Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Building a 3D Indoor Scanner

Semester Thesis

Jonas Bächli

baechlij@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Manuel Eichelberger, Simon Tanner
Prof. Dr. Roger Wattenhofer

June 6, 2017

Acknowledgements

I would like to thank my supervisors Manuel Eichelberger and Simon Tanner for their inputs in the weekly meetings during the development of the thesis as well as the constructive criticism concerning this report. I would also like to thank my family and friends for the feedback and support.

Abstract

This thesis dives into still unknown possibilities of virtual reality and explores the feasibility of using a 3D scanner built into a smartphone to virtually visit remote locations. It focuses on the reduction of the polygon count required to represent a scanned scenery in a time and memory efficient way, introducing the developed algorithms to achieve this and showing how the data is gathered and processed. Results are displayed and evaluated in terms of quantity and quality, depicting strengths and limitations of the implementation. Finally, the initial goals are compared to what was achieved, leading to the outlook of possible additions in the future.

Contents

Acknowledgements i

Abstract ii

1 Introduction 1

1.1 Motivation . . . 1

1.2 Related Work . . . 2

1.3 Contributions . . . 2

2 Google Tango 3

2.1 Application . . . 4

3 Method 5

3.1 Polygonal Mesh . . . 5

3.2 Traversing Google Tango Data . . . 6

3.3 Plane Fitting . . . 8

3.3.1 Plane Equation . . . 8

3.3.2 The Growing Algorithm . . . 9

3.4 Replacing the Meshes . . . 11

4 Results 13

4.1 Evaluation Setup . . . 13

4.2 Living room . . . 13

4.2.1 Quantitive Evaluation . . . 16

4.2.2 Qualitative Evaluation . . . 18

5 Conclusions and Future Work 20

Bibliography 22

Chapter 1

Introduction

1.1 Motivation

While virtual and augmented reality (VR and AR) are gaining popularity in niche markets such as gaming or training (flight simulators, surgery simulators, and more), the public has not yet experienced its capabilities besides some exceptions such as Pokémon Go. However, with the emergence of affordable consumer devices, those technologies will most likely change how we accomplish certain day-to-day tasks. A conceivable use case is flat or house hunting: the current owners could scan their home and share this data with possible interested parties. Prospective new tenants on the other hand would then be able to explore their potential new living space, maybe even try to fit in virtual objects such as existing furniture.

In this thesis we use a Google Tango enabled smartphone, which features an easy-to-use 3D scanner, to explore the possibilities of using such a device to virtually visit remote locations, focusing on the handling of the vast amount of data being produced while scanning a scenery through the means of 3D reconstruction. One way to reduce the amount of data accumulating in point cloud based scan is to find context for points, for example whether or not they are part of a larger surface corresponding to a parametric primitive. This allows to get rid of data points, since they can easily be replaced by a primitive. This thesis tries to find plane primitives and uses the resulting limitations applicable to those points to delete points which do not contribute any geometric value.

1.2 Related Work

There is already a lot of existing research in the field of 3D reconstruction, even some using the Google Tango Platform [1] as well.

An example is [2], this paper explores large scale scene reconstruction, using a Google Project Tango tablet to gather data interactively. This work focuses on finding filtering options for outliers, determining the reliability of scanned data. CHISEL [3], another Google Tango based project, aims to improve memory management, allowing scans of large areas with high resolutions on a portable device. This is accomplished by quickly filtering data and throwing away unessential measurements. In this thesis this task is performed by the use of the Google 3D Reconstruction Library [4].

The PCL (Point Cloud Library) is introduced in [5], establishing a standard library for 3D reconstruction from point cloud data. This open source library collects a multitude of algorithms usable for many different tasks. However due to incompatibility with the runtime used in this thesis it was decided to implement an independent solution.

In [6] a framework for the segmentation of point cloud data into surfaces is presented, fitting in geometric primitives to resolve missing data. And last but not least, [7] discusses feature extraction from point clouds, focusing on kitchen environments. This thesis takes a similar approach, using plane primitives to detect planar surfaces; however rather than using point-clouds as data source it uses preprocessed data coming from the Google 3D Reconstruction Library [4].

1.3 Contributions

The contributions in this thesis are the following:

- Just-in-time exploration of the data provided by Google Tango, transforming the data into an adequate representation.
- Exploring the acquired dataset to find planes using a growing algorithm.
- Using the discovered planes to reduce the amounts of vertices and triangles necessary to store the 3D data.

Chapter 2

Google Tango

The Google Tango platform [1] enables Android devices to build 3D models of the environment. This is done using a multitude of sensors, such as wide-angle cameras, depth-sensing cameras, motion cameras, gyroscopes and accelerometers. Those sensors, combined with a software stack allow applications to generate point clouds, provide virtual/augmented reality experiences and more. The software stack behind the scenes implements technologies ranging from computer vision to SLAM (Simultaneous Localisation and Mapping) to combine the different sensor readouts into one consistent representation of the surroundings.

