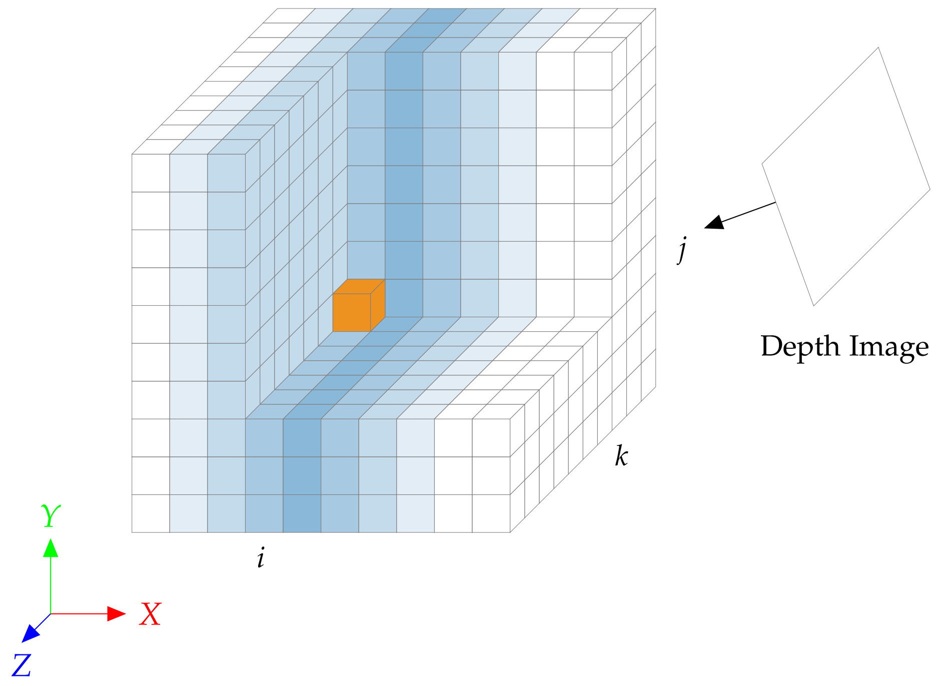

Figure 3.13: Integrate Depth Image into Voxel Grid (Resolution = 10^3)

$$sdf = \|t - V_g\| - d \tag{3.10}$$

$$sdf \overset{!}{\geq} -d_{trunc} \tag{3.11}$$

$$tsdf = \min(\max(\frac{sdf}{d_{trunc}}, -1), 1) \tag{3.12}$$

$$tsdf_{average} = \frac{tsdf_{prev} \cdot w_{prev} + tsdf \cdot w}{w_{prev} + w} \tag{3.13}$$

$$w = \min(w_{prev} + 1, w_{max}) \tag{3.14}$$

For getting a better overview of the even mentioned depth integration, all required steps are summarized in Listing 3.1.

```
for $i$, $j$, $k$ in TSDF model
    $V_g$ ← calculate global position
    $V$ ← transform to current coordinate system
    $i$, $j$ ← project into image plane
    if $i$, $j$ in depth image
        $d$ ← acquire depth value
        if $d$ is valid
            $sdf$ ← calculate SDF value
            if $sdf$ is greater or equal than $-d_{trunc}$
                $tsdf_{average}$ ← calculate average TSDF value
                $w$ ← increment weight
                save $tsdf_{average}$ and $w$ at voxel $i$, $j$, $k$
```

Listing 3.1: Integrate Depth into Model

3.1.6 Integrate Color into Model

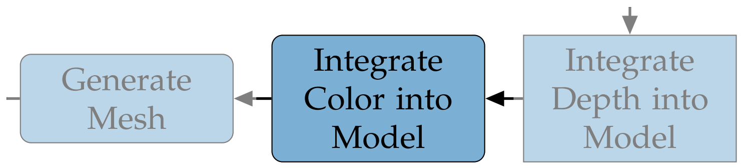

Figure 3.14: Pipeline - KinectFusion - Integrate Color into Model

The optional integration of the color image may follow the depth integration directly (see Figure 3.14). This form of integration is not defined by KinectFusion, therefore the proposed method is taken from the KinectFusion based implementation of PCL. While not being a requirement of the 3D reconstruction pipeline, the color is an important perception cue of the human beings.

The handling of the color is very similar as compared to the depth. Analog to the TSDF volume, an additional volume for storing the color, is introduced. This volume has the same properties, but instead of using a TSDF value and a weight per voxel, an RGB vector is considered. The integration is also processed, by iterating over the volume's voxels and projecting them into the image plane of the current coordinate system. But instead of calculating the

SDF value, the euclidean distance is estimated, for defining the integration criteria. If this distance does not exceed the already introduced threshold d_{trunc}, the RGB components, defined as a vector of three short typed values, will be integrated. The average RGB vector $rgb_{average}$ is calculated as shown in Equation 3.15. Obviously, this is pretty much the same formula as used for the calculation of the TSDF's average value (see Equation 3.13). But instead of incrementing associated weights in every integration, the weighting is done by involving a constant value w_{const} only. This way, the to be integrated color has always the same importance. As the TSDF value is defined by a specific range, going from -1 to 1, the three color values are valid between the range of 0 and 255 (see Equation 3.16).

$$rgb_{average} = \frac{rgb_{prev} \cdot w_{const} + rgb}{w_{const} + 1} \tag{3.15}$$

$$rgb_{average} = \min(\max(rgb_{average}, 0), 255) \tag{3.16}$$

3.1.7 Generate Mesh

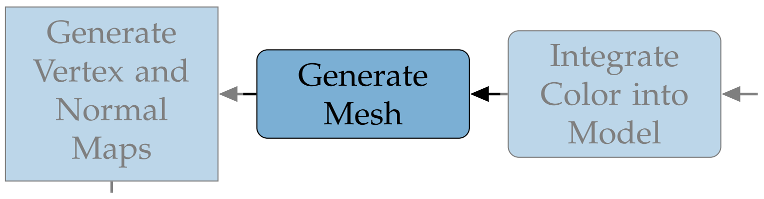

Figure 3.15: Pipeline - KinectFusion - Generate Mesh

At this stage a triangle mesh may be constructed (see Figure 3.15). The creation of a mesh is not needed for the pipeline, but in most cases the preferred representation of the result. If this task is planned to be executed, then it may be performed either in a defined interval or at the end only once. This is because of being an calculation intensive task and furthermore of avoiding the generation of much data. A common approach for getting such a mesh is to use the Marching Cubes algorithm (Lorensen and Cline, 1987). This is also true for this pipeline, by being the chosen method for this task. The output of the applied algorithm is in from of a triangle mesh and by being directly acquired from the TSDF model, placed in the global coordinate system.

If the color integration is activated in the pipeline, then also the colors need to be applied onto the mesh. This is simply done by searching for every vertex of the previously created mesh, the corresponding color value in the 3D color volume. Because of being in the same coordinate system, this involves only a calculation of the voxel index, based on the global vertex position (see Equation 3.9). Finally by using this index, the color of the voxel can be fetched from the color volume. By doing this for every single vertex, the complete color information is applied onto the mesh.

3.1.8 Generate Vertex and Normal Maps

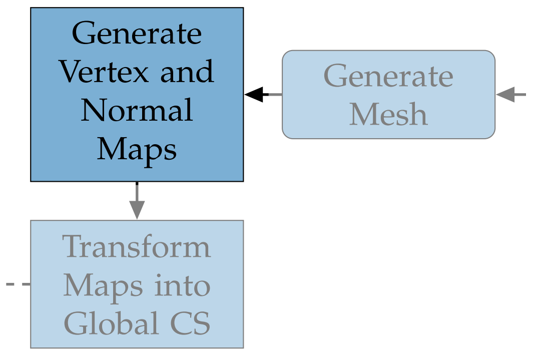
Generate Vertex and Normal Maps
Generate Mesh
Transform Maps into Global CS

Figure 3.16: Pipeline - KinectFusion - Generate Vertex and Normal Maps

The second to last step of the pipeline deals with generating synthetic vertex and normal maps (see Figure 3.16). This is done by raycasting the implicit surface information out of the currently saved TSDF model (Parker et al., 1998). By emitting rays, for every index pair of the to be generated vertex map onto the model, the appropriate global position can be calculated and stored. Actually, the ray stops when it enters the transition between negative and positive TSDF values, means when the surface is hit. This is done for each i, j index pair of the resulting map. By taking the neighbors of the estimated position into account, the surface can be predicted and from that the normal information is derived. This results in getting the vertex and also the normal map, positioned in the global coordinate system, already in one shot. While both maps are created in the original size only, their two minimized versions have to be created too. The procedure for acquiring this vertex and normal map pyramid is the same as described in Section 3.1.2. But instead of averaging the scalar values, the now available 3D points and vectors have to be used.

The generated maps are stored in the previous map containers for the use in the next iteration. In the first iteration of the pipeline, this step is omitted and the current maps are simply adapted for being used as the previous maps in the next run. By using, in this case, the vertex and normal maps acquired from the bilaterally filtered input, instead of extracting the information out of an under-determined model, a better result is to be expected.

3.1.9 Transform Maps into Global Coordinate System (2)

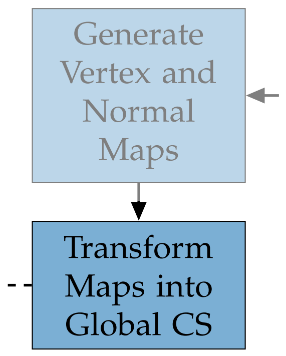
Generate Vertex and Normal Maps
Transform Maps into Global CS

Figure 3.17: Pipeline - KinectFusion - Transform Maps into Global Coordinate System (2)

The previous step produces the maps already in the global coordinate system, but as shown in Figure 3.17, a transformation may follow. Actually this is not required, in the way how the raycasting is performed. By extracting the information directly out of the TSDF model, the resulting points are already positioned in the global coordinate system. Though the proposed pipeline illustrates an abstract approach, in which other methods for the generation of the maps could be applied. In fact, this is the case in the next section, where the maps are obtained by a different method and a transformation has to follow.

3.2 Deformable 3D Reconstruction - Template-Based

The first variant for facing the non-rigid reconstruction problem runs in two different stages. In the first stage, it is assumed that no deformation happens and the input data is used for a static 3D reconstruction. Then, at some specific frame $F = F_{start} - 1$, the depth integration is stopped. Finally the observation of any deformation is going to be honored in the following iterations, marking the second stage. This template-based approach is similar to the work of Zollhöfer et al. (2014), in which a static reconstruction is also performed first. Therefore this involves two different pipelines, having a defined timestamp when they are going to be switched. Before the specific time has been reached, the KinectFusion pipeline is run as already shown in Section 3.1 and illustrated in Figure 3.1. After that, a triangle mesh is created. This mesh is used in the following as the container, replacing the TSDF volume, for representing the current state. Then the actual, newly proposed pipeline comes in place, performing the tasks as shown in Figure 3.18. Its key functionality consists of a combination of observing and integrating any occurring deformations. These transformations, in form of translations, are integrated into the mesh directly. Therefore no additional storage model is needed. Because of having a fixed set of vertices defined in the mesh, the template-based method does not have to deal with the uncertainty of making the decision if a point is newly occurring or has only been moved. This makes that method very stable.

The exchanged path of the new pipeline is colored orange, while the start and the end, colored blue, stay the same. Though their methods have to be slightly modified to fulfill the newly needed behavior. These modifications of the already available blocks and the newly introduced path will be discussed below. Because of no need to change anything in the first three tasks, they are going to be skipped in the description below.

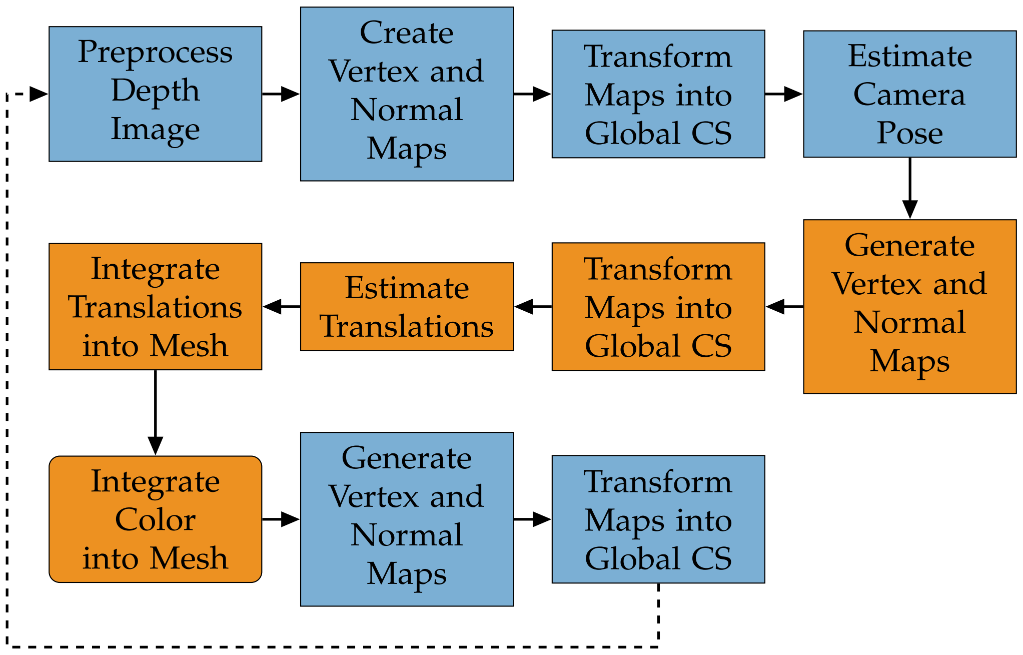
Preprocess Depth Image
Create Vertex and Normal Maps
Transform Maps into Global CS
Estimate Camera Pose
Generate Vertex and Normal Maps
Transform Maps into Global CS
Estimate Translations
Integrate Translations into Mesh
Integrate Color into Mesh
Generate Vertex and Normal Maps
Transform Maps into Global CS

Figure 3.18: Pipeline - Deformable 3D Reconstruction - Template-Based

3.2.1 Estimate Camera Pose

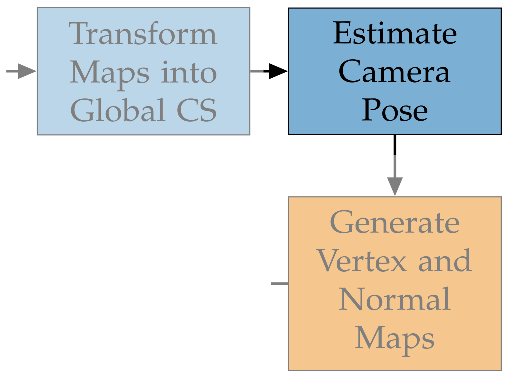
Transform Maps into Global CS
Estimate Camera Pose
Generate Vertex and Normal Maps

Figure 3.19: Pipeline - Deformable 3D Reconstruction - Template-Based - Estimate Camera Pose

The first step which needs to be modified for the use in the new approach deals with the estimation of the camera pose (see Figure 3.19). Because of being only interested into a single body instead of capturing a complete scene, the content of the vertex and normal map pyramid is pre-filtered, using a bounding box. Its size is equal to that of the TSDF volume, to contain only the relevant data of the model's frustum, after the filtering. This modification affects the ICP algorithm only and enables the movement of the body itself, by not having any static data in the background anymore, for example. Therefore the system does not distinguish between a camera or a body movement.

