VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

3D Reconstruction on iOS

3D rekonstrukce na iOS

2018 Marek Šimoník

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Bachelor Thesis Assignment

Student:	**Marek Šimoník**
Study Programme:	B2647 Information and Communication Technology
Study Branch:	2612R025 Computer Science and Technology
Title:	3D Reconstruction on iOS 3D rekonstrukce na iOS
The thesis language:	English

Description:

The aim of this thesis is a 3D scene reconstruction using a RGB-D camera located on a mobile device. The final application will be able to display a 3D model of the scanned space.

Complete the following steps in your thesis:
1. Read and study available algorithms for 3D reconstruction of objects and scenes [1, 2, 3].
2. Implement the required algorithms using available libraries.
3. Test your implementation properly on the mobile device.
4. Properly document your solution.

References:

[1] D. Hernandez-Juarez and A. Chacón and A. Espinosa and D. Vázquez and J.C. Moure and A.M. López: Embedded Real-time Stereo Estimation via Semi-global Matching on the GPU, International Conference on Computational Science 2016, 80, pp. 143-153, issn: 1877-0509, (2016)
[2] Chang, Ting-An and Lu, Xiao and Yang, Jar-Ferr: Robust stereo matching with trinary cross color census and triple image-based refinements, EURASIP Journal on Advances in Signal Processing, issn: 1687-6180, (2017)
[3] O. Kähler, V. Adrian Prisacariu, C. Yuheng Ren, X. Sun, P. Torr and D. Murray,: Very High Frame Rate Volumetric Integration of Depth Images on Mobile Devices, IEEE Transactions on Visualization and Computer Graphics, vol. 21, no. 11, pp. 1241-1250, (2015)

Supervisor: **Ing. Jan Gaura, Ph.D.**

Date of issue: 01.09.2017
Date of submission: 30.04.2018

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA
Fakulta elektrotechniky a informatiky
1D16
OSTRAVA

[illegible]

doc. Ing. Jan Platoš, Ph.D.
Head of Department

[illegible]

prof. Ing. Pavel Brandštetter, CSc.
Dean

I hereby declare that this bachelor's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, April 30 2018 .... [illegible]

I hereby agree to the publishing of the bachelor's thesis as per s. 26, ss. 9 of the Study and Examination Regulations for Bachelor's Degree Programmes at VŠB – Technical University of Ostrava.

Ostrava, April 30 2018

[illegible]
. . . .

I would like to thank my supervisor Ing. Jan Gaura, Ph.D. for his kindness, benevolence and excellent advices while consulting technical details.

Abstrakt

Tato bakalářská práce popisuje implementaci řetězce pro 3D rekonstrukci z RGBD snímků v reálném čase, určené pro Apple iPhone X s TrueDepth kamerou. Nejdříve je podán přehled běžných přístupů k rekonstrukci, následován popisem algoritmů a technik použitých v této práci. Nakonec jsou popsány implementační detaily zvoleného rekonstrukčního řetězce spolu s popisem výkonnosti implementované aplikace.

Klíčová slova: 3D rekonstrukce, iOS, RGBD, TrueDepth kamera, TSDF

Abstract

This bachelor thesis describes implementation of a real-time RGBD-based 3D reconstruction pipeline suited for Apple's iPhone X with the TrueDepth camera. First, an overview of common approaches to the reconstruction problem is made, followed by a description of the underlying algorithms and techniques used in the thesis. Finally, the implementation details of the application pipeline are presented with performance overview of the implemented application.

Key Words: 3D reconstruction, iOS, RGBD, TrueDepth camera, TSDF

Contents

List of symbols and abbreviations

SL	–	Structured Light scanning
ToF	–	Time-of-Flight scanning
RGB	–	Red Green Blue
RGBD	–	Red Green Blue Depth
px	–	Pixel
CPU	–	Central Processing Unit
GPU	–	Graphics Processing Unit
SIMD	–	Single Instruction Multiple Data
SDF	–	Signed Distance Field
TSDF	–	Truncated Signed Distance Field
API	–	Application Programming Interface
IMU	–	Inertial Measurement Unit
ICP	–	Iterative Closest Point
CUDA	–	Compute Unified Device Architecture
GCD	–	Grand Central Dispatch
GPGPU	–	General-Purpose programming on Graphics Processing Units
POSIX	–	Portable Operating System Interface
QoS	–	Quality of Service
PLY	–	Polygon File Format
FPS	–	Frames per Second
MRI	–	Magnetic Resonance Imagining
AR	–	Artificial Reality
DoF	–	Degrees of Freedom
SGM	–	Semi-Global Matching
I/O	–	Input/Output

List of Figures

List of Tables

Listings

1 Introduction

Today it is easy to take photos and videos of objects to remember the past, but they are a bit flat. Would not it be better to capture the past in 3D? Mobile devices are arguably the most ubiquitous AR platform in the world today with the help of Apple's ARKit [1]. Despite the mass propagation of AR, users are still a mere consumers of AR content.

I think the ideal start of producing 3D content is scanning — with further abilities to edit it and couple with real-world objects. This was one of the reasons behind my decision to create a scanning application that would be easy to use without doing tedious pre-scanning setup, like scanning with the Kinect [2] requires.

And because Apple announced the iPhone X with the front facing "depth sensor" — the TrueDepth camera — which is based on the same technology as Kinect v1, it was clear there was no better way to create high-quality 3D content anywhere and at any time.

2 Approaches to 3D Reconstruction

There are several ways to perform 3D reconstruction; those can be divided into two groups regarding the scanners they are used with; *photogrammetry* and *scanning* (laser, Structured Light (SF) or Time-of-Flight (ToF) scanning).

2.1 Ways of Obtaining Data

The photogrammetry methods use images of the scanned object from various poses, to reconstruct the final object. Because this method uses visible-light images (taken by an ordinary commodity camera) only, it does not have range limitation and can be used for reconstruction of everyday objects, rooms, up to large-scale areas from aerial or satellite photos (e.g. 3D view of cities in Google Maps). The disadvantage is that there are ambiguities when computing the depth, which can result in considerable reconstruction errors. Such methods are also known as *passive*, because they do not interfere with environment.

The active scanning methods are invasive, meaning the scanners interfere with the scanned scene e.g. by illuminating it via infrared light (structured light), or light pulses. Output of such scanners can be either range image[1] (for SL and ToF), or 3D point cloud (for laser scanning). Even though the scanners are accurate, some of them (SL, ToF) can be used in a limited range only (up to a few meters). For the rest of this thesis, only sensors that output data at real-time[2] rates will be assumed.

2.2 Reconstruction Methods

Most of the reconstruction approaches require considerable computing power to run at real-time rates. To satisfy this requirement, the underlying algorithm should be parallelizable, with the possibility to run on a GPU, to leverage the high number of cores and SIMD architecture. The computation is often split between CPU (which can perform tracking) and GPU (which performs image-related operations). Both active and passive reconstruction methods follow a similar general reconstruction pipeline;

1. obtaining data from a sensor and producing a range image,
2. computing the current global pose of the sensor — *tracking*,
3. using the current pose to fuse the range image into the reconstructed 3D model.

[1]A 2D image where each pixel holds the distance from the camera sensor to the nearest object in scene.
[2]Ideally at ≥ 30 FPS.

2.2.1 Producing Range Image

This is the first step in a reconstruction pipeline and is performed in the case of photogrammetry. The range (depth) image is computed from a pair[3] of RGB images. It is required that the camera is calibrated (see section 3) and the image pair is rectified[4] (see section 4.1).

After the raw range image is available (either by computing from RGB images or obtained from a scanner) it can be post-processed by applying a median or an edge-preserving[5] filter(s), to mitigate noise. Note that after obtaining the range image, the type of sensor used is abstracted away for the subsequent stages of pipeline.

2.2.2 Tracking

Next, it is needed to compute the global pose of the captured range image. This might be done in several ways;

- by using a pair of RGB images — find point correspondences between the image pair and compute pose via essential matrix [5],
- by using a pair of range images — a variant of the ICP algorithm (see section 5.1) can be used,
- via a combination of the above approaches — create a variant of ICP where we optimize two weighted terms – geometric (pure ICP) and photometric (RGB).

2.2.3 Creating Global Model

In the last step of a pipeline, the range image is fused with the global model; each range image contains noise, but the fusion of multiple images produces a smoothed average which results into mitigation of noise in the global model. Various techniques can be used to create the global model;

- Point Clouds [6],
- Surfels [7],
- (Truncated) Signed Distance Fields (see section 7.5.2).

[3]Multiple images can be used for more precise results (Multiview Stereo reconstruction [3]). In recent years, several Machine Learning approaches have been developed which can hallucinate depth even from single RGB image [4].

[4]Rectification is not required per se, but it is more convenient to work with and faster to process rectified image pair.

[5]Usually a variant of bilateral filter.

3 Camera Calibration

Before a range image could be generated (in the case of photogrammetry) and also subsequently used to update 3D model, it is necessary to know the internal parameters of the camera sensor (for both RGB and RGBD sensors).

By knowing the parameters, it is possible to project arbitrary 3D point from the *camera coordinate space* into the homogeneous 2D *image coordinate space* (see Figure 1). Inverse projection is also possible (a 2D point from the image c.s. can be projected into the camera c.s., but depth for the point must also be provided).

A simple camera model – *pinhole camera* – is used. This model assumes no lenses and therefore no image distortions.

All these internal parameters are commonly expressed in the 3×3 *intrinsic matrix* K of a pinhole camera (see Equation 1). A camera-space point Q can be transformed into image-space point q as $q = KQ$ (see sections 3.1 and 3.2 for derivation).

$$K = \begin{bmatrix} f_x & s & P_x \\ 0 & f_y & P_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

where:

f_x = horizontal focal length [px]
f_y = vertical focal length [px]
s = image skew
P_x = horizontal offset (from top-left corner of image) of the Principal Point [px]
P_y = vertical offset (from top-left corner of image) of the Principal Point [px]

Note that there are 2 values of focal length in K, even though the camera has only one focal length (in world units). The reason for this is that f_x and f_y are expressed in pixels (this will be useful when triangulating real-world position from pixel position and depth (see section 4.5)). When the scanner sensor's pixels are not perfectly square, then $f_x \neq f_y$, because they are scaled to compensate rectangularity of the sensor's pixels.

The skew factor s expresses the shear of sensor's pixels. However, because the current sensors' pixels are rectangular, there is zero skew ($s = 0$).

If K is not provided by the scanner vendor, it has to be found by calibration via checkerboard pattern [8]. Such calibration is out o scope of this thesis, because Apple's APIs provide intrinsic matrices for cameras.

3.1 Using K to Project Points to the Image Coordinate Space

Let's assume a point $Q = \begin{bmatrix} Q_x & Q_y & Q_z \end{bmatrix}^T$ in the camera-space (see Figure 1) that we want to project into the image-space, to obtain the corresponding 2D coordinates $q = \begin{bmatrix} q_x & q_y \end{bmatrix}^T$. To

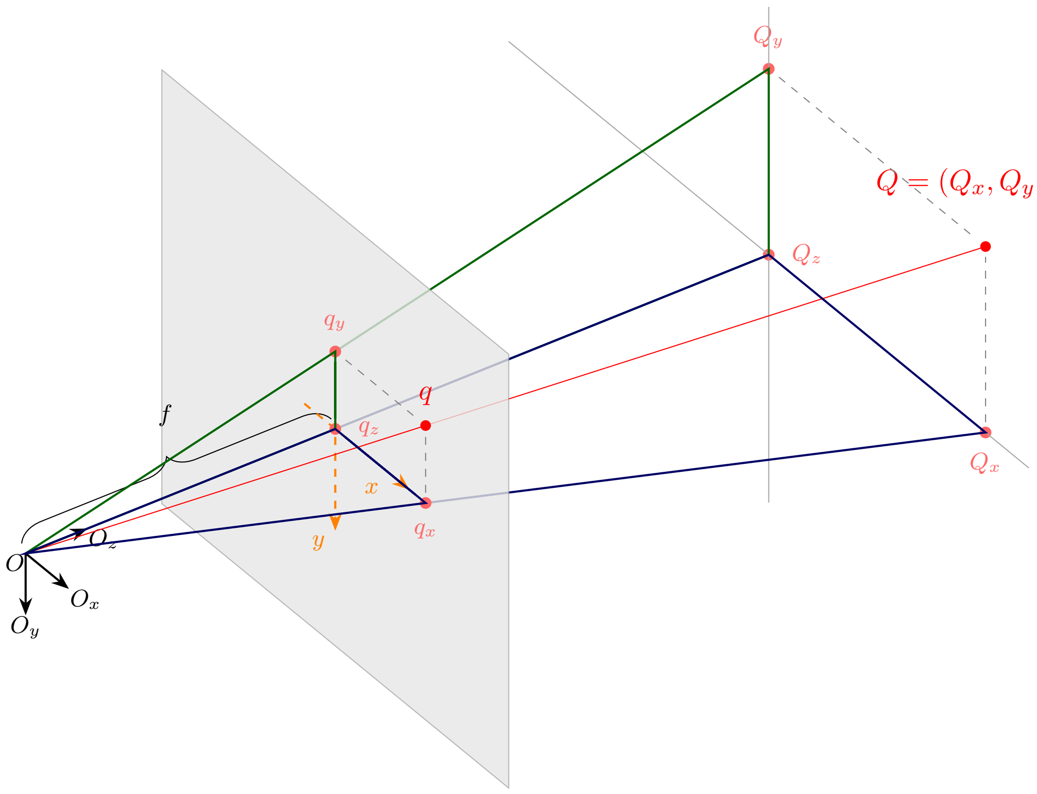


Figure 1: Pinhole camera model.

simplify the projection, we project the x and y components of Q individually. First we start by projecting the x component; we form 2 similar triangles $\triangle OQ_zQ_x$ and $\triangle Oq_zq_x$. It can be seen that by similar triangles it holds that:

$$\begin{aligned} \frac{q_x}{f_x} &= \frac{Q_x}{Q_z} & \qquad \frac{q_y}{f_y} &= \frac{Q_y}{Q_z} \\ q_x &= \frac{Q_x}{Q_z} f_x & \qquad q_y &= \frac{Q_y}{Q_z} f_y \end{aligned} \tag{2}$$

Now we have obtained coordinates q in the image-space (which has origin in the principal point P), and want to transform them to the pixel-space coordinates q'. Pixel- and image-spaces are only offsetted by the principal point P, therefore we can just add the offset $P = \begin{bmatrix} P_x & P_y \end{bmatrix}^T$ (relative to the pixel-space) to q to obtain $q' = q + P$.

