OULUN SEUDUN
AMMATTIKORKEAKOULU

Juha Hyvärinen

SURFACE RECONSTRUCTION OF POINT CLOUDS CAPTURED WITH MICROSOFT KINECT

SURFACE RECONSTRUCTION OF POINT CLOUDS CAPTURED WITH MICROSOFT KINECT

Juha Hyvärinen
Bachelor's Thesis
Spring 2012
Degree Programme in Information Technology and Telecommunications
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology and Telecommunications

Author: Juha Hyvärinen
Title of thesis: Surface Reconstruction of Point Clouds Captured with Microsoft Kinect
Supervisors: Pekka Alaluukas (OUAS), Jarkko Vatjus-Anttila (CIE)
Term and year of completion: spring 2012 Pages: 42 + 4 appendices

The main motivation behind this thesis was to create a new method of content creation for virtual spaces. This thesis focuses on how an already captured point cloud has to be processed for the optimal results of the object reconstruction.

In this thesis realXtend Tundra, which is an open source framework for virtual spaces, is used as a target platform. The point cloud processing is integrated with Tundra in such a way that a user can import real world objects to the Tundra virtual space with an easy-to-use procedure. This thesis is based on the work done in the bachelor thesis of Miika Santala, who researched the capturing of real world objects to a point cloud by using Microsoft Kinect and the Point Cloud Library libraries (38). In this thesis, the open source Point Cloud Library was used for the point cloud processing.

The thesis was commissioned by the Chiru project, Intel and Nokia Joint Innovation Center, Center for Internet Excellence, University of Oulu, which concentrates on researching and developing new 3D based user interaction paradigms.

As a result, a working process for object capturing was achieved. A real world object can be captured to a point cloud and processed to the Ogre mesh, which can be added to a virtual space scene. The whole toolset is implemented to the realXtend Tundra framework as a module.

Keywords: Microsoft Kinect, realXtend, point cloud, Point Cloud Library, surface reconstruction

CONTENTS

SYMBOLS AND ABBREVIATIONS

3D	Three-dimensional
API	Application Programming Interface
Chiru	Name of the project started by Intel, Nokia and CIE
CIE	Center for Internet Excellence
LLUDP	Linden Labs UDP, an UDP based protocol for Second Life
MLS	Moving Least Squares
Ogre	Object-Oriented Graphics Engine
OpenNI	Open Natural Interaction
PCL	Point Cloud Library
PCD	Point Cloud Data -file format
SSE	Streaming Single Instruction, Multiple Data Extensions
VTK	Visualization Toolkit
XML	Extensive Markup Language

1 INTRODUCTION

The purpose of this bachelor thesis is to enable an object reconstruction for the virtual space environment by reconstructing a 3D object from a point cloud. The point cloud is a data structure holding multi-dimensional data in it. Most commonly, point clouds are used to represent three-dimensional data and usually those are X, Y and Z coordinates.

In this thesis, it is described how point cloud data can be transformed into a polygon mesh, which can be imported to a realXtend Tundra scene. The need for the implementation comes from the point of view of a user who may want to create a personalized virtual space but is not skilled in 3D modeling. The research goal of this thesis was to achieve easy importing of any objects into a virtual space where it can be manipulated and used as a building block for a fully personalized virtual space.

The 3D acquisition from a physical object to a point cloud is done by a co-worker and the results are described in the bachelor thesis of Miika Santala (38). The field of this thesis is limited to reconstructing an object from a point cloud that has already been captured converting the resulting mesh to the Ogre mesh format and importing it into the realXtend Tundra scene. Tundra is an open source framework for virtual spaces. The Ogre mesh can be used directly in Tundra. The whole dataflow is integrated to Tundra; hence it works seamlessly with other software components in that framework. All algorithms used in this thesis were profiled by measuring the time consumed in the algorithm code. Also, the rendering times of the captured objects were measured and the most expensive code blocs were identified.

This thesis is a part of a larger project which aims to create new user experiences and study 3D user interfaces in a virtual space environment. This particular part of the work supports the possibilities for content creation in virtual spaces. The captured objects could be then transferred between different devices and virtual spaces. This way, it is easy to create fully personalized spaces into a virtual environment without any previous experience in 3D modeling or knowledge about the technology under the hood.

This thesis was commissioned by the Chiru project, Intel and Nokia Joint Innovation Center, Center for Internet Excellence, University of Oulu. The project focuses on researching the 3D-internet and the new user interface paradigms.

2 RESEARCH PROBLEM

In this thesis, the main goal was to achieve capturing of a real world object, which can be used in virtual spaces. It was predefined a requirement of the Chiru project that only a CPU (Central Processing Unit) of a computer can be used for data processing. Commonly this kind of data processing is done with a GPU (Graphics Processing Unit). Due to the limitations in the processing power, the requirement for real time processing was not the primary concern and the target was set to the semi-real time process. In this thesis, a method is proposed how the point cloud can be transformed into a mesh format supported by the realXtend Tundra.

In the literature, the object capturing and surface reconstruction has got quite much attention, since more and more content for virtual environments is needed and the accuracy levels required have also risen (1, 5-7, 10, 13, 16, 18, 19, 26, 31, 35-38). The problem of the object capturing can be divided into several sub-problems: cloud smoothing, surface reconstruction, surface coloring, hole filling, point cloud filtering and mesh simplification. Furthermore, the point cloud processing can be divided into four main steps: pre-processing, determination of the global topology for the surface of the object, generation of the polygonal surface and post-processing. (37.)

Microsoft Kinect is a commercial depth camera with RGB color data. Kinect is designed to work as a game controller for the Microsoft Xbox 360 recognizing the motion of the player. The data acquired from Kinect is low resolution and inherently noisy even though it is compelling compared to other commercially available depth cameras (18). Still, it is sufficient for the object capturing and reconstruction in the range where the measurement errors are the smallest (16). The two main problems with the scanned point cloud data are the holes in the data cloud and the scattered points due to the scanning errors. In addition, It is not possible to build solid objects from partial point clouds.

During the surface reconstruction, the unevenly divided points can cause holes in the final mesh. A bigger problem is the missing data from a large surface area, and because of that it is impossible to determine programmatically whether it should be solid or if there really is supposed to be a surface gap. Usually, the missing data results from the scans with less than a perfect camera positioning. The restrictions for the camera placement can arise from hardware limitations; Kinect, for example cannot, capture any data closer than 0.5 meters. (6.) Also, uneven surfaces are difficult to capture, since scans from every direction would be needed.

Typical artefacts in the acquired data are holes caused by the sensor restrictions or measurement error based noise (37). Both of these defects can cause outliers to a resultant cloud. Techniques have been developed to avoid these kinds of errors. One removes outliers with a robust algorithm and creates a solid object from the remaining points. The resulted object is intended to be solid even when there are large numbers of points outside the surface (31). Another technique tries to create methods able to learn called neural network based surface reconstruction methods (10).

When the visually best possible clone from the original object is required, coloring the resulted mesh is maybe even more important than the shape. There are three possible ways to color the mesh. The first one is to use an external material file, which refers to a color texture map file with the given surface parameters such as lighting. The second one is to directly add color data into each polygon in the mesh. The third one is to use 3D texture materials which consist of a set of sliced 2D textures. The resulting set of planes is blended together with the mesh. Each polygon in the mesh can obtain texture data from the corresponding part of the 3D texture cube based on the coordinates of the polygon (4). In this thesis, the polygon coloring technique is used. Finally, some methods are needed to export the produced mesh in a format that can be imported into a Tundra scene.

3 USED TOOLS AND TECHNOLOGIES

This work is based on open source libraries and frameworks. In this chapter, these technologies are presented with more detailed information.

3.1 Redmine

Redmine is a flexible open source project management tool published under the GNU General Public License v2. It is written with the Ruby on Rails framework. It is cross-platform and cross-database software. Its user interface is web-based and therefore it can be used with almost any client hardware from mobile browsers to desktop computers. (34.)

Project management tools are used for supervising the work, marking and tracking the changes in the tasks that have to be done and estimating the time needed to complete larger tasks. The Chiru project uses Redmine for the management.

3.2 Git

Git is an open source distributed version control system, originally created by Linus Torvalds for the Linux kernel development management. It is designed for the speed and efficiency on large projects. (35.) With Git, each developer always has a full local copy of the repository with the development history. Therefore, a network connection is not mandatory for changing branches or for viewing the commit history after the repository is loaded to a local computer.

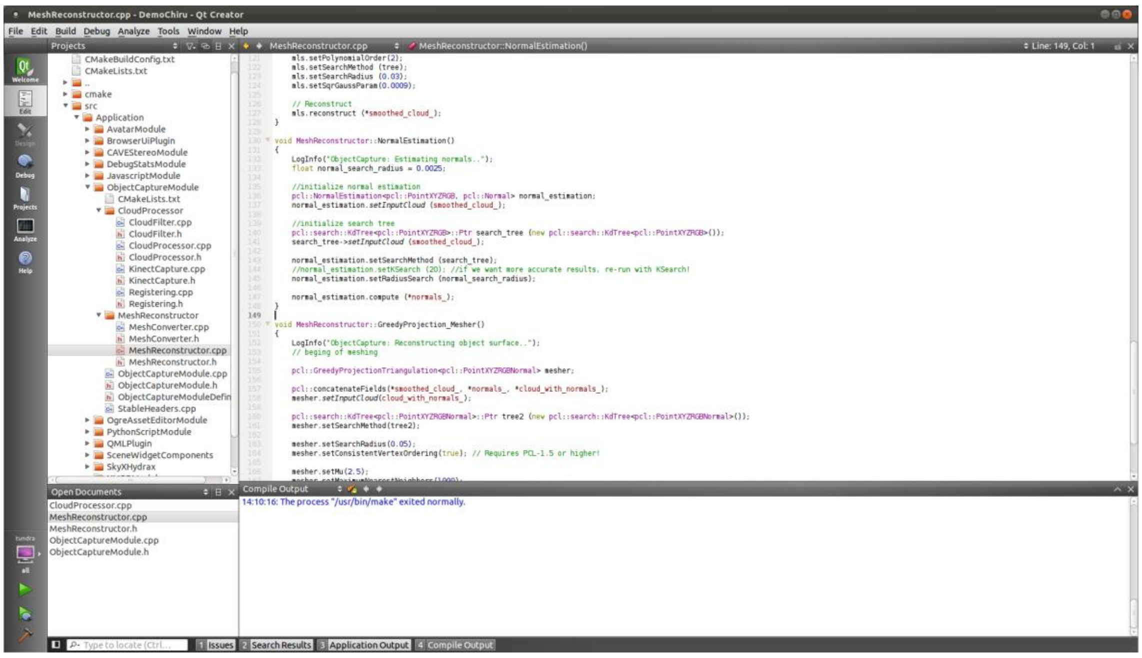
In the Chiru project, Github was used as a central repository server, storing all the data in a cloud. Even if a local hard drive was to break or the computer stolen, all data would be protected. In Github, it is possible to store projects in the Git format and from there the data is usable with the Git tools. Github is free for an unlimited number of developers for public projects, without any storage capacity limitations. Only when a private repository is needed, the service has a price tag. (9.)

3.3 Qt Framework and Qt Creator

Haavard Nord started to develop the Qt framework in 1991, and the first version became available in May 1995. The most famous feature of Qt is a concept for signals and slots, and it came up in 1992 by Eirik Chambe-Eng. From the very beginning, one of the main goals of Qt has been to create a framework that can be run natively with GUI on Unix, Macintosh and Windows.