The other change compared to the step mentioned in Section 3.1.4 is related to the possible skipping of the data integration, when the resulting camera matrix is too similar to the previous one. Obviously, a body may also deform, regardless, whether the camera is moving or not. Without this change, a possible deformation would not be observed in this case, by skipping the following tasks.

3.2.2 Generate Vertex and Normal Maps (1)

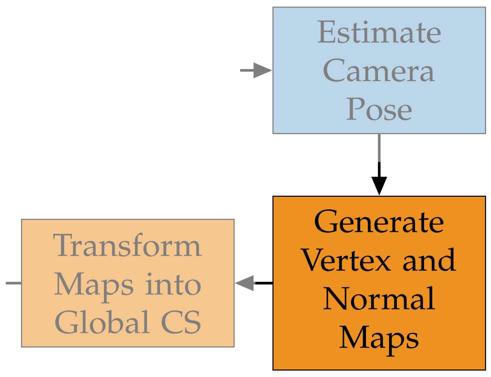
Estimate Camera Pose
Transform Maps into Global CS
Generate Vertex and Normal Maps

Figure 3.20: Pipeline - Deformable 3D Reconstruction - Template-Based - Generate Vertex and Normal Maps (1)

The generation of the vertex and normal maps marks the beginning of the orange colored, exchanged path (see Figure 3.20). This task treads only with the previous maps, representing the TSDF model. Therefore the output of this step is stored in the previous maps container. Theoretically this step is not needed at this point, but it may improve the quality of the translation estimation. The idea behind that step is to get the previous vertex and normal maps rendered in the updated current camera pose. Therefore the content of the maps will become more accurate and the handling of them becomes easier too, because of having to take care of only one camera pose anymore.

In contrast to the original KinectFusion system, the vertex map is now generated by doing a rasterization instead of doing a raycasting. Because of having now a different form of input data, the generation has to be performed in an alternate way. While the raycasting in KinectFusion is done directly out of the TSDF model, now a triangle mesh is used as the source representation for that task. Obviously, at this point a triangle mesh has to be available already.

The following applied rasterization procedure is based on the work of Scratchapixel (2018). For getting a rasterized map out of a mesh, every single face needs to be processed. First, the faces, in form of triangles have to be transformed into the current coordinate system and projected into the image plane. They are now in the same plane as the target vertex map and the triangles are ready to be virtually drawn onto the map. This is done by iterating over all triangles and also all destination indices of the to be created vertex map, performed in a cascaded fashion. For the detection, if a to be drawn vertex is inside the currently processed triangle, the edge function (Pineda, 1988) is used. This is done by estimating the relative position of the vertex to all three edges of the triangle. If the vertex is in all three cases either on the right side or on the edge itself located, then the vertex is definitely in the triangle. While this works on usual front faces, back faces will not get treated properly by this test. Therefore, in this case, the ordering of the vertex indices, defined by the triangles are changed beforehand. The detection of a back face is done by checking the normal direction of the triangle beforehand. If an index pair of the target map has multiple candidates for inclusion, then their positions are going to be compared. In fact, the one with the lowest Z-Value is chosen. For improving the speed of the rasterization, all triangles are only processed in the area of their precalculated bounding box, instead of processing all points in the target map. For acquiring the interpolated Z-Value at the target position, the barycentric coordinates are needed. They are calculated using the results of the edge function. At the end, the three inverted Z-Values of the triangle are used to get the correct interpolated resulting value with the help of their current barycentric coordinates.

Having the goal to get the lowest Z-Value on every single target index pair, the previous described algorithm has to be processed sequentially. This results in gaining a long run for the rasterization process, especially because of expecting a big triangle mesh as input. Therefore the idea is to produce multiple temporarily vertex maps in parallel, by dividing the mesh into parts. Because of having no knowledge about the neighborhood, the segmenting is simple done by rendering every n-th triangle using an own thread. The result of this parallel approach is in form of multiple vertex maps, containing no reasonable values when considered on their own. But by merging them together, taking always the lowest Z-Value on every index pair, a valid representation of the mesh is produced in form of a single vertex map. While this procedure requires more memory, it improves the speed drastically without loosing quality.

Finally, the normals are calculated the same way as described in Section 3.1.2. Though additionally, the orientations of the normals are checked for showing in the right Z-Direction. This means, all normals having a negative Z-Value, are turned by negating all three components of the vectors. The creation of the vertex and normal map pyramid is also performed identically to the procedure in the raycasting task (see Section 3.1.8).

3.2.3 Transform Maps into Global Coordinate System (1)

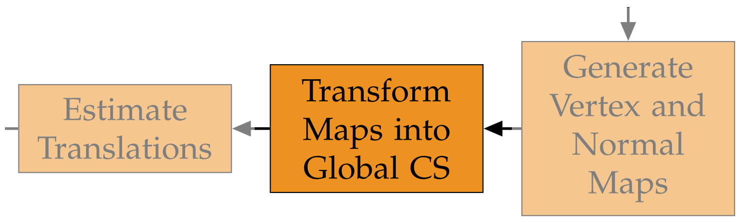
Estimate Translations
Transform Maps into Global CS
Generate Vertex and Normal Maps

Figure 3.21: Pipeline - Deformable 3D Reconstruction - Template-Based - Transform Maps into Global Coordinate System (1)

Now the previously acquired maps and the maps derived from the current input are transformed into the global coordinate system (see Figure 3.21). Therefore the only difference between both maps may be a possible occurring deformation, which will be determined in the next step.

3.2.4 Estimate Translations

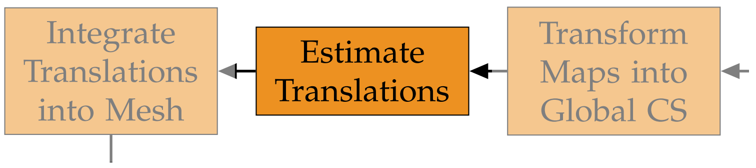
Integrate Translations into Mesh
Estimate Translations
Transform Maps into Global CS

Figure 3.22: Pipeline - Deformable 3D Reconstruction - Template-Based - Estimate Translations

Now the most interesting part of the new approach starts to happen, by dealing with the estimation of possible arising translations (see Figure 3.22). To keep track of deformations, there has to be some logic for detecting it. The idea is to use something similar to ICP, but getting one transformation for every single point, instead of only one common global one. Therefore the image is split into multiple small patches, based on the neighborhood. These patches are used for estimating the actual transformation. By using the property of the vertex and normal maps to contain an organized point cloud, the neighbors of every point are already known. The smallest possible size of a patch results by taking one neighbor in all directions of a vertex and normal map pair (see Figure 3.23). Obviously, the resulting patch contains of nine points in this case. Generally, by using Equation 3.17, the number of involved points can be calculated.

$$\text{Number of points} = (\text{Number of neighbors} \cdot 2 + 1)^2 \tag{3.17}$$

To get a more robust result, the number of used points may be increased. For example, by defining the patch to take five neighbors, would result in 121 points. Though, for a better demonstration of the to be shown approach, always a patch size of nine points will be used in the following. Obviously,

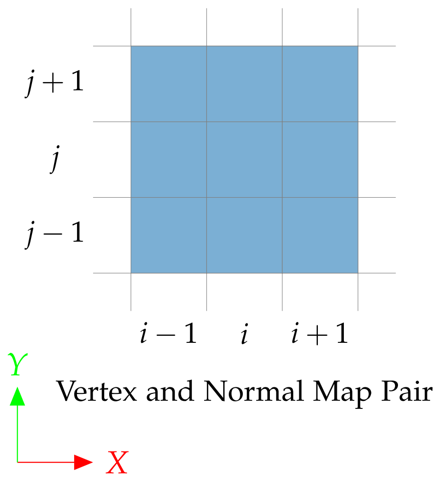

Figure 3.23: Patch (Number of Neighbors = 1)

the number of the to be created patches is equal to the number of points contained in the vertex and normal maps. The forming of the patches is performed on the previous map pair. Their to be found correspondences will form the opposite point pattern of the current maps. This means the transformations are estimated from the previous onto the current maps. For making things easier, faster and to save memory, only the translations will be acquired, instead of dealing with costly transformation matrices.

As shown in Figure 3.24, the translation estimation is split into five different subtasks. Similar to the pyramid approach, used in the ICP task, this pipeline is executed multiple times. But in contrast of increasing the resolutions of the used maps in each pyramid step, the size of the patches, more precisely, the number of used neighbors are varied. In fact, this number will be decreased in every cycle by involving five, three and two neighbors in each iteration respectively. This results in getting a more fine-grained translation map in every iteration.

Search Correspondences

The used attributes for finding the best point pairs to form the correspondences, are the position and the normal of the points. This means that only

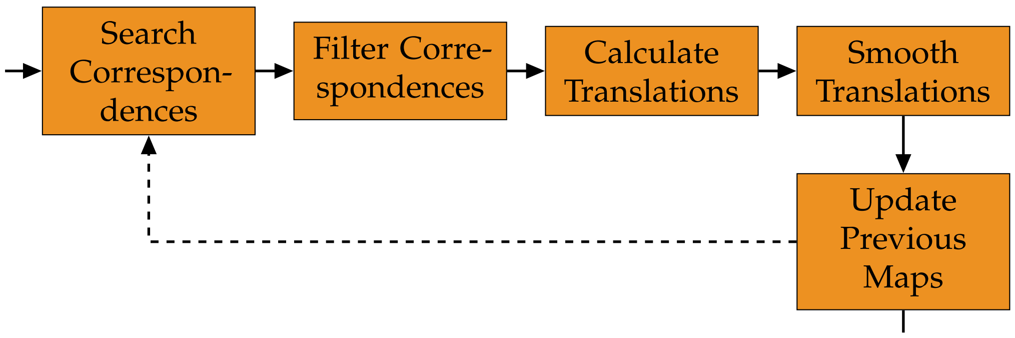

Figure 3.24: Pipeline - Deformable 3D Reconstruction - Template-Based - Estimate Translations - Subtasks

points which have a valid normal can either be acquired or considered as a correspondence. The measure for its quality is defined by its distance and the normal deviation, in which the combination of both should be as small as possible. To get this value, the cross product between the normal of the previous normal map and the difference between the positions of the current and previous vertex maps is calculated (see Equation 3.18). This metric is also known as "normal shooting" (Rusinkiewicz and Levoy, 2001; Y. Chen and Medioni, 1991; Weisstein, 2018). For a better comparison, the vector is broken down to a scalar value d_4, by calculating the vector's length.

$$d_4 = \|N_{g,prev} \times (V_g - V_{g,prev})\|^2 \tag{3.18}$$

Generally, the search of the correspondences is performed in both directions by using the previous maps as reference first and finally the current one. Therefore the output of this stage is in form of two correspondences maps, though only one of them will be actually used for the translation estimation.

The search range for finding a correspondence in the other map is also defined by a patch. The currently processed point from the previous vertex and normal maps will be compared with all other points of the search patch of the current map pair (see Figure 3.25). The center of the patch is defined by the same index as used for the reference maps. By including the surrounding neighbors, the patch is fully determined. Involving one neighbor in each direction, as already shown previously, will result in having nine possible correspondences candidates (see Equation 3.17). Again this size will be used for the illustration, but actually the number of 5 neighbors have been chosen for this method. If a point obtains multiple correspondences, having the same calculated distance value d_4, the point with the smallest distance in relation to the indices, is taken. For getting a better overview, the procedure of finding the correspondences is summarized in Listing 3.2.

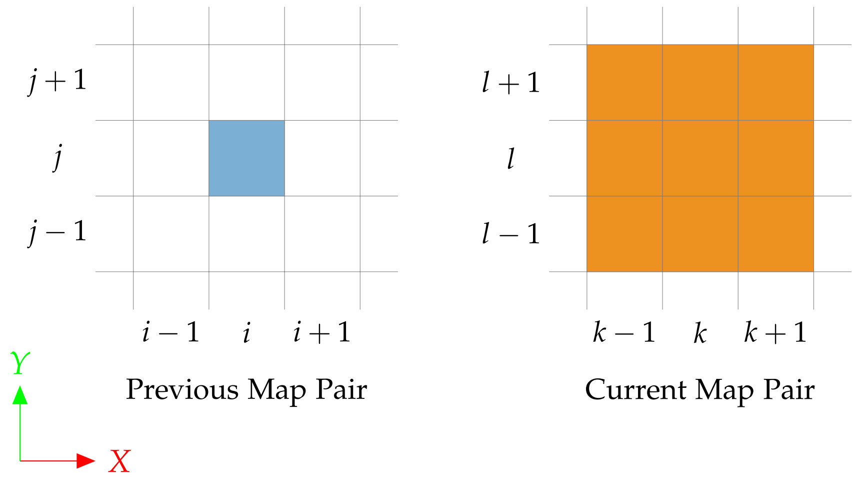

Figure 3.25: Search Range between Map Pairs (Number of Neighbors = 1)

```
for $i$, $j$ in map pair
    for $k$, $l$ in patch of $i$, $j$
        $d_4 \leftarrow$ calculate distance
        if $d_4$ is minimum
            save $d_4$ and $k$, $l$ at indices $i$, $j$
```

Listing 3.2: Search Correspondences

Filter Correspondences

As soon as the best point pairs are estimated, they are checked whether they are good enough to be further used or not. This decision will be taken in two steps, by using two different methods.

The first step at the filtering stage deals with comparing the acquired correspondences maps, one containing the relation from the previous to the current maps while the other defines the other direction. If two correspondences, one in each direction, contain the exactly same point pairs, then the requirement of the first test is fulfilled and the correspondence of the previous map pair will be preserved. Otherwise it is rejected and will not be available for further use anymore. This type of filtering greatly improves the quality of the correspondences and ensures that no target point will be used multiple times as correspondence. At the end of this stage, only the correspondence map which contains the mapping from the previous onto the current vertex and normal maps, will remain.

The second and also last step in the filtering process is responsible for preserving correspondences only, which fulfill some defined geometric thresholds. To achieve this, every point pair is checked by its euclidean distance d_1 (see Equation 3.4), normal angle deviation d_5 (see Equation 3.19) and normals to camera deviations d_6 and d_7 (see Equations 3.20 and 3.21). If one of these properties exceed one of the respectively beforehand defined thresholds e_i, then the appropriate correspondence will be rejected (see Equation 3.7). In contrast to Equation 3.5 in Section 3.1.4, the normal deviation is calculated the same way as in the original KinectFusion method (see Equation 3.6), for also being aware of the direction of the normals. Furthermore it can be noticed, that the real angular dimensions are calculated, by using the arccos function.

$$d_5 = \arccos(N_g \cdot N_{g,prev}) \tag{3.19}$$

$$d_6 = \arccos(N_{g,camera} \cdot N_{g,prev}) \tag{3.20}$$

$$d_7 = \arccos(N_{g,camera} \cdot N_g) \tag{3.21}$$

The four calculated values are also used for defining the quality of a correspondence, which develops into a weight w_c, as shown in Equation 3.22. The values are normed in respect to their corresponding maximum valid threshold values e_i. The weights, associated to their correspondences will become important in the next section.

$$w_c = 1 - \frac{\frac{d_1}{e_1} + \frac{d_5}{e_5} + \frac{d_6}{e_6} + \frac{d_7}{e_7}}{4} \tag{3.22}$$

Calculate Translations

So for getting a translation for any single point implies, beside its own correspondence, also the use of up to eight additional point pairs to satisfy the definition of a patch. Therefore the transformation estimation algorithm can be applied to every point, where at least its own correspondence is known. If some neighbors do not have any correspondence, then they are simply not involved in the estimation process. This means that the content of the patch can vary between one and nine point pairs. See Figure 3.26 for a possible patch, used for the translation calculation. This patch example shows, that not all valid points from the previous vertex and normal maps have a correspondence. Therefore only four point pairs of both map pairs are involved into the translation estimation.

