UNIVERSITÀ DI PISA

Department of Computer Science
Bachelor in Computer Science

IMPROVING THE SUPPORT FOR 3D SCANNED DATA IN MESHLAB AND PYMESHLAB

Supervisors:

Paolo Cignoni

Alessandro Muntoni

Presented by:

Gabriele Pappalardo

Academic Year 2020/2021

Abstract

MeshLab, and its Python counterpart PyMeshLab, have been widely used for processing 3D scanned data. However, some functionalities of MeshLab that are entirely interactive have not been transferred into the scriptable framework under Python. Moreover, an emerging file format for distributing scanned data still is not supported by MeshLab. In this thesis, we have improved the capability of MeshLab to handle different kinds of 3D scanned data by adding the support for 3D scanned LIDAR data (E57). Moreover, by refactoring the core part of the alignment tools, we allowed higher flexibility in their usage in MeshLab and made them available in PyMeshLab.

Contents

Chapter 1

Introduction

This chapter will introduce MeshLab and PyMeshLab, the open-source mesh processing system where I have developed the contribution described in this thesis. The second section of this chapter will talk about the MeshLab's plugin architecture and will introduce a general description about the plugins that I have implemented to improve the support for the loading and processing of 3D scanned data.

1.1 Introduction to MeshLab

MeshLab [1] (https://meshlab.net) is an open-source, extensible program written in C++ for the elaboration and the processing of meshes, developed by the *Visual Computing Lab* of *ISTI-CNR*. The software offers many tools useful for editing, cleaning, recovering, inspecting, rendering, texturing and converting meshes. MeshLab comes under the *GPL 3.0* license, and it's available on Windows, Linux and macOS.

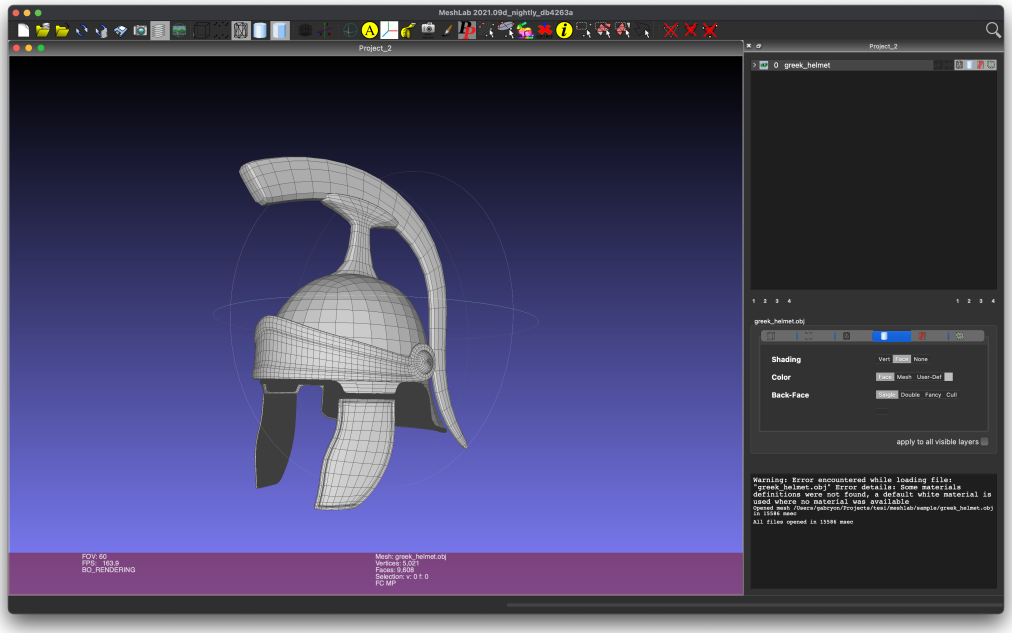


Figure 1.1: **MeshLab** running on *macOS*.

MeshLab is compiled in two versions: single-precision (32 bit) and double-precision (64 bit) modes. Its source code can be found on ISTI-CNR GitHub repository at this link: https://github.com/cnr-isti-vclab/meshlab.

From a developer's perspective, an important characteristic of MeshLab is that it has been designed to be modular and extensible thanks to a plugin architecture that will be explained in Section 1.2.

1.1.1 MeshLab's Goals and Impact

The idea behind MeshLab is to provide a set of mostly automatic tools for processing, analyzing and showing 3D models. These characteristics allowed the software to be used in various environments: *for the creation of assets in gaming industry* [2] [3] [4], *for* [5], *for the analysis of meshes obtained by 3D scanning*, *for the visualization of biomedical data* [6], *for surgical simulations field* [7] [8] and *for the conversion of 3D data between different formats*. MeshLab has established itself as an open-source main software for the data management of objects acquired by 3D scanners and for the treatment and preprocessing of models to be used for 3D printing.