Qt has always had dual licensing models, one for open source products and another for commercial use. (3.) Currently, Qt is licensed under the GNU Lesser General Public License (LGPL) version 2.1 by Nokia. Additionally, its commercial licensing is maintained by Digia and it is called Qt Commercial. (11.) The realXtend project is build by using the Qt framework with the open source licence.

Qt Creator is a development environment designed for the Qt based software development and it is deployed with the Qt framework. Qt Creator (Figure 1) supports the highlighting for multiple programming languages, and projects can be created either with qmake or with cmake.



```
    mls.setPolynomialOrder(2);
    mls.setSearchMethod (tree);
    mls.setSearchRadius (0.03);
    mls.setSqrGaussParam(0.0009);

    // Reconstruct
    mls.reconstruct (*smoothed_cloud_);
}

void MeshReconstructor::NormalEstimation()
{
    LogInfo("ObjectCapture: Estimating normals..");
    float normal_search_radius = 0.0025;

    //initialize normal estimation
    pcl::NormalEstimation<pcl::PointXYZRGB, pcl::Normal> normal_estimation;
    normal_estimation.setInputCloud (smoothed_cloud_);

    //initialize search tree
    pcl::search::KdTree<pcl::PointXYZRGB>::Ptr search_tree (new pcl::search::KdTree<pcl::PointXYZRGB>());
    search_tree->setInputCloud (smoothed_cloud_);

    normal_estimation.setSearchMethod (search_tree);
    //normal_estimation.setKSearch (20); //if we want more accurate results, re-run with KSearch!
    normal_estimation.setRadiusSearch (normal_search_radius);

    normal_estimation.compute (*normals_);
}

void MeshReconstructor::GreedyProjection_Mesher()
{
    LogInfo("ObjectCapture: Reconstructing object surface..");
    // beging of meshing

    pcl::GreedyProjectionTriangulation<pcl::PointXYZRGBNormal> mesher;

    pcl::concatenateFields(*smoothed_cloud_, *normals_, *cloud_with_normals_);
    mesher.setInputCloud(cloud_with_normals_);

    pcl::search::KdTree<pcl::PointXYZRGBNormal>::Ptr tree2 (new pcl::search::KdTree<pcl::PointXYZRGBNormal>());
    mesher.setSearchMethod(tree2);

    mesher.setSearchRadius(0.05);
    mesher.setConsistentVertexOrdering(true); // Requires PCL-1.5 or higher!

    mesher.setMu(2.5);
```

```
14:10:16: The process "/usr/bin/make" exited normally.
```

FIGURE 1. User interface of Qt Creator

3.4 RealXtend

RealXtend is an open source framework for virtual worlds and the name is owned by the realXtend Foundation. The development of realXtend began in 2007 and it was officially launched a year later. RealXtend is licensed under the Apache 2 license and therefore anyone can use its source code freely, even in commercial closed source projects. A copy of realXtend licence is presented in Appendix 1. The initial funding came from the City of Oulu, private investors and local companies. The development work has mainly been done in Oulu. The idea is originally from Juha Hulkko, who wanted an open source virtual world platform to compete with the restricted, closed source systems. (12.)

From the beginning of the development, the target was to create a generic platform that could be easily modified to fulfill for specific needs. The architecture of the Tundra scene is designed to be extensible in a way that nothing is hardcoded as opposed to OpenSimulator with the Second Life (LLUDP) protocol, which is mostly predefined into the platform. The functionality of the scene is defined by the entities in it. Each entity can contain multiple entity components, which then defines the functionality of that entity. For example, when the entity component EC_RigidBody is a part of the entity, it also belongs to the physics world and is under the laws of physics defined in the scene. (2.)

In 2011 the realXtend association was created. The association coordinates the collaboration and development work between the interested parties. Users, developers and companies can propose the needed functionality or sub-projects to the association and it can suggest these proposals to the Foundation. The Foundation then starts to find funding for these suggestions. Currently, there are seven member companies in the association (32). Similar organizational setups are also used in Blender and Mozilla. (27.)

The current incarnation of the realXtend is Tundra, which is a combined server and viewer application for virtual worlds. In spring 2012, the realXtend Tundra has reached to release the version 2.3.2. A Tundra client (Figure 2) is connected to the technology demonstration scene made by Ludocraft.

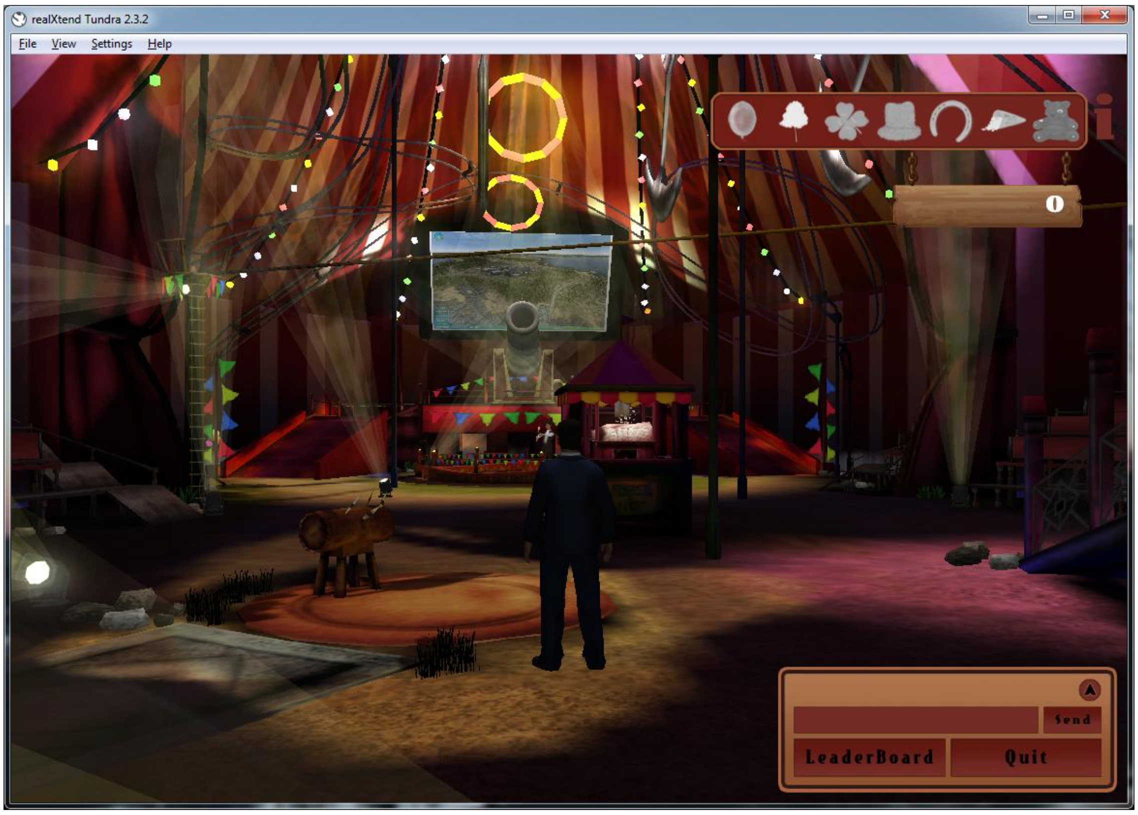


FIGURE 2. RealXtend Tundra client, connected to Ludocraft Circus scene

3.4.1 Tundra Module

In Tundra, it is possible to extend the functionality by creating a plug-in type of module. The Kinect module is an example of the extensions made such as a module, making it possible to guide an avatar in a virtual space with a movement captured by the Kinect. Other modules available are, for example, scripting support for JavaScript and Python.

At the startup it can be defined which modules are loaded. With this design, the memory footprint of the running software and the startup time are significantly lower, compared to a case where all available modules would be loaded at the startup. (33.)

3.4.2 Ogre

Ogre (Object-Oriented Graphics Rendering Engine) is used in Tundra as a graphics rendering engine. As the name says, it is a rendering engine as opposed to a game engine. It is maintained and developed by a relatively small core team, but a growing community also contributes new features to it. Since Ogre is not a scripting kit, it requires more knowledge of the programming than commercial game engines, at least intermediate skills in C++ and in the object oriented programming are essential. (17.) Currently, Ogre supports Direct3D and OpenGL as rendering targets. The Direct3D support includes the DirectX versions 9 and 10 (20), but there is a preliminary support for DirectX 11 as well, and it should be included to the next version of Ogre. There is no official estimation for the release date but unofficially it will be released by the end of 2012. The version number is going to be 1.8, also known as Byatis. (22.) Ogre is cross-platform software, which can be used with Windows, Linux and Mac OSX.

Since most of the OpenGL code is integrated into Ogre, it offers a possibility to use a lower level OpenGL Application Programming Interface (API) together with an easier-to-use high level API. A part of this higher level API offers a possibility to create 3D objects directly from the program code and then reuse those objects with every rendering frame without a need to specifically redraw them. In Ogre, this functionality for the object creation is wrapped inside of ManualObject class (21). Self created objects can also be serialized and saved into a hard disk with build-in functionality, if needed.

The Ogre scene is an abstract representation of a 3D environment and it may contain various components, for example, static geometry such as terrain, meshes such as chairs or avatars, lights for illumination of the scene and cameras for viewing the scene (23). There are different kind of scenes varying from indoor scenes mainly consisting of hallways and rooms populated with a furniture and artwork to large outdoor scenes consisting of a terrain of rolling hills, trees and grass.

3.5 Point Cloud Library

The Point Cloud Library (PCL) is an open source project for point cloud processing. It is released under the BSD licence and it is free for commercial and research use. PCL is aimed to be a cross-platform library and it is successfully compiled and deployed on Linux, Windows, MacOS and Android. (28.)

PCL provides many state-of-the art algorithms for point cloud processing. PCL is designed to be a fully templated, modern C++ library written with the performance and efficiency in mind. The data structures used in PCL use SSE optimizations, and with the third party libraries it supports the multi-core parallelization. The algorithms in PCL are designed to work together in a way that the result from the first method can be passed on to the next processing method. The data is passed around using the Boost shared pointers, and therefore no data copies are needed when the data is already present in the main memory of the system. (1.)

For the data acquisition, PCL provides the OpenNI (Open Natural Interaction) wrappers API, which makes it possible to use many publicly available capturing devices such as Primesense PSDK, Microsoft Kinect, Asus XtionPro and Live (25). PCL can also export and import multiple different data formats with its import and export implementations. This design makes it possible to capture data with a different computer than from the one used for the final data processing. (29.)

PCL is integrated with the Visualization Toolkit (VTK) to achieve its own visualization library. With this library, it is possible to quickly test and prototype the results received from the point cloud processing. The visualization works with n-dimensional point cloud structures. In this thesis, the visualization library was used to visualize the point data, the wire frame model from the reconstructed data and the surfaces created with the surface reconstruction. The properties and settings available for the visualization are, for example, colors for points and background, point size and opacity.

PCL is also shipped with a pre-build dedicated viewer program for the point cloud visualization called PCD Viewer. With the PCD Viewer it is possible to select what data the user wants to be visible. For the polygon meshes, different combinations for inspection are vertex data with and without colors, a wireframe model or a model with surfaces. Also, the surfaces and wireframe model can be colored with the color data specified in the vertex.