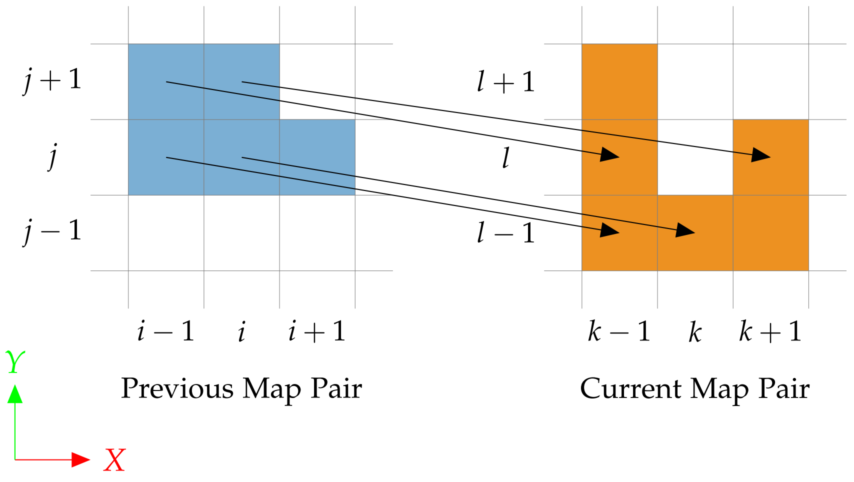

Figure 3.26: Possible Translation Patch (Number of Neighbors = 1, Number of Correspondences = 4)

General Approach. For getting now a resulting translation for every point, an algorithm proposed by Umeyama (1991) can be used. This algorithm tries to minimize the mean squared error for a given set of point patterns, to acquire a rotation matrix and a translation vector. Additionally it can also handle scaling, but this is not a requirement in this case and has been therefore reduced from its original formula (see Equation 3.23). By getting a rotation matrix R and a translation vector t as outcome, a translation is easily extracted. In fact, this just involves a multiplication as shown in Equation 3.2, by using the to be transformed point's position, followed by a subtraction of its unmodified position.

$$E_2 = \min\left(\frac{1}{n} \cdot \sum_{i=1}^{n} \| V_{g,i,prev} - (R \cdot V_{g,corr(i)} + t) \|^2\right) \tag{3.23}$$

Simple Approach. Because of having only the need for the translations, a simpler calculation is performed, as shown in Equation 3.24. This is simply a mean value calculation of the position differences of the found point pairs of the appropriate patch.

$$t_j = \frac{1}{n_{corr(j)+1}} \cdot \left(\sum_{i=1}^{n_{corr(j)}} (V_{g,corr(i)} - V_{g,i,prev}) + (V_{g,corr(j)} - V_{g,j,prev}) \right) \tag{3.24}$$

Weights. The use of weights may greatly improve the result of an algorithm and adds also more control for manipulating the actual output. Therefore two different types of weightings are used for estimating the translations. The first one deals with the information of the acquired correspondences, while the other takes care of the geometry of the patch.

By incorporating the weights, estimated at the filtering stage, correspondences with a high deviation will not dramatically influence the resulting translation, but will be still considered. As shown in Equation 3.22, the weight w_c becomes 0 when a correspondence reaches the maximum of all thresholds, while is set to 1, if the correspondence does perfectly fulfill all requirements.

Next to the weights based on the correspondence information, an additional factor is introduced to respect the relative distances of the currently used indices within the estimation patch. This results in getting a higher importance for the points found in the center, compared to the ones located at the border. The index values i and j are defined by the previous maps. The weight $w_{d,i}$ is calculated by using the squared euclidean distance, followed by a division by the squared maximum distance (see Equation 3.25). The values Δi_{max} and Δj_{max} are defined by the number of neighbors used for the estimation.

$$w_{d,i} = 1 - \frac{\Delta i^2 + \Delta j^2}{\Delta i_{max}^2 + \Delta j_{max}^2} \tag{3.25}$$

Both weights are simply combined by a multiplication, resulting in the weight $w_{cd,i}$ (see Equation 3.26).

$$w_{cd,i} = w_{c,i} \cdot w_{d,i} \tag{3.26}$$

Simple Approach with Weights. The integration of the resulting weight $w_{cd,i}$ is done by extending Equation 3.24, forming the final calculation of the translation as shown in Equation 3.27.

$$t_{j,w} = \frac{1}{n_{corr(j)+1}} \cdot \Big(\sum_{i=1}^{n_{corr(j)}} (V_{g,corr(i)} - V_{g,i,prev}) \cdot w_{cd,i} + (V_{g,corr(j)} - V_{g,j,prev}) \cdot w_{cd,j} \Big) \tag{3.27}$$

By applying the calculation of Equation 3.27 on every patch, nearly all translations are estimated and the deformation between both vertex maps is basically determined. The actually used algorithm for executing the simple approach including the handling of the weights for estimating the translations for the whole vertex map, is shown in Listing 3.3.

```
for $i$, $j$ in map pair
    for $k$, $l$ in patch of $i$, $j$
        if $k$, $l$ has correspondence
            $t_{j,w} \leftarrow$ calculate and append translation
    $t_{j,w} \leftarrow$ norm translation
    save and accumulate $t_{j,w}$ at indices $i$, $j$
```

Listing 3.3: Estimate Translations

Smooth Translations

This step is important for filtering out peeks, which may result if a patch is under-determined in form of very less correspondences. The reason for this is, that the previous calculation of the translations does not care about the number of used correspondences. Therefore the estimation produces also an output, even though if only a single correspondence is defined in a patch. Additionally to the actual averaging, this step damps also the translations at the borders. The smoothing is done by simply calculating the mean values of all translations and by involving the translations of their neighbors (see Equation 3.28). This means that there is no distinction between acquired and not acquired translations and therefore positions, which do not have a translation acquired, will affect their neighbors in form of damping. For calculating the resulting translation, also the weighting factor $w_{d,j}$ is involved.

$$t_{j,smooth} = \frac{1}{n_{neigh(j)+1}} \cdot \left(\sum_{i=1}^{n_{neigh(j)}} t_{neigh(j),i} \cdot w_{d,neigh(j)} + t_j \cdot w_{d,j} \right) \tag{3.28}$$

Update Previous Maps

The resulting translation map contains the absolute difference between the previous and the current vertex map. This pipeline is an iterative process and therefore only the relative translations have to be calculated in every iteration. Therefore this step accumulates the absolute translations onto the original previous vertex map, but instead of overwriting its values, they get stored onto a temporary map. This map will be actually used to represent the previous vertex map in the next iteration. Additionally in this step, the normal map is recalculated, because of changes in their related vertex map. In fact, the result of the normal calculation is also stored in a temporary map.

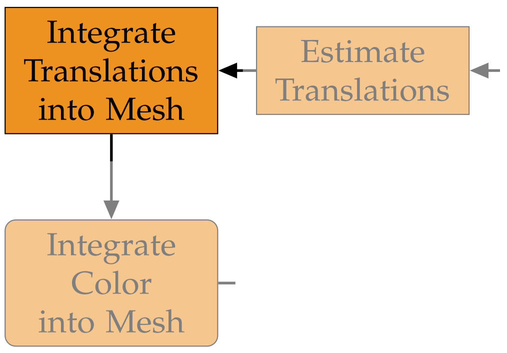

Figure 3.27: Pipeline - Deformable 3D Reconstruction - Template-Based - Integrate Translations into Mesh

3.2.5 Integrate Translations into Mesh

The previously acquired translation map is now ready to be used for getting integrated (see Figure 3.27). The integration into the triangle mesh is also done by a projection, the same method as used for the TSDF model integration. But instead of having an equidistant voxel grid, the mesh consists of scattered vertices. Therefore the iteration has to be performed over these vertices. As usual, they are transformed into the current coordinate system and projected into the image plane. In contrast to the TSDF model integration, where the depth image is used for accessing the projected position, now the vertex map is used for this. So, if there exists a valid vertex in the vertex map, using the indices i and j, a distance metric between the vertex and the currently processed vertex of the mesh will be calculated. This is done by calculating their difference, followed by applying the euclidean norm. Otherwise, the position of the to be integrated translation cannot be determined and therefore the integration for this voxel is canceled. Also if the distance exceeds a given threshold value d_{max}, the integration for this vertex will be aborted too. This threshold is similar to its counterpart d_{trunc}, found in the TSDF model integration, but may have a different value. In the other case, the not yet rounded indices, represented as floating point numbers, are used to acquire a bilinearly interpolated translation $t_{b,j}$. Generally, this interpolation promises to acquire a more accurate translation on the required position, instead of simply accessing it by using the rounded indices. Afterwards, the translation is scaled with the reciprocal distance, normed to the defined maximum distance d_{max} (see Equation 3.29). Finally, the result t_j is directly accumulated onto the position of the processed mesh vertex.

$$t_j = t_{b,j} \cdot (1 - \frac{\|V_j - V_{g,prev}\|^2}{d_{max}}) \tag{3.29}$$

Figure 3.28 presents an example how an accumulation of translations on a triangle may look like. One triangle, picked out of the initially created mesh, more precisely at frame $F_{start} - 1$, is shown on the left hand side. A possible path of the triangle, resulting in running two iterations of the pipeline and therefore integrating the translations also twice, can be seen by following the black colored arrows. The absolute translations are indicated in the form of dashed lines, connected with the respectively vertices of frame $F_{start} - 1$ and $F_{start} + 1$.

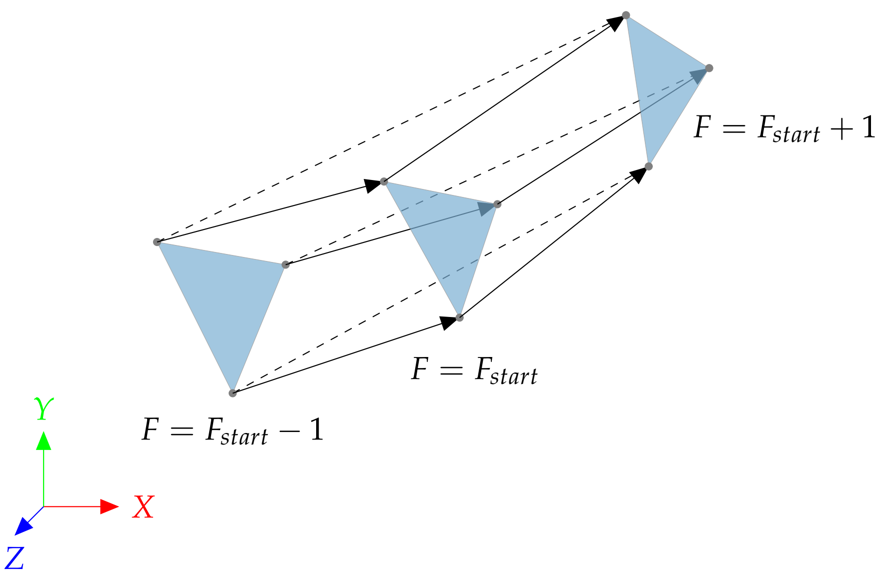

Figure 3.28: Possible Accumulation of Translations on a Triangle

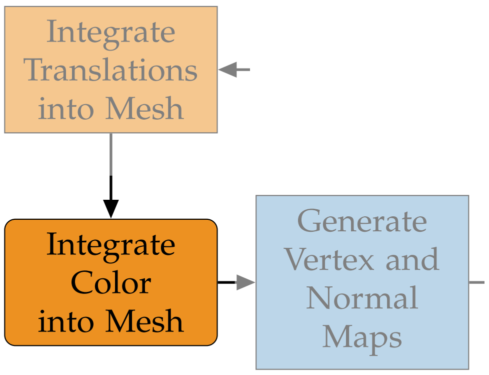
Integrate Translations into Mesh
Integrate Color into Mesh
Generate Vertex and Normal Maps

Figure 3.29: Pipeline - Deformable 3D Reconstruction - Template-Based - Integrate Color into Mesh

3.2.6 Integrate Color into Mesh

By having now the deformed mesh available, the optional integration of the color may happen (see Figure 3.29). The only difference, compared to the integration into the 3D color volume, as described in Section 3.1.6, is, that the integration loops over scattered vertices instead of equidistant aligned voxels. Therefore there is no need to calculate the global vertex position, by having the appropriate vertex positions already. Because of handling the color after the integration of the translations, the resulting indices of the projection can be directly used for accessing the RGB values of the color image.

3.2.7 Generate Vertex and Normal Maps (2)

Now, yet another generation of the maps is taking place, to render the currently deformed mesh (see Figure 3.30). This task is equivalent to the already mentioned step for the generation of the maps (see Section 3.2.2), but now the generation is not only because of quality reasons. It is required to update the vertex and normal maps to represent the newly integrated data of the mesh, for the usage in the next iteration.

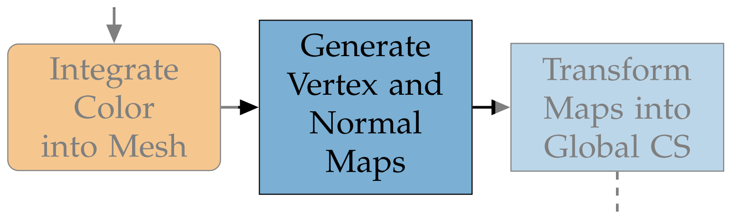
Integrate Color into Mesh
Generate Vertex and Normal Maps
Transform Maps into Global CS

Figure 3.30: Pipeline - Deformable 3D Reconstruction - Template-Based - Generate Vertex and Normal Maps (2)

3.2.8 Transform Maps into Global Coordinate System (2)

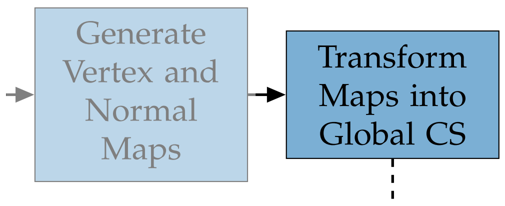
Generate Vertex and Normal Maps
Transform Maps into Global CS

Figure 3.31: Pipeline - Deformable 3D Reconstruction - Template-Based - Transform Maps into Global Coordinate System (2)

At the end, the rasterized maps are transformed into the global coordinate system, using the current camera pose (see Figure 3.31).

3.3 Deformable 3D Reconstruction - Model-Based

In contrast to the previous pipeline, the more dynamic and also more complex variant is shown in Figure 3.32. Obviously, there are many steps needed to handle the deformation additionally to the actual reconstruction. In this variant, the basic idea is to insert a translation layer, which transforms the points before its integration and also before its output generation, but in this case in the reverse direction. This idea is taken from the work of Newcombe, Fox, and Seitz (2015), in which a similar procedure has been shown. To get such a functionality, an additional storage model is needed to save the deformations. In contrast to the template-based method, the deformation can be tracked already from the beginning, in especially at the second frame $F = 2$. Therefore the pipeline behaves only identical to the KinectFusion system in the first iteration. The reason for this is, that there is no deformation information available at the first run. This means, that nearly all new steps are skipped in the first iteration. The only outlier is the creation of the triangle mesh, because the mesh is already needed in the second iteration.