In practice, the intrinsic matrix K is used to transform the camera-space point Q to homogenous 2D pixel-space point $\dot{q}$ that can be dehomogenized into pixel-coordinates q';

$$\begin{aligned}
\dot{q} &= KQ \\
&= \begin{bmatrix} f_x & 0 & P_x \\ 0 & f_y & P_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Q_x \\ Q_y \\ Q_z \end{bmatrix} = \begin{bmatrix} Q_x f_x + Q_z P_x \\ Q_y f_x + Q_z P_y \\ Q_z \end{bmatrix} \\
q' &= \begin{bmatrix} \frac{\dot{q}_x}{\dot{q}_z} \\ \frac{\dot{q}_y}{\dot{q}_z} \end{bmatrix} = \begin{bmatrix} \frac{Q_x}{Q_z} f_x \\ \frac{Q_y}{Q_z} f_y \end{bmatrix}
\end{aligned} \tag{3}$$

3.2 Using K for Inverse Projection to Camera Coordinate Space

The inverse projection can be derived analogously. If we know the pixel coordinates q', camera-space depth Q_z of the projected point and intrinsic matrix K, the camera-space point can be computed as $Q = Q_z K^{-1} \begin{bmatrix} q'_x & q'_y & 1 \end{bmatrix}^T$.

4 Photogrammetry Approach — Generating Depth Image From Stereo RGB Pair

Although not used in the current pipeline, the photogrammetry approach to generation of depth images is briefly described — to give the reader a general idea of how to approach 3D reconstruction without RGBD data —, because an earlier version of this thesis used it.

The range image (depth image) can be generated from a pair of visible-light images with overlapping views. The goal is to compute depth for each pixel in the image (for the left or right). To be able to compute depth for a pixel at coordinate ${}^{l}c \in \mathbb{R}^2$ from the left image ${}^{l}I$, we need to realise that the world-space point ${}^{c}p \in \mathbb{R}^3$, which corresponds to ${}^{l}c$ (i.e. which was projected onto the image plane of ${}^{l}I$ under the pixel ${}^{l}c$), must also be visible from the viewpoint of the right image ${}^{r}I$ under another pixel coordinate ${}^{r}c$.

For successful computation of depth image from the left image ${}^{l}I$, we need to find for each one of its pixels the corresponding pixel in the right image ${}^{r}I$. To find the corresponding point for ${}^{l}c$ from ${}^{l}I$, we could search in the 2D neighborhood of ${}^{l}c$ from ${}^{r}I$, nevertheless this approach would be computationally demanding.

To speed-up and simplify the search for correspondences, an image transformation called *rectification* can be applied to both images — the images are resampled via 2 projective transformations (a 3×3 matrices) in such way that makes the images appear as if the relative movement between the cameras that took them was only translational [9]; this makes the rows of pixels of the image pair correspond to each other (i.e. real-world points $p \in \mathbb{R}^3$ project to the same rows of ${}^{l}I$ and ${}^{r}I$ after rectification), which reduces the problem of correspondence search to 1D — pixel ${}^{l}c$ from ${}^{l}I$ has its corresponding pixel at the ${}^{l}c_y{}^{\text{th}}$ row of pixels of the image ${}^{r}I$ — it is only needed to find the shift $d = {}^{l}c_x - {}^{r}c_x$, so called *disparity*. After finding disparity for each pixel, the depth can be triangulated by knowing the intrinsic matrix K of the image pair (see section 4.5).

4.1 Rectification

There are several ways to rectify a pair of images; methods exist for both uncalibrated (intrinsic matrices K and relative pose between the two images are unknown) and calibrated cameras (Ks and relative camera poses are known). Because one of the reconstruction approaches I have tried utilized ARKit, which provides both world pose and intrinsic matrix, I will focus only on rectification in the calibrated case.

Rectifying a pair of images in the calibrated case means changing only the orientation (rotation) of the cameras (without changing their position) — i.e. modifying the rotational part of the camera-to-world matrices of the left (M_l) and right (M_r) images, so that the image planes are co-planar and their epipolar lines[6] are horizontal and parallel with the baseline of the two cameras.

A common rotation for both cameras needs to be found. Following [9], the axes of the new, common rotation matrix R_n can be defined in the following way:

- x axis as the baseline of the 2 cameras,
- y axis orthogonal to both the new x axis and the old left z axis,
- z axis is given as the axis orthogonal to the new x and y axis.

To obtain the 3×3 transformations that will be applied to the homogeneous image coordinates, we need to remap the original rotation matrix to the new, common one for both the left and right images [9].

4.2 Generating Disparity Image

After we have a rectified stereo image pair, we can create the disparity image for them. The techniques for disparity image generation can be categorized into three groups:

- local methods — the simplest ones; those utilize only local data around the pixel position — usually, they compare a small window of pixels around the source pixel in left image and around the same-sized window of pixels on the corresponding row in the right image (similar to pattern matching),
- semi-global methods — the best trade-off between quality and speed — these methods compute the same cost as the local methods, but on top of that perform piecewise optimization (i.e. applying a smoothness constraint on disparity values in each pixel-row separately — this can lead to typical striking artifacts),
- global methods — methods that compute disparity as an optimization problem — e.g. constraining the disparity to smoothly vary across the whole image.

4.3 Local Methods

Local methods are highly parallelizable, but do not ensure smooth disparity guesses — disparity guessed are not coupled via any smoothness constraint. These methods assume a maximal disparity of d_{max} pixels (e.g. 50), depending on image resolution. A typical method of this type does for each pixel at location ${}^{l}c$ from the left image ${}^{l}I$ compare a small (e.g. 11×11) window of pixels around ${}^{l}c$ from ${}^{l}I$ with the same-sized windows of pixels around the pixel ${}^{l}c$ in ${}^{r}I$ up to pixel ${}^{l}c - \begin{bmatrix} d_{max} & 0 \end{bmatrix}^T$ by a simple loss metric. The disparity value with the lowest value of

[6]Epipolar line E_r for a point ${}^{l}c$ from image ${}^{l}I$ is the line located on the image ${}^{r}I$, created by projecting points of the ray originating in the left camera's optical center and passing through the pixel ${}^{l}c$ of ${}^{l}I$ onto the image ${}^{r}I$. It is the line where the possible point correspondences for ${}^{l}c$ from ${}^{l}I$ are to be found in ${}^{r}I$.

loss metric represents the selected disparity. The three most common examples of loss functions are described in the following sections. By empirical observation, the results of using SAD and SSD are almost the same, but the CENSUS transform has considerably higher quality results (particularly when used with semi-global disparity-generation methods).

4.3.1 SAD (Sum of Absolute Differences)

This metric takes the two windows of pixels, for each corresponding pair of pixels computes the absolute value of their difference and sums the individual results into a single value. The smaller the value, the closer the match.

4.3.2 SSD (Sum of Squared Differences)

This is nearly identical to SAD, except that instead of summing absolute values of differences, the differences are squared — this penalizes outliers with quadratically higher cost.

4.3.3 CENSUS

The CENSUS transform requires preprocessing of the left ${}^{l}I$ and right ${}^{r}I$ images, i.e. creating a new pair of images ${}^{l}I'$ and ${}^{r}I'$ in the following way: a pixel's value at c in I' is formed as the concatenation of 0 or 1 bits based on whether the center pixel's value at c in I is greater than its neighboring pixels in a small window (say 9×7). This procedure is applied on both ${}^{l}I$ and ${}^{r}I$, producing ${}^{l}I'$ and ${}^{r}I'$.

To perform the comparison of similarity, we do not need to compare image windows like in the case of SAD or SSD — the 2 pixels at ${}^{l}c$ in ${}^{l}I'$ and ${}^{r}c$ in ${}^{r}I$ already contain these informations; we just XOR the 2 pixels' values and count the number of 1 bits in the result. The smaller the number of 1s, the more similar the 2 patches of image are.

Several extensions and improvements of the CENSUS transform exist; an e.g. the Center-Symmetric Census Transform (CSCT) [10] which creates the bit-string by comparing pairs of pixels symmetrical by the center of 9×7 window. So far all the described methods worked on grayscale images, there exists a variant that takes into consideration all RGB channels, the Trinary Cross Color Census, which is also being combined with different shapes of sampling window [11].

4.4 Semi-Global Disparity Generation

The output of most local methods is noisy and in most cases non-continuous. To improve the quality of disparity textures without compromising the computation time, semi-global method of generating disparity can be used; prior to implementing the RGBD reconstruction pipeline, I implemented reconstruction approach which used Semi-Global Matching (SGM) technique of [10].

This SGM technique is based on constraining the rows, columns and (optionally) diagonals of the output disparity image to contain piece-wise smooth values of disparity; first, the CSCT 3D cost volume is computed — for each pixel from the left image, the CSCT cost is computed for disparities in range $[0;\ d_{max}]$ (hence *3D* cost volume of size image width $\times$ image height $\times\, d_{max}$).

Then for each direction[7] r, a separate cost volume is created. Let us take the left-to-right direction as an example of creating a cost volume: each 2D xz slice of the cost volume is computed independently of each other (hence *semi-global* optimization) and each line in the z direction of that slice is computed at once, serially processing one z-line after another from left to right (i.e. increasing in the x direction of the slice).

Every element of the n^{th} z-line has its value updated as the sum of its CENSUS penalty value and the smallest penalty value from the $(n-1)^{\text{th}}$ z-line, with a small penalty $P1$ added in the case the smallest penalty from $(n-1)^{\text{th}}$ z-line was 1 pixel away from the current disparity value, and higher penalty $P2$ added in case the smallest previous disparity was more than 1 pixel away. No extra penalty is added if the smallest disparity from the previous step is the same as the current one. Cost volumes for other directions are created analogously.

After all cost volumes for every required direction are computed, they are summed together and the final disparity value for a pixel is given by selecting the disparity value with the smallest cost from the summed 3D cost volume for the given pixel. Usage of several directions mitigates the negative effect of streaking artifacts. For further details refer to [10].

4.5 Generating Depth Image From Disparity Image

After the disparity texture is found, we can create a depth texture out of it via triangulation [12] if we know the intrinsic matrix K of the rectified image pair:

$$Z = \frac{Bf}{d} \tag{4}$$

where:

Z = the computed depth
B = the baseline between cameras
f = the focal length
d = the disparity

[7]Possible directions are: left-to-right, right-to-left, bottom-to-top, top-to-bottom, left-up-to-bottom-right, bottom-right-to-left-up.

5 Tracking

Precise and robust tracking of position is a necessary step in every reconstruction pipeline. However, in practice the computed pose of sensor accumulates small errors over time which result into deviations (drift) from ground truth pose.

Drift is caused by noise in the input data and needs to be accounted for if longer reconstruction sessions are to be held. It can be mitigated by using global tracking techniques (loop closure, bundle adjustment, local maps of magnetic field, etc.) or by incorporating external IMU[8] measurements into tracking algorithm (e.g. as initial guesses of pose).

Trackers try to compute/guess position delta between the current frame captured by sensor and a reference frame. The reference frame can be either the previous frame of the sensor (a.k.a. *frame-to-frame* tracker) or a previous render of the scene (a.k.a. *frame-to-model* tracker).

Frame-to-frame (F2F) trackers suffer from noise, which is present in both the tracked and the reference frame. Due to the noise, F2F trackers are unstable and cannot sustain even short scanning sessions without introducing severe artifacts into the scan and eventually completely failing.