The development of PCL is financially supported by many companies including Willow Garage, NVIDIA, Google and Toyota. The development also happens in other organizations and companies geographically distributed around the world, for example by Intel, Icar, Australian National University and Eindhoven University of Technology. (28.) Since the PCL provides all required functionality from the data acquisition to the surface creation, it was selected to the implementation of this thesis.

3.6 Meshlab

Meshlab is an open source project designed to be a portable and extensible system for processing and editing the unstructured 3D triangular meshes. The Meshlab is licensed under the terms of the GNU General Public License (GPL). Even though it is free to use, everyone using it in an official or a commercial project should explicitly cite that Meshlab has been used in that project. Also, a small description about the project it was used for should be posted to the Meshlab user forums. (14.)

From the beginning of the Meshlab development, there have been three main goals in mind; ease of use, so that it can be used without a strong knowledge about the 3D modeling; scanning oriented, meaning that it should focus on the mesh processing instead of the mesh editing; and efficiency so that it should be capable of handling millions of primitives which often is the case when scanned objects are involved. (7.)

Meshlab is based on the VCG library and it has been developed at the Visual Computing Lab of ISTI-CNR. The project started in the late 2005 with a funding from European Commission, and the initial work was mainly done by a handful of students. Many of those initial developers continued working with Meshlab even after their studies did not require it anymore. (39.) Currently, Meshlab provides a full tool chain for the object recreation from the scanner raw data to the final clean ready-to-be-used 3D models (7).

The main features and tools of Meshlab, listed by the developers, include various kinds of filters, and more can be done if those do not fulfill the requirements. The mesh cleaning filters are made for the removal of duplicated, unreferenced vertices, null faces and small isolated components. The remeshing filters provide edge collapse simplification functionality with a possibility to preserve the texture coordinates.

For the surface reconstruction, there is a whole toolset. With the toolset the surfaces can be recreated from the points, and the surfaces can also be subdivided. The merging of multiple meshes is possible with the surface reconstruction tools. Under the remeshing tools there are also filters for feature preserving smoothing and for hole filling. For the mesh healing, Meshlab provides interactive mesh painting tools for color painting, selection painting and smoothing.

Last but not least, the 3D scanning tools in Meshlab provide a possibility for the alignment with the Iterative Closest Point algorithm based on the range map for putting the meshes into the same reference space and register different raw range maps. The rendering included in Meshlab is based on the OpenGL Shaders making it possible to write an individual shader to suit the requirements precisely. (13.)

3.7 Polygon Mesh

The polygon mesh is a widely used data format to store 3D models, when those are being drawn with hardware. The elements of the 3D models are typically vertices, edges, faces, polygons and surfaces. Every vertex has its own position, and two vertices create an edge. Three edges create one face, hence the face is a closed set of edges.

A polygon consists of a set of faces, and if a system supports multi-sided faces, there is no difference between the polygons and faces. Usually, the rendering systems do not have a support for multiple sided faces, which leaves no choice than to present the polygons with multiple faces.

The surfaces, also called as a smoothing group, can provide additional data for the mesh calculations in the renderer. With this extra data, it is possible to create relatively round objects only from few faces. In the high resolution meshes, the smoothing groups are obsolete, since the faces are already so small. In a polygon mesh, each polygon is constructed from three vertices and each vertex can be part of the multiple polygons (Figure 3).

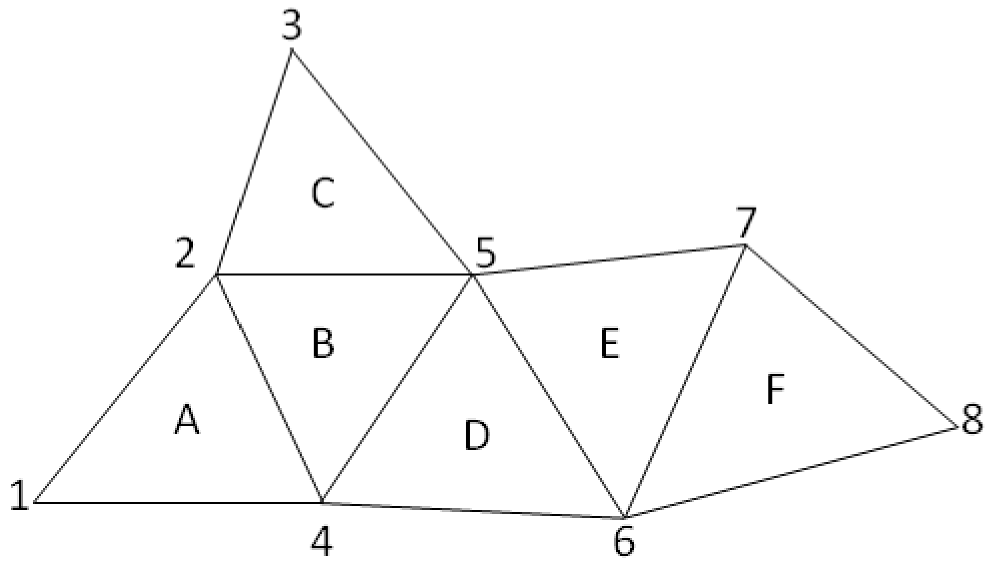


FIGURE 3. Polygon mesh visualization. Faces are marked with alphabets from A to F and vertices are marked with numbers from one to eight.

In Figure 3 it can be seen that each face of the polygon mesh is composed from three vertices. For example, face A is constructed from vertices one, two and four. Also, it can be seen that most of the vertices are a part of the multiple faces, for example vertex two is a part of faces A, B and C. (5.)

4 POINT CLOUD PROCESSING METHODS

Implementation of the used cloud processing methods is modular, and therefore the methods can be easily changed. Some of these methods, presented here, were used in the final implementation, and some only during the research stage. In this chapter, these different processing algorithms are described only briefly.

4.1 Point Cloud

A point cloud is a data structure holding multi-dimensional data in it. Most commonly, the point clouds are used to represent the three-dimensional data and usually those are X, Y and Z coordinates. The common usage for the point data is to represent an outer surface of some object. Beside the geometrical coordinates, the point clouds can contain color data for each point, normal directions or both. When data is added to a point, the cloud dimensions increase. When the color data is added to the three-dimensional geometric cloud, it becomes four-dimensional.

Stanford University have collected various point clouds of the laser scanned objects into their 3D scanning repository, since not all the interested research parties had the possibility to get laser scanners back then. (40.) All point clouds in the repository are free to use for research purposes. The most famous scanned object in the scanning repository of Stanford University is the Stanford bunny. The bunny was originally scanned in 1993 and it is widely used by different studies as a test point cloud.

Industrial quality equipment is not mandatory anymore for the object scanning, since with the current consumer class equipment, such as Microsoft Kinect or Asus XTionPRO, it is possible to acquire point clouds with a decent accuracy. Therefore, the point cloud acquisition has got more interest from curious developers.

4.2 Point Cloud Filtering

The clouds can be filtered with various different ways. The methods used in this thesis are based on the range filtering or density filtering.

In the range filtering, the decisions are solely made based on the point distance from the camera. In this filtering method, the points outside of the given range are removed from the resulted cloud. A common way to implement this kind of filtering technology is to use a pass through method.

Density filtering is an abstract name for a point cloud filtering step and it can be done with several different algorithms. In this step, the point cloud density is usually reduced based on a range between the points in a cloud. This usually reduces the cloud size significantly with the object still being recognizable.

In the pass through filtering, all points with matching values in a predefined value range are passed through without any modifications to their data. The pass through filtering is one of the basic filtering methods and it is fast and simple to use. It is implemented in PCL as a template class and therefore supports all predefined point types. (29.)

In the voxel grid filtering, the original point cloud is divided into tiny boxes with a defined leaf size. The amount of filtering can be controlled with the leaf size. The average values from the points inside every box are calculated and a new point cloud is produced from the values. The resultant point cloud has as many points in it as there are boxes in the original cloud. (30.)

4.3 Moving Least Squares

The Moving least squares (MLS) is a smoothing algorithm for the point cloud data. It is used for approximating the original surface presented by the vertices in a local neighborhood in the point cloud. MLS is considered to belong to a class of the meshless interpolation methods, and there are several competitive ways to implement the MLS method (19). All those methods try to achieve the noise reduction in the point data which represents the surface. MLS used in PCL does its approximation by using a bivariate polynomial height function, which is defined on a reference plane. Also, the normal values for each point in the cloud are calculated and stored for the later use during the MLS processing. (26.)

4.4 Normal Estimation

In the computer graphics, the surface normals are used for generating the correct shadows based on the lightning. In the surface reconstruction, the normals define the direction of the polygon faces. The normal estimation of the geometric surface is usually a trivial task, but the points in the point cloud do not have any surface, they only represent one. For this reason, the task is non-trivial.

There are basically two different approaches for the normal estimation. The first one is to reconstruct the surface, which the points represent, and calculate the normals from that. The second one is to approximate the normal data directly from the point cloud. Since many methods doing the surface reconstruction need normals as their input data, the estimation is commonly done directly from the point data. (36.)

4.5 Greedy Projection Triangulation

The Greedy projection triangulation works by keeping a list of the possible points, where a mesh can be grown. This method continues to extend a mesh as long as there are points available. The Greedy projection triangulation can handle the unorganized point clouds coming from multiple scans with multiple connected parts. The best results are achieved in the situations where the processed cloud is locally smooth and the transitions with different density areas are also smooth.

In PCL, the triangulation projection is working locally but some parameters can be set to specify the wanted features. There are parameters for setting the neighborhood size for search, the search radius for the maximum length of the edge as well as the minimum and maximum angles for the triangles. In cases with sharp edges, the maximum angle between the surfaces can also be specified. Also, in some cases it is essential that all the surfaces are facing the same direction. In this case, the normal data can be forced to be consistent in the dataset. Appendix 2 presents a picture of a polygon mesh with the inconsistent normal data, and therefore all surfaces are not facing the same direction. (30.) The Triangulation algorithm itself could fairly easily support the parallel processing, but it is not implemented in PCL and it is not known if the implementation of a multithreaded version is even planned (24).

5 IMPLEMENTING OBJECT RECONSTRUCTOR

Before choosing which algorithms to use in the implementation, they were first tested in a standalone application. If the algorithm worked and the results were adequate, it was added to the Tundra integration as well. The algorithm testing was first done with different surface reconstruction methods with various different point clouds. This approach provided valuable information about the initial performance of the algorithms available, and based on that information, it was decided if any additional processing steps were needed.

In the beginning of this work, the actual acquisition code was not in a usable state yet, so all of the initial tests were done with the high quality laser scanned point clouds from the 3D repository of Stanford University.

It is not recommended to use extremely accurate clouds for reconstruction tests in cases where the actual input data is much more inaccurate. For this reason, the testing was changed to use the raw clouds acquired by using Microsoft Kinect as soon as possible. An ideal situation for the object capturing is when the object is about one meter away from the Kinect camera (Figure 4).