For this thesis, a Lenovo Phab 2 Pro was used. This device offers all the necessary sensors required by the Google Tango platform and is backed by a Google Tango optimised processor and 4 GB of RAM.

Figure 2.1: The Lenovo Phab 2 Pro [8]

2.1 Application

Google Tango offers multiple APIs to access its generated 3D data. At the moment, developers can choose from a C/C++ API, a Java API and an Unity SDK. As visualised in Figure 2.2, the Google Tango APIs run on top of the Google Tango Service. This service handles the data collection from the sensors and serves the user with preprocessed data.

Unity is a popular game engine, offering multi-platform support [9] for most platforms in use today. It also offers 3D capabilities right out of the box without the need to rely on a third party 3D engine, which led to the decision to use the Unity SDK for this thesis instead of using the C or Java API.

The application built in this thesis makes use of the 3D Reconstruction Library [4] [10]. This C library is exposed through the Unity SDK and provides mesh representations of the environment, created from the point cloud. Those meshes already offer an estimation of the surfaces and can be configured to include additional data, for example colours and normals. The environment is divided into voxels, which, similar to pixels in 2D images refer to a single value in a 3D environment, arranged in a regular three dimensional grid. To access those voxels, one uses the so called *GridIndex*, a struct designating a cube of 16 x 16 x 16 voxels in the volume grid, where the referenced unit is defined by the set resolution of the Google Tango framework.

Furthermore, Math.NET Numerics [11], an open source C# numerical library was used to solve linear equations.

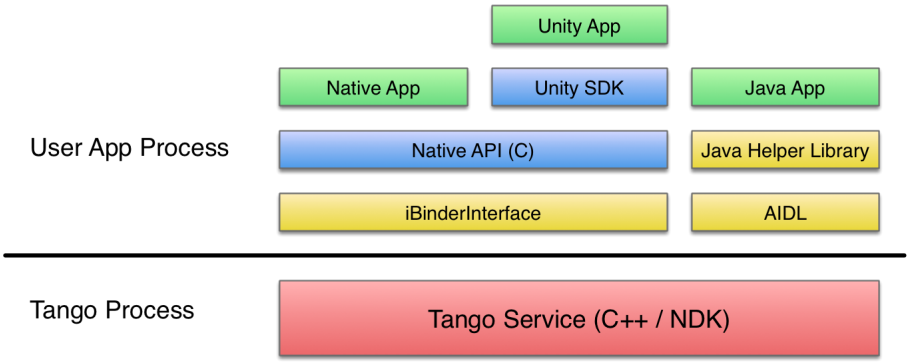

Figure 2.2: Overview of the Google Tango development stack [12]

Chapter 3

Method

This chapter introduces the algorithms used to process the Google Tango 3D data, find planes in this data and finally replace those planes with size-optimised meshes.

3.1 Polygonal Mesh

The term mesh used throughout this thesis refers the so-called polygonal mesh, a data structure used in 3D modelling to represent surfaces, consisting of vertices, edges and faces. The faces are usually triangles, as there is a large amount of existing graphics hardware specifically optimised to handle this shape. Figure 3.1 shows an example mesh, also displaying the boundary edge, the set of edges belonging to exactly one triangle. Additional data, such as colour information, normal vectors, texture mappings and more might be stored together to represent a 3D surface.

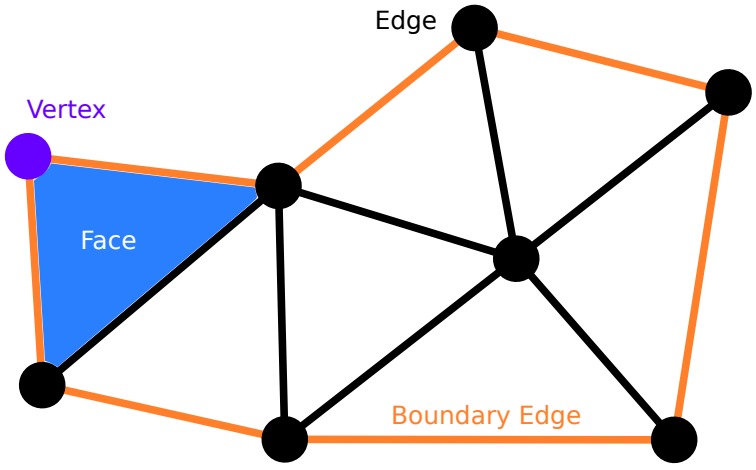

Figure 3.1: A simple example mesh

3.2 Traversing Google Tango Data

As mentioned above, the data retrieved from Google Tango is divided into individual meshes, each spanning the detected surface in the volume of one cube, called a *GridIndex*. The application keeps track of the processed *GridIndices* by storing the meshes in a hash table, mapping the *GridIndices* to the respective mesh.