Frame-to-model (F2M) trackers have only one source of noise (the tracked image) — the reference image is rendering of scanned model, which is smoothed-out (by averaging and integrating previous data), which makes the tracker relatively robust to noise and even faster movements of the sensor do not cause failure of tracking. The reconstruction quality is highly dependent on good tracking *and vice versa* in this case.

Trackers can be further divided into two groups; sparse and dense. Sparse trackers use only some parts of the tracked and reference images (keypoint-based tracking), whereas dense trackers utilize all informations (pixels) from the images. This thesis is concerned only with dense tracking [2]. The dense tracking technique used in this thesis is a variant of the Iterative Closest Point (ICP) algorithm [13].

5.1 ICP

ICP is a general algorithm that is used (or rather its variants) when 2 point-clouds[9] need to be aligned and the (usually affine) transformation between them is to be found. ICP is composed of 2 key components (the differences between used components form different ICP variants); the first is *data association technique*, the second *base error metric*;

Imagine 2 point clouds, $\mathcal{A}$ and $\mathcal{B}$, both of which contain only a single vertex. The problem of finding transform between the two clouds is very easy, because we assume that the point from $\mathcal{A}$ corresponds to the point in $\mathcal{B}$. However, when there are multiple points in both clouds, it is necessary to find the corresponding point pairs in $\mathcal{A}$ and $\mathcal{B}$ — this is addressed by different *data association techniques.*

[8]Inertial Measurement Units — those include gyroscopes, accelerometers, magnetometers and barometers.

[9]A set of points, in this case points $p \in \mathbb{R}^3$.

After point correspondences are found, a way of measuring how closely a pair of points aligns is needed — such metric is called *base error metric.*

In this thesis, the frame-to-model type of tracker is used — the reference frame consists of 2 textures: *points map* (2D `float` texture that holds the world positions of rendered points from the scanned model) and *normals map* (2D `float` texture containing normals for points from points map).

5.2 Selecting Data Association Technique

The source of RGBD frames is supposed to be able to operate at least at 30FPS and small inter-frame motion is assumed, which allows using world-pose of the reference image M_r (4×4 homogeneous camera-to-world matrix) as the initial guess M_{t_0} for the pose M_t of the tracked image ${}^tI$ (i.e. the new, tracked frame is assumed to be approximately at the same world position as the old reference image).

After the initial guess of pose M_{t_0} is determined, point associations can be made; for each pixel of the tracked image ${}^tI$ (range image) at coordinate $c \in \mathbb{R}^2$ we wish to find a corresponding pixel coordinate r of the points map texture (so we will be able to measure their alignment). We do it by unprojecting c into a temporary world space point z, and then projecting z onto the image plane of the reference image, obtaining r (followed by optionally sampling the corresponding point from the points map):

$$\begin{aligned}
z &= M_t \begin{bmatrix} {}^tI_c {}^tK^{-1} \begin{bmatrix} c_x \\ c_y \\ 1 \end{bmatrix} \\ 1 \end{bmatrix} \\
g &= {M_r}^{-1} z \\
g' &= {}^rK \begin{bmatrix} g_x & g_y & g_z \end{bmatrix}^T \\
r &= \frac{1}{{g'}_z} \begin{bmatrix} {g'}_x \\ {g'}_y \end{bmatrix}
\end{aligned} \tag{5}$$

where:

${}^tI_c$ = the depth value from the tracked range image ${}^tI$ under pixel c
${}^tK$ = the intrinsic matrix of the tracked range image
${}^rK$ = the intrinsic matrix of the reference's points map

This data association technique is called *projective data association* and has been used in [2] and [14].

5.3 Selecting Base Error Metric

Let $\mathcal{A}$ be the set of points $p \in \mathbb{R}^3$ projected from the tracked range image into its camera-space, and let $\mathcal{B}$ be the set of points $q \in \mathbb{R}^3$ from points map of the reference image.

After we have found the corresponding pairs of points $(p, q)\colon p \in \mathcal{A},\ \ q \in \mathcal{B}$, as described in section 5.2, we need to measure how well do they align with each other — a penalty[10] function. An example of naive penalty function (so called *point-to-point* metric) is $L_n(p, q) = \|M_t p - q\|_2^2$; it computes squared distance between the two points without considering their local features (normals, local curvature, etc.). It was showed it is not suitable for real-world environments, because the data association step can in many cases find incorrect correspondences, which this penalty function does not account for.

When scanning in real-world environments, the scanned objects are in most cases continuous surfaces — this is being exploited by a so called *point-to-plane* metric that, apart from positions of the pair of points, utilizes also the normal ${}^q n$ of the reference point q and measures the squared distance from the tracked (world) point p to the plane defined by ${}^q n$ and q:

$$L\left(p, q, {}^q n\right) = \left(M_t p - q\right) {}^q n \tag{6}$$

5.4 Composing Tracker

The tracker used in this thesis is based on projective data associations and the point-to-plane metric. ICP tries to find a pose delta (pose increment) $\hat{M} \in \mathbb{R}^{4\times 4}$ that after being applied to the current n^{th} tracked frame's pose guess M_{t_n} as $M_{t_{n+1}} = \hat{M} M_{t_n}$ would cause the new $((n+1)^{\text{th}})$ guess to improve alignment of $\mathcal{A}$ and $\mathcal{B}$.

In practice, the pose increment $\hat{M}$ is not computed directly, but rather it is parametrized in some way, because the 4×4 matrix has 16 degrees of freedom (DoF), whereas $SE(3)$ pose, which we are trying to find, has only 6 DoF. The parameters used in this thesis are x, y, z for translation delta and α, β, γ for angle deltas.

$\hat{M}$ is composed of rotation submatrix $R \in SO(3)$ and translation vector $t \in \mathbb{R}^3$. To compute a rotation matrix, the trigonometric functions $\sin(\alpha)$ and $\cos(\alpha)$ are needed. However, usage of trigonometric functions would introduce nonlinearity into formulas, which would complicate derivatives of such expression and add computational burden (trigonometric functions are expensive to evaluate). This is the reason why a linear approximation of $SO(3)$ rotations is used, which for small angles around 0 radians assumes $\sin(\alpha) \approx \alpha$ and $\cos(\alpha) \approx 1$ [15]. This approximation is valid for our purposes, as small inter-frame motion is assumed. The rotation approximation R is derived from the exact $SO(3)$ rotation R' in Equation 7. Right-hand-side coordinate system with x axis pointing east, y axis pointing south and z axis pointing forward

[10] Also called *cost* or *loss* function.

(away from reader) is assumed.

$$\begin{aligned}
{R'}_x &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}, \quad {R'}_y = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}, \quad {R'}_z = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
R' &= {R'}_z {R'}_y {R'}_x = \\
&= \begin{bmatrix} \cos\beta\cos\gamma & \cos\gamma\sin\alpha\sin\beta - \cos\alpha\sin\gamma & \cos\alpha\cos\gamma\sin\beta + \sin\alpha\sin\gamma \\ \cos\beta\sin\gamma & \cos\alpha\cos\gamma + \sin\alpha\sin\beta\sin\gamma & \cos\alpha\sin\beta\sin\gamma - \cos\gamma\sin\alpha \\ -\sin\beta & \cos\beta\sin\alpha & \cos\alpha\cos\beta \end{bmatrix} \\
R' \approx R &= \begin{bmatrix} 1 & -\gamma & \beta \\ \gamma & 1 & -\alpha \\ -\beta & \alpha & 1 \end{bmatrix} \\
\hat{M} &= \begin{bmatrix} 1 & -\gamma & \beta & x \\ \gamma & 1 & -\alpha & y \\ -\beta & \alpha & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{7}$$

To describe how to compute the pose increment $\hat{M}$, the core of the ICP algorithm needs to be introduced; ICP is a dense tracker which means it iterates through all valid point pairs (p, q) (described in section 5.3) and accumulates their point-to-plane errors into a global penalty function $E(x, y, z, \alpha, \beta, \gamma) \in \mathbb{R}^6 \mapsto \mathbb{R}$ that returns value serving as a metric of how well does the pose delta $\hat{M}$ (which is computed from the translation and angle deltas $x, y, z, \alpha, \beta, \gamma$) align the two point clouds $\mathcal{A}$ and $\mathcal{B}$:

$$\begin{aligned}
E(x, y, z, \alpha, \beta, \gamma) &= \sum_{(p,q)} \left(1 - \frac{p_z - R_{min}}{R_{max} - R_{min}}\right) L_\delta \left(\left(\hat{M}(x, y, z, \alpha, \beta, \gamma) M_{t_n} p - q\right)^r n\right) \\
L_\delta(q) &= \begin{cases} q^2, & \text{if } |q| \le \delta \\ 2q\delta - \delta^2, & \text{if } |q| > \delta \end{cases}
\end{aligned} \tag{8}$$

where:

R_{max} = the maximum depth value that a tracker works with (usually 2 meters)
R_{min} = the maximum depth value that a tracker works with (usually 5-10 centimeters)
L_δ = the Huber norm function

Because the reliability of depth sensor measurements decreases with increasing distance of measured surface, every point pair is weighted appropriately; the tracker works only for those points from tracked image, whose depth is in the range $[R_{min};\ R_{max}]$ (valid points) and the weight is determined to be $\left(1 - \frac{p_z - R_{min}}{R_{max} - R_{min}}\right)$. To further robustify the tracker against outliers (caused by invalid data association), the Huber loss L_δ is used instead of the square function $f(x) = x^2$.

To simplify further notation, let us redefine $E(w)$ as a function of vector $w = \Big[x, y, z, \alpha, \beta, \gamma\Big]^T$. To find a guess of $\hat{M}$, we need to minimize the value of $E(w)$ ($\arg\min_w E(w)$), which can be done via Newton-Raphson optimization method, which iteratively minimizes the gradient $\frac{\partial E}{\partial w}(w)$ of $E(w)$ by finding the Δw that minimizes the linear approximation of the gradient:

$$\begin{aligned}
\frac{\partial E}{\partial w}(w + \Delta w) &= \frac{\partial E}{\partial w}(w) + \frac{\partial^2 E}{\partial w^2}(w)\Delta w \\
\frac{\partial E}{\partial w}(w + \Delta w) &= 0 \\
\frac{\partial E}{\partial w}(w) + \frac{\partial^2 E}{\partial w^2}(w)\Delta w &= 0 \\
\frac{\partial^2 E}{\partial w^2}(w)\Delta w &= -\frac{\partial E}{\partial w}(w) \\
\Delta w &= -\left(\frac{\partial^2 E}{\partial w^2}(w)\right)^{-1}\frac{\partial E}{\partial w}(w)
\end{aligned} \tag{9}$$

First, the gradient $\frac{\partial E}{\partial w}(w)$ is derived:

$$
\begin{aligned}
\frac{\partial L_\delta}{\partial k}(k) &= \begin{cases} 2q, & \text{if } |q| \le \delta \\ 2\delta\,\mathrm{sgn}(k), & \text{if } |q| > \delta \end{cases} \\
\frac{\partial E}{\partial w}(w) &= \sum_{(p,q)} \left(1 - \frac{p_z - R_{min}}{R_{max} - R_{min}}\right) \frac{\partial L_\delta}{\partial k} \left(\overbrace{\left(\hat{M}(w) M_{t_n} p - q\right) {}^r n}^{v} \right) \frac{\partial v}{\partial w} \\
\frac{\partial v}{\partial w} &= \partial_w \left(\begin{bmatrix} 1 & -\gamma & \beta & x \\ \gamma & 1 & -\alpha & y \\ -\beta & \alpha & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x \\ s_y \\ s_z \\ 1 \end{bmatrix} \begin{bmatrix} {}^r n_x \\ {}^r n_y \\ {}^r n_z \\ 0 \end{bmatrix} - \overbrace{\begin{bmatrix} q_x \\ q_y \\ q_z \\ 1 \end{bmatrix} \begin{bmatrix} {}^r n_x \\ {}^r n_y \\ {}^r n_z \\ 0 \end{bmatrix}}^{\substack{\text{can be discarded,} \\ \text{does not contain } w}} \right) \\
&= \partial_w \left(\begin{bmatrix} s_x & - & s_y\gamma & + & s_z\beta & + & x \\ s_x\gamma & + & s_y & - & s_z\alpha & + & y \\ -s_x\beta & + & s_y\alpha & + & s_z & + & z \\ 0 & + & 0 & + & 0 & + & 1 \end{bmatrix} \begin{bmatrix} {}^r n_x \\ {}^r n_y \\ {}^r n_z \\ 0 \end{bmatrix} \right) \\
&= \begin{bmatrix} {}^r n_x \\ {}^r n_y \\ {}^r n_z \\ {}^r n_z s_y - {}^r n_y s_z \\ {}^r n_x s_z - {}^r n_z s_x \\ {}^r n_y s_x - {}^r n_x s_y \end{bmatrix}
\end{aligned} \tag{10}
$$

The next step is deriving the hessian $\frac{\partial^2 E}{\partial w^2}(w)$:

$$
\begin{aligned}
\frac{\partial^2 L_\delta}{\partial k^2}(k) &= \begin{cases} 2, & \text{if } |q| \le \delta \\ 0, & \text{if } |q| > \delta \end{cases} \\
\frac{\partial^2 E}{\partial w^2}(w) &= \sum_{(p,q)} \left(1 - \frac{p_z - R_{min}}{R_{max} - R_{min}}\right) \frac{\partial^2 L_\delta}{\partial k^2} \left(\overbrace{\left(\hat{M}(w) M_{t_n} p - q\right) {}^r n}^{v} \right) \frac{\partial v}{\partial w} \left(\frac{\partial v}{\partial w}\right)^T
\end{aligned} \tag{11}
$$

We use $w = \vec{0}$ as the initial guess. Now we are able to compute Δw and from it the pose increment $\hat{M}$ that we use to update the current pose guess M_{t_n} to $M_{t_{n+1}} = \hat{M} M_{t_n}$. We refine the pose guess in this way in several iterations until the penalty function is sufficiently low or perform fixed number of iteration regardless penalty function value. The last computed guess is assigned as the final pose of the tracked frame: $M_t = M_{t_{last}}$. For further details about the concrete tracker implementation of this thesis, refer to section 7.4.1.