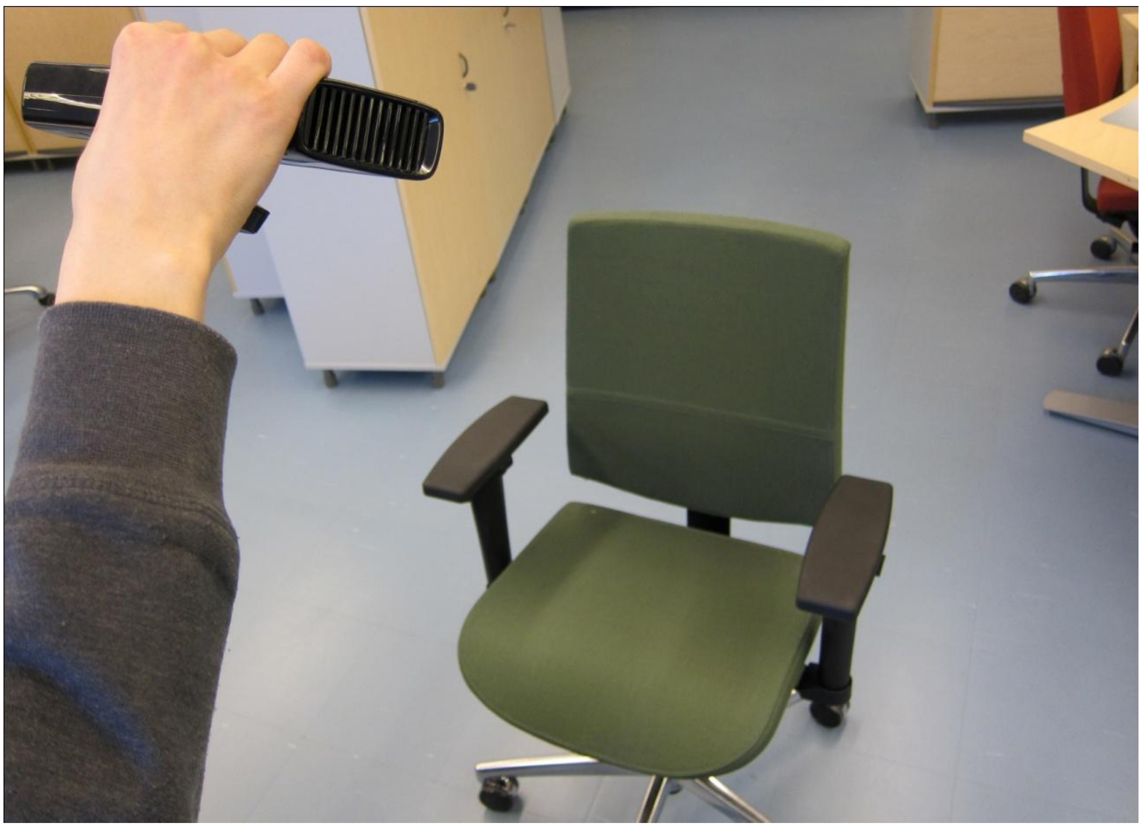


FIGURE 4. Picture of an office chair capturing situation

5.1 Cloud Filtering

The final filtering process implementation was done by a co-worker but it is briefly explained, since it is an important part of the object capturing work flow and therefore also of the surface reconstruction (38). Even though the filtering is an important processing step for the real use case, all surface reconstruction methods described here can be applied also to the unfiltered clouds.

With the point cloud filtering two main problems had to be solved. Filtering is used to reduce the time needed for the surface reconstruction and finding the object of interest. An unfiltered point cloud captured with Microsoft Kinect (Figure 5) contains 307200 points in total, but some of the points may contain only color data. If the position of the point is unknown in the depth field, it cannot be visualized in a 3D environment. Therefore, only the points including also depth data are visualized.

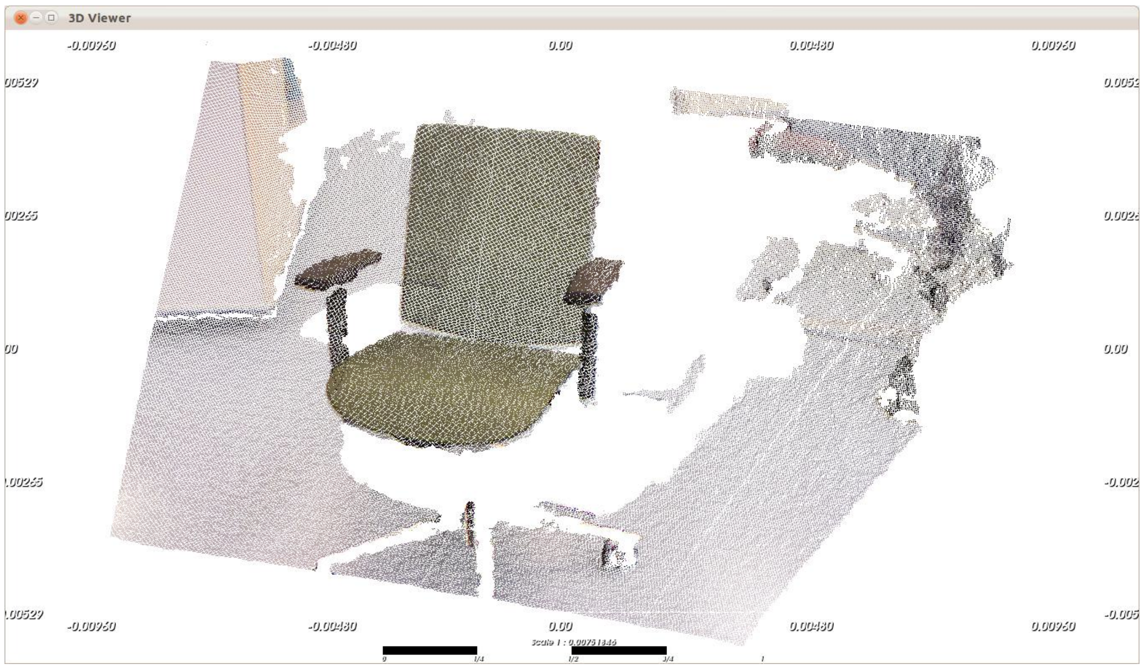


FIGURE 5. Unfiltered point cloud captured with Microsoft Kinect and visualized with the visualization library of PCL

During the filtering process the following three different filtering methods were applied:

1. Filtering by depth
2. Filtering by density
3. Cluster extraction

In the depth filtering state, all points that are further than 1.4 meters away from the camera are removed. The pass through filtering was used for the implementation of this step. The minimum range of the Kinect sensor is 0.5 meters, thus the points from the range of 0.5 meters to 1.4 meters were preserved.

The density filtering was done by using a voxel grid filter. The voxel grid leaf size was set to 0.005, which corresponds to 0.5 millimeters in the real world. After the density filtering, the number of points was reduced by 81.1% in average.

The third and the last applied filter is a cluster extraction, which is achieved with the point cloud segmentation. The segmentation removes the largest planar component found from the point cloud after the depth filtering. It is highly likely that this planar is either a floor or a table under the object of interest. In the tests, this method worked in most cases. Only in some rare cases, partials from the floor were left to the final point cloud. Also, sometimes the floor part of the cloud was falsely recognized to be an object of interest. The final point cloud after all the filtering methods were executed (Figure 6) is the input data for the mesh reconstruction.

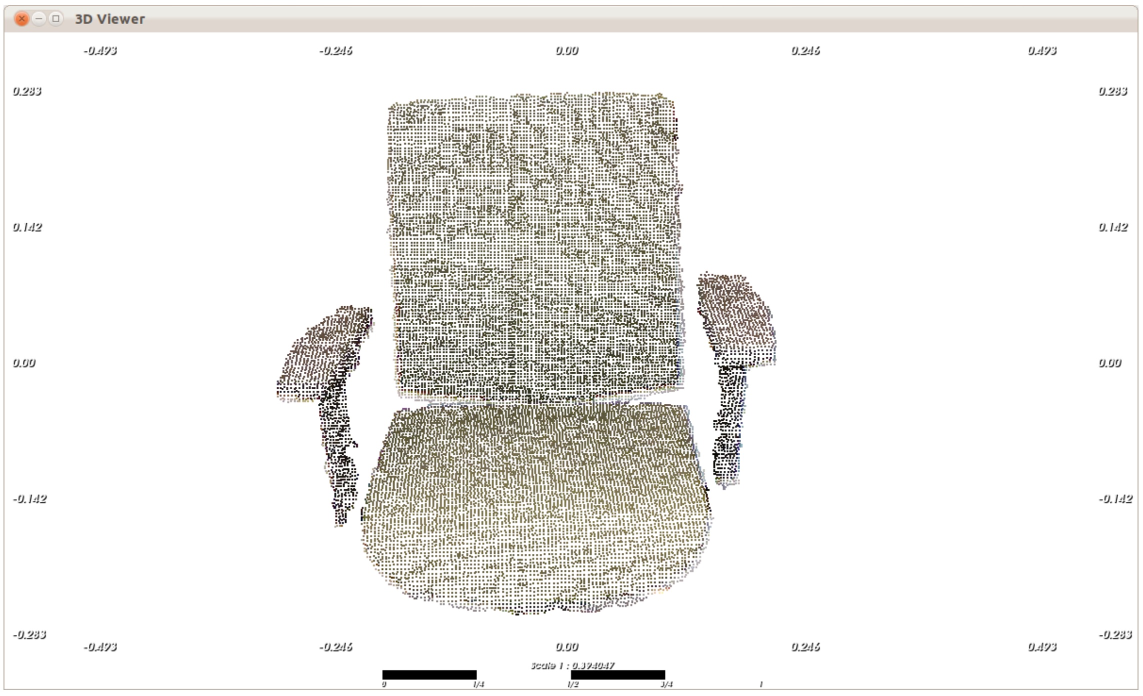


FIGURE 6. The final point cloud after all filtering methods are applied. The point cloud has 22539 points total in it.

5.2 Mesh Reconstruction

The mesh reconstruction is divided into two phase procedure, where the first step is cloud smoothing and the second is surface reconstruction. From the general data flowchart of the implementation (Figure 7), the phases can be seen more accurately. The point cloud smoothing is done with the MLS algorithm and the surfaces are reconstructed from the smoothed point cloud.

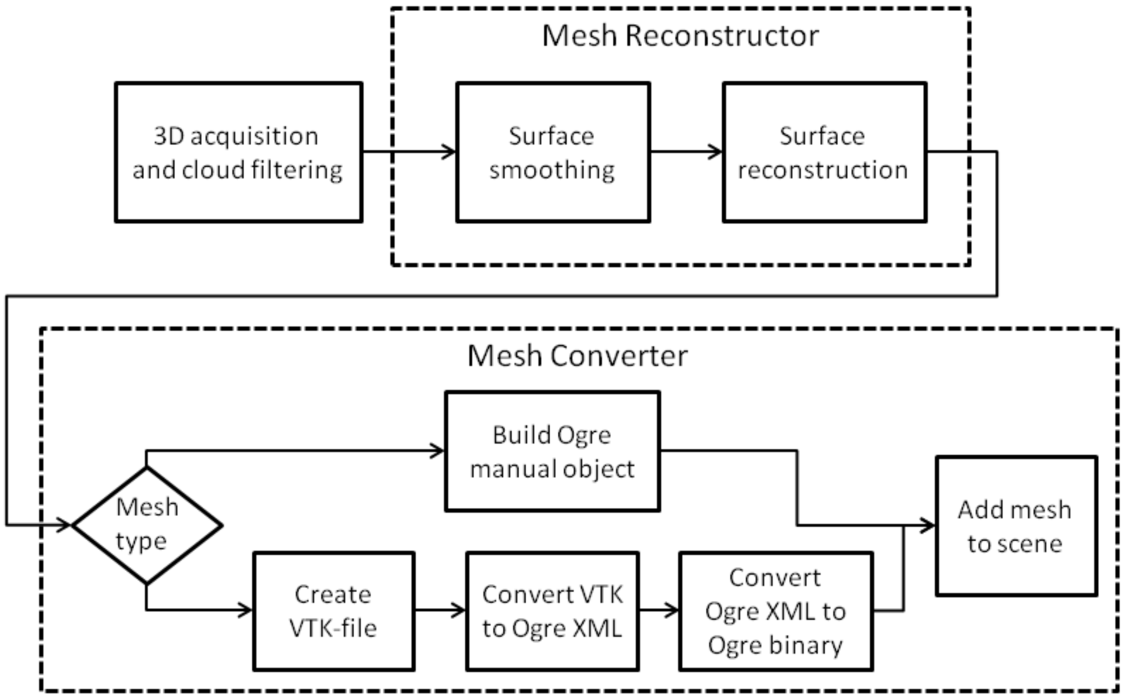


FIGURE 7. General data flowchart of the implementation

In the implementation, the decision in which the mesh conversion method was used had to be done in the program code by changing one signal connection inside the ObjectCaptureModule and it could not be changed during the runtime.