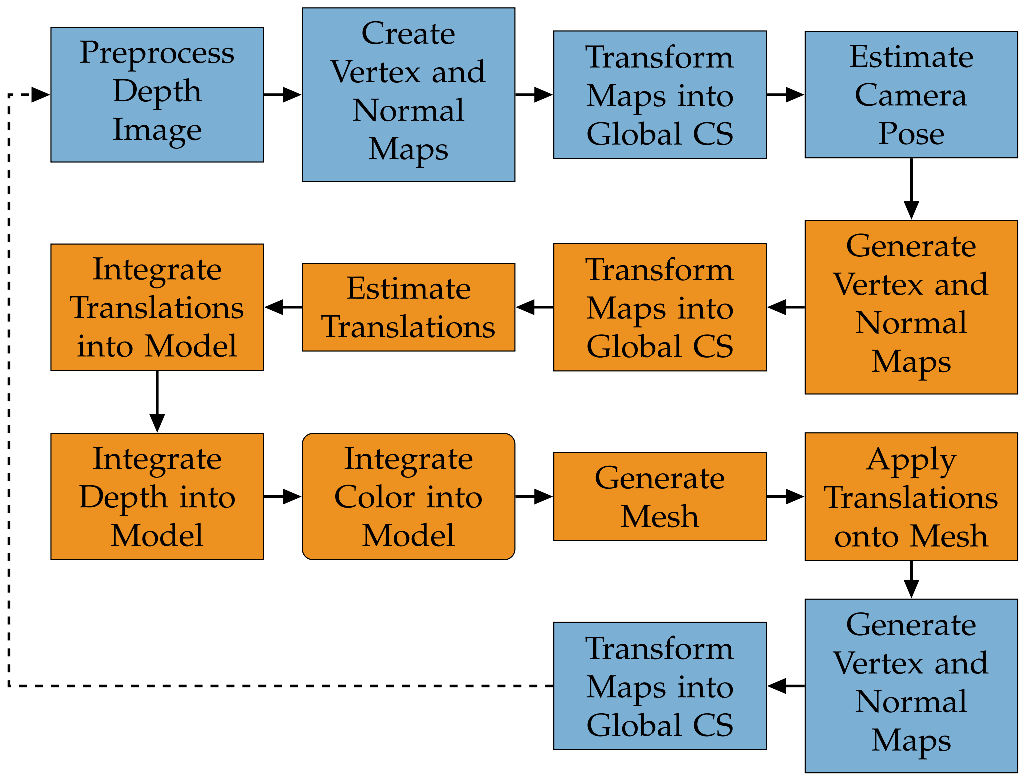
Preprocess Depth Image
Create Vertex and Normal Maps
Transform Maps into Global CS
Estimate Camera Pose
Generate Vertex and Normal Maps
Transform Maps into Global CS
Estimate Translations
Integrate Translations into Model
Integrate Depth into Model
Integrate Color into Model
Generate Mesh
Apply Translations onto Mesh
Generate Vertex and Normal Maps
Transform Maps into Global CS

Figure 3.32: Pipeline - Deformable 3D Reconstruction - Model-Based

As already illustrated in the previous pipeline graphic, the exchanged path of the original KinectFusion system is colored orange. This second variant for handling the deformation can be interpreted as an extension to the template-based variant, mentioned in the previous section. In fact, all modifications made to the tasks of KinectFusion, apply also to this pipeline. Therefore, only the remaining new steps, related to the introduction of the translations model, will be discussed below.

3.3.1 Integrate Translations into Model

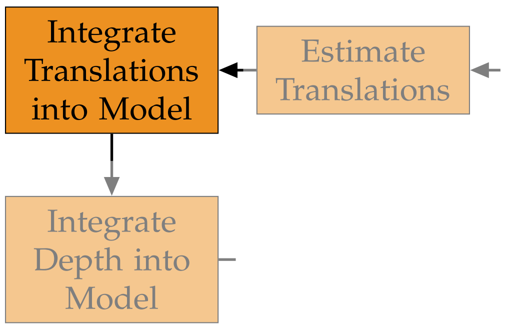

Figure 3.33: Pipeline - Deformable 3D Reconstruction - Model-Based - Integrate Translations into Model

The first new step in the second variant of the new approach integrates the translations into the translations model, instead of applying them directly onto the mesh, as done in the template-based method. (see Figure 3.33). For this, also a similar technique as for the integration into the TSDF model is used. The translations model has the same structure as the TSDF model, by also being defined as a fixed cubic voxel grid. The difference is that it is responsible for the storage of the deformation information, in contrast of holding static body information. The deformation is specified by a translation vector, holding three floating point values.

To insert the translations, every single voxel has to be covered. If a voxel has not yet acquired a corresponding value, held by the currently processed indices in the TSDF model, the voxel would be already skipped at the beginning, because then there is nothing to deform. Otherwise the voxel index is used to calculate its global position V_g, the same way as done in the integration into the TSDF model. Then this resulting position is accumulated onto the translation, stored on the currently processed voxel (see Equation 3.30). After that, the position $V_{g,translated}$ is transformed into the current coordinate system and projected into the image plane, as usual. As already done in the template-based variant, now the previous vertex map is used for acquiring the current position. By calculating the euclidean norm of the difference between that position and the already translated global voxel position, the distance can be determined. If this value exceeds a constant threshold d_{max}, then the integration would be stopped at this point for this voxel. Because of being somehow associated with the TSDF model, by using its weights for example, this distance needs be set to the same value as the maximum truncated distance d_{trunc}. In contrast to the template-based method, in which this distance may be different. Otherwise, the to be integrated translation can be calculated. As already described for the template-based variant in the previous section, the to be integrated translation $t_{b,j}$ is acquired by a bilinearly interpolation. Further on, the interpolated vector is linearly scaled, using the corresponding weight of the TSDF model. The translation is multiplied with the reciprocal weight, normed to a defined maximum weight (see Equation 3.31). This scaling is done as long as the weight has not reached the maximum weight. The idea behind that is to settle down the TSDF volume, before the integration of the translations is performed in the whole extent. By linearly scaling the translation vector, the deformation is already integrated from the beginning, though will not get that big importance initially. Alternatively the integration may be started as soon as some fixed defined weight has been reached on the appropriate voxel. But in fact, not all weights of the to be integrated area will have the same weight and therefore would not start accepting the translations at the very same time. This would result in getting an irregular deformed surface. These both arguments are the reason for applying the additional scaling. Finally, the result of this calculation t_j is accumulated onto the translation of the current voxel.

$$V_{g,translated} = V_g + t_{model} \tag{3.30}$$

$$t_j = t_{b,j} \cdot \min(\frac{w_j}{w_{max}}, 1) \tag{3.31}$$

As already illustrated of how the accumulation of the translations is applied on a triangle (see Figure 3.28), Figure 3.34 shows the pendant for the use in the model-based variant. In contrast to the former method, the voxels are not translated actually. The position, especially the three indices of a voxel, stay obviously the same. But in fact, the global vertex positions are translated for simulating the voxels new virtual positions. Therefore, in this variant, there is always the need to add the translation to the global vertex position, when doing any kind of operation. As the translation integration of this method does start right from the beginning, a voxel may get an translation information already at the second run of the pipeline. This is illustrated in the figure, by showing a translation from frame $F = 1$ to $F = 2$. As the translations model holds a single vector on each voxel, only the resulting translation of the two accumulations are kept and the intermediate translation is lost. This summed movement is visualized with a dashed line.

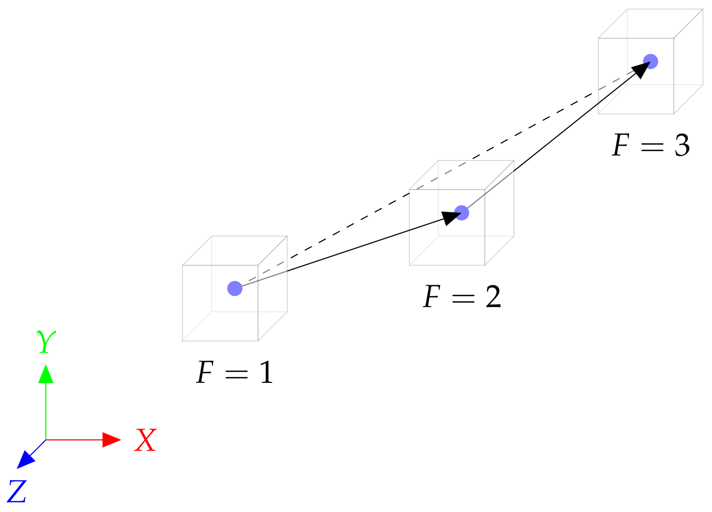

Figure 3.34: Possible Accumulation of Translations on a Voxel

3.3.2 Integrate Depth into Model

In this step, the raw depth image is integrated into the TSDF model (see Figure 3.35). This is done similar as it is achieved in the original KinectFusion system. The first difference is that every voxel from the TSDF model is transformed with its corresponding translation before its integration. This vector is found in the translations model and is applied to the global voxel position V_g, derived from the indices of the voxel. This affects the projection into the image plane, in especially in form of their resulting indices, used for accessing the to be integrated depth value. The second and last difference is, that the integration is only done as long as the maximum weight w_{max} of

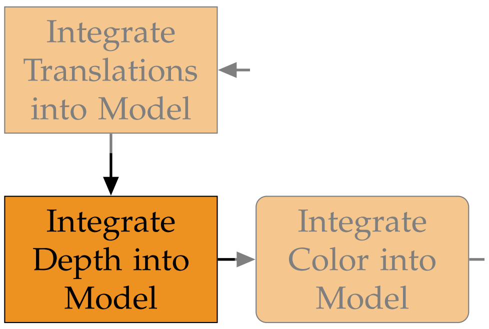
Integrate Translations into Model
Integrate Depth into Model
Integrate Color into Model

Figure 3.35: Pipeline - Deformable 3D Reconstruction - Model-Based - Integrate Depth into Model

the currently processed voxel has not been reached. The following steps are processed the same way as done in the KinectFusion method.

3.3.3 Integrate Color into Model

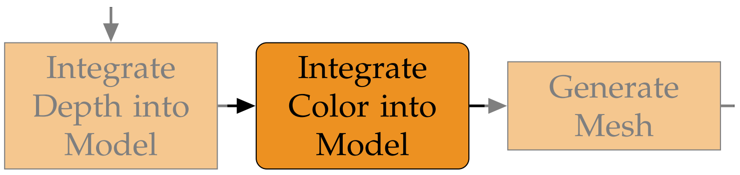

Figure 3.36: Pipeline - Deformable 3D Reconstruction - Model-Based - Integrate Color into Model

As like in the pipeline described in Section 3.1, the optional color integration may follow the depth integration (see Figure 3.36). This task is also only slightly modified, compared to the already mentioned integration into the color volume (see Section 3.1.6). For handling the translations, the global voxel positions V_g have to be translated, before the coordinate system transformation and the following projection is performed. This is done in the same way as for the integration of the depth, described in Section 3.3.2.

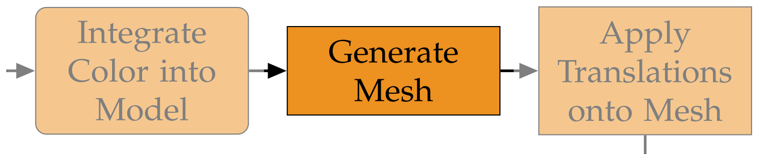

Figure 3.37: Pipeline - Deformable 3D Reconstruction - Model-Based - Generate Mesh

3.3.4 Generate Mesh

At this step, the current model has to be transformed into a different representation in order to be used for a later generation of the vertex map (see Figure 3.37). In the original KinectFusion approach, the creation of a mesh is not needed and a vertex and normal map is directly acquired from the TSDF model. But as argued by Innmann et al. (2016), a direct raycasting is not easy possible, when the deformation is stored in the forward direction. This is also true for the translations model. Therefore the current storage model is transformed into a triangle mesh for the use in a later rasterization. The creation of the mesh is done the same way, as described in Section 3.1.7. In fact, this mesh represents the canonical state only, without having any deformation yet applied.

3.3.5 Apply Translations onto Mesh

Finally, the previously generated mesh is updated to represent the deformation state of the currently stored translations model (see Figure 3.38). The previously produced mesh represents the canonical state of the model, without containing any translation information. To apply that information, every single vertex in the triangle mesh has to be transformed with its related translation, stored in the translations model. Even though, the mesh and the translations model are using the same coordinate system, the finding of the corresponding translations has to involve some calculations, especially in form of an interpolation. For getting a proper translation from a discretized model for any vertex in a continuous mesh, the first task is to find the eight indices, which enclose that currently processed vertex. With that information, an accurate resulting translation vector can be calculated, by using the

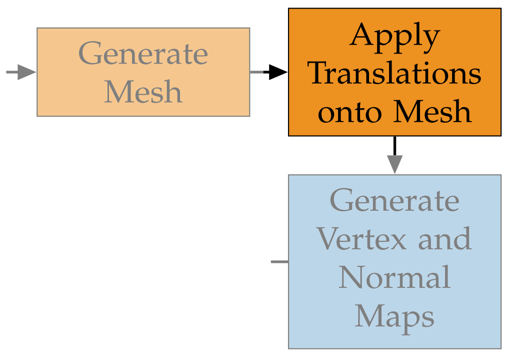

Figure 3.38: Pipeline - Deformable 3D Reconstruction - Model-Based - Apply Translations onto Mesh

resulting eight translation vectors, associated with the found indices. The calculation itself is done by running three times a trilinear interpolation, one for each dimension. The result of this calculation is simply accumulated onto the position of the currently processed vertex.

4 Implementation

By designing and implementing the previously presented approach, a complete framework has been evolved. Besides its main functionality, it offers a well set of classes from containing a simple data structure up to including some high-level methods. Therefore it provides everything what is needed to develop a 3D reconstruction method. For this, it is a system, which can be easily adapted and extended, to fit any particular needs. The framework has been realized by using C++ as programming language. By exploiting its nature design, many object-oriented methods have been used, to create a logical and clean application. Besides the main functionality, the calculation intensive tasks have been implemented in the CUDA programming language. In fact, this interface enables the usage of the massive parallelism capabilities of a GPU.

4.1 Toolkits

Following toolkits have been used for the implementation:

- OpenCV 3.1
- Eigen 3
- PCL 1.8
- CUDA 7.5
- Qt5
- Various camera driver e.g.: libfreenect, OpenNI2 or librealsense2

Besides the required toolkits, the compiler has to support the C++11 standard. This standard extends the core functionality of C++ with some modern features. One of these new features, extensively used in the implementation, deals with threading. The class *std::thread* provides an easy to use container for making use of parallel processing, in this case, obviously on the CPU. In Section 4.4 some of its applied use cases will be shown. Another adapted feature of C++11 involves the implementation of callbacks. The combination of the class *std::function* and the function *std::bind* provides an easy and clean interface for encapsulating any method, to be used as a callback function. This greatly simplifies the communication between objects.

For defining the building process of the framework, qmake has been used, which is delivered by Qt. In the developed building scripts, some optional switches have been integrated, to be able to enable or disable components, depending on the requirements. For example, this is especially useful when only one type of camera is planned to be used for the recording, which in fact would yield the application to one single driver. By disabling all others, using these switches, only the required driver has to be installed on the computer.

4.2 KinectFusion

As already noted in Chapter 3, KinectFusion is used as the basis for the new approach. The used implementation of the KinectFusion system has been acquired from the PCL. For easier understanding, testing and also modifying this part of that external toolkit, the source code has been copied into the workspace and activated for the development.