6 Brief Overview of Used iOS Technologies

Before describing the implementation details of the implemented application, a few terms and technologies need to be clarified. The reconstruction pipeline relies on the Apple's TrueDepth camera (a Structured Light sensor) which is, in the time of writing of this thesis, contained only in the iPhone X that runs on the iOS operating system. Applications for iOS can be written several programming languages; C/C++, Objective-C/Objective-C++, Metal Shading Language, and Swift (all languages are compiled). The CPU portion of the reconstruction pipeline is written in Swift, meanwhile the GPU portion in Metal Shading Language.

6.1 Metal

Before Metal [16] was introduced in 2014, developers would use OpenGL ES for programming graphics rendering tasks on GPU. Metal allowed to utilize the GPU not only for image-rendering, but also for GPGPU tasks (like CUDA) — that makes Metal comparable to Vulkan and Microsoft's DirectX.

Metal composes of for two parts:

- Metal Shading Language — that is used to program fragment, vertex and compute-kernel functions that run on GPU, it's a variant of C++14,
- Metal framework — that runs on CPU and provides API for GPU memory management (e.g. managing creation of buffers, textures, heaps), function compilation and command execution scheduling (execution of GPU functions).

Metal is available for both macOS and iOS. However, unlike on macOS, Metal on the iOS platform is based on the *shared memory* model — the RAM and VRAM memories are accessible both from CPU and GPU, meaning there is no need for expensive memory copying between "host" and "device" (like in CUDA) — the CPU can access GPU buffers without any overhead.

Moreover it supports atomic operations on integers and recently obtained support for *quad-group* shuffle instructions[11] for devices with the Apple A11 chip (in the time of writing of this thesis only on the iPhone 8/Plus and iPhone X).

6.2 Grand Central Dispatch (GCD)

GCD is a framework which provides abstraction model for parallelism and concurrency (it is based on POSIX threads). The core concept of GCD are *queues*. Code can be submitted into a queue and depending on the type of the queue, the code is executed either serially (in the order it was submitted into the queue), or concurrently (but not necessarily parallelly) — that applies for *serial* and *concurrent* queues respectively.

[11] Those are similar to SIMD shuffle instructions of CUDA, but instead of operating on the whole warp (32 consecutive threads in CUDA), they operate only on 4 consecutive threads (hence *quad*-group).

A queue also belongs to one of the QoS groups: User Interactive, User Initiated, Utility and Background, which defines its priority. The first two groups are intended for real-time execution, meanwhile the last two for long-running tasks.

6.3 ARKit

This is the Augumented Reality framework that Apple introduced in 2017. It is a visual-inertial odometry solution that among other features provides estimation of global position of device at scale (in meters) and the intrinsic matrix K of the camera. This enables easy rendering of overlay content (e.g. 3D models of objects).

7 Selected Reconstruction Pipeline

Before settling down for the final solution, I implemented a photogrammetry approach to reconstruction that relied on Apple's ARKit 1.0 framework for tracking pose[12], [9] for rectifying a pair of RGB images (see section 4.1) and Semi-Global Matching (see [10] for further details) for computing disparity from the rectified pair. The computed disparity images were transformed into depth images that were fused (similarly to [2]), into a $512 \times 512 \times 512$ 3D texture on GPU containing TSDF values, weights and colors. The results were not good (no details of scanned surface could be captured) — primarily because of ARKit pose drifting and also due to noisy depthmaps (caused mainly by incorrect rectification due to imprecise pose guesses). This approach was draining the phone battery fast and caused rapid heating. It was clear it could not be used for casual reconstructions.

Given the target of high-resolution reconstruction at real-time or interactive rates, I decided to leverage the TrueDepth camera[13] of the new Apple iPhone X to obtain accurate depth measurements.

7.1 Application Capabilities

The application supports scanning a scene at a fixed resolution (this is the reconstruction pipeline), polygonising the saved scan into PLY [17] file format and finally displaying the polygonalized model "in Artifical Reality" via ARKit. A user can also download the exported PLY model to a computer via iTunes File Sharing. The application GUI is kept simple as can be seen on the Figure 2.

7.2 Existing RGBD Reconstruction Pipelines

Before deciding on the final algorithm to be used for 3D reconstruction, an overview of papers concerned with real-time scanning is given. The terms described in the following sections are properly explained later in this section.

7.2.1 KinectFustion

The most popular pipeline is KinectFusion [2], which uses an ICP variant with point-to-plane metric and projective data association (see section 5.1) for frame-to-model tracking and fuses live stream of range images into a 3D texture containing TSDF, weight and color data. The system is implemented on GPU. Given the limited GPU memory and constraints of 3D texture size, it is typically used with 512^3 grid of voxels, to be able to run at realtime rates (over 30FPS) on commodity GPUs. This inherently creates trade-off between size of scanned area and resolution.

[12]Details of algorithms used by ARKit 1.0 are not publicly known, nevertheless it is known to use bundle adjustment, IMU measurements and dead-reckoning. It seems to run only on CPU.

[13]Which is a Structured Light range sensor, similar to Kinect v1.

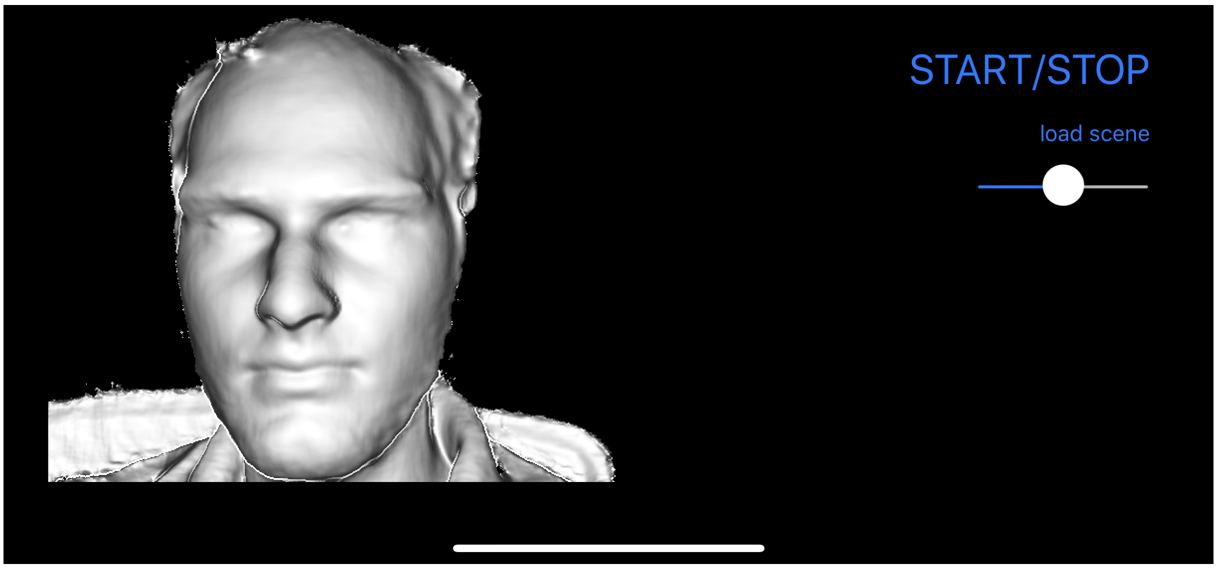


Figure 2: The application GUI during live scanning.

7.2.2 Real-time 3D Reconstruction at Scale using Voxel Hashing

The [18] paper proposes solution to the KinectFusion's limited scanning area problem. It uses the same tracker, but instead of saving the global model to a static, ineffective 3D texture, it stores the TSDF, weight and color data into chunks of 8^3 voxels, called voxel-blocks. Voxel-blocks are created only in the truncation range of a given pixel from range image, which creates a sparse structure, and are stored on a "heap" (ordinary GPU buffer). The heap, containing voxel-blocks, is managed by a small, lightweigth *hashtable*, which is also a GPU buffer.

The hashtable stores metadata about voxel-blocks — when a voxel-block is about to be saved, its world (integer) position $p = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix}$ is hashed by a special hash function (see section 7.5.4) and its output $h = \text{hash}(p)$ is used as index to the hashtable GPU buffer. On a specified index, there are stored metadata about voxel-block it holds: the world position of voxel-block, pointer into heap containing TSDF, weight and color data of the voxel-block, n (typically 3) cells for combating potential overflow and an index into an overflow linked list (in case all 1+3 cells are occupied due to overflow).

7.2.3 Very High Frame Rate Volumetric Integration of Depth Images on Mobile Devices

The previous paper used a complex system for handling parallel creation of new voxel-blocks that *prevented* race conditions (which could happen when multiple threads tried to insert voxel-blocks with colliding (the same) hashes of their position into the hashtable) that included using locks in GPU kernels and ensuring that every voxel-block was added in single iteration. This way of handling collisions, together with a hashtable that included 3 more positions at single index for saving collided voxel-blocks was causing bottleneck.

Authors of [14] came with a solution for this bottleneck. They decided to *embrace* the collisions instead of explicitly handling them; the situation when multiple threads try to insert voxel-blocks to an identical place in the list of voxel-blocks-to-allocate-in-current-frame is not explicitly handled; the last thread to write into the location is the "winner" and its values are preserved in the list under the collided key. That makes this approach not entirely deterministic[14] and hard to conduct reliable tests.

It also means that not all voxel-blocks are guaranteed to be allocated in a single iteration of the algorithm — if there are multiple voxel-blocks with the same hashed key, only one of them is going to be allocated. However, this is not a problem in real-life, high frame rate environment, as the missing voxel-blocks are allocated the next iteration of the algorithm.

7.3 Selecting Reconstruction Pipeline

Given the real-time reconstruction constraint, I decided to implement the [14] paper. This paper has been implemented in the open-source library InfiniTAM [13] that served as an inspiration for my implementation (the [14] paper does not contain all necessary information that would suffice for implementation).

The pipeline of [13] assumes that processing of a single frame is done within the sampling rate of the range image data source (e.g. live camera feed or an offline dataset saved on disk). Have the current frame not been processed before a new frame (i.e. range image) arrives, a delay between physical scanner movements and currently processed (and displayed to the user) frame would start to accumulate. Not only would this result into bad user experience, it would also consume all available device resources (GPU, RAM) continuously without a chance to idle (and save battery), moreover the old range images would queue-up in memory (because the data source would pass current range image to the callback routine), causing depletion of RAM, tracker failure and eventually application crash. To prevent this issue, I decided to parallelize tracking and reconstruction (processing of range image) by running the two tasks in parallel in 2 GCD serial queues.

7.4 Tracker Queue

The tracker queue is implemented as a serial GCD queue. It is the entry point of the reconstruction pipeline. A data source (live TrueDepth camera feed or disk) calls the reconstruction pipeline's callback `processFrame()` *on this queue* when a new RGBD frame is ready. The callback function reads the current RGBD data from the data source, immediately calls the frame-to-model tracker to process the new frame and blocks until the tracker finishes — note that it is assumed that tracker finishes before the data source has the next RGBD frame ready.

[14]The reason of this non-deterministic behavior is that the order, in which the GPU executes thread-blocks (and therefore memory-write instructions) is not always deterministic and depends on the implementation of the GPU warp scheduler (to borrow CUDA terminology).