The most important fields of the mesh representation in Unity are vertices, triangles and colours. The vertices on their own just define a point in space, while triangles, referencing these vertices, form the surface. Colour data is used to shade the resulting surface. Between vertices the rendered colour is interpolated from the vertices around it. The triangles are also used to relate vertices to each other, building neighbourships by adding each one of the three vertices of a triangle as a neighbour of the other two vertices. This is a simple and fast algorithm, but it is limited to neighbours inside of one mesh only.

To be able to overcome this limitation, a struct *GlobalVertex* was introduced (Listing 3.1). This struct specifies one vertex in one mesh, therefore making it possible to differentiate every existing vertex. This allows a vertex A in *GridIndex* gi_A to reference a vertex B in *GridIndex* gi_B, hence enabling neighbourships across the *GridIndex* border. The *GlobalVertex* also keeps references to its mesh and its absolute position, making it one of the most essential datatypes used, allowing easy mappings between the Google Tango given representation and the global view over the entire 3D model.

Listing 3.1: GlobalVertex struct

```
public struct GlobalVertex
{
        public Tango3DReconstruction.GridIndex gridIndex;
        public int vertexIndexInMesh;
        public Mesh mesh;
        public Vector3 actualVertex;
}
```

To discover neighbours between adjacent *GridIndices* (Algorithm 1), we first create a list of non-empty *GridIndices*, adjacent to the current *GridIndex*. For each adjacent *GridIndex*, we try to find a common plane between the current *GridIndex* and the adjacent *GridIndex*. This is achieved through the use of *BorderBoxes*, which define the minimum and maximum value of the x, y and z coordinate values in each mesh. If two adjacent *GridIndices* gi_A and gi_B share a common border across one axis, we are able to easily define a plane between gi_A and gi_B. Iterating through the vertices in gi_A and gi_B and checking the distance to the newly created plane allows us to build two sets with vertices in those two *GridIndices* that are in close proximity to the other *GridIndex*. By finding pairs

of vertices in those two sets that are very close to each other, we can add those vertices as their respective neighbours, closing the gap between *GridIndices*.

```
input: GridIndex giA
input: GridIndex giB
bbA = BorderBox (giA);
bbB = BorderBox (giB);
find common plane p between giA and giB using the border boxes;
for each vertex v in giA do
    if distance (p,v) < margin then
        add to planeA;
    end
end
for each vertex v in giB do
    if distance (p,v) < margin then
        add to planeB;
    end
end
for each vertex vA in planeA do
    for each vertex vB in planeB do
        if distance (vA, vB) < margin then
            add vA as a neighbour of vB;
            add vB as a neighbour of vA;
        end
    end
end
```

Algorithm 1: Finding neighbours across *GridIndices*

Establishing neighbourships on a per *GridIndex* base allows us to parse this data just-in-time, or more precisely when a vertex inside of a new *GridIndex* is added to a plane set (see below). This potentially decreases processing time since unused *GridIndices* are never parsed and leads to earlier visible results; being able to use the established relationships required to find planes without having to wait for the parsing of the entire data. It also makes it possible to change the environment during runtime without having to re-parse the whole mesh.

Neighbours are stored using a hash table, having a *GlobalVertex* as the key and a list of *GlobalVertices* as its neighbours.

3.3 Plane Fitting

3.3.1 Plane Equation

The main equation used is the well known plane equation Equation (3.1), rewritten as Equation (3.2).

$$ax + by + cz + d = 0 \tag{3.1}$$

$$\frac{ax}{c} + \frac{by}{c} + \frac{d}{c} = -z \tag{3.2}$$

This allows us to do a simple least square approach to solve the linear equation

$$Ax = \vec{c}$$

where A corresponds to a matrix with n rows containing $(x_i\ y_i\ 1)$, with n being the total amount of vertices currently in the data set and i specifying one vertex in the set with it's coordinates x_i, y_i, z_i. The vector $\vec{c}$ contains n rows of $-z_i$.