One of the most interesting part of KinectFusion's implementation deals with the storage model. As described by KinectFusion and implemented in the PCL, the storage of the TSDF model is completely realized using the memory of the GPU. In the implementation, the model's fixed size is set to 512 for each dimension, resulting in 512^3 points. Generally, when having to store the content of one voxel of the model, consisting of one floating point value for representing the TSDF value and one integer value for the definition of the weight, an amount of 32 bits + 16 bits = 48 bits memory would be needed. Taking the resolution into account, the complete TSDF model would require 768 MB of video memory. Though the implementation of PCL, based on the definition of KinectFusion, encodes both values using short as data type. This decreases the consumption of memory to 16 bits +

16 bits = 32 bits per point and therefore minimizes the total requirement amount to 512 MB.

The main class of the KinectFusion implementation is called *KinfuTracker*. It offers some high level methods for controlling the pipeline, as shown in Figure 3.1. Basically by defining the initial camera position and filling the pipeline with a depth image in every iteration, the system is able to do its job in form of a 3D reconstruction. For this, nearly all of its essential steps are implemented in CUDA. One exception is a numeric solver, used for estimating the camera pose in the Iterative Closest Point (ICP) step. But this task is not very expensive in the sense of calculation effort and therefore falls not much into account. All in all, by fulfilling some conditions, the pipeline is able to run in real time.

4.3 Deformable 3D Reconstruction

Analog to the reference implementation, a main class, named *KinfuCustom* has been created. It is a superset of *KinfuTracker*, therefore the new class supports everything what *KinfuTracker* does, but with the extend of the new approach. *KinfuCustom* offers an interface for both new pipelines, shown in Figures 3.18 and 3.32. In fact, it supports also the KinectFusion pipeline (see Figure 3.1), but by offering a slightly different behavior due to some modifications made in its tasks. As already noted in the previous chapter, some methods of KinectFusion had to be modified. But instead of directly changing the code, the methods have been newly implemented to adapt the reference implementation as little as possible. At the end of the development, there is only a minor difference left between the forked code and the original implementation of PCL. One of this changes affects the TSDF model. Its size has been reduced from 512^3 to 256^3, because of the additional memory consumption of the new translation model. By having the requirement of being in the same resolution as its counterpart, the TSDF model, and for storing three floating point values per voxel, the capabilities of a typical GPU would be exceeded. More precisely, this would result in the amount of $512\text{ MB} + 3 \cdot 512\text{ MB} = 2\text{ GB}$ for storing both models, in total. Additionally the parameters of the bilateral filter have been adjusted, for performing a less intensive filtering. Whilst its output of the KinectFusion pipeline is only used for the alignment of the vertex maps, it is now also needed for the translation estimation process, where a less modified input is preferred.

4.4 Structure

The implementation is divided into the following six logical categories:

- Core
- Devices
- Gui
- Kinfu
- Loader
- Tasks

4.4.1 Core

As the name already implies, the core category contains all basic classes.

One of the core functionality deals with data structures. The most important ones are related to the data of images, point clouds and meshes. For the first type, a class named *Image* (see Figure 4.1) is responsible for managing color and depth images. Besides holding the data, the class offers also methods for loading, saving and manipulating an image. The data itself is stored in form of an object, instanced out of the *cv::Mat* class, defined in the OpenCV toolkit. Analog to the *Image* class, a class named *Mesh* takes care of all geometry related data, especially in form of point clouds and meshes. For storing its underlying data, represented by 3D points, a structure called *Vertex* has been introduced. It uses the class *Eigen::Matrix*, defined in the Eigen toolkit, for managing its content. The *Mesh* class actually holds a list of objects instanced out of the *Vertex* structure. Additional there is a class called *Camera* which is responsible for the mapping of the real world onto the virtual one and the other way around. Similar to the *Image* class, it uses mainly the *cv::Mat* class for storing its relevant content.

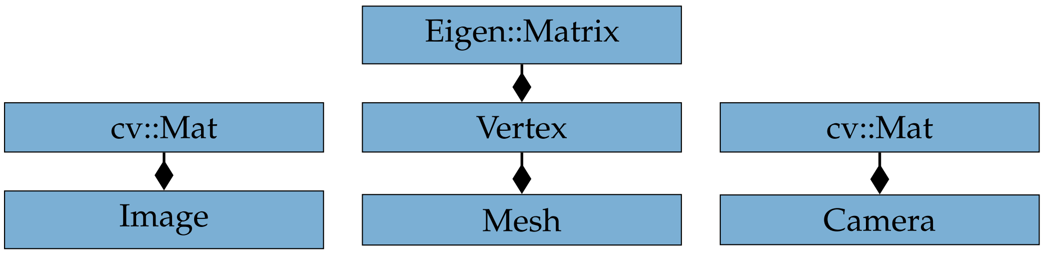

Figure 4.1: Class Diagram - Data Structures

For serializing the content of the *Image* and *Camera* classes, native OpenCV methods are used. OpenCV offers functions for directly writing and reading the data of the *cv::Mat* class into a file, using a graphics format. These functions are called *cv::imwrite* and *cv::imread*. Based on the given filename, especially focusing on the extension, the appropriate format is chosen by the underlying logic. The first candidate to be used as the graphics file format was the Portable Network Graphics (PNG) (Duce et al., 2003) format, because it is well-known and has therefore great support. It is able to store an image containing three channels, each having a depth of 8 bits and also one channel describing a depth of 16 bits. The second, more exotic representation, is needed for storing depth images, besides the usual form of RGB images. Another requirement on the chosen format is to encapsulate the data in a lossless manner, for having the best possible quality in the recording stage. This is also fulfilled by the definition of the PNG format. The show-stopper for actually not using it was its calculation intensive compression. In spite of the implementation of a saving queue (see Section 4.4.2), the system was not able to store two images, the RGB and the depth one, simultaneously in a frequency of 30 Hz during the recording. Therefore this format was not acceptable for this use case and another file format called Portable Any Map (PNM) (Netpbm, 2013) has been considered. It fulfills both previous mentioned requirements and furthermore uses no compression, which in fact means no calculation effort. Its specialized formats Portable Pixel Map (PPM) (Netpbm, 2016b) and Portable Gray Map (PGM) (Netpbm, 2016a) are used for storing the RGB and the depth images respectively. For handling the storage of all attributes in the *Camera* class, which involves multiple variables in different types, the OpenCV class named *FileStorage* has been applied. Again, based on the given extension in the method calls, the appropriate format is selected by the routines in underlying implementation. By having in this case no real requests to the format, the simple description language, YAML Ain't Markup Language (Ben-Kiki, Evans, and Net, 2009) has been chosen, to serialize the data. In contrast to the image and camera related classes, the *Mesh* class uses self-developed methods for reading and writing its data. For this, the chosen structure is defined by the PLY Polygon File Format (Turk, 1994). It has been implemented to support the loading and saving of the files in form of ASCII text and also binary form.

Another important part of the core functionality deals with the interaction between the framework and third-party toolkits, such as OpenCV and PCL. For this, the classes *OpenCVUtils* and *PCLUtils* encapsulate selected methods and algorithms to offer an easy and comfortable access to the respectively toolkit.

4.4.2 Devices

The implementation supports different input devices, in form of interacting with different drivers. The handling of each specific device driver is implemented in a separate class. By having some functionality in common, the classes are derived by a base class called *Device*. The main task of the specific classes is to communicate with their related device drivers. Each device runs in an own thread, serving the recorded data with the help of callbacks. These callbacks are used to forward the data to the GUI related parts, for providing the data for visualization. In case of a running recording, the captured frames are simply stored to the previously defined target directory. The saving is done with the help of a queue, which is polled in a defined time interval, using multiple threads for doing the actual storing. This multithreaded approach tries to decrease the impact of the recording overhead by balancing the load. Additionally every specific device class has the possibility, by overwriting a method in the sense of object-oriented programming, to perform a customized postprocessing on the to be stored image.

So, by having to support different devices, offering various approaches for communicating with their internals, a common interface has to be available.

This problem can be solved with the help of a design pattern called Adapter, proposed by Gamma et al. (1995). The idea behind this scheme is to define a unified interface for accessing different already available implementations. By using the methods defined in the base class *Device*, any specific device may be accessed in the same manner. Any request to the generic *Device* class is forwarded to the specific device dependent implementation involving one of the classes *FreenectDevice*, *OpenNIDevice* or *RealSenseDevice*.

For handling all connected devices, a class called *DeviceManager* (see Figure 4.2) exists. It stores all created devices and forwards the calls to the desired targets.

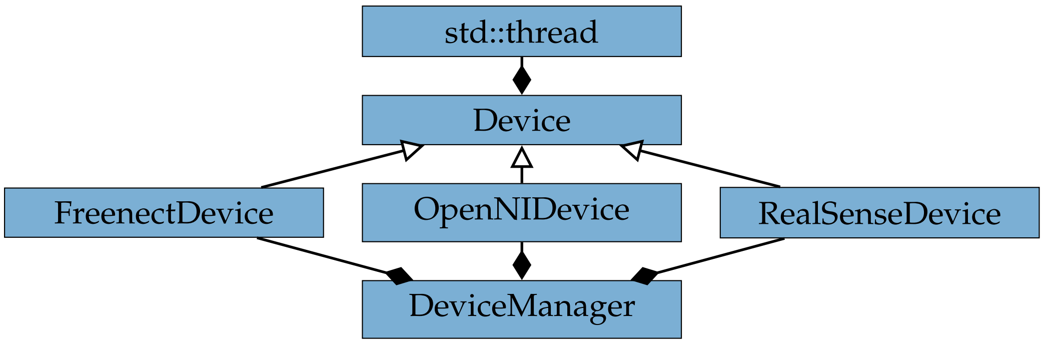

Figure 4.2: Class Diagram - DeviceManager

Because of administrating the state of all available devices, the *DeviceManager* class has to be instanced only once. Therefore it is implemented to follow the rules of the creational pattern named Singleton (Gamma et al., 1995). By forbidding a manual creation of an object out of the class, this logic has to be handled internally. By calling its static method *getInstance*, which is the interface for its usual access, an instance of *DeviceManager* is returned. If there is non available, means no object has been created yet, its construction gets triggered indirectly.

4.4.3 Gui

The graphical user interface (GUI) is the main entry point for the user. The *MainWindow* class (see Figure 4.3) acts as a controller and in dependency of the user's choice the program flow is determined. To keep things segmented, this is the only category where Qt related code can be found. Next to the general implementation of the GUI, this category includes also two specialized widgets for handling the visualization of the to be displayed data, named *ImageWidget* and *MeshWidget*. The *MeshWidget* class is derived from *ImageWidget* and is therefore a superset of it.

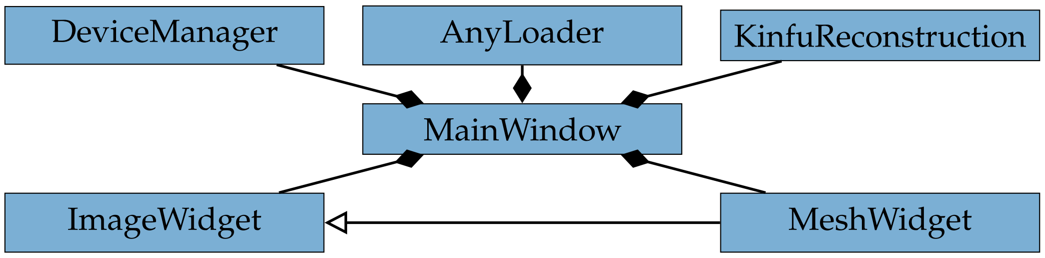

Figure 4.3: Class Diagram - MainWindow

For the communication between its instantiated classes callbacks are used. Mainly this involves transferring data of the already proposed classes *Camera*, *Image* and *Mesh*.

4.4.4 Kinfu

This is the place for all pipeline related developments. As already noted in Section 4.3, the class *KinfuCustom* (see Figure 4.4) offers a high level access to all necessary methods. This methods can be nearly mapped to the described tasks in the pipeline shown in the Figures 3.18 and 3.32. The structure of the class is based on the equivalent KinectFusion implementation found in the PCL's class *KinfuTracker*. Besides that class, all additional CUDA related code is placed in this category too. This code is simply in the form of independent functions, extending the CUDA routines found in the KinectFusion implementation of PCL.

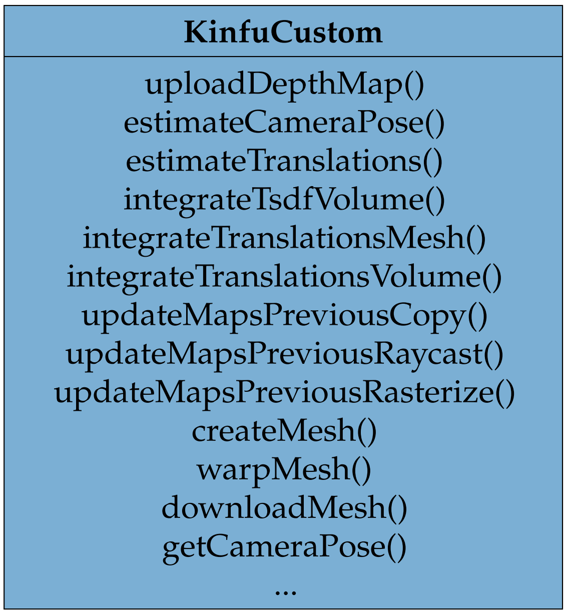

Figure 4.4: Class - KinfuCustom

Obviously the class *KinfuCustom* encapsulates a lot of functionality, while offering only some simple methods for controlling its internal activities. This conforms to the structural design pattern Facade, introduced by Gamma et al. (1995). Basically the given requests are forwarded to some specific methods or functions of the internally handled references. Therefore *Kinfu-Custom* may be interpreted as a black box from the outside, while offering access to a complex subsystem.

4.4.5 Loader

For the visualization of varying data (video) a templated class *AnyLoader* is available (see Figure 4.5). This class is able to operate with data of the *Image* and *Mesh* classes. Basically it supports any class which implements a method called *load*. After setting a target frequency, defined in frames per second, a beforehand specified callback will be regularly called in the desired interval, while providing the requested data. This process runs in an own thread for enabling a parallel execution. The actual control logic, for example for starting or stopping the playback is implemented in the class *Loader*, which is actually the parent of *AnyLoader*.

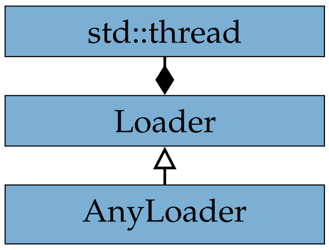

Figure 4.5: Class Diagram - AnyLoader

4.4.6 Tasks

The created application can be considered as a construction kit for experimenting with different approaches of 3D reconstruction. Therefore any additional task can be added easily. The base class for such a task is called *DepthReconstruction*, which provides already some required underlying routines. It runs in an own thread and by deriving it, a specific reconstruction type can be implemented. In fact, such a class is *KinfuReconstruction* (see Figure 4.6), which controls the reconstruction pipeline by calling methods of the *KinfuCustom* class. Its results, mainly in the form of the class *Mesh*, are forwarded by also using callbacks. In dependency of the defined parameters in the GUI, the task executes either the pipeline of KinectFusion or one of the other two available pipelines previously proposed.