5.2.1 Surface Smoothing

The MLS algorithm was chosen for the smoothing, because the initial test results with it were very good and it was well documented. The smoothing was an optional step, but it was selected for the data path since it made the final meshes look much better. As a comparison, in an unsmoothed mesh reconstruction (Figure 8) the edges are rough and the surfaces are not clean.

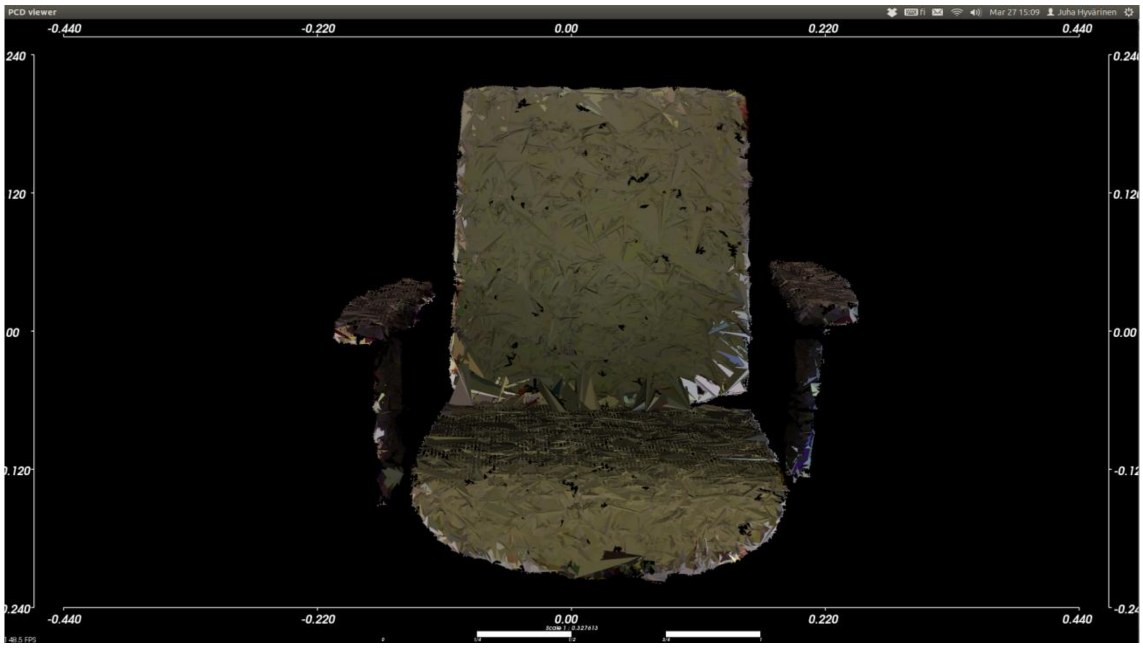


FIGURE 8. Polygon mesh created from captured point cloud and meshed with greedy projection triangulation

The MLS algorithm calculates the real position for the surface by estimating possible errors in the position data based on the positions of the neighboring points. All points in the neighborhood are then moved on the estimated surface. When the search radius is too large, the shape of the original surface is most probably lost during the MLS processing.

The MLS processing needs XYZ-data as an input and the all other inputted data fields are preserved. Optionally, it is possible to calculate the normal directions from the point data in the cloud with the MLS algorithm. The decision is left for the programmer whether the directions are wanted to be calculated with the MLS algorithm. They can also be calculated somewhere else if needed. Since the used MLS process supports the multithreading, it was decided to calculate even the normal data with the MLS algorithm.

The parameters needed by the MLS algorithm were tested manually to find which values gave the best results with the used input data. Some initial values for the testing were found from the PCL tutorials. The used parameters in the MLS process are presented in Appendix 3.

The MLS processing was done with a version supporting the multithreaded processing. That way the processing time could be somewhat reduced. The resulting clouds, meaning the smoothed clouds and normals for each point, are combined into one point cloud containing five dimensions, which includes the following data: the point positions in the X, Y and Z axes, the point color data in the RGB format and the point normal data.

5.2.2 Surface Reconstruction

For the surface reconstruction, it was decided to use the greedy triangulation algorithm provided by PCL. The input point cloud for the triangulation is the output point cloud from the MLS process. In the greedy triangulation process, the surfaces are estimated by creating triangles from the point data. Each created triangle represents one face in a final polygon mesh.

The usual number of the points in the input cloud in the test scenarios was between 20 000 and 30 000 points. Still, it needs to be remembered that only one input cloud was used in the tests and the number of the points will be much higher if the multiple clouds are combined. Hence, the filtering should be more aggressive if the input consists of more than one cloud.

The finalized mesh is presented in Figure 9, after the MLS algorithm has been used for the point cloud smoothing and the surface reconstruction has been done with the greedy projection triangulation.

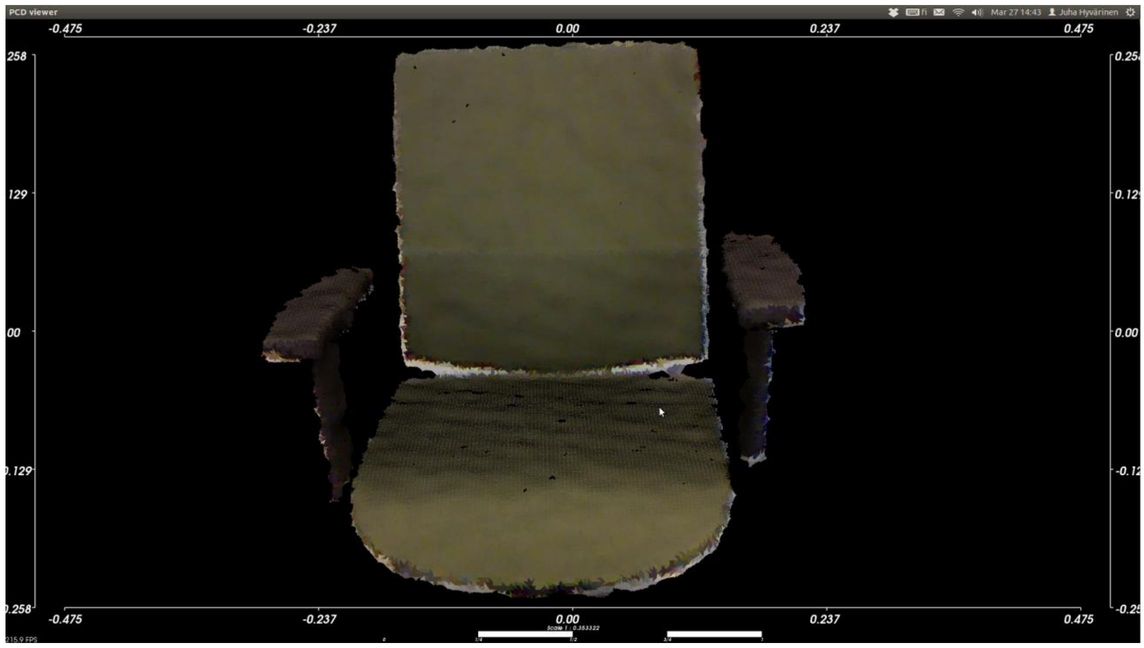


FIGURE 9. Polygon mesh created from smoothed point cloud and surfaces reconstructed with greedy projection triangulation

5.3 Mesh Conversion

The polygon mesh is converted either to a VTK file, which is written to the hard disk or to an Ogre ManualObject, which can be imported directly to the Tundra scene with the memory copying. The implementation using the VTK files was done first because it was a more straight forward method and the working demonstration was needed quickly. This was done even though it was known from the beginning that using system calls from the program code to execute the third party executables and scripts is bad design. In the mesh converter, the earlier constructed polygon mesh is transformed to an Ogre mesh and there are two possible ways to achieve this conversion.

5.3.1 Mesh Conversion with External Tools

The polygon mesh, which resulted from the mesh conversion with external tools, was saved to the disk in the VTK file format. Saving the polygon mesh was done with the exporting functionality in PCL. The supported file formats for the exporting were VTK and Point Cloud Data (PCD). The VTK format was chosen to the implementation, since the scripts implemented earlier for other tasks were extended relatively easily to support the VTK file format for the data conversions as well.

The conversion from the VTK file to an Ogre XML file was done with a Python script. The script was written earlier by a co-worker for another task and it was extended with the VTK file reading support. The Ogre XML file was then converted to the Ogre binary format with a command line tool, called OgreXMLConverter, which was bundled in the Ogre tools. Also, this conversion was first done by reading the XML file from the hard disk and after the conversion written as an Ogre binary file back to the hard disk.

5.3.2 Mesh Conversion with Ogre ManualObject

Another option for the polygon mesh conversion to the Ogre mesh format is implemented by using the Ogre ManualObject class. The difference to the external tool chain is that with ManualObject all conversions can be done in the main memory.

Three steps are needed for the data conversion from the polygon mesh to the Ogre ManualObject:

1. The vertex color data from the input cloud has to be normalized because the Ogre ManualObject uses the floating point values between 0.0 and 1.0. In the polygon mesh, the color data is saved as an 8-bit unsigned integer, meaning it can have values between 0 and 255.
2. The vertex position data, normalized color data and normal data have to be added to the ManualObject. This data is read from a smoothed point cloud, which was combined with the point normal data.
3. The vertices in the ManualObject have to be indexed; otherwise it could not be possible to convert the ManualObject to an Ogre mesh. The indexing data is read from the polygon mesh and it is added to the ManualObject after all vertices have been placed.

The ManualObjects can also be converted into an Ogre mesh, which can be saved to the hard disk with the serialization functionality of Ogre for later use.

5.4 Integration with Tundra

The implementation of this thesis was done as a Tundra module. The module had two data processing layers in it. The main layer handled the communications with Tundra, the user input and the messages from the scripting layer, which provided a primitive user interface for the object capturing. The architectural design of the integration is presented in Figure 10.