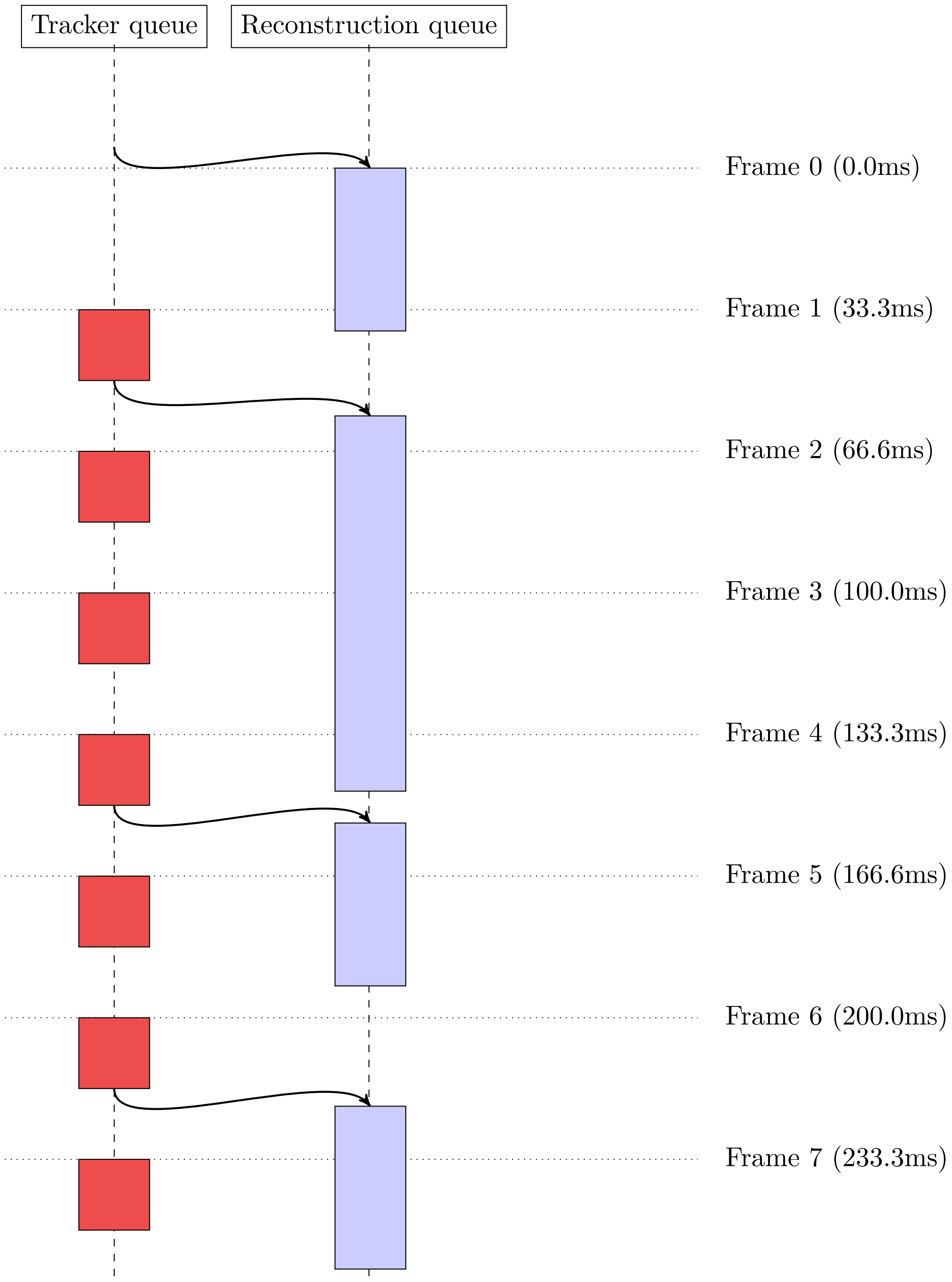


Figure 3: Reconstruction pipeline overview. The two GCD serial queues are portrayed. On the Tracker queue, the newly processed RGBD frames are passed, which are tracked in *every* frame. When the Reconstruction pipeline is idle, the Tracker queue schedules reconstruction work to it. The processing time of one frame by the Reconstruction queue is variable due to different number of visible voxels in every view.

After the tracker computed a new world-pose (for the current RGBD frame), and if no previous frame is being processed in the Reconstruction queue, the current RGBD frame (RGB image with accompanying range image), together with the newly computed world-pose, is asynchronously passed to the `addDepthmap()` method *on the Reconstruction queue*, which ensures the reconstruction step (that accounts for majority of the time spent by processing RGBD frame) does not block the tracker queue, so it can be ready for processing the next RGBD frame (see the diagram in Figure 3).

The tracker computes pose guess for *every* new frame — even though the current frame cannot be processed by the Reconstruction queue, because a previous frame is still being processed and has not ended processing yet. This is useful because the currently computed pose is used as the initial guess of pose for the next RGBD frame. For live reconstruction, the TrueDepth camera is used and is configured to stream 640×480px RGBD images at 30FPS. The Algorithm 1 shows the pseudocode of the tracker queue.

Algorithm 1 Pseudocode for the reconstruction pipeline's `processFrame()` callback.

```
1: function PROCESSFRAME(intrinsicMatrix)
2:     frame = dataSource.getCurrentRGBDFrame()
3:     newPosition = tracker.computeNewPosition(frame, newPosition)
4:     if not isFrameBeingProcessed then
5:         isFrameBeingProcessed = true
6:         model.addDepthImage(frame, newPosition, intrinsicMatrix)    ▷ Run asynchronously
7:     end if
8: end function
```

7.4.1 Tracker Configuration

The used frame-to-model tracker uses point-to-plane metric with projective data association and robust Huber norm in the cost function with depth-based weighting (see section 5.1 for more details). First, the tracker creates 4-level image pyramid from the tracked image (640×480, 320×240, 160×120 and 80×60px). However, because the tracked range image is noisy, only the top 3 (smallest) levels are used by the tracker (it was empirically determined that using all 4 levels of the pyramid caused the tracker to fail more often, than using only the top 3 levels).

The next step is performing 2 Newton-Raphson optimization iterations[15] for each level of the image pyramid, starting from the smallest image — the world-position of the previously tracked frame is used as the initial guess.

Because small inter-frame movement is assumed, and to increase robustness of the tracker, rotation-only optimization is performed for the top 2 smallest images. After the rotation guess is

[15]By observing the effect of different number of iterations, two steps showed to be sufficient for tracking quality. Moreover it was also observed that the residual error of even and odd iterations was converging to 2 (slightly) different values. I suspect the parameters are gradually bouncing to the bottom of a local convex hull of the cost function.

computed, the tracker optimizes for both rotation and translation (with the rotation computed in previous steps as the initial guess) in the next pyramid level (320×240px).

The tracker uses the rendered scanned model as its reference image (i.e. rendered points map texture (range image projected into global space) and normal map generated from the points map). Note that the same rendered model image can be used for several frames (usually 1-3 frames) when the Reconstruction queue has not finished processing a new frame yet. After a new range image has been processed by the Reconstruction queue (which includes rendering of the global model), the newly rendered model's points map is *copied* to the tracker (because the reconstruction stage can write data to the original texture in the Reconstruction queue in parallel with the Tracker queue). The resolution of the rendered reference image is 320×240px (to speed-up the rendering stage).

7.5 Reconstruction Queue

The reconstruction stage takes range image and its world-pose (the camera-to-world matrix) M with intrinsic matrix K as the inputs and refines the scanned model. It runs on GPU by utilizing Metal compute kernels (similar to CUDA kernels). The scanned model is internally represented as a Truncated Signed Distance Field (TSDF) with the raw data stored in an array of structures called *voxel-blocks* which are stored sparsely and managed by a lightweight hashtable with excess list (all terms are explained in the subsequent sections).

7.5.1 Signed Distance Fields

Signed Distance Fields (SDF) are used to define/represent geometry in implicit manner as opposed to the classical explicit approach of polygonal geometry. A 3D Signed Distance Field is a function $s\colon \mathbb{R}^3 \mapsto \mathbb{R}$ that for an arbitrary point $p = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix}$ defines *the shortest* distance to the surface:

$$s(p) = \begin{cases} > 0, & \text{if } p \text{ is outside of every object} \\ = 0, & \text{if } p \text{ is exactly on the surface of an object} \\ < 0, & \text{if } p \text{ is inside of an object} \end{cases} \tag{12}$$

For example, the SDF function for a ball with radius $r \in \mathbb{R}$ located at position $q \in \mathbb{R}^3$ could be written as $s(p) = \|p - q\|_2 - r$. Note that meanwhile polygonal model can express only surfaces, SDFs on the other hand allow to also express volume (due to the fact they are signed). SDFs are commonly used to express surfaces of implicit functions and complex shapes (such as fractals). They have many advantages; it is simple, for example, to compute normals in arbitrary point. In practice, normals are computed via central differences by the function $\hat{n}_s(p)$ (using a small delta step $\epsilon \in \mathbb{R}$):

$$
\begin{aligned}
\epsilon_x &= \begin{bmatrix}\epsilon & 0 & 0\end{bmatrix}^T, \quad \epsilon_y = \begin{bmatrix}0 & \epsilon & 0\end{bmatrix}^T, \quad \epsilon_z = \begin{bmatrix}0 & 0 & \epsilon\end{bmatrix}^T \\
n_s(p) &= \frac{1}{2\epsilon}\begin{bmatrix} s\left(p+\epsilon_x\right) - s\left(p-\epsilon_x\right) \\ s\left(p+\epsilon_y\right) - s\left(p-\epsilon_y\right) \\ s\left(p+\epsilon_z\right) - s\left(p-\epsilon_z\right)\end{bmatrix} \\
\hat{n}_s(p) &= \frac{n_s(p)}{\|n_s(p)\|_2}
\end{aligned}
\tag{13}
$$

When the SDF function is not or cannot be defined analytically (i.e. when there is no continuous definition of the function) — like in the case of MRI scans or 3D reconstruction —, it needs to be sampled only at some points in space — usually at the centers of cubes called *voxels* that form a 3D grid in space (the higher the resolution of the grid (the smaller the edge of one cube), the more details can be captured). Most of the time it is needed to sample SDF values at points that are not exactly at the centers of the cubes — in that case the SDF value is interpolated from neighbor voxels via tri-linear interpolation. For the rest of this thesis, only SDFs that are defined via this quantization on voxels are assumed (i.e. not analytical functions).

7.5.2 Truncated Signed Distance Fields

In the problem of 3D reconstruction, one of the requirements for the underlying data structure that holds the scanned model is the ability to quickly update it. Althought SDFs are implicitly defined and thus suitable for such a task, they are defined "globally" — when we want to add a new object to the "scene", it is <u>*not*</u> sufficient to update only the SDF values in a local neighborhood of the surface! Instead, the whole space of the modelled function (i.e. all of the voxels) needs to be traversed and updated in the places, where the newly added object would change the function value. This is not viable in practice, where there are tight constraints on frame-processing time budget.

It is the reason, why a relaxed, local variant of SDFs was introduced; Truncated Signed Distance Fields (TSDF) that only require the SDF function $s(p)$ to define its exact value in a predefined distance — the *truncation region* $\xi \in \mathbb{R}$ — from the surface. The truncation region needs to be greater than the voxels edge size (in practice usually 4-10$\times$). The TSDF function $s_t(p)$ is defined in the following way:

$$
s_t(p) = \begin{cases} \frac{s(p)}{\xi}, & \text{if } p \text{ is within the truncation region from the surface} \\ 1, & \text{otherwise} \end{cases}
\tag{14}
$$

7.5.3 Updating TSDF Model from RGBD Image

Now that it is clear a 3D model can be represented by a TSDF function sampled on regular 3D grid of voxels, we need a data structure to hold the TSDF function's samples. A simple, non-scalable, naive way is using a 3D texture[16] of `floats` on GPU, like the original KinectFusion [2] did.

In practice a TSDF model (e.g. the 3D texture) should be updated by a stream of range images; apart from the current range image R, the intrinsic matrix K and world-pose M (camera-to-world matrix) of the image are also going to be needed to update the TSDF.

In a parallel implementation on GPU, a compute kernel is run in such way that one thread corresponds to one voxel. A thread takes the homogenous world position of its assigned voxel's center p, transforms it to the frame (i.e. camera) coordinate system via the world-to-camera matrix (the inverse of M) — into ${}^cp$ — and finds the pixel coordinate c on the range image R, into which the voxel's center p was projected (voxels that are located behind the camera image plane will project with negative homogeneous z-coordinate and are discarded — TSDF values will not be updated for such voxels). The depth value R_c from that pixel is read back and compared with depth value ${}^cp_z$ of the voxel's center in the camera coordinate space:

$$\begin{aligned}
p &= \begin{bmatrix} p_x & p_y & p_z & 1 \end{bmatrix}^T \\
{}^cp &= M^{-1}p \\
c' &= K \begin{bmatrix} {}^cp_x & {}^cp_y & {}^cp_z \end{bmatrix}^T \\
c &= \begin{cases} \text{invalid}, & \text{when } c'_z \leq 0 \text{, point is behind the camera plane} \\ \frac{1}{c'_z} \begin{bmatrix} c'_x \\ c'_y \end{bmatrix}, & \text{when } c'_z > 0 \text{, point is in front of the camera plane} \end{cases}
\end{aligned} \tag{15}$$

When p projects in front of the camera image plane, the voxel's TSDF value ${}^pv_{tsdf}$ can be updated by simple averaging; note that to perform averaging it is needed to keep the number of samples per voxel as the *weight* ${}^pv_w \in \mathbb{R}$ stored together with TSDF value. The distance $d = {}^cp_z - p_z$ between p and ${}^cp_z$ is measured and the new TSDF value is obtained as $s_t(d)$ (see Equation 14). Voxel's values are then updated in the following way (in that order):

$$\begin{aligned}
{}^pv_{tsdf} &:= {}^pv_w {}^pv_{tsdf} + s_t\left({}^cp_z - p_z\right) \\
{}^pv_w &:= {}^pv_w + 1
\end{aligned} \tag{16}$$

[16]A 3D texture is used instead of a simple array, because on GPU, textures do have optimized access to texels — pixels that are close in space are also stored in close proximity in memory — this is commonly achieved by using Z-order curves (or ideally Hilbert curves) as a hashing functions to compute index of texel in memory based on the 3D location of the texel. These techniques can be also implemented in software for developer-defined 1D arrays, however the GPU has this feature implemented in hardware and it is thus faster to leverage textures.