Solving this equation leads to the values for $a' = \frac{a}{c}, b' = \frac{b}{c}, c' = 1, d' = \frac{d}{c}$, defining the plane $a'x + b'y + c'z + d' = 0$ with normal vector Equation (3.3). This plane equation is then normalised: with d'' scaled $d'' = \frac{d'}{|\vec{n}|}$, and a'', b'' and c'', the respective members of the normalised vector $\vec{n_u} = \frac{\vec{n}}{|\vec{n}|}$, therefore giving us the normalised plane equation $a''x + b''y + c''z + d'' = 0$.

$$\vec{n} = \begin{pmatrix} \frac{a}{c} \\ \frac{b}{c} \\ 1 \end{pmatrix} = \begin{pmatrix} a' \\ b' \\ c' \end{pmatrix} \tag{3.3}$$

The error of a point is defined as the distance D from the plane, given by Equation (3.4). Since the normal vector $\vec{n_u}$ is normalised, D can be simplified to Equation (3.5).

$$D = \frac{|a''x + b''y + c''z + d''|}{\sqrt{a''^2 + b''^2 + c''^2}} \tag{3.4}$$

$$D = |a''x + b''y + c''z + d''| \tag{3.5}$$

Therefore the total error E across the entire plane can easily be calculated by using Equation (3.6).

$$E = \sum_{i=1}^{n} \left| a''x_i + b''y_i + c''z_i + d'' \right| \tag{3.6}$$

3.3.2 The Growing Algorithm

With the Google Tango data parsed into a more suitable representation, we are now able to find planes in our data sets.

The process is initiated by setting a starting point, either a randomly selected vertex or a chosen vertex. Then a seed is built by directly adding vertices from the neighbouring set to the plane set without any checks, until we reach a certain size, currently set to 10 vertices. As soon as the start seed size is reached, we calculate the first plane parameter estimation as explained above.

Having acquired the first plane estimation, we can now try to grow the plane, according to Algorithm 2. Every plane set member is either in the bounding box set, so potentially still has neighbours not yet in the set that might fit the plane estimation; or they belong to the implied inside set, where every neighbour already belongs to the plane; or it has an error big enough to eliminate it from further consideration and therefore belongs to an out-of-bounds set.

Iterating through the neighbours in the bounding box set yields a set of potential new vertices to add to the plane set. In the same step we can also lazily update the bounding box set by keeping track of how many neighbours of a vertex we are able to add to the potential new vertices set; if there are no neighbours added at all, we can safely remove this vertex from the bounding box set, since all of its neighbours either belong to the plane set or the out-of-bounds set.

In a next step the vertices in the potential new vertices set are rated according to their error/distance D from the plane and those ratings are then sorted in ascending order. In a last step the decision on which vertices to add to the plane set, which to keep active for later consideration and which to add to the out-of-bounds set is made. In one grow-step only a limited amount of vertices can be added to the plane, given by Equation (3.7). This limits the amount of vertices added due to a bad estimation. The plane estimation is updated after each grow-step to make sure the estimation is as accurate as possible.

$$\begin{aligned} lowerGrowSize &= \max(\#verticesInPlane/10, minGrowSize) \\ growSize &= \min(lowerGrowSize, maxGrowSize) \end{aligned} \tag{3.7}$$

The plane keeps growing until either the total error E exceeds a threshold $E_{max} = \#verticesInPlane \cdot allowedErrorPerVertice$, in which case the plane is rejected, or no more vertices are found that match the criterion to be added to the plane, which results in a valid plane. After growing, the algorithm decides whether or not the plane is worth keeping. This is decided based on the quantity of the included vertices; if the plane contains 1000 vertices or more, it is considered worthwhile to optimise its vertex and triangle count, otherwise it is rejected.

```
Function GetNeighboursOfBoundingBox():
    for each vertex v in BoundingBox do
        for each neighbour n of v do
            if (n not in Plane) and (n not in OutOfBounds) then
                set flag that v still has neighbours to potentially grow to;
                add to list PotentialNewVertices;
            end
        end
        if v has no more vertices to grow to then
            remove v from BoundingBox;
        end
    end
    return PotentialNewVertices
end
Function ChooseKBestVerticesFromList(k, PotentialNewVertices):
    for each vertex v in PotentialNewVertices do
        calculate error D for v;
        rate v by D and add to list Rated;
    end
    sort Rated ascending;
    for each vertex v in Rated do
        if (rating(v) < allowable error) and (less than k vertices added)
        then
            add v to the Plane;
        end
        else if rating(v) > throw away boundary then
            add v to OutOfBounds;
        end
    end
end
Function Grow(k):
    PotentialNewVertices = GetNeighboursOfBoundingBox ();
    ChooseKBestVerticesFromList (k, PotentialNewVertices);
end
```

Algorithm 2: Description of the growing step

3.4 Replacing the Meshes

After finding a suitable plane, the compiled set is once again converted, this time into a representation adequate for reduction. The main difference is that triangles are now instances, holding references to its vertices and vice versa. However, due to the isolated nature of the meshes acquired from Google Tango and the requirements of the polygon reduction algorithm needing an uniform mesh, the vertex pairs found in Algorithm 1 have to be merged into one vertex, combining the meshes from multiple *GridIndices* into one consistent mesh.