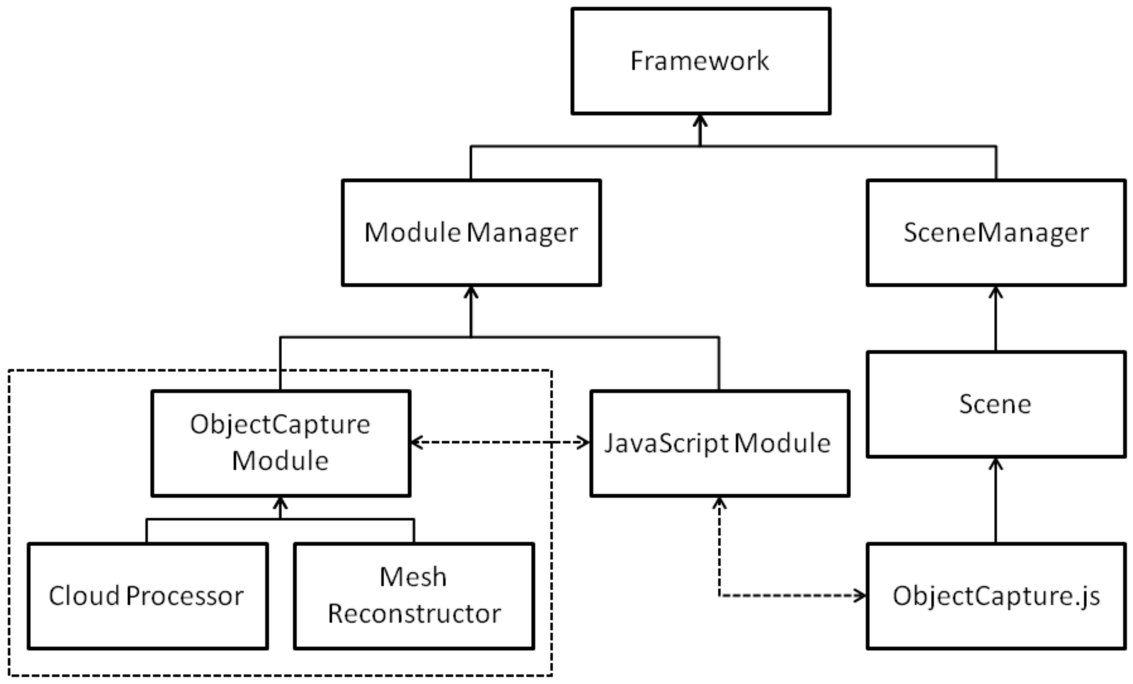


FIGURE 10. Architectural picture of ObjectCaptureModule integration to Tundra

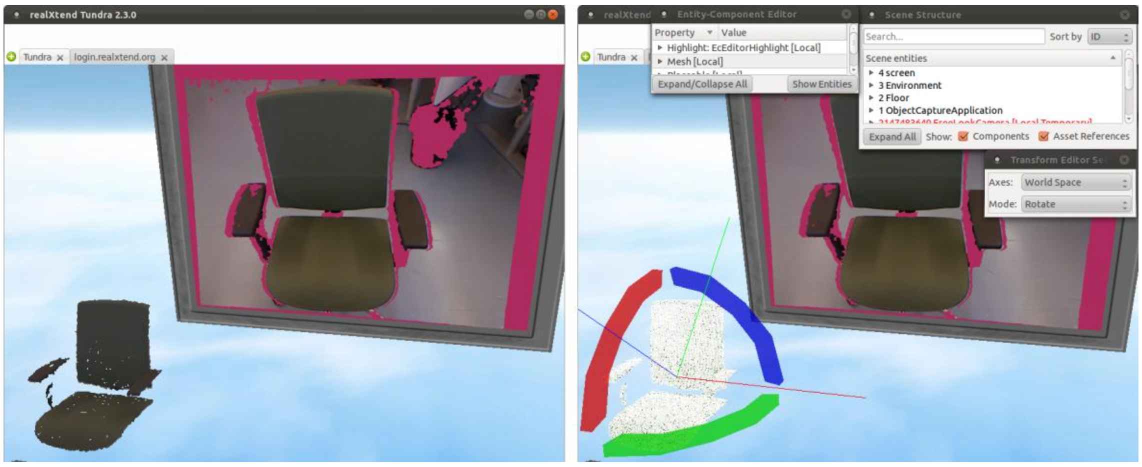
The point cloud capturing and the point cloud filtering were done in the CloudProcessor, which was done in a separate work (38). The surface reconstruction, the mesh conversion to the Ogre mesh and exporting it to a Tunda scene was done in the MeshReconstructor. The processed data and acknowledgments about the processing state were transferred between the classes with the Qt signals. All the data processing was done in a separate thread to ensure that the program was usable all the time without freezing the user interface.

The data from the CloudProcessor was passed back to the ObjectCaptureModule, which then forwarded the data to the MeshReconstructor. The mesh reconstruction was implemented as a separate class and it was designed to work independently after the initial data was given for the reconstruction. This design made the multithreading easy.

The final transformation from the polygon mesh, created with PCL, was done in the mesh converter class. Within the mesh converter, the earlier reconstructed polygon mesh was transformed to the Ogre ManualObject. During this step the vertices are copied from the polygon mesh to the Ogre ManualObject with normalized color values for each vertex, and then the polygons are indexed.

A new entity was created to the Tundra scene by the MeshConverter and it also added all the entity components needed for the visualization of the reconstructed object. These entity components were EC_Placeable and either EC_OgreCustomObject or EC_Mesh, based on which conversion method was used. The EC_Placeable identifies the location of the entity in a Tundra scene. The EC_OgreCustomObject and the EC_Mesh make it possible to represent a mesh in a Tundra scene. The EC_OgreCustomObject converts the Ogre ManualObject to the Ogre mesh internally.

Also, a small scene was created for testing and demonstration purposes. When an object was added to the scene, a user could manipulate the created mesh with tools provided by Tundra, which included the possibility to rotate, move and scale the object. The object selected for the editing was highlighted and the manipulators for the object editing appeared around the object. Figure 11 is from the test scene after an office chair has been captured with the Microsoft Kinect.



Axes: World Space
Mode: Rotate

FIGURE 11. On the left side there is a reconstructed object in the Tundra test scene. The Kinect output is shown on the gray screen object in the scene. On the right side the reconstructed object is being manipulated with the tools provided by Tundra.

6 RESULTS

The surface reconstruction from a point cloud and conversion from a polygon mesh to the Ogre mesh format was implemented to Tundra. With the toolset provided, it is possible to capture simple objects to a virtual space with the Microsoft Kinect. This process is highly automated and only few clicks are needed from the user. Earlier experience about the object capturing or advanced technical knowledge is not required.

With the implementation, recognizable objects can be created and imported to a virtual space. Since the integrated data acquisition only supports the capturing of one point cloud per object, there are some restrictions for the shape and the type of the captured object. The objects with a high number of small details which have to be seen, or a shape which needs scans from multiple directions, cannot be captured. The best results are achieved with the objects which do not have any shiny surfaces and a shape, which is recognizable even if only one capture has been taken. A better data acquisition has already been developed, but it has not yet been integrated to Tundra.

All measurements were run on a computer with an Intel Q8400 processor running at 2.66GHz, a main memory of 4GB, an nVidia Quadro 600 -graphics card and a hard drive with a capacity of 500GB. The operating system used in the tests was Linux Ubuntu 11.10. In a sense of the processing power, it represents an average computer, a few years old.

6.1 Surface Smoothing

The surface smoothing with the MLS process was tested in the Tundra client by capturing objects of different sizes with the Microsoft Kinect. The number of points in the final filtered point cloud varied from 3045 to 34789 points. The time consumed in the MLS process was measured for each test case. Figure 12 presents how long the cloud smoothing took with the MLS.

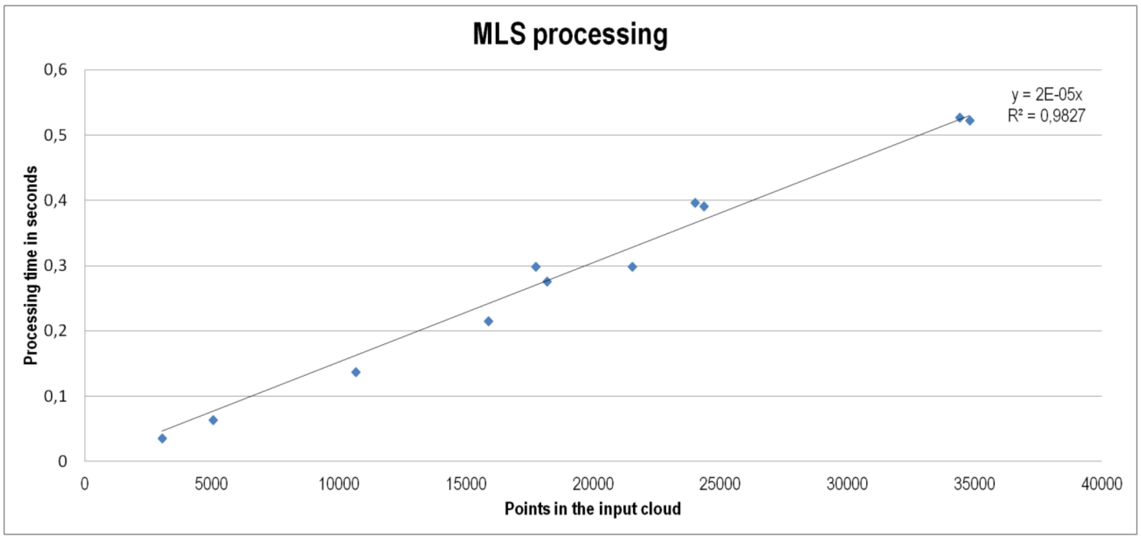


FIGURE 12. Time needed for the MLS surface smoothing

From the measured data can be seen that the time needed for the MLS processing grows almost linearly when the number of the points increases. The calculated correlation factor R^2 for the linear regression is 0.9827 out of a maximum value of 1.0, meaning that small variations with the measured results are most probably caused by measurement errors. The cloud smoothing of an office chair used as a test object took averagely 0.35 seconds.

6.2 Surface Reconstruction

The measurements for the surface reconstruction were done in the Tundra client. The input point cloud used for the surface reconstruction came from the MLS smoothing process. The time consumed in the surface reconstruction program code was measured (Figure 13).

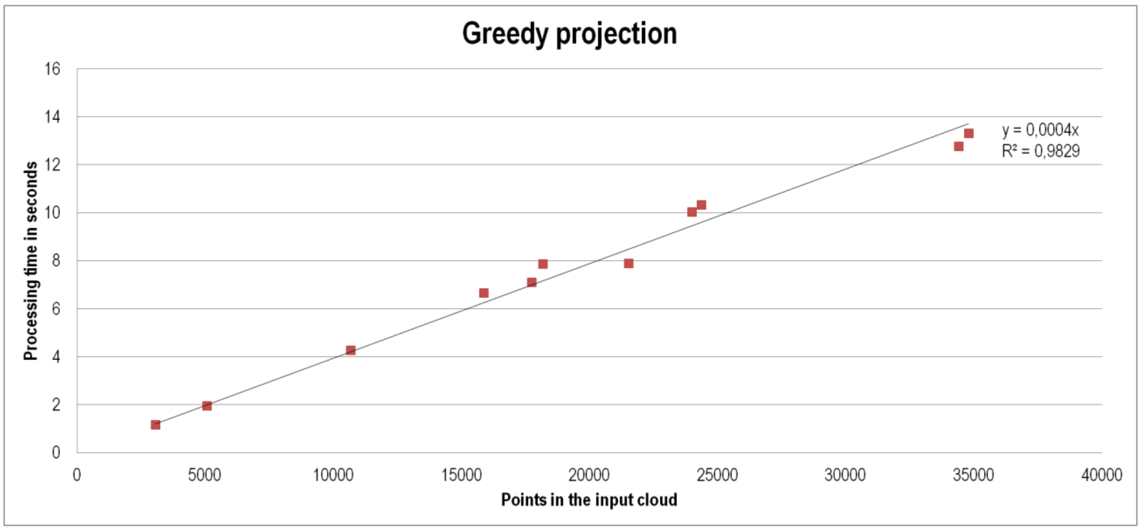


FIGURE 13. Time needed for surface reconstruction with greedy projection triangulation

The time needed for the surface reconstruction grows almost linearly when the number of the points increases. The surface reconstruction of an office chair used as a test object took averagely 10 seconds.