Table 1: Contents of the `htbl_meta_t` structure.

PROPERTY	DATATYPE	USAGE
position	`int3`	position of the voxel-block
heap_idx	`int`	offset into the heap where the voxel-block data is stored
next	`htbl_meta_t*`	pointer to the next voxel-block with collided hashkey

When scanning, we usually want to capture not only the 3D structure, but often also color information. This is done by storing additional variable v_c per voxel for color. It is updated either by averaging (like in the case of TSDF) or by setting new value of color each time the voxel is processed (due to imprecision in scanning, the color averaging approach often results in blurry textures). The color is sampled from the RGB part of RGBD image — from the pixel at location c.

7.5.4 Hashtable-based Saving of Voxel-Blocks

Empirical observations reveal that saving TSDF data into a static 3D texture is wasteful, as most of the space is unutilized (free space). The paper [14] proposes a sparse data structure that enables voxels to be saved only in near neighborhood of the scanned surface, thus saving space.

Voxels cannot be allocated individually, but rather in $8 \times 8 \times 8$ blocks called *voxel-blocks* that are stored on *heap* (an ordinary GPU buffer/array). Voxel-blocks' data include TSDF values, weights and colors and are stored in 3 separate buffers.

Metadata about allocated voxel-blocks are saved and managed by a lightweight hashtable structure of a fixed size H_s (usually in order of hundreds of thousands elements), where every cell in the hashtable contains a `htbl_meta_t` structure. See Table 1 for overview of the `htbl_meta_t` structure.

Each voxel-block has associated with it a non-unique key via the hashing function $h(q) = (q_x 73856093) \oplus (q_y 19349669) \oplus (q_z 83492791) \bmod H_s$ that generates hash from the global integer position q of voxel-block and serves as the index into the hashtable. Due to a limited size of the hashtable, collisions are inevitable.

This is solved by appending an unordered excess array of `htbl_meta_t` structures at the end of the hashtable and adding a new property to `htbl_meta_t` — a pointer to another `htbl_meta_t` — so that voxel-blocks with colliding keys can form chains/linked-lists (see Figure 4). The excess (collided) voxel-blocks's metadata are stored in the excess array.

When a new voxel-block at position q needs to be allocated, first its hash-key is computed via $k = h(q)$. The hash is then used as the index to the hashtable, and if the cell is empty, metadata at that cell are updated. In the case the cell already contains a record, it means a collision occurred; if the pointer property `next` of the cell points to another cell, we follow it; we follow the chain until the pointer `next` is empty (`null`). Then a new cell from the excess

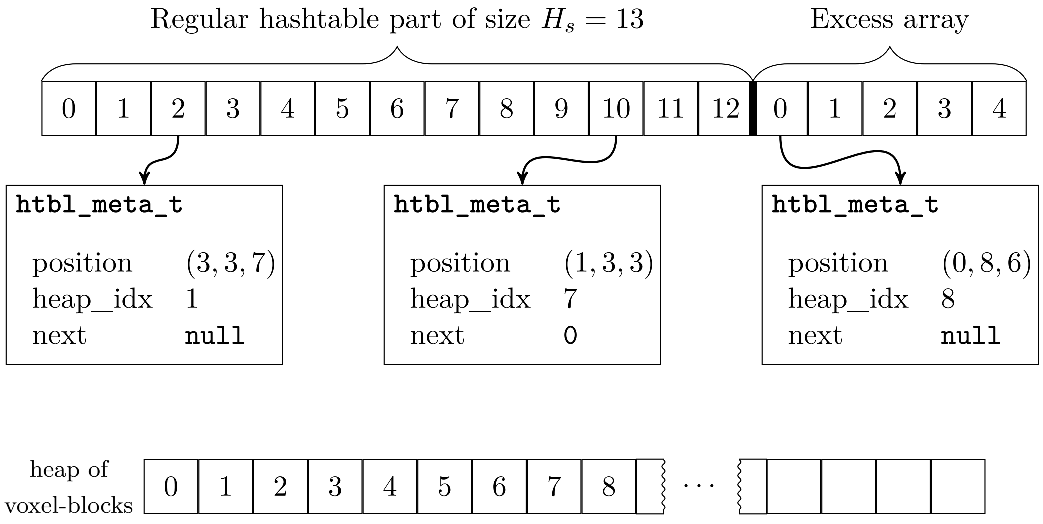


Figure 4: The hashtable structure (on top) with the heap of voxel-blocks. Note that hashtable entry at position 10 contains pointer to a voxel-block with the same hash key. The collided voxel-block is stored in the excess array.

array is reserved for our block (by incrementing the atomic variable holding index of the next free excess cell) and metadata can be inserted. Finally, we reserve the storage for voxel-block in the heap (also by incrementing the atomic variable holding the next free pointer on heap).

7.5.5 Reconstruction Stage

It was described how to update a single voxel from RGBD frame data and how voxel-blocks are allocated. Now it is needed to fuse the two steps together into a single pipeline. The application's reconstruction stage follows these steps:

1. mark visible voxel-blocks that are already allocated and mark unallocated voxel-blocks,
2. allocate voxel-blocks that are marked as unallocated and mark them as visible,
3. perform visibility check for all voxel-blocks marked as visible and unmark invisible voxel-blocks,
4. update TSDF values for all voxels in each visible voxel-block (like in section 7.5.3),
5. render the model (render points map, normals map and colors map).

Before describing the above steps, it is needed to say the reconstruction runs on GPU and uses the buffers described in Table 2.

Step 1 All voxel-blocks can be flagged as either visible of invisible. The visible ones are updated in Step 4. When processing a new range-image, we need to determine, which voxels are going to be updated. And because we cannot allocate single voxels, we select whole voxel-blocks

Table 2: GPU buffers used by the reconstruction step.

BUFFER	DATATYPE	SIZE	USAGE
`visibility`	`bool`	htbl+ex[†]	flag determining whether the voxel-block is visible
`visible_vbs`	`int32`	htbl+ex	reduced visibility buffer (used when integrating/updating TSFD)
`num_visible`	`int32`	1	size of the reduced array (number of elements in visible_vbs)
`nxt_vba_idx`	`int32`	1	the next free index in the heap (atomic)
`nxt_excess_idx`	`int32`	1	the next free index in the excess array of hashtable (atomic)
`vbs_to_alloc`	`int32`	htbl[‡]	positions of voxel-blocks marked for allocation
`htbl_pos`	`int3`	htbl+ex	part of hashtable; position of voxel-blocks
`htbl_offset`	`int32`	htbl+ex	part of hashtable; index into excess array (for excess voxel-blocks)
`htbl_vbs`	`int32`	htbl+ex	part of hashtable; index into heap (for TSDF data)
`vbs_tsdf`	`fp16`	htbl+ex	part of heap; TSDF values
`vbs_weight`	`fp16`	htbl+ex	part of heap; weight of measurements
`vbs_color`	`uchar4`	htbl+ex	part of heap; color

[†] Size of the regular hashtable together with excess array.
[‡] Size of only the regular hashtable.

instead. The goal is to update/allocate only those voxel-blocks that are within the truncation range ξ of the surface defined by the range image. Such voxel-blocks are determined by traversing the truncation range for each pixel of the range image (1 pixel = 1 GPU thread). Truncation range $[{}^{s}r;\ {}^{e}r]$ for a pixel c is defined as a line segment on the pixel's ray ${}^{c}\hat{p}$ centered around the projected pixel ${}^{c}p$ and stretching ξ units into both positive and negative direction of the ray (K is intrinsic matrix, M is camera-to-world matrix, ξ is the truncation length);

$$\begin{aligned} {}^{c}p &= {}^{c}RK^{-1}\begin{bmatrix} {}^{c}p_x & {}^{c}p_y & 1 \end{bmatrix}^T \\ {}^{c}\hat{p} &= \frac{{}^{c}p}{\|{}^{c}p\|_2} \\ {}^{s}r &= M\begin{bmatrix} {}^{c}p - \xi {}^{c}\hat{p} & 1 \end{bmatrix}^T \\ {}^{e}r &= M\begin{bmatrix} {}^{c}p + \xi {}^{c}\hat{p} & 1 \end{bmatrix}^T \end{aligned} \tag{17}$$

Each voxel-block on that line segment is either marked as visible (in the `visibility` buffer), if it has already been allocated, and if it have not been allocated yet, the position of the voxel-block is saved under its hash-key in the `vbs_to_alloc` buffer (multiple threads can write to the same place — this is expected; only the last thread to write has its values saved).

Step 2 The `vbs_to_alloc` buffer is traversed (1 buffer index = 1 GPU thread) and all marked voxel-blocks (i.e. their positions) are allocated (as described in section 7.5.4). After a voxelblock gets allocated, it is marked as visible in the visibility buffer.

Step 3 The `visibility` buffer is not cleared at each new frame — instead it is preserved between frames, which is the reason why all visible voxel-blocks need to be checked if those are visible also in the current view. This is done by projecting all 8 corners of the tested voxel-block into the image-plane and if at least one of the corners is visible (i.e. within bounds of the image-plane), it is considered visible. In the opposite case, it is unmarked as visible in the visibility buffer.

Step 4 All visible voxel-blocks from the visibility array are collected and reduced into the `visible_vbs` buffer (the number of visible voxel-blocks is saved into `num_visible`). Then the visible voxel-blocks are updated — 1 voxel per 1 GPU thread (i.e. 1 voxel-block per 1 threadgroup/block run as $8 \times 8 \times 8$ block of threads (in CUDA terminology)). Voxels are updated as described in section 7.5.3.

Step 5 After the model has been updated, it is rendered. Output of the renderer are 3 `float` textures; points map (holds the world positions of points on the visible surface from the current view), normals map (holds normals for the points from points map) and color map (holds colors of the points). Rendering is accomplished as raymarching in a limited rendering (depth) range (say $[0;\ 2]$ meters). Each pixel of the output texture is processed by 1 GPU thread. The rendering compute kernel computes only points map and color map (a ray corresponding to a pixel is marched until it hits a surface (TSDF value < 0) or until it is within the rendering range) — the normals map is derived from the points map via central differences in screen-space. Rendering of the user preview from the 3 textures is done in separate rendering pipeline; Phong shading model is used. After rendering has finished, the pipeline gets notified and the newly rendered texture trio is copied into the tracker (together with the pose and intrinsic matrix) — semaphore is used to ensure atomicity of copying.

7.6 Polygonizing Scanned TSDF (Exporting into Polygonal Formats)

After scanning a model, it is often wanted to share it with other devices and edit it on a desktop computer. The most common way of sharing models are polygonal file formats, therefore a polygonalization technique for TSDF is required.

Note that our TSDF is a scalar function sampled on grid (voxel-blocks) with model surface at the 0 isolevel. The most common way of polygonising such functions is to use the technique called Marching Cubes [19] that polygonises a scalar function sampled on the *corners* of voxels.

The procedure starts by defining a function value that will serve as the isoline. Marching cubes provides means of generating triangles from a single voxel (a cube) with function values sampled at its corners. To fully polygonise the scanned model means polygonising all of its voxel-blocks. To polygonise a voxel-block, it is not sufficient to load only that voxel-block — voxels of our TSDF model have the function values sampled in the *centers* of its voxels, meanwhile

Marching cubes (MC) requires corners as the sampling points — a $2 \times 2 \times 2$ block of voxels from a voxel-block can be polygonised by MC.

Note that to polygonise a single voxel-block, we need additional function samples at the boundaries (in all 6 directions) from the neighboring voxel-blocks. However if all voxel-blocks of the scan are to be polygonised, only data from 3 neighboring directions are needed to be loaded.

7.6.1 Polygonising a Single MC Voxel

A MC voxel is a cube with function samples at all 8 corners. The goal is to create several triangles *within* the cube that would best describe the geometry of surface contained within the cube. The surface level is given by the isoline — when the function value at a point is less that the isoline value, the point is considered to lie *inside* the object. When the function value is greater than or equal to the isoline value, the point is on the other hand assumed to be outside the object to be polygonised.