Since the mesh we want to simplify is a planar surface, but probably has an irregular edge, we keep the boundary edge as it is. Therefore we have to identify the boundary of our mesh. This task is accomplished by Algorithm 3, using the fact that a boundary edge is part of exactly one triangle, as defined in Section 3.1.

for *each triangle in Triangle* **do**
  increase edge count for each included edge;
end
add every edge with count 1 to the boundary set;

Algorithm 3: Boundary detection algorithm

The algorithm used to reduce the polygon count is loosely based on [13], a standard algorithm for polygon reduction. It requires each edge of the mesh to have an assigned weight, defining the cost of collapsing it, allowing it to decide which edges to collapse and merge its vertices into one. Merging two vertices u and v hereby means to move vertex u to vertex v, thus allowing the algorithm to move the inside vertices outwards, while the edge vertices remain in place. Due to the fact that the mesh to be reduced is a flat plane, we can simplify the cost function used to determine the cost of merging vertex u into vertex v to Equation (3.8).

$$cost(u,v) = \begin{cases} maxCost & u \in Boundary \\ distance(u,v) & \text{otherwise} \end{cases} \tag{3.8}$$

With established weights for each edge the only thing remaining is to actually merge the vertices according to Algorithm 4. While the desired vertex count is not reached, the edge uv with the lowest weight assigned to it is found, then the vertices u and v are merged together. Here we make use of the fact that triangles and vertices reference each other, allowing us to quickly find the triangles containing u. If a triangle contains the edge uv, it is removed as there would only be two vertices remaining. Otherwise the vertex u is replaced with v. Vertex u can now be removed from the dataset and the former neighbours of u are now updated to be neighbours of v.

```
while reduction possible do
    find edge uv with lowest cost;
    Merge (u,v);
end
Function Merge(u,v):
    store neighbours of u;
    for each triangle adjacent to u do
        if triangle contains v as well then
            remove triangle;
        end
        else
            replace u with v in triangle;
        end
        remove u;
        restore neighbour relationship between neighbours of u and v;
        update edge costs;
    end
end
```

Algorithm 4: Reducing the polygon count

Chapter 4

Results

This chapter shows the results obtained by running the implementation on a laptop computer, due to the fact that the code has not yet been ported to the Android app. The laptop has a 2.5 GHz Intel Core i7 processor, 16 GB of RAM and provides a NVIDIA GeForce GT 750M as a graphics card.

4.1 Evaluation Setup

The data was gathered using a slightly modified version of the MeshBuilderWithColor example [14] app provided by Google, improving its exporting capabilities. After scanning, the data is exported to a Wavefront .obj file [15], a simple, text-based 3D file format. This file is then imported into the Unity desktop app, restoring the representation used in the smartphone app to enable code sharing over multiple platforms. Finally, the implementation is run according to Chapter 3, choosing random starting points until no more planes are found during 30 seconds.

4.2 Living room

The scene explored is a living room as depicted in Figure 4.1 and Figure 4.2, showing a photo of the scene as well as a shaded wireframe render of a similar view in the Unity Editor. Also visible in the wireframe renders is that Google Tango struggles with reflective and/or black surfaces such as the window, the chairs or the table in the example.

Figure 4.1: The living room, view 1

Figure 4.2: The living room, view 2

4.2.1 Quantitive Evaluation

On the smartphone, the scan process took 1:56 minutes, exporting it as an .obj file took 3:12 minutes. The resulting file contained a total of 171,286 vertices and 845,775 triangles.

After importing the geometry into the desktop app, the algorithm detected and reduceed planes. The results displayed in Figure 4.3 and Table 4.1. 27 planes were detected and reduced, resulting in a reduction of 64.4 % for vertices and 78 % for triangles. The processing took 215.53 seconds or about 3 minutes and 35 seconds. Table 4.2 shows how the total time is shared between the different processes. Note that 26.1 % of the total time is spent by the Unity runtime; a process we have little influence on. The surprisingly long time needed to delete vertices is explained through the fact that each *GridIndex* touched by the resulting plane needs to be visited, the corresponding vertices removed and the whole mesh updated without destroying existing relations.

In terms of file size, before reduction the .obj file was 26996208 bytes or 25.746 MiB. After reduction the size shrank to 7027509 bytes respectively 6.702 MiB, a 73.969 % reduction in file size.