With the multithreading, the processing time of the greedy projection triangulation could be reduced significantly, since the algorithm itself should scale well with the parallel processing. The implementation of the greedy projection triangulation in the PCL did not support the multithreading.

6.3 Mesh Conversion

The conversion times were benchmarked with both implemented methods by running 11 different sized polygon meshes through the conversion methods. The time consumed by these processes was measured (Figure 14).

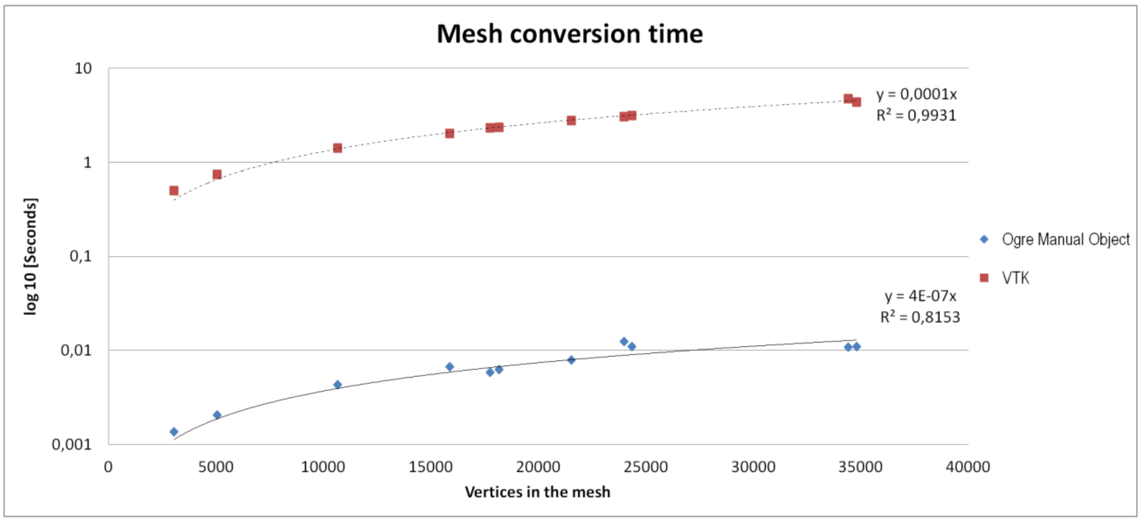


FIGURE 14. Mesh conversion times with two competitive methods, by using external tools and by creating Ogre ManualObject directly in memory

The time needed for the data conversion with the external tools grew linearly with the size of the input polygon mesh. The time needed for the processing can be estimated when the number of the vertices is known. The formula for this estimation is presented in Figure 14 next to the VTK curve. Most likely the conversion speed was limited by the speed of the hard drive, when saving the temporary files needed by the process.

As can be seen from the measurement results in Figure 14, the time needed for the Ogre ManualObject conversion grew almost linearly in the direct relation to the size of the inputted polygon mesh. The conversion time needed for the Ogre ManualObject creation was measured to be between 245 and 442 times faster than the conversion with the external tools. The conversion with the ManualObject is likely to be even more efficient if the processing times could have been measured more accurately. With such small processing times of the ManualObject, the background processes running on the same computer can have a large impact on the results.

6.4 Object Rendering

Since two different data paths were implemented from the polygon mesh to the Ogre mesh, even the rendering performance was briefly inspected in these cases. The tests were run in a simple test scene where there were two static objects and two captured objects. The captured objects were reconstructed from the same input cloud and therefore they both had exactly the same number of vertices and indices. The processing time needed for one Tundra frame was calculated from the average frame time values during a few seconds, in a situation where the mouse cursor was over the object. Only one of the captured objects was visible at the time. These frame times are presented in Figure 15.

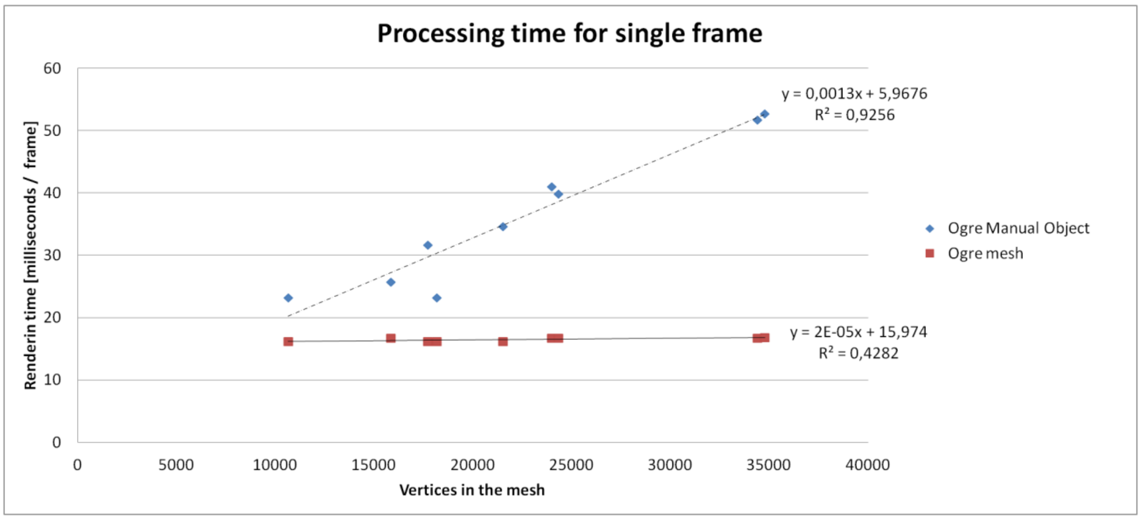


FIGURE 15. Processing time for single frame when number of vertices varies in rendered object

As it can be seen from Figure 15, the mesh created with the Ogre ManualObject needs significantly more processing time for the rendering. Furthermore, the time needed for the rendering grew linearly when a rendered mesh has more vertices in it. For some reason, the mesh converted from the VTK file did not have a similar behavior, and the rendering time was very close to the constant one regardless of the amount of the vertices. The reason for this behavior was tried to be solved with the build-in profiler in Tundra.

The results from the profiling showed that the slowest block was the EC_Mesh_Raycast. When the custom object was used, the EC_Mesh_Raycast consumed from 81% to 91% of the processing time per one frame. With a mesh loaded from the hard disk, the EC_Mesh_Raycast needed only from 8% to 24% of the processing time in a single frame.

Furthermore, if the Ogre ManualObject was serialized to the hard disk and then again loaded from there into the EC_Mesh, it was rendered as fast as the mesh converted from the VTK file. The reason for this remained a mystery, since there was no time to investigate that behavior within the timetable of this thesis. Most probably, the EC_Mesh and EC_OgreCustomObjects are treated differently by the ray casting.

The hardware culling was turned off from the Ogre material file to achieve the mesh being visible from all directions. By default, Ogre does the culling for the faces in the mesh in order to save processing time. When the culling is activated, only the front side of each face is rendered and therefore it is invisible from the backside. The Ogre material file used for the reconstructed objects is presented in Appendix 4. The same material file was used for both mesh types. The mesh color is saved per vertex since this way every face in a mesh has its own color based on the color defined in the vertices.

6.5 Saved Binary Meshes

It should also be noted that when the meshes are saved to the hard disk in an Ogre binary format, the version exported from an Ogre ManualObject is significantly smaller in size. The reason for this is unknown since the input data is exactly the same in both situations, and in the resulting mesh, there was no difference in the number of vertices or indices.

The VTK file is listed here as a comparison. It was used as the input data for the external tools converting method. The VTK was saved in a text format and others were saved as Ogre binary meshes. In Figure 16, the file sizes in the three used file formats are presented.

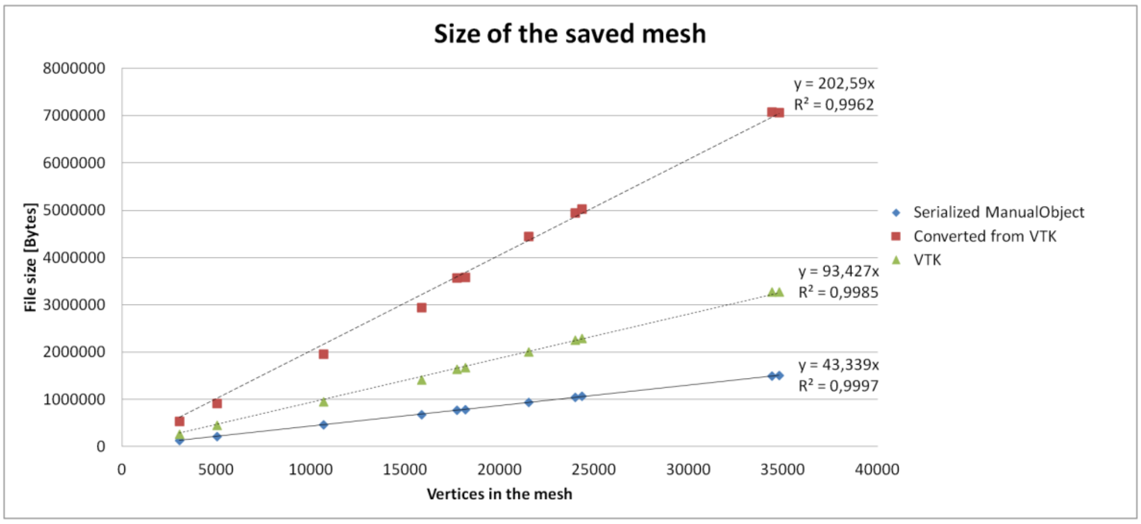


FIGURE 16. Mesh file sizes with different conversion methods

As expected, a mesh serialized from an Ogre ManualObject becomes linearly larger when the mesh contains more vertices. With the VTK file and the Ogre mesh converted from it, there was some variety in the size, and both of those were always larger than the mesh created from the ManualObject. The reason for such a large difference between the two methods, which should create identical data, could not be identified within this thesis because of the time limitations.

7 CONCLUSION

A working data flow for the object capturing was implemented. A simple real world objects can be captured to a point cloud and processed to the Ogre mesh. Created mesh can be imported to the Tundra scene quickly and easily. The whole toolset is implemented to the realXtend Tundra framework as a module. The processing time for the surface reconstruction of a captured office chair is about ten seconds.

The main problems which were found are associated to the polygon faces. In the PCL version 1.4, it is not possible to force all polygon faces into one direction, and for this reason, the mesh seems to be full of holes (Appendix 2). This problem was managed to get around by forcing the hardware culling off for the imported objects from the material file specified for the object. Basically it doubles the rendering time for a single object. PCL 1.5 introduces new options, which allow the forcing of all polygons facing to one direction. Another problem, which even the newer version would not solve, was that the direction for the faces could not be forced, and in the tests which was ran, those always faced to the wrong direction. Part of the problem was that the captured objects were only partials from the real world objects.

Future work will include investigations about the problems which were found during the measurements. As it was expected, the mesh conversion times from a polygon mesh to the Ogre ManualObject were significantly shorter than the converting times with the external tools. Even though, it was unexpected that there was such a large difference in the rendering times, since the input data was identical to both converting methods.