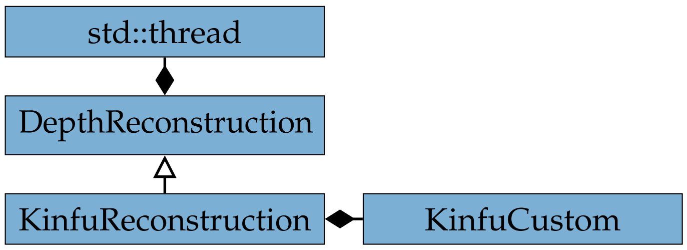

Figure 4.6: Class Diagram - KinfuReconstruction

5 Results

For demonstrating the proposed methods including their implementations, the focus is placed on single bodies. The first device which is considered to do the recording task and to produce the required input data, is named Xbox 360 Kinect. This device has been published by Microsoft and was originally developed for video games. It has become very popular and therefore is easily accessible to the consumers. But by having not the capabilities of capturing the depth under the distance of about half a meter, the resolutions of the integrated cameras will not be exploited when used for smaller bodies. The consequence of this would be to loose important depth information, when there is a small distance to the recorded object. Therefore a different device has to be chosen which fulfills this requirement. Intel has introduced a near field camera, called RealSense SR300, which is able to capture depth information for distances starting at 0.2 m. While supporting the same depth resolution as the Kinect of 640x480, it also grabs color images in a higher resolution than its counterpart, in the amount of 1920x1080 points. Therefore this device will be used for the recording.

For testing the already presented approach, facial expressions are chosen. This is realized by centering the head into the recording frustum of the camera and by statically positioning the recording device. At the beginning, the head is in a forward facing position. Then the head turns to the right, to the left and finally back to its initial position. Except of the back and the top, the head is now fully captured. Then the actual deformation process starts, testing the support of the non-rigid processing pipeline. This is done by starting with smiling, going back into the initial state and finally followed by another movement of the mouth.

Starting with the recording data used as input for the proposed pipelines, some important frames are shown in Figure 5.1. The visualization of the raw depth maps is illustrated by gray nuances. In general, the darker the color, the smaller the distance to the camera. Derived from that, the face is obviously the nearest part of the captured body. Points where the depth cannot be estimated, for instance by being to far away from the camera, are colored black. While the first row of the figure shows the movement of the head, the second and third rows illustrate the facial expressions.

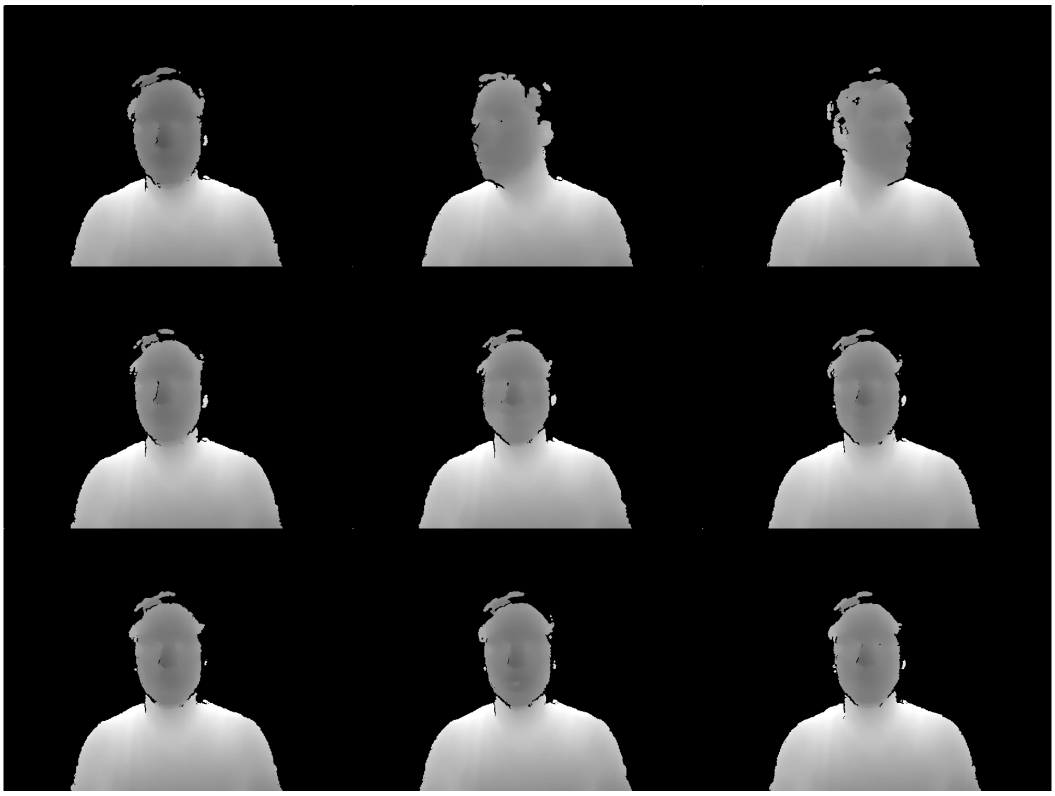
Figure 5.1: Facial Expressions - Depth Image

Figure 5.2 shows the results of the projection of the depth images. The illustrated vertex maps are already bilaterally filtered and clipped to the bounding box of the concerned volume. This is the reason for the absence of the torso. Additionally the maps are already transformed into the global coordinate system. All frames are rendered using the same camera pose to demonstrate the alignment of the ICP algorithm. The displayed points are connected with its neighbors forming a quad, in order to achieve a better visualization. Also the normals can be indirectly observed by focusing on the shading. The shown timestamps are equal to those of the previously presented depth images. As obvious, the vertex maps only contain the current recording information, like the depth images. This can best be observed by looking at the images shown at the top in the middle and on the right side of the figure, where the half of the head is missing.

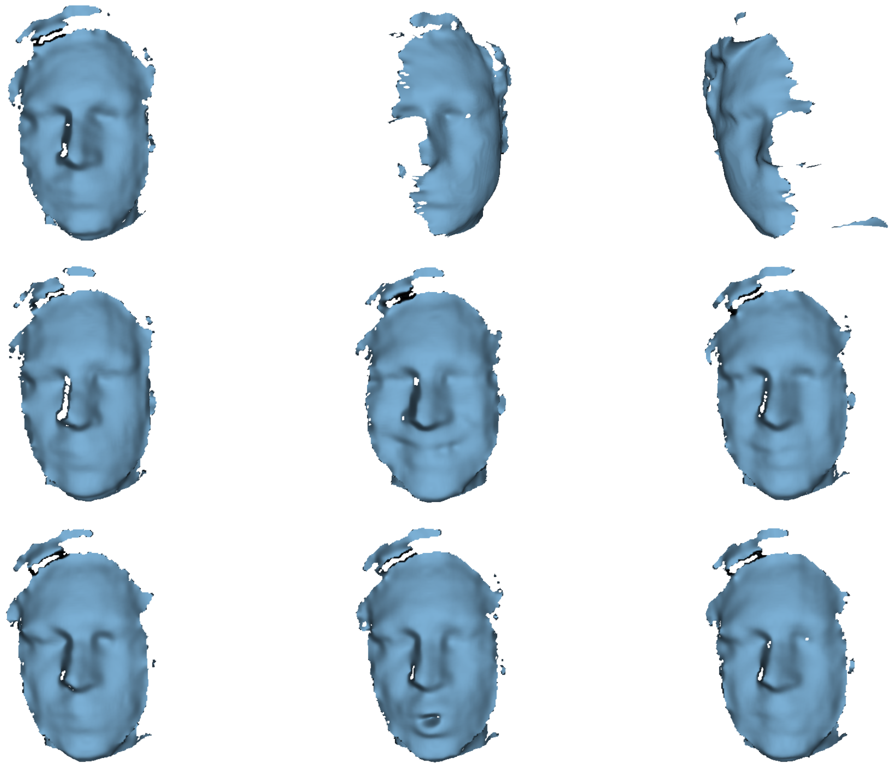
Figure 5.2: Facial Expressions - Current Vertex Map

The data model where all the captured depth information is stored is shown in Figure 5.3. It is split up into three different timestamps each of them shown in a different row, while every row represents three different viewing angles. As the head is not being perfectly aligned to the camera, some depth values of its left side are known at the beginning and therefore have already been integrated. For the right side no data is available at that moment. In the second row can be seen that the left side of the head got extended with some additional data. In the third row it is visible that also the right side has got caught up and therefore the TSDF model has acquired some new values. Obviously at this stage the head is already fully captured. By comparing the three shown timestamps it can be noticed that the TSDF values do not vary very much. The main reason for this is the use of the weighted integration (see Equation 3.13). For the visualization of the TSDF model the data range from -0.99 to 0.99 has been chosen, whereas the color map ranges from red to blue respectively. The resulting surface, represented by the value 0 is shown in black color and forms the transition between the colors red and blue. This can be spotted best on the image of the top left hand side.

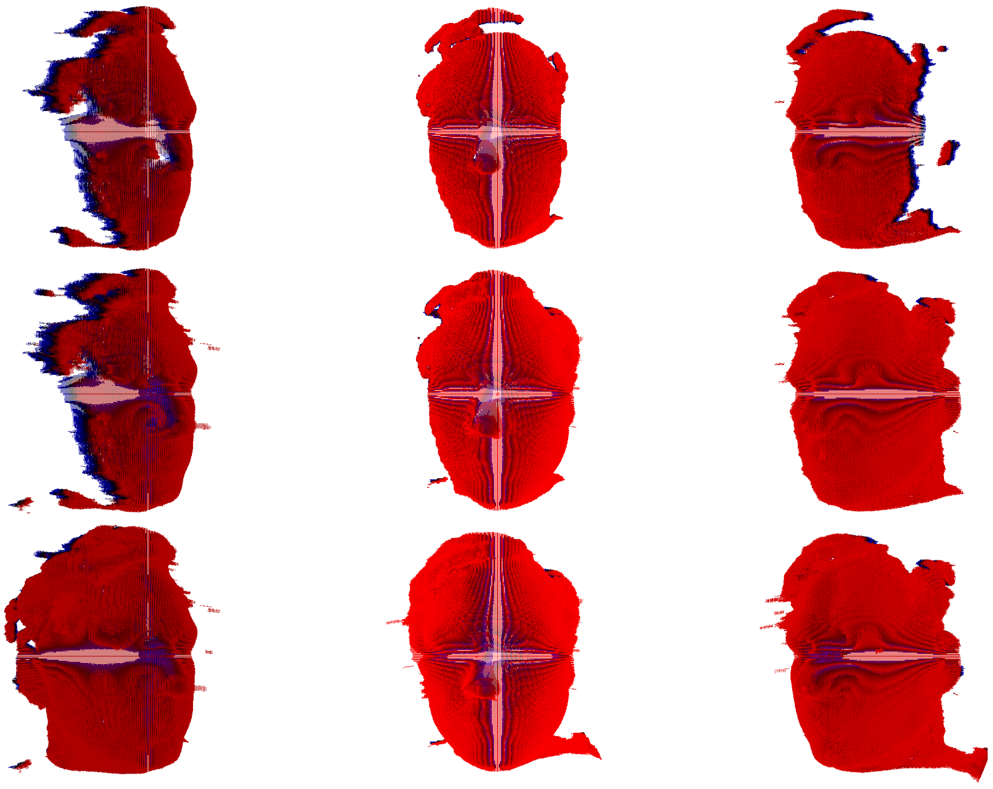

Figure 5.3: Facial Expressions - Template-Based - TSDF Model

In Figure 5.4 the most important results produced by the pipelines are shown. Like in Figure 5.3, three different timestamps from three different angles are visualized. In the first row the results of the basic KinectFusion system are shown. As mentioned in Section 3.2, the template-based method is identical to KinectFusion up to a certain point. The result before passing this exact timestamp is shown in the first row. Starting with the second row, the interesting results related to the proposed methods are illustrated. As seen in the middle row, a complete mesh deformation has happened. Especially in the area of the mouth and the cheeks, a big modification can be noticed. In the third row the facial movement has already been faded away and theoretically the mesh should look exactly like the canonical mesh, in the first row.

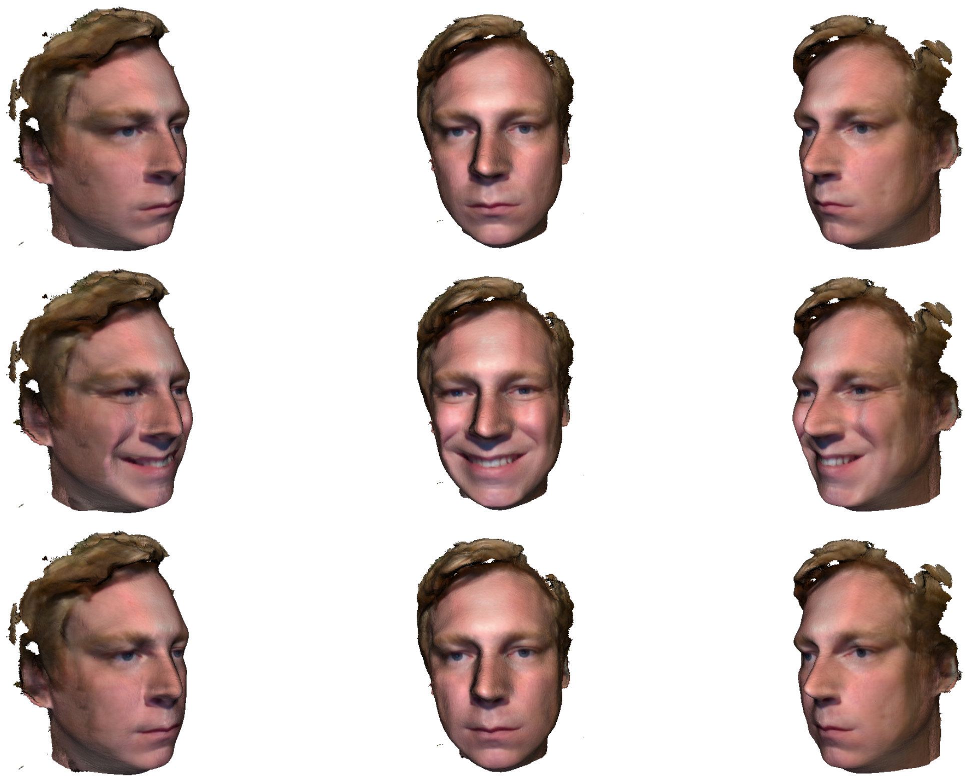
Figure 5.4: Facial Expressions - Template-Based - Mesh

For demonstrating the real movement of the surface, a red colored demo point is used as an indicator. Instead of usually integrating the color images into the model, a synthetically generated image is used. It consists of a red point drawn on a white background and is integrated only at a specific timestamp. In all other iterations of the pipeline, the color integration is therefore skipped. Depending on the camera position derived of the chosen timestamp, the resulting mesh will have the point placed anywhere on its surface. Figure 5.5 shows the point located a bit over the left cheek. On the illustration can be seen, how the point moves and deforms on the surface over time.