Name	Before	After	% Reduction
Vertices	171,286	60,905	64.443 %
Triangles	845,775	185,685	78.046 %
Planes	0	27	n/a

Table 4.1: The removed vertices and triangles in numbers

Process	Seconds	% of Total Time
Importing geometry	4.955	2.299
Finding start vertices	0.556	0.258
Parsing Google Tango Data	38.867	18.033
Growing planes	42.569	19.751
Reducing polygon count	49.404	22.922
Deleting the original meshes	22.673	10.520
Inserting the reduced meshes	0.259	0.120
Other (Unity)	56.246	26.097

Table 4.2: Time used by the processes

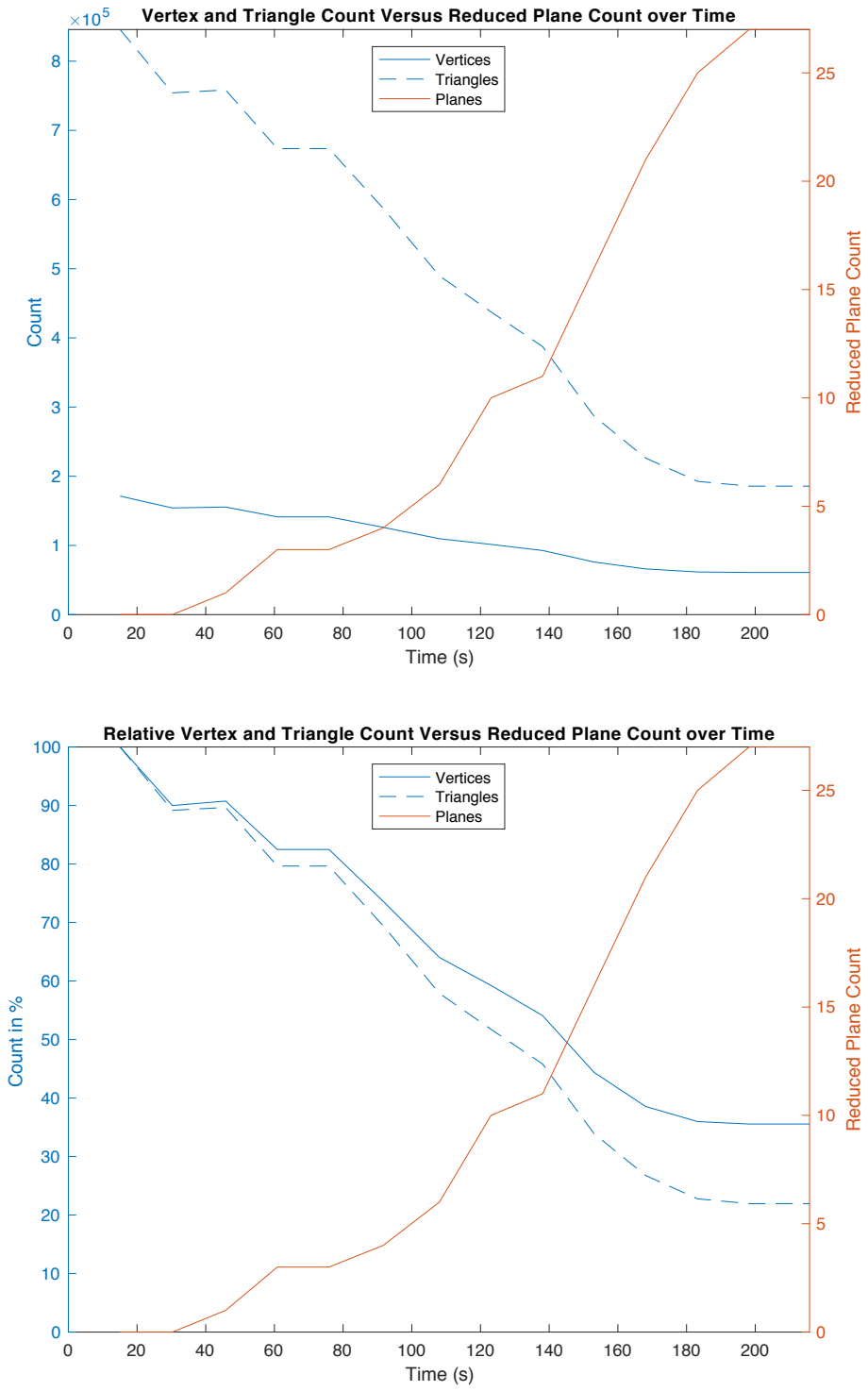

Figure 4.3: Plots displaying vertex, triangle and reduced surface count over time, the upper in absolute numbers and the lower relative to the starting count.

4.2.2 Qualitative Evaluation

Figure 4.4 and Figure 4.5 show the scenery after reduction. The vertices and triangles belonging to a recognised plane are removed and replaced with a reduced version, coloured uniformly across the entire plane. Most of the time the algorithm reliably recognises planes and reduces their polygon count, however the algorithm does not always work as expected: the door (the sea-green surface in Figure 4.4) grew across the floor, along the plane defined by the door, thus not only inaccurately detecting the door but also preventing the floor to be identified as a connected surface.