Given that there are 8 corners and a corner can be in 2 states — either inside or outside the object —, there exist 2^8 possible configurations of points and accompanying triangle meshes. In practice tables with all 256 hardcoded triangle configurations are used [19]. The binary information whether a corner is or is not inside an object is represented by a single bit for each of the 8 corners — this creates 1-byte index into the lookup table with triangle configurations.

The exporter is written in Objective-C++ and it is not constrained by the size of the input TSDF model and also is not RAM-bounded — voxel-blocks (and the 7 neighbors of each one) are loaded in small batches of up to 10 000 voxel-blocks, until all voxel-blocks of the scanned model are polygonised. GCD was used to parallelize loading of voxel-blocks from disk.

7.7 Displaying the Polygonised Model in AR

After the model is exported, it can be displayed in AR. World-pose tracking is handled by Apple's ARKit framework and rendering is done via Apple's SceneKit rendering framework (by manually importing the PLY model). Example of viewing a scanned model in AR can be sen on Figure 5.

7.8 Used Constants

The reconstruction pipeline requires determining constants that, among others, influence reconstruction quality and the maximum size of the scanned model. Overview can be seen in Table 3. Values were chosen experimentally.

7.9 Configuration Comparison and Performance Overview

Separation of the application into two parallel queues allowed real-time reconstruction even at high resolutions. The tracker queue, which processes stream of RGBD images, is able to be run

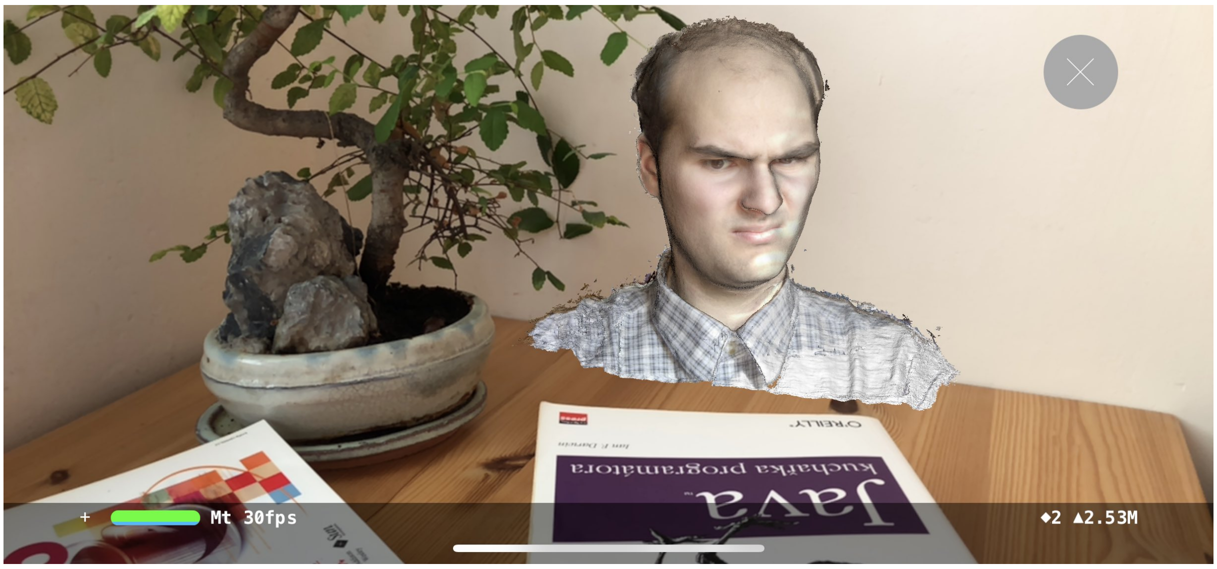


Figure 5: The application during preview of scanned and exported model.

at 30FPS assuming the constants for tracker from Table 3 are set. The influence of reconstruction resolution on pipeline duration times can be seen in figures 7 and 8. The accompanying reconstruction results can be seen in Figure 6. All benchmarks were done on the same dataset.

Table 3: Used pipeline constants.

NAME	VALUE	DESCRIPTION
R_{min}	$0.1m$	The start of allowed depth range.
R_{max}	$1.5m$	The end of allowed depth range.
${}^rI_s$	320×240px	The size of the rendered reference image (points-, normals- and color-map.).
${}^tI_s$	640×480px	The size of the tracked reference image (from live camera feed.).
H_s	120 000	The size of the pure part of the hashtable.
H_e	95 000	The size of the excess array of the hashtable.
—	$320 \times 240, 160 \times 120, 80 \times 60$px	Used tracker pyramidal levels (for the tracked frame).
—	2	Number of iterations per pyramidal level.

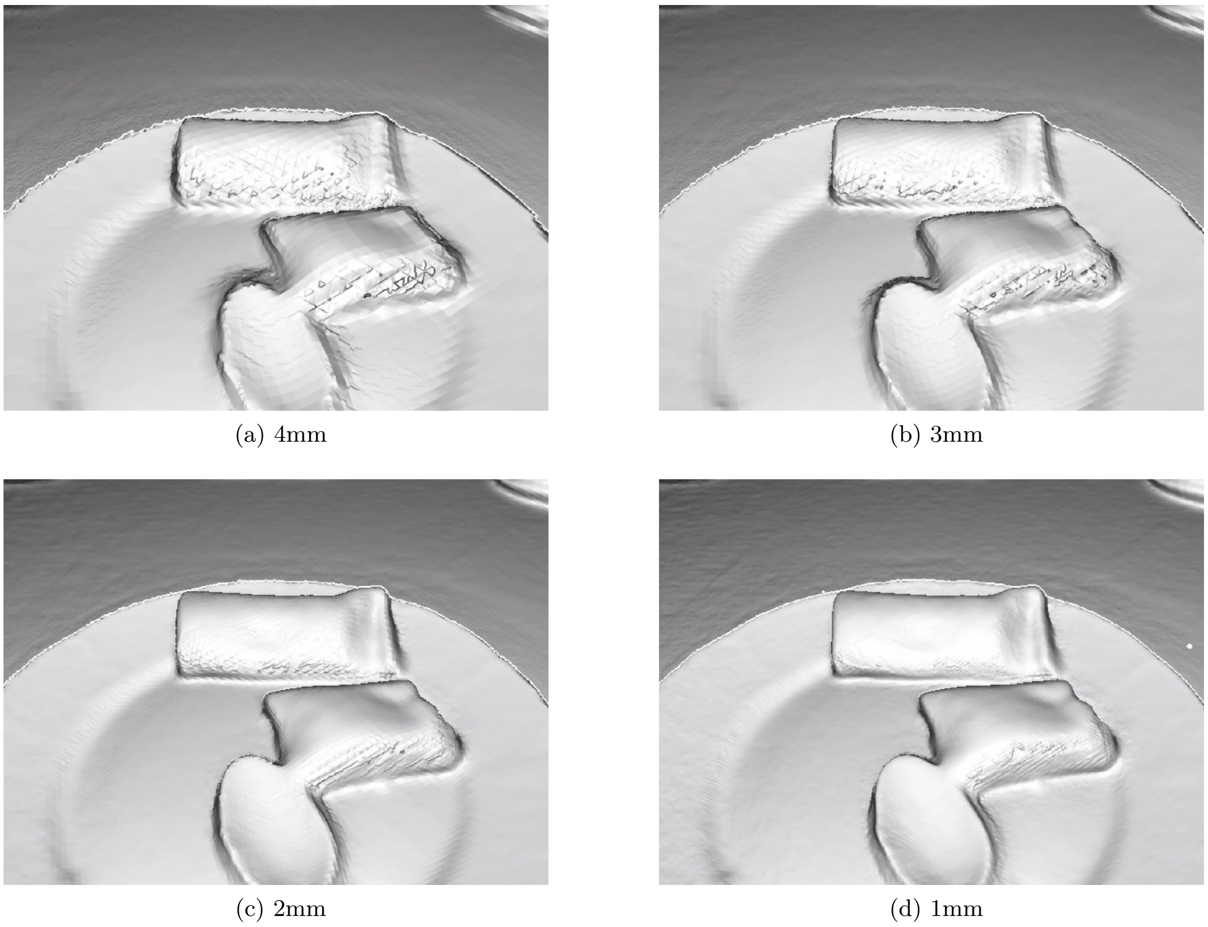
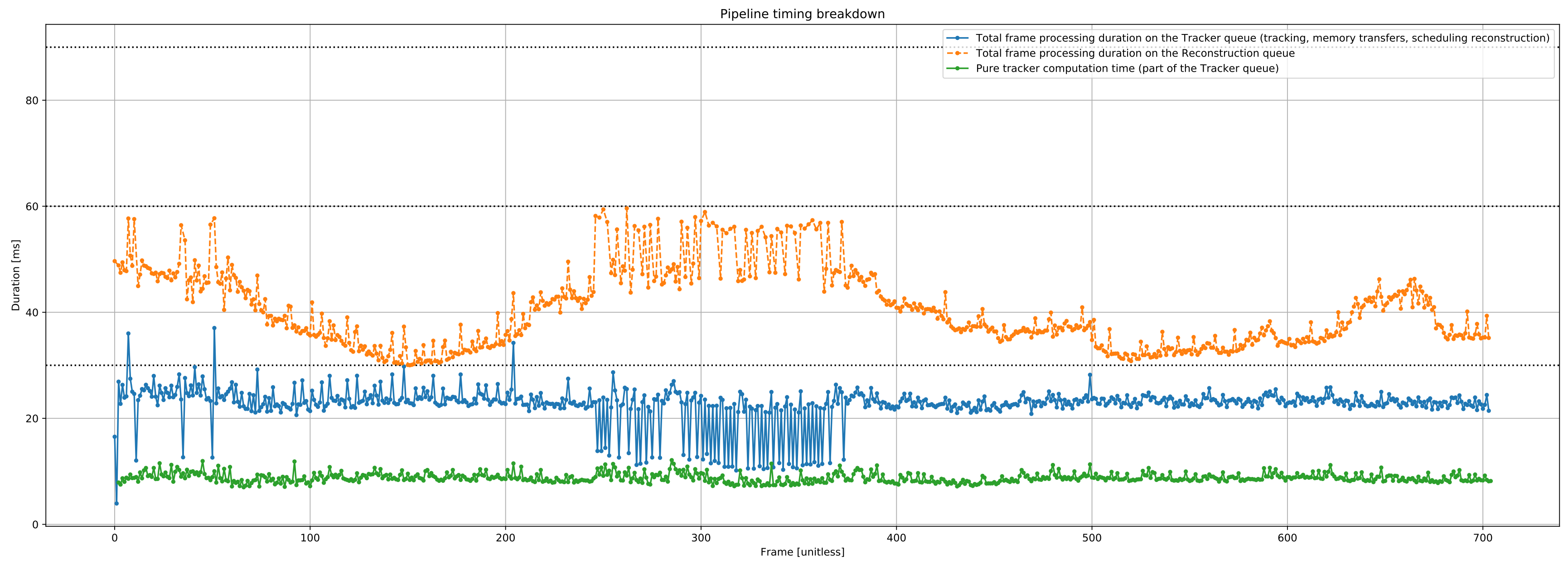
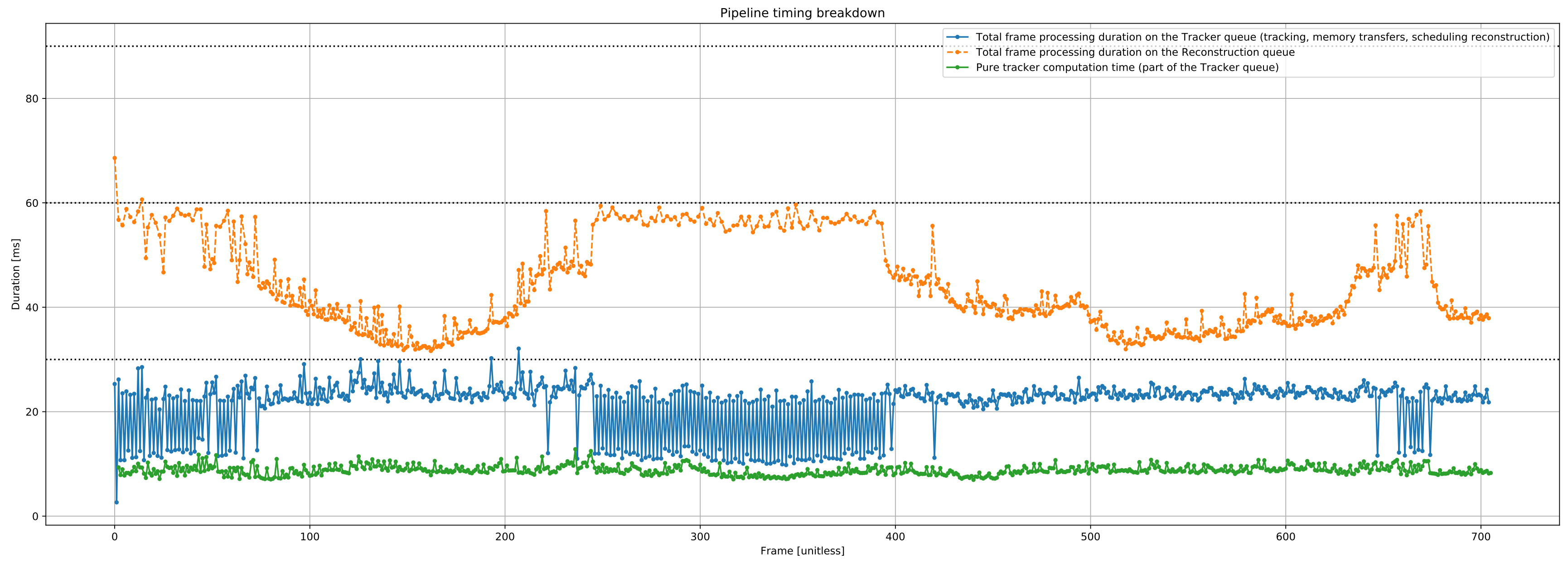

(a) 4mm (b) 3mm (c) 2mm (d) 1mm

Figure 6: Detail of a scanned model that shows influence of scanning resolution on model quality. The resolutions of a voxel are 4, 3, 2 and 1 millimeters respectively.