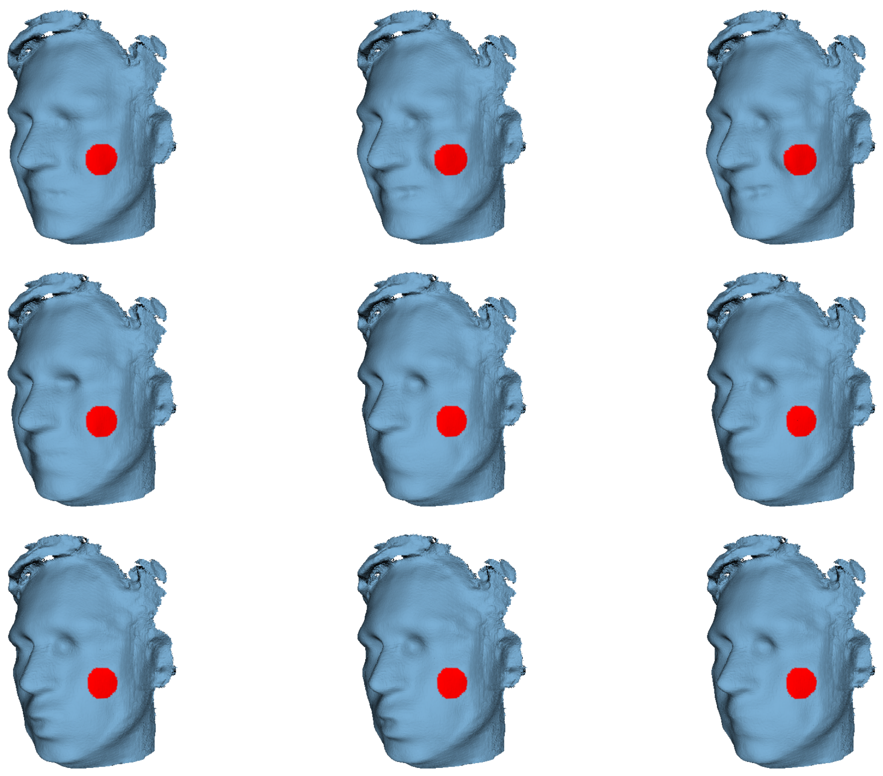

Figure 5.5: Facial Expressions - Template-Based - Demo Point on Mesh

Because of covering now the results of the template-based method, there is no translation model available. As a result it is not present for illustration. Though the translations can be derived by subtracting the positions of the already translated mesh from their canonical counterparts. The result of this operation is shown in Figure 5.6. The difference is visualized in form of yellow colored arrows pointing from the vertices of the canonical mesh to the corresponding ones of the deformed mesh. By looking at the images, it can be noticed that the peek in the sense of translations happens in the first row. This finding may be compared by looking at the same area of the previous illustration. Also as noted previously, at the end there should no longer be any deformation, which can be generally confirmed by looking at the results of the third row.

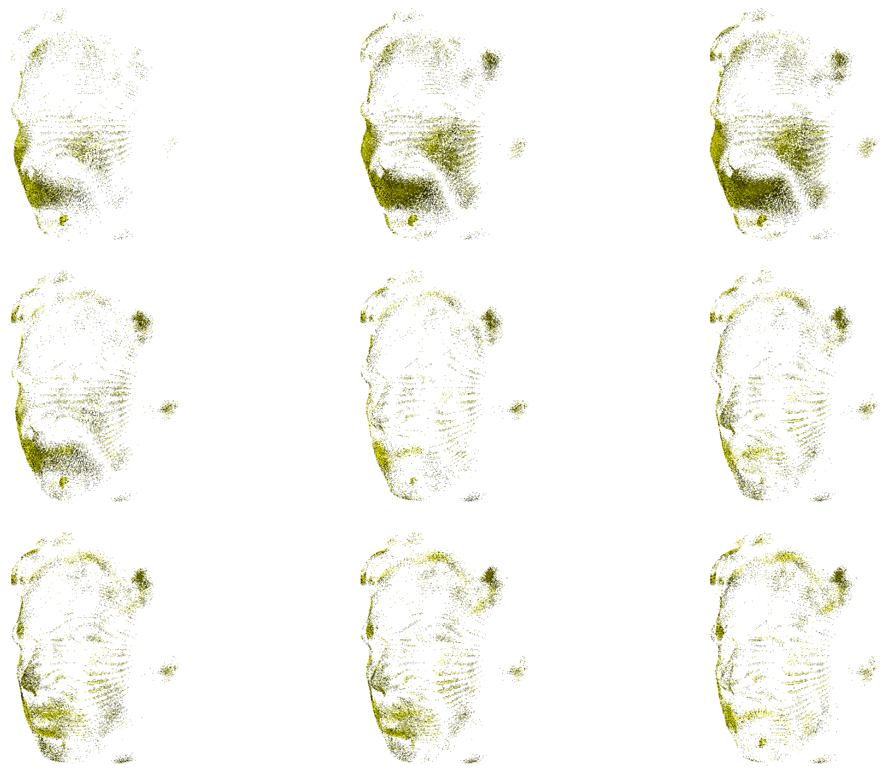

Figure 5.6: Facial Expressions - Template-Based - Translations on Mesh

So far only results of the template-based method have been shown. Although it can be assumed that they look pretty much the same compared to their counterparts of the model-based method, two selected frames of both variants are shown in Figure 5.7. In this illustration the mesh is rendered from the front, using again the demo point as an indicator. On the left hand side the output of the template-based variant is shown, whereas on the other side the outcome of the model-based pipeline is presented. Obviously there cannot be spotted any real difference.

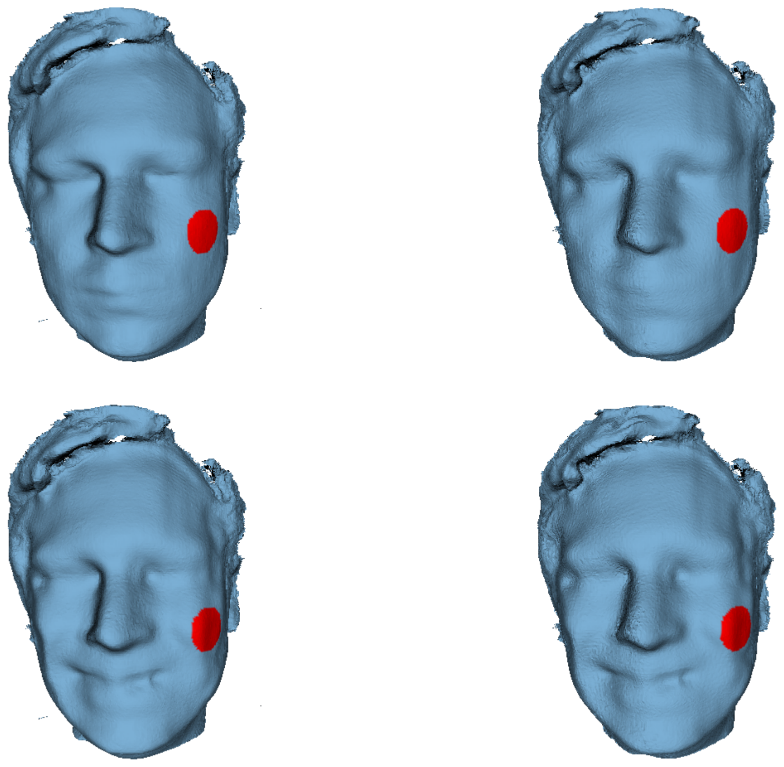

Figure 5.7: Facial Expressions - Comparison - Demo Point on Mesh

By not having seen any difference of both variants in the previous illustration, the focus now is placed on their translations. Again the same timestamps are chosen for the comparison. In the first row of Figure 5.8 a discrepancy of the translations is already apparent. While the template-based variant starts the tracking of the deformation not before a specific timestamp, the model-based variant handles this right from the beginning. Therefore it observes a translation for nearly every single point, even if it is just a small one. This can be seen by looking at the first row on the right hand side, where the surface of whole head has acquired some translations. In the bottom row some difference in the visualization can be observed as well. In conclusion there are some small deviations between both models, but the translations in the important areas around the mouth and the cheeks look pretty much the same.

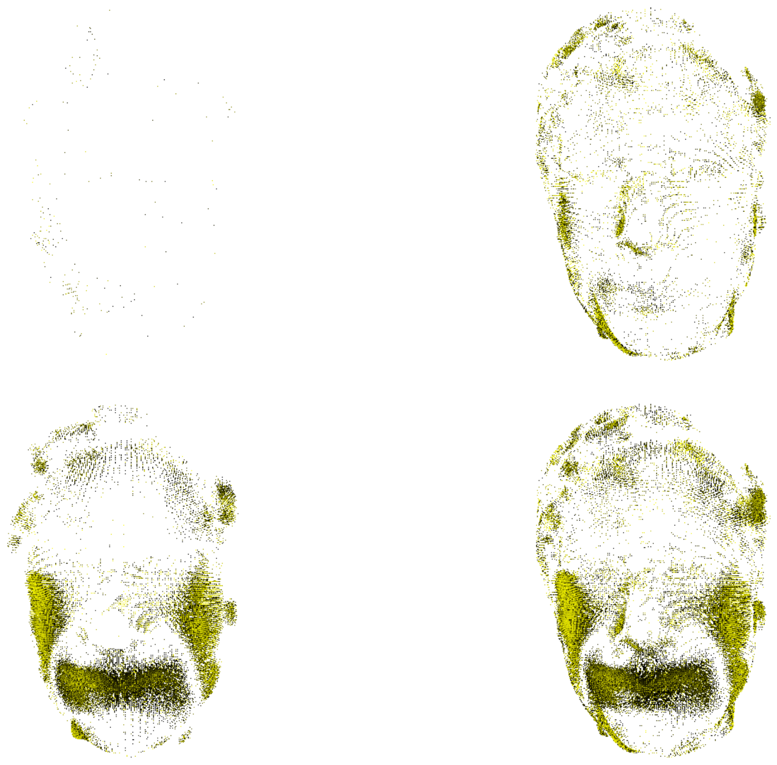

Figure 5.8: Facial Expressions - Comparison - Translations on Mesh

6 Validation

The validation is going to be executed in the terms of quality and resources. This process is applied to the template-based as well as to the model-based method, by using the sample of the previous chapter.

For demonstrating the quality of the estimation of the translations and their mapping either onto the mesh or into the model, the occurring deviation in two timestamps is visualized in Figure 6.1. The figure contains the same timestamps as the two last graphics of Chapter 5 (see Figures 5.7 and 5.8). The rasterized vertex maps produced at the end of both pipelines, represent pretty exactly the mesh and the model respectively. By comparing one of these maps with the bilaterally filtered current vertex map, taken from the same pipeline iteration, the difference between the input and the model can be easily determined. Because both maps are either recorded or rendered from the exact same camera viewpoint, the difference can be directly acquired by subtracting the positions, accessed by using the same i, j indices. For visualizing the deviation, yellow arrows are drawn, pointing from the points of the previous map to the ones of the current map. Obviously, it can be seen that the maps of both variants do not really differ in the main part of the face. Only the borders show a quiet huge deviation. But this is expected and also desired. This effect results from the smoothing process found in the translation estimation task (see Section 3.2.4).

Besides the quality, also the use of the resources is an important factor for quantifying a system. Therefore both variants of the proposed approach are compared with the implementation of KinectFusion in form of processing time and memory usage. Because of doing all calculation intensive tasks on the GPU and holding all its data, the load of the video memory gets considered. The graphics adapter used in the test system is labeled NVIDIA Quadro 1000M which serves 2 GB of video memory while having 96 CUDA

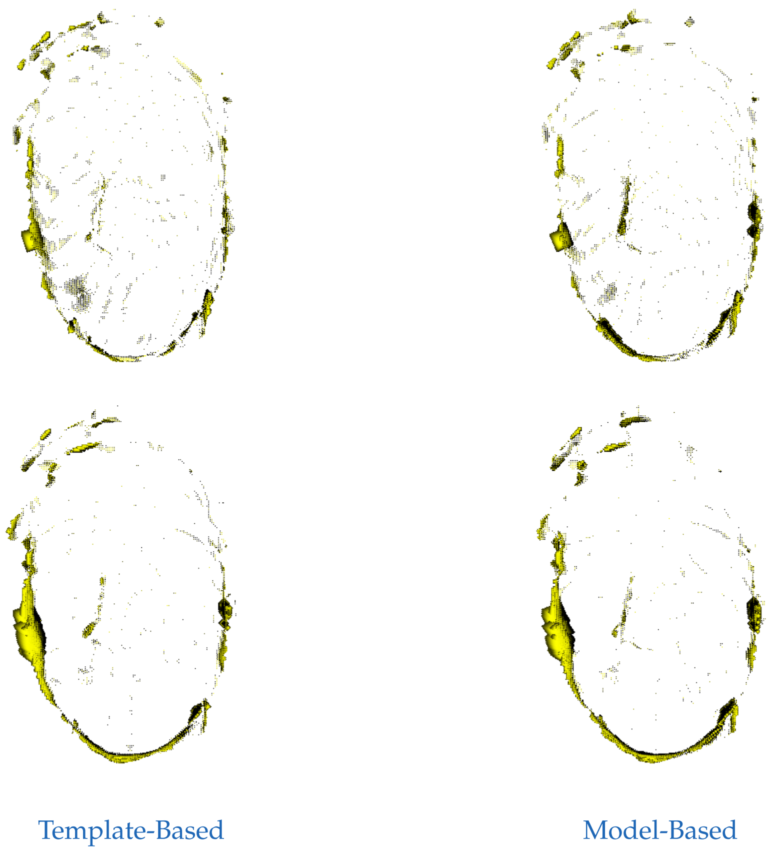

Figure 6.1: Facial Expressions - Validation - Deviation

cores. The resolution of the TSDF model, the color model and in case of the model-based variant, also the translations model is set to 256^3. The used example contains 321 frames to be processed, while in case of the template-based method its actual approach is active for the last 151 frames only. The tests run with and without color integration, therefore in the first case the color model needs to be allocated and processed too. The results obtained by the test scenario are shown in Table 6.1. By looking at the numbers it can be obviously seen that the template-based variant is nearly as fast as KinectFusion while using about the double amount of memory. Comparing those two variants with the model-based one, a doubled time consumption and a much higher memory usage can be noticed. The color integration seems to affect the resources in all three cases identically by triggering a constant offset in time and also memory manner. Basically, the main causer for the memory usage are the fixed sized models (TSDF, color and translations), which are created depending on the used case. Additionally to them the storage of the mesh is also requiring many bits on the GPU, by being defined in fixed size too. The mesh structure is specified to be able to hold two million triangles, regardless of whether or not used. Also it has to be noted, that KinectFusion does not require a mesh for working properly, in contrast to the new approach. But the creation of the mesh is activated for getting the result in the preferred representation, compared to a point cloud for example. In fact, this optional step has an influence on the results of time and memory consumption. The last culprit, occupying some perceptible amount of memory in case of the new approach, is caused by the applied rasterization technique. To be able to perform a fast rasterization, 128 vertex maps are allocated for running the algorithm in the same amount of threads in parallel (see Section 3.2.2).

System	**Color**	**Time (s)**	**Memory (MB)**
KinectFusion	No	53.2	292
	Yes	56.7	379
Deformable 3D Reconstruction - Template-Based	No	65.7	541
	Yes	69.8	633
Deformable 3D Reconstruction - Model-Based	No	124.6	797
	Yes	129.6	884

Table 6.1: Facial Expressions - Validation - Resources

7 Conclusion and Future Work

As shown by the results (see Chapter 5) the template-based method and also the model-based method appear to work pretty well with the applied use case. The resulting meshes do have a good quality when the object is in the rigid state as well as in a deformed. Additionally the coloring offers a realistic representation of the recorded object of the real world. Besides the quality, the speed is a second satisfying property of the proposed system. The fast reconstruction is achieved by doing nearly all calculations on the GPU. A comparison of both methods shows, that the template-based variant is about twice as fast as the model-based one. The first one nearly reaches the processing time of the PCL's KinectFusion implementation. When there is no deformation expected until the complete object has been captured, the template-based method may be chosen. In this case there are no additional resources needed for saving the translations in the memory of the GPU and therefore the free space may be used to increase the resolution of the TSDF model.