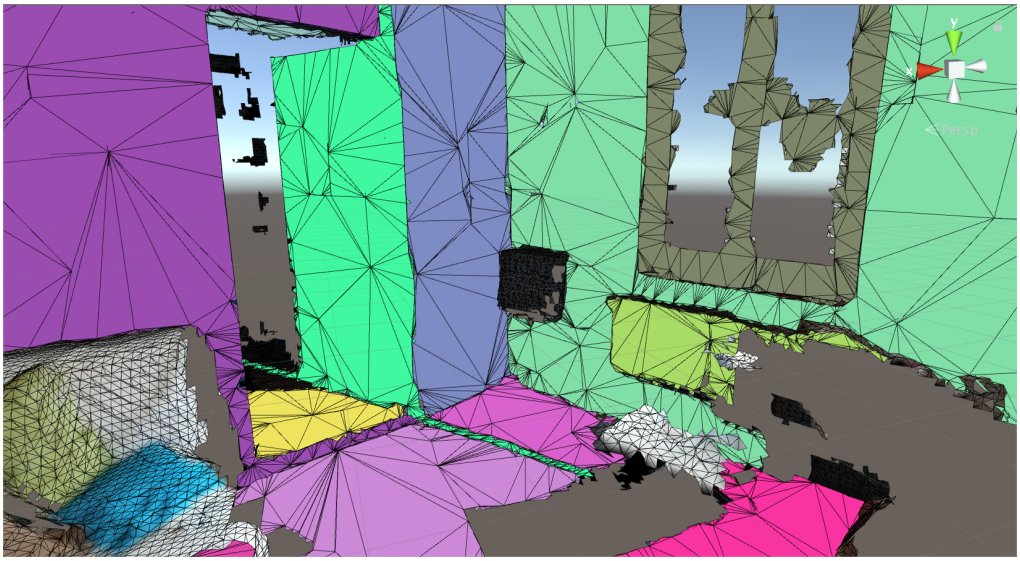

Figure 4.4: The living room after reduction, view 1

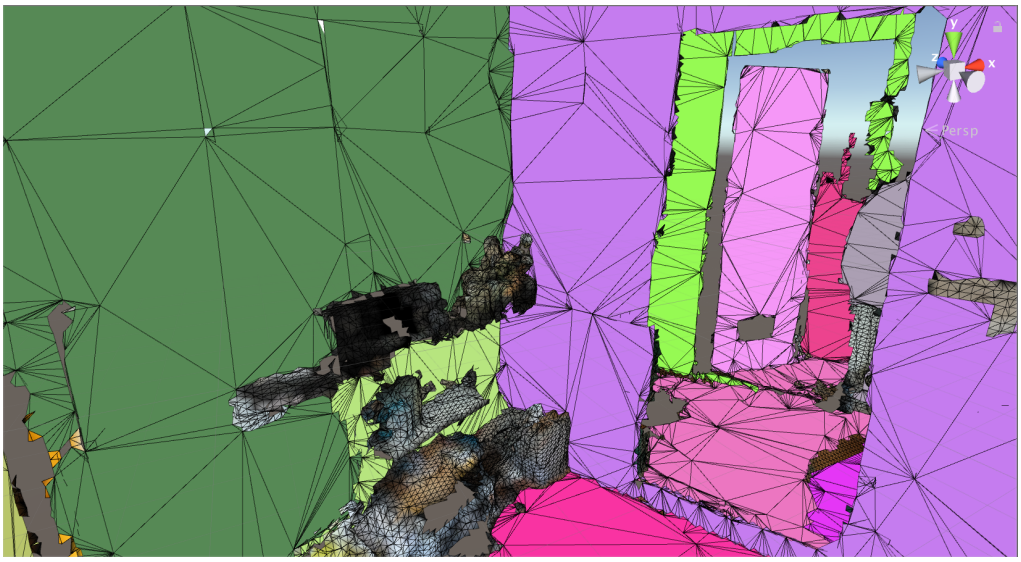

Figure 4.5: The living room after reduction, view 2

Chapter 5

Conclusions and Future Work

This thesis lays the foundation on reaching its eventual goal, creating an app to easily scan an indoor environment and share the collected data. Even though this goal has not been met in the sense that there is now an existing app, a lot progress has been made regardlessly. The data exposed by Google Tango is acquired in an Android app, exported to a file and imported on a desktop machine, where it is then processed to reduce the total vertex and triangle count required to represent a given indoor scene through the use of a set of developed algorithms.