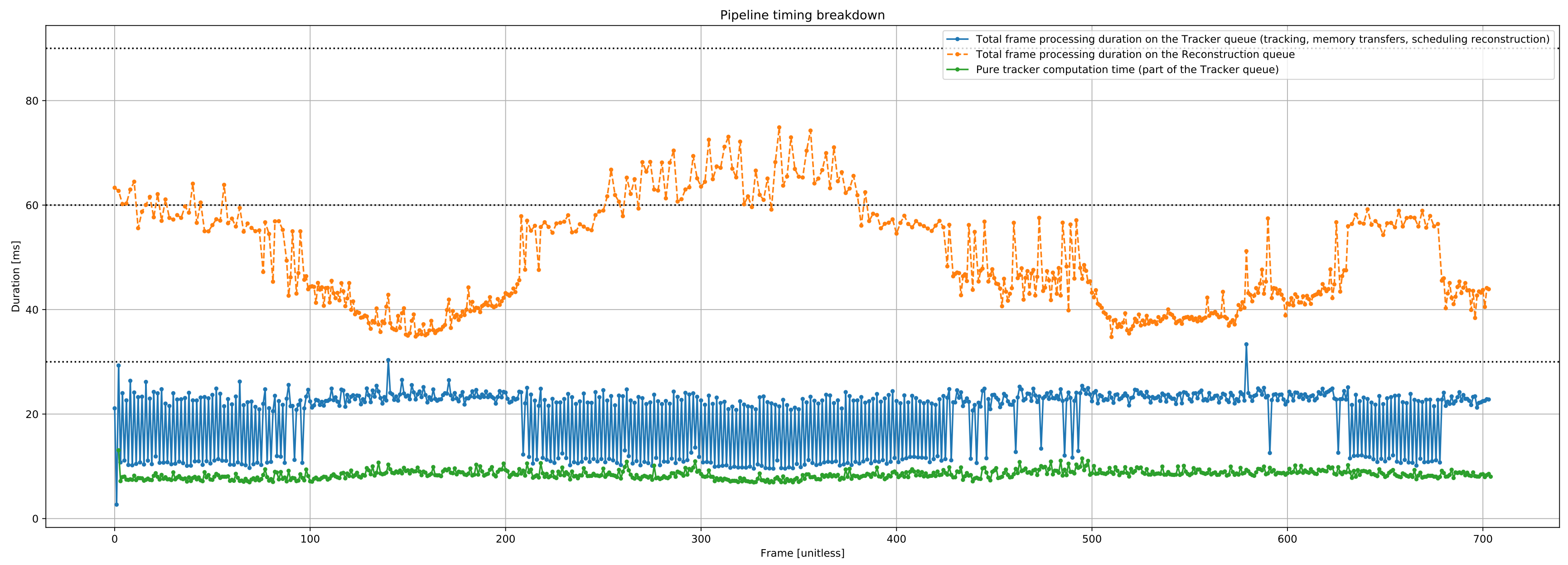


(a) 4mm

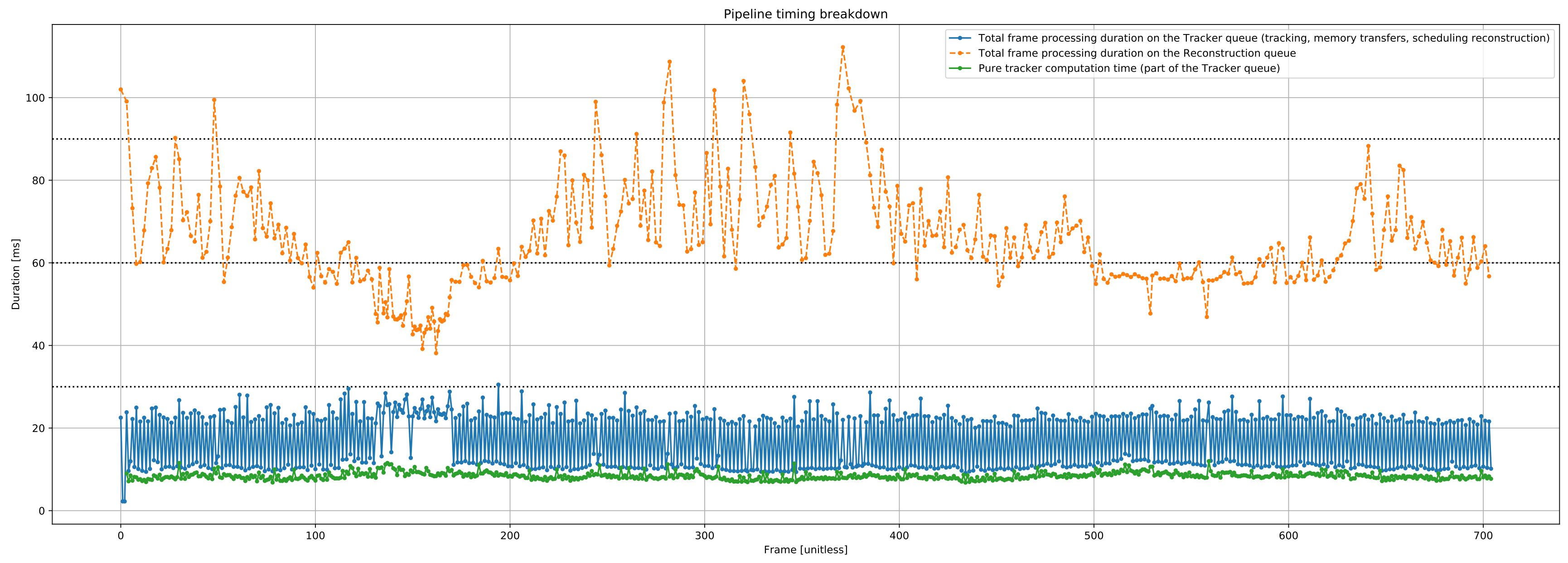


(b) 3mm

Figure 7: Pipeline statistics



(a) 2mm



(b) 1mm

Figure 8: Pipeline statistics

8 Future Work

There are several components in the application that need to be improved; namely the tracker which is not robust enough to sustain longer periods of scanning without introducing drift-induced artifacts (e.g. duplication of geometry due to angle drift). Although I tried to add IMU-based initial guesses of rotation into the reconstruction pipeline, the tracker would eventually fail after vigorous movement — the reason of the failure could be furthermore investigated as it should mitigate drift artifacts.

Scanning of unbounded space is another possible feature to implement — the pipeline was designed to be future-proof by having the reconstruction queue be able to work (e.g. wait for disk I/O) during several frames while tracking in parallel. And finally, the GUI could be improved e.g. by adding user-switchable reconstruction precision handles and more ergonomic controls.

9 Conclusion

The assignment of this bachelor thesis — being able to scan and show the scanned scene — was successfully accomplished, moreover the application supports even exporting the scanned model into a polygonal file format for future usage on desktop computer. After making a stereo-RGB reconstruction pipeline based on ARKit and failing to achieve the specified goals with it (being able to produce detailed models and maintaining longer scanning sessions), despite the presumed precision of tracking of ARKit, a new approach to scanning was taken — RGBD reconstruction with the iPhone X.

By researching existing real-time RGBD reconstruction pipelines, one of them [14] was selected (after mutual comparison of several pipelines), optimized and parallelized to be able to run with the limited computing power (compared to desktop GPUs) of mobile environment and implemented. The implemented pipeline saves scans into a sparse data structure, which is vital for systems with small amount of RAM and VRAM memories (like mobile devices are). The application is future-proofed for potential enhancements (e.g. in the form of tracker relocalization or implementation of space-unbounded scanning) by having the tracking and reconstruction parts running in parallel.

It was empirically observed that the application is able to sustain moderately long scanning sessions, yet under the assumption of smooth and relatively slow movement of the scanner (i.e. mobile phone). Even though the scanning application is not by far fully ideal, I believe it could be useful for quick, everyday scanning.

References

[1] Apple, “ARKit.” [Online]. Available: https://developer.apple.com/arkit/

[2] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, “Kinectfusion: Real-time dense surface mapping and tracking,” in *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, Oct 2011, pp. 127–136.

[3] S. Galliani, K. Lasinger, and K. Schindler, “Massively parallel multiview stereopsis by surface normal diffusion,” June 2015.

[4] B. Li, C. Shen, Y. Dai, A. van den Hengel, and M. He, “Depth and surface normal estimation from monocular images using regression on deep features and hierarchical crfs,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 1119–1127.

[5] Cyrill Stachniss, “Photogrammetry II - 03b - Epipolar Geometry and Essential Matrix (2015/16).” [Online]. Available: https://www.youtube.com/watch?v=8pV-1GFsavA

[6] Radu Bogdan Rusu and Steve Cousins, “3D is here: Point Cloud Library (PCL),” in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[7] M. Keller, D. Lefloch, M. Lambers, S. Izadi, T. Weyrich, and A. Kolb, “Real-time 3d reconstruction in dynamic scenes using point-based fusion,” in *2013 International Conference on 3D Vision - 3DV 2013*, June 2013, pp. 1–8.

[8] “Camera Calibration.” [Online]. Available: https://docs.opencv.org/3.1.0/dc/dbb/tutorial_py_calibration.html

[9] A. Fusiello and L. Irsara, “Quasi-euclidean epipolar rectification of uncalibrated images,” *Machine Vision and Applications*, vol. 22, no. 4, pp. 663–670, Jul 2011. [Online]. Available: https://doi.org/10.1007/s00138-010-0270-3

[10] D. H. Juárez, A. Chacón, A. Espinosa, D. Vázquez, J. C. Moure, and A. M. L. Peña, “Embedded real-time stereo estimation via semi-global matching on the GPU,” *CoRR*, vol. abs/1610.04121, 2016. [Online]. Available: http://arxiv.org/abs/1610.04121

[11] T.-A. Chang, X. Lu, and J.-F. Yang, “Robust stereo matching with trinary cross color census and triple image-based refinements,” *EURASIP Journal on Advances in Signal Processing*, vol. 2017, no. 1, p. 27, Mar 2017. [Online]. Available: https://doi.org/10.1186/s13634-017-0462-3

[12] “Depth map from stereo images.” [Online]. Available: https://docs.opencv.org/3.1.0/dd/d53/tutorial_py_depthmap.html

[13] V. A. Prisacariu, O. Kähler, S. Golodetz, M. Sapienza, T. Cavallari, P. H. S. Torr, and D. W. Murray, “Infinitam v3: A framework for large-scale 3d reconstruction with loop closure,” *CoRR*, vol. abs/1708.00783, 2017. [Online]. Available: http://arxiv.org/abs/1708.00783

[14] O. Kahler, V. Adrian Prisacariu, C. Yuheng Ren, X. Sun, P. Torr, and D. Murray, “Very high frame rate volumetric integration of depth images on mobile devices,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 21, no. 11, pp. 1241–1250, Nov. 2015. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2015.2459891

[15] K.-L. Low, “Linear least-squares optimization for point-to-plane icp surface registration.” [Online]. Available: https://www-new.comp.nus.edu.sg/~lowkl/publications/lowk_point-to-plane_icp_techrep.pdf

[16] Apple, “Metal.” [Online]. Available: https://developer.apple.com/metal/

[17] P. Bourke, “PLY - Polygon File Format.” [Online]. Available: http://paulbourke.net/dataformats/ply/

[18] M. Niessner, M. Zollhöfer, S. Izadi, and M. Stamminger, “Real-time 3d reconstruction at scale using voxel hashing,” *ACM Trans. Graph.*, vol. 32, no. 6, pp. 169:1–169:11, Nov. 2013. [Online]. Available: http://doi.acm.org/10.1145/2508363.2508374

[19] P. Bourke, “Polygonising a scalar field,” 1994. [Online]. Available: http://paulbourke.net/geometry/polygonise/