Even though having achieved good results, there are some limitations left to be considered, which lead to a reduction in the range of applications. Basically, the system is only able to capture surface deformation and would fail if a complete body deformation, for example in form of a topology change, would happen. In principle this has three different reasons: Firstly, the use of only one input device, obviously, offers only one angle of view and captures the information just from one side. This means any deformation, which happens outside of the camera's viewing frustum, for instance taking place at the back of an object, would lead to a missing observation. In general, the proposed system could be extended to support multiple devices simultaneously to avoid this problem, as already proposed by Dou et al. (2016). But the objective of this thesis was to depend on only one device for the recording. Secondly, the integration of the translations does take care only of the current viewing depth within some small degree, but the manipulation needs to be applied onto a complete 3D mesh or model. This happens because the integration of the translations is processed pointwise, by doing the iteration over the vertices or voxels. This further results in not taking care of the connectivity of the neighbors of the mesh or model respectively. For the template-based method a possible solution would be to use an algorithm which is known as as-rigid-as-possible surface deformation (Sorkine and Alexa, 2007). By applying this algorithm, using the estimated translation map for the definition of the applied keypoints and their movement as input, unrelated connected parts of the mesh would get moved accordingly to ensure a smooth surface also at the borders. Definitely this would solve that issue, but having a negative side-effect in form of a huge calculation effort. This is because of having to process the whole mesh for this calculation. Finally, the last reason for not being able to capture the full body deformation is that the correspondences estimation task does not always output the correct point pairs. Therefore the translations estimation is not able to produce correct results in this area too. This is one reason why the color integration is still continued, although the depth integration has been already stopped. The culprit for the incorrect estimation of the point pairs lies in the finding of the closest but not the most appropriate correspondences. Even though this sometimes results in a somehow incorrect translation map, the deformed mesh will stay more regularized as if the correspondences would be more correct. This effect is triggered by picking closer vertices and applying less translations on the related regions. By avoiding this big movements of the vertices, as a consequence, big faces will occur less often. The same also applies for the opposite case where shrinking would lead to many small faces. This incorrect estimation yields to smoother deformations in overall. This can be observed for instance by looking at the area around the mouth (see Figure 5.5), when a big deformation happens (e.g.: smiling). Usually the stretching would be much higher at this area and therefore would lead to relatively big faces. In order to get a more realistic deformation, an additional method for estimating the correspondences could be used. Using in addition to the depth input, the color input to detect Scale Invariant Feature Transform (SIFT) (Lowe, 1999; Lowe, 2004) features, as shown by Innmann et al. (2016), the real movement of points could be observed. The detection and the description of these features is processed frame by frame and furthermore used for the matching of the images. By doing this globally for all frames, also the problem of loop-closure can be solved (Innmann et al., 2016). Unfortunately the SIFT method is very computational costly and therefore not easy applicable for real-time applications.

Another improvement of the model-based pipeline would be to take care of the rotations too, additionally to the currently estimated translations. This would result in a more exact description of the deformation model and maybe further improve the result. The use of the rotation has been avoided to not exploit the available video memory for the storage of the deformation model only. While having the requirement of saving three floating point numbers per voxel in the current system, at least three additional numbers would be required to handle the rotation too. As described in the Section 4.2, the use of one single number becomes very expensive when being extrapolated to a complete 3D voxel grid. Besides that, the other reason for omitting the rotations was to simplify some calculations in the sense of complexity and processing effort. For instance, one of such calculation is the bilinear interpolation, applied to integrate the translations into the mesh or model (see Sections 3.2.5 and 3.3.1).

Summarized, the proposed approach provides good results when used in its desired use case. By having the choice of the two different variants, the advantages and disadvantages could be weight up first in the view of the needed applied scenario. While having the big benefits in form of quality and speed, when using the template-based method, the model-based method supports deformation already from the beginning. After having discussed some related work (see Chapter 2), showing other impressive approaches and results, there is still room for improvements.

Bibliography

Andrews, K. (2012). *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria (cit. on p. xv).

Ben-Kiki, O., C. Evans, and I. d. Net (2009). *YAML Ain't Markup Language (YAML^{TM}) Version 1.2*. URL: `http://yaml.org/spec/1.2/spec.html` (visited on 09/02/2018) (cit. on p. 62).

Besl, P. J. and N. D. McKay (1992). "A Method for Registration of 3-D Shapes." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2, pp. 239–256. ISSN: 0162-8828. DOI: `10.1109/34.121791` (cit. on pp. 5, 19).

Chen, J., D. Bautembach, and S. Izadi (2013). "Scalable Real-Time Volumetric Surface Reconstruction." In: *ACM Trans. Graph.* 32.4, 113:1–113:16. ISSN: 0730-0301. DOI: `10.1145/2461912.2461940` (cit. on p. 11).

Chen, Y. and G. Medioni (1991). "Object Modeling by Registration of Multiple Range Images." In: *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, pp. 2724–2729. DOI: `10.1109/ROBOT.1991.132043` (cit. on pp. 21, 38).

Collet, A., M. Chuang, P. Sweeney, D. Gillett, D. Evseev, D. Calabrese, H. Hoppe, A. Kirk, and S. Sullivan (2015). "High-Quality Streamable Free-Viewpoint Video." In: *ACM Trans. Graph.* 34.4, 69:1–69:13. ISSN: 0730-0301. DOI: `10.1145/2766945` (cit. on pp. 10, 11).

Curless, B. and M. Levoy (1996). "A Volumetric Method for Building Complex Models from Range Images." In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, pp. 303–312. ISBN: 0-89791-746-4. DOI: `10.1145/237170.237269` (cit. on pp. 5, 6, 22).

Dou, M., S. Khamis, Y. Degtyarev, P. Davidson, S. R. Fanello, A. Kowdle, S. O. Escolano, C. Rhemann, D. Kim, J. Taylor, P. Kohli, V. Tankovich, and S. Izadi (2016). "Fusion4D: Real-Time Performance Capture of Challenging Scenes." In: *ACM Trans. Graph.* 35.4, 114:1–114:13. ISSN: 0730-0301. DOI: 10.1145/2897824.2925969 (cit. on pp. 11, 12, 81).

Duce, D. et al. (2003). *Portable Network Graphics (PNG) Specification (Second Edition)*. URL: https://www.w3.org/TR/2003/REC-PNG-20031110 (visited on 09/02/2018) (cit. on p. 61).

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN: 0-201-63361-2 (cit. on pp. 63, 65).

Guo, K., F. Xu, T. Yu, X. Liu, Q. Dai, and Y. Liu (2017). "Real-Time Geometry, Albedo, and Motion Reconstruction Using a Single RGB-D Camera." In: *ACM Trans. Graph.* 36.3. ISSN: 0730-0301. DOI: 10.1145/3083722 (cit. on p. 9).

Innmann, M., M. Zollhöfer, M. Nießner, C. Theobalt, and M. Stamminger (2016). "VolumeDeform: Real-Time Volumetric Non-Rigid Reconstruction." In: *Computer Vision – ECCV 2016*. Ed. by B. Leibe, J. Matas, N. Sebe, and M. Welling. Cham: Springer International Publishing, pp. 362–379. ISBN: 978-3-319-46484-8 (cit. on pp. 9, 10, 54, 82, 83).

Izadi, S., D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon (2011). "KinectFusion: Real-Time 3D Reconstruction and Interaction Using a Moving Depth Camera." In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST '11. Santa Barbara, California, USA: ACM, pp. 559–568. ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047270 (cit. on pp. 5, 7, 13, 17, 22).

Kazhdan, M., M. Bolitho, and H. Hoppe (2006). "Poisson Surface Reconstruction." In: *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*. SGP '06. Cagliari, Sardinia, Italy: Eurographics Association, pp. 61–70. ISBN: 3-905673-36-3 (cit. on p. 11).

Keller, M., D. Lefloch, M. Lambers, S. Izadi, T. Weyrich, and A. Kolb (2013). "Real-Time 3D Reconstruction in Dynamic Scenes Using Point-Based Fusion." In: *2013 International Conference on 3D Vision - 3DV 2013*, pp. 1–8. DOI: 10.1109/3DV.2013.9 (cit. on pp. 6, 7).

Li, H., B. Adams, L. J. Guibas, and M. Pauly (2009). "Robust Single-View Geometry and Motion Reconstruction." In: *ACM Trans. Graph.* 28.5, 175:1–175:10. ISSN: 0730-0301. DOI: 10.1145/1618452.1618521 (cit. on p. 11).

Lorensen, W. E. and H. E. Cline (1987). “Marching Cubes: A High Resolution 3D Surface Construction Algorithm.” In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’87. New York, NY, USA: ACM, pp. 163–169. ISBN: 0-89791-227-6. DOI: 10.1145/37401.37422 (cit. on p. 28).

Lowe, D. G. (1999). “Object Recognition from Local Scale-Invariant Features.” In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2, pp. 1150–1157. DOI: 10.1109/ICCV.1999.790410 (cit. on pp. 9, 82).

Lowe, D. G. (2004). “Distinctive Image Features from Scale-Invariant Keypoints.” In: *Int. J. Comput. Vision* 60.2, pp. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94 (cit. on pp. 9, 82).

Netpbm (2013). *The PNM Format*. URL: http://netpbm.sourceforge.net/doc/pnm.html (visited on 08/13/2018) (cit. on p. 61).

Netpbm (2016a). *PGM Format Specification*. URL: http://netpbm.sourceforge.net/doc/pgm.html (visited on 08/13/2018) (cit. on p. 61).

Netpbm (2016b). *PPM Format Specification*. URL: http://netpbm.sourceforge.net/doc/ppm.html (visited on 08/13/2018) (cit. on p. 61).

Newcombe, R. A., D. Fox, and S. M. Seitz (2015). “DynamicFusion: Reconstruction and Tracking of Non-Rigid Scenes in Real-Time.” In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 343–352. DOI: 10.1109/CVPR.2015.7298631 (cit. on pp. 8–10, 48).

Newcombe, R. A., S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon (2011). “KinectFusion: Real-Time Dense Surface Mapping and Tracking.” In: *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pp. 127–136. DOI: 10.1109/ISMAR.2011.6092378 (cit. on pp. 5, 6, 13, 17).

Nießner, M., M. Zollhöfer, S. Izadi, and M. Stamminger (2013). “Real-Time 3D Reconstruction at Scale Using Voxel Hashing.” In: *ACM Trans. Graph.* 32.6, 169:1–169:11. ISSN: 0730-0301. DOI: 10.1145/2508363.2508374 (cit. on pp. 6, 7).

Orts-Escolano, S., C. Rhemann, S. Fanello, W. Chang, A. Kowdle, Y. Degtyarev, D. Kim, P. L. Davidson, S. Khamis, M. Dou, V. Tankovich, C. Loop, Q. Cai, P. A. Chou, S. Mennicken, J. Valentin, V. Pradeep, S. Wang, S. B. Kang, P. Kohli, Y. Lutchyn, C. Keskin, and S. Izadi (2016).

"Holoportation: Virtual 3D Teleportation in Real-Time." In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. UIST '16. Tokyo, Japan: ACM, pp. 741–754. ISBN: 978-1-4503-4189-9. DOI: 10.1145/2984511.2984517 (cit. on p. 12).

Parker, S., P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan (1998). "Interactive Ray Tracing for Isosurface Rendering." In: *Proceedings of the Conference on Visualization '98*. VIS '98. Research Triangle Park, North Carolina, USA: IEEE Computer Society Press, pp. 233–238. ISBN: 1-58113-106-2 (cit. on p. 29).

Pineda, J. (1988). "A Parallel Algorithm for Polygon Rasterization." In: *SIGGRAPH Comput. Graph.* 22.4, pp. 17–20. ISSN: 0097-8930. DOI: 10.1145/378456.378457 (cit. on p. 34).

Rusinkiewicz, S. and M. Levoy (2001). "Efficient Variants of the ICP Algorithm." In: *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*, pp. 145–152. DOI: 10.1109/IM.2001.924423 (cit. on pp. 20, 38).

Rusu, R. B. and S. Cousins (2011). "3D is here: Point Cloud Library (PCL)." In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China (cit. on p. 13).

Scratchapixel (2018). *Rasterization: a Practical Implementation*. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation (visited on 08/13/2018) (cit. on p. 34).

Slavcheva, M., M. Baust, D. Cremers, and S. Ilic (2017). "KillingFusion: Non-Rigid 3D Reconstruction Without Correspondences." In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5474–5483. DOI: 10.1109/CVPR.2017.581 (cit. on p. 10).

Slavcheva, M., W. Kehl, N. Navab, and S. Ilic (2016). "SDF-2-SDF: Highly Accurate 3D Object Reconstruction." In: *Computer Vision – ECCV 2016*. Ed. by B. Leibe, J. Matas, N. Sebe, and M. Welling. Cham: Springer International Publishing, pp. 680–696. ISBN: 978-3-319-46448-0 (cit. on p. 10).

Sorkine, O. and M. Alexa (2007). "As-Rigid-As-Possible Surface Modeling." In: *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*. SGP '07. Barcelona, Spain: Eurographics Association, pp. 109–116. ISBN: 978-3-905673-46-3 (cit. on pp. 7, 82).

Sumner, R. W., J. Schmid, and M. Pauly (2007). "Embedded Deformation for Shape Manipulation." In: *ACM SIGGRAPH 2007 Papers*. SIGGRAPH

'07. San Diego, California: ACM. DOI: 10.1145/1275808.1276478 (cit. on pp. 8, 11).

Tomasi, C. and R. Manduchi (1998). "Bilateral Filtering for Gray and Color Images." In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pp. 839–846. DOI: 10.1109/ICCV.1998.710815 (cit. on p. 15).

Turk, G. (1994). *The PLY Polygon File Format*. URL: http://web.archive.org/web/20170502135307/http://www.dcs.ed.ac.uk:80/teaching/cs4/www/graphics/Web/ply.html (visited on 09/02/2018) (cit. on p. 62).

Umeyama, S. (1991). "Least-Squares Estimation of Transformation Parameters Between Two Point Patterns." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13.4, pp. 376–380. ISSN: 0162-8828. DOI: 10.1109/34.88573 (cit. on p. 41).

Voit, K. (2018). *novoid/LaTeX-KOMA-template: Generic Template For Midsize and Larger Documents Based on KOMA script classes*. URL: https://github.com/novoid/LaTeX-KOMA-template (visited on 09/06/2018) (cit. on p. xv).

Weisstein, E. W. (2018). *Point-Line Distance–3-Dimensional*. URL: http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html (visited on 08/13/2018) (cit. on p. 38).

Zollhöfer, M., M. Nießner, S. Izadi, C. Rehmann, C. Zach, M. Fisher, C. Wu, A. Fitzgibbon, C. Loop, C. Theobalt, and M. Stamminger (2014). "Real-Time Non-Rigid Reconstruction Using an RGB-D Camera." In: *ACM Trans. Graph.* 33.4, 156:1–156:12. ISSN: 0730-0301. DOI: 10.1145/2601097.2601165 (cit. on pp. 7–9, 11, 31).

Zollhöfer, M., P. Stotko, A. Görlitz, C. Theobalt, M. Nießner, R. Klein, and A. Kolb (2018). "State of the Art on 3D Reconstruction with RGB-D Cameras." In: *Computer Graphics Forum (Eurographics State of the Art Reports 2018)* 37.2 (cit. on p. 12).