However, there is still a lot to be done to get closer to the initial goal, including the porting of the desktop code to the Android platform. Another task is performance optimisation; it should be possible to use one data representation for all the steps implemented, getting rid of the time needed to translate between different data representations. However, with potential realtime applications in mind, this is quite a challenge due to the fact that new, updated Google Tango data should still be included, even if the *GridIndex* has already been reduced. Furthermore, improving the used algorithms would also be worthwhile, for example to include normal vectors to define planes to counteract misbehaviours such as the door plane extending across the floor in Figure 4.4 or reducing shorter boundary edge subsets which form a line into bigger edges using an improved cost function. And finally, as this is a seed based algorithm, it suffers from the same drawback as pretty much every other seed based algorithm: If the seed is bad, the outcome might not be what you would expect, thus one might try to optimise the chosen seeds or reject certain seeds.

Feature-wise there are also quite a few possible enhancements:

- Textures and UV-mapping: At the moment, the colour information obtained from Google Tango is completely tied to the vertex data. Therefore, reducing vertex count also reduces the colour information density. Using separate textures (either acquired directly from the camera / Google Tango framework or projected using the vertex colour information) and a corresponding UV-mapping would allow the user to keep details like wallpapers while still reducing the vertex count.

- Normal and bump mapping: This technique allows keeping a higher level of detail with a lower polygon count, since finer details are displayed using special textures, in which instead of colour geometry (normal vectors or heightmaps) is stored and then rendered by the GPU. This might allow to depict details such as clocks or framings while keeping the polygon count low.

- Room detection: Another potential feature would be to detect rooms. This would not only enhance the user experience since this is how we as users relate to rooms but also open potential optimisations, because it would allow the app to only keep parts of the scanned geometry in memory while currently unused parts would be loaded as needed.

- Measurements: As an addition to the last feature, it would be nice to get room measurements directly, allowing users to relate the virtual world with real life distances. As a supplement, it might be possible to scan existing furniture and try to fit it in virtual reality to see whether or not something would fit into a certain spot.

Bibliography

[1] Google: Tango developer overview. https://developers.google.com/tango/developer-overview [Online; accessed 25-May-2017].

[2] Schöps, T., Sattler, T., Häne, C., Pollefeys, M.: 3d modeling on the go: Interactive 3d reconstruction of large-scale scenes on mobile devices. In: 3D Vision (3DV), 2015 International Conference on, IEEE (2015) 291–299

[3] Klingensmith, M., Dryanovski, I., Srinivasa, S., Xiao, J.: Chisel: Real time large scale 3d reconstruction onboard a mobile device using spatially hashed signed distance fields. In: Robotics: Science and Systems. (2015)

[4] Google: Tango 3d reconstruction library c api reference. https://developers.google.com/tango/apis/c/reconstruction/reference/ [Online; accessed 25-May-2017].

[5] Rusu, R.B., Cousins, S.: 3d is here: Point cloud library (pcl). In: Robotics and Automation (ICRA), 2011 IEEE International Conference on, IEEE (2011) 1–4

[6] Rusu, R.B., Blodow, N., Marton, Z.C., Beetz, M.: Close-range scene segmentation and reconstruction of 3d point cloud maps for mobile manipulation in domestic environments. In: Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on, IEEE (2009) 1–6

[7] Rusu, R.B., Marton, Z.C., Blodow, N., Dolha, M., Beetz, M.: Towards 3d point cloud based object maps for household environments. Robotics and Autonomous Systems **56**(11) (2008) 927–941

[8] Lenovo: Lenovophab2pro. http://shop.lenovo.com/ISS_Static/WW/campaigns/2016/tango/images/Phab2-pic5-big.jpg [Online; accessed 25-May-2017].

[9] Unity: Unity multiplatform. https://unity3d.com/unity/multiplatform [Online; accessed 26-May-2017].

[10] Google: Tango.tango3dreconstruction. https://developers.google.com/tango/apis/unity/reference/class/tango/tango3-d-reconstruction [Online; accessed 25-May-2017].

[11] Math.NET: Math.net numerics. https://github.com/mathnet/mathnet-numerics [Online; accessed 06-June-2017].

[12] Google: Tango api diagram. https://developers.google.com/tango/images/apis/APIDiagram.png [Online; accessed 25-May-2017].

[13] Melax, S.: A simple, fast, and effective polygon reduction algorithm. Game Developer **11** (1998) 44–49

[14] Google: Project tango unitysdk examples. https://github.com/googlesamples/tango-examples-unity [Online; accessed 06-June-2017].

[15] Wavefront: Object files (.obj). http://www.martinreddy.net/gfx/3d/OBJ.spec [Online; accessed 06-June-2017].