Optimizations for the rendering path of the Ogre ManualObject should be done since currently the imported Ogre ManualObject is very heavy to render and it cannot be exported to other formats than Ogre mesh. To fix this, some work should be done with the optimization and possibly create separate texture materials for the object, which can be rendered on top of the captured object. In addition, a possibility to use 3D textures should be investigated. This way, the amount of vertices in an object could be dramatically reduced without losing the accuracy compared to the original object.

Some strange behavior was also found from the data conversions, and it will be researched further in the future. When the mesh was saved as a VTK file and converted from that to the Ogre binary mesh, file size was five times larger than the file serialized directly from the Ogre ManualObject to the Ogre binary mesh.

There is also a need to create support for importing and exporting meshes, for other formats preferably at least in a Collada format (8). This part of the work should also include the Collada support reimplementation for Tundra 2, which was originally made by Joni Mikkola for Tundra 1 (15).

REFERENCES

1. 3D is here: Point Cloud Library (PCL). Robotics and Automation (ICRA), 2011 IEEE International Conference on; 2011.

2. Alatalo T. An Entity-Component Model for Extensible Virtual Worlds. Internet Computing, IEEE 2011 sept.-oct.;15(5):30.

3. Blanchette J & Summerfield M. 2008. C++ GUI Programming with Qt 4, Second Edition. the United States of America: Prentice Hall.

4. Boada, I. Navazo, I. Scopigno, R. Multiresolution volume visualization with a texture-based octree. The Visual Computer. Volume 17, Number 3 (2001), 185-197.

5. Botsch M, Steinberg S, Bischoff S, Kobbelt L. OpenMesh - a generic and efficient polygon mesh data structure. 2002.

6. Bernardini F, Rushmeier H. Volume 21 (2002), number 2 COMPUTER GRAPHICS forum The 3D Model Acquisition Pipeline.

7. Cignoni, P. 2008. Meshlab: an Open-Source 3D Mesh Processing System. Ercim news 73 April 2008, 47-48.

8. COLLADA - Digital Asser Schema Release 1.5.0. Date of data acquisition 4 April 2012. Available at: http://www.khronos.org/files/collada_spec_1_5.pdf.

9. Github Plans & Pricing. Date of data acquisition 18 April 2012. Available at: https://github.com/plans.

10. Neural mesh ensembles 3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004. Proceedings. 2nd International Symposium on title   Neural mesh ensembles; sept.; ; 2004.

11. Nokia. Qt Licensing. Date of data acquisition 10 April 2012. Available at: http://qt.nokia.com/products/licensing/.

12. Manninen T. Hollström H. Pirkola J. & Wittenberg V. REALXTEND: Avoimen lähdekoodin virtuaalimaailmajärjestelmä realXtend luo pohjan 3D Internetille. Date of data acquisition 23 April 2012. Available at: http://www.kauppalehti.fi/5/i/yritykset/lehdisto/hellink/tiedote.jsp?selected=kaikki&oid=20081120/12270739024580.

13. MeshLab project page. Date of data acquisition 18 April 2012. Available at: http://meshlab.sourceforge.net/.

14. Meshlab licensing. Date of data acquisition 18 April 2012. Available at: http://sourceforge.net/apps/mediawiki/meshlab/index.php?title=Licenses.

15. Mikkola, J. Collada Importer for RealXtend, Oulu University of Applied Science, Bachelor's Thesis, 2011

16. Interactive 3D modeling of indoor environments with a consumer depth camera. Proceedings of the 13th international conference on Ubiquitous computing New York, NY, USA: ACM; 2011.

17. Junker, G. Pro OGRE 3D Programming (Expert's Voice in Open Source). The United States of America: Apress; 2006.

18. KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera. Proceedings of the 24th annual ACM symposium on User interface software and technology New York, NY, USA: ACM; 2011.

19. Kolluri R. Provably good moving least squares. ACM Trans.Algorithms 2008 may;4(2):18:1-18:25.

20. Ogre Current Features. Date of data acquisition 4 April 2012. Available at: http://www.ogre3d.org/tikiwiki/Current+Ogre+Features.

21. Ogre API Documentation: ManualObject. Date of data acquisition 4 April 2012. Available at: http://www.ogre3d.org/docs/api/html/classOgre_1_1ManualObject.html.

22. Ogre ByatisNotes. Date of data acquisition 4 April 2012. Available at: http://www.ogre3d.org/tikiwiki/ByatisNotes.

23. Ogre Scene. Date of data acquisition 4 April 2012. Available at: http://www.ogre3d.org/tikiwiki/tiki-index.php?page=-scene.

24. Oliker, L.; Biswas, R.; , Parallelization of a dynamic unstructured algorithm using three leading programming paradigms, Parallel and Distributed Systems, IEEE Transactions on , vol.11, no.9, pp. 931- 940, Sep 2000

25. OpenNI organization. Date of data acquisition 20 April 2012. Available at: http://75.98.78.94/default.aspx.

26. Pauly, M. 2003. Shape modeling with point-sampled geometry. ACM Transactions on Graphics 22 (3), 641-650.

27. Presentation of realXtend.org by Toni Alatalo at NVWN Montly Meeting. Date of data acquisition 20 April 2012. Available at: http://nordicworlds.net/2011/04/16/presentation-of-realxtend-org-by-toni-alatalo-at-nvwn-monthly-meeting/.

28. Point Cloud Library: About. Date of data acquisition 6 April 2012. Available at: http://pointclouds.org/about.html.

29. Point Cloud Library API Documentation. Date of data acquisition 6 April 2012. Available at: http://docs.pointclouds.org/1.5.1/.

30. Point Cloud Library Tutorials. Date of data acquisition 6 April 2012. Available at: http://pointclouds.org/documentation/tutorials/.

31. Spectral surface reconstruction from noisy point clouds. Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing New York, NY, USA: ACM; 2004.

32. RealXtend association. Date of data acquisition 23 April 2012. Available: http://realxtend.wordpress.com/realxtend-association/.

33. realXtend: a Platform for Networked 3D Applications. Date of data acquisition 10 April 2012. Available at: https://github.com/realXtend/doc/blob/master/acm_multimedia/overview.rst.

34. Redmine, Redmine wiki. Date of data acquisition 4 April 2012. Available: http://www.redmine.org/projects/redmine/wiki/.

35. Loeliger, J. 2009. Version control with Git. O'Reilly Media: New York.

36. Estimating surface normals in noisy point cloud data. Proceedings of the nineteenth annual symposium on Computational geometry New York, NY, USA: ACM; 2003.

37. Remondino F, Institut für Geodäsie und Photogrammetrie (Zürich). From point cloud to surface the modeling and visualization problem. Zurich: ETH, Swiss Federal Institute of Technology Zurich, Institute of Geodesy and Photogrammetry; 2003.

38. Santala M, 3D Content Capturing and Reconstruction Using Microsoft Kinect Depth Camera, Oulu University of Applied Science, Bachelor’s Thesis, 2012.

39. Scarano, V. 2008. Meshlab: an Open-Source Mesh Processing Tool. Eurographics Association 2008, 129-136.

40. The Stanford 3D Scanning Repository. Date of data acquisition 4 April 2012. Available at: http://graphics.stanford.edu/data/3Dscanrep/.

REALXTEND LICENCE

REALXTEND LICENCE

communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

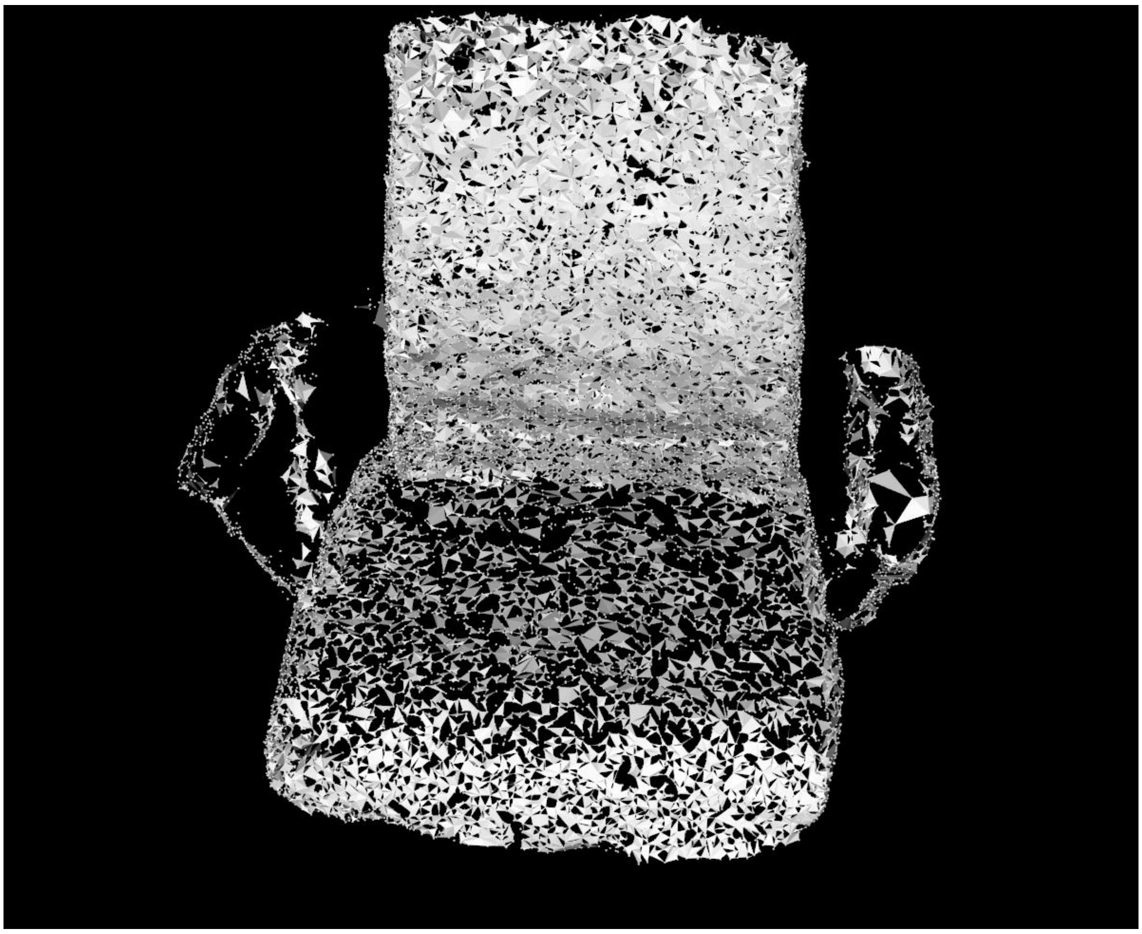
(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one

REALXTEND LICENCE

REALXTEND LICENCE

POLYGON MESH WITH INCONSISTENT NORMAL DATA



APPENDIX 3

MLS PROCESSING PARAMETERS

Polynomial fit = true
Polynomial order = 2
Search method = KdTree<pcl::PointXYZRGB>
Search radius = 0.03
SqrGaussParam = 0.0009

APPENDIX 4

MATERIAL FILE FOR CAPTURED OBJECT

```
material CapturedObject
{
   technique
   {
      pass
      {
         cull_hardware none
         ambient vertexcolour
         diffuse vertexcolour
         specular vertexcolour
         emissive 0.1 0.1 0.1
      }
   }
}
```