The software had a strong impact on the "*Cultural Heritage*" field, where 3D scanning, visual presentation and rapid prototyping are now widely accepted as enabling technologies, provided that their use is not limited by the cost and complexity of using survey systems and commercial software. Particularly with the emergence of low-cost 3D scanning technologies (such as photo reconstruction techniques), MeshLab has become the software of choice for managing these data.

1.2 MeshLab's plugins architecture and goals

As said in the introduction of MeshLab, it was designed to be extensible, meaning it's possible to write separate plugins that allow the addition of new processing capabilities. These additional components are easily writable by anyone who knows how to code using the C++ programming language. According to the kind of functionality they add, there exist various categories of plugins inside MeshLab:

1. **IOPlugin**: allows importing meshes from new file formats (such as `.E57`, see Section 3).
2. **FilterPlugin**: implements automatic, black box processing algorithms that can be applied to one or more meshes according to defined parameters (like random mesh coloring).
3. **EditPlugin**: provides tools that need some kind of interaction with the mesh. Editing tools are the most complex and most tightly coupled with the rest of MeshLab, given the fact that they can grab GUI events (like the mouse events) and customize the rendering process.
4. **RenderPlugin**: are used to customize the rendering process of the displayed meshes.
5. **DecoratePlugin**: are used to implement the *decorators*, which are visualization aids and glyphs that are overlaid on the rendered mesh to help show some data about a mesh (e.g. the normals of the surface).

All the plugins inherit the `MeshLabPlugin` and `MeshLabPluginLogger` interfaces, that are the base for all MeshLab plugin classes. The first class represents the library file of the plugin, since all the MeshLab's plugins are compiled as *dynamic libraries*, so they can be loaded during the runtime. The second one is a utility class used to log information about the plugin itself. All the plugins are loaded using the `PluginManager`, which is a container object designed to load at runtime the dynamic libraries and organize them in order to expose the plugin functionalities. The `PluginManager` and all the Plugin interfaces are part of the so-called *MeshLab Common Framework*. The *MeshLab Common Framework* does not contain GUI data structures and routines (which are part of the MeshLab GUI), but just the core functionalities, interfaces and data structures of the software that need to be shared between the GUI (or a CLI like PyMeshLab, see section 1.3) and the plugins. In the Figure 1.2 all the plugins categories contained in MeshLab are shown.

For the *objectives* of this work, only two plugin categories were used: the *IOPlugin* and the *FilterPlugin*, that will be explained in detail in the next sections.

For anyone who would like to create a new plugin for MeshLab, just see the examples contained in this repository: https://github.com/cnr-isti-vclab/meshlab-extra-plugins.

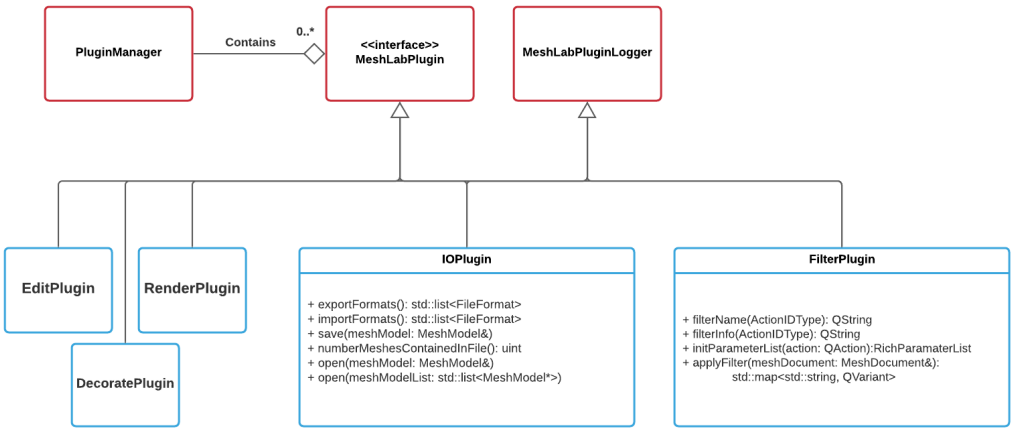


Figure 1.2: UML [9] class diagram for MeshLab's Plugins. The Plugin Manager manages all the plugins loaded inside MeshLab. Only the **IOPlugin** and the **FilterPlugin** inner methods are shown because they are the main used in this work.

1.2.1 IOPlugin

This category of plugins has several methods used to load and save images, and to load 2D images as well. For example, MeshLab supports the new *glTF* [1] file format created by *Khronos Group*, which is designed to become the new standard for three-dimensional models and scenes [10]; the authors themselves defined the standard as "the *JPEG* for 3D". The support for this file format is possible thanks to `IOglTF` plugin, which is contained inside MeshLab. The Figure 1.3 shows a glTF file loaded inside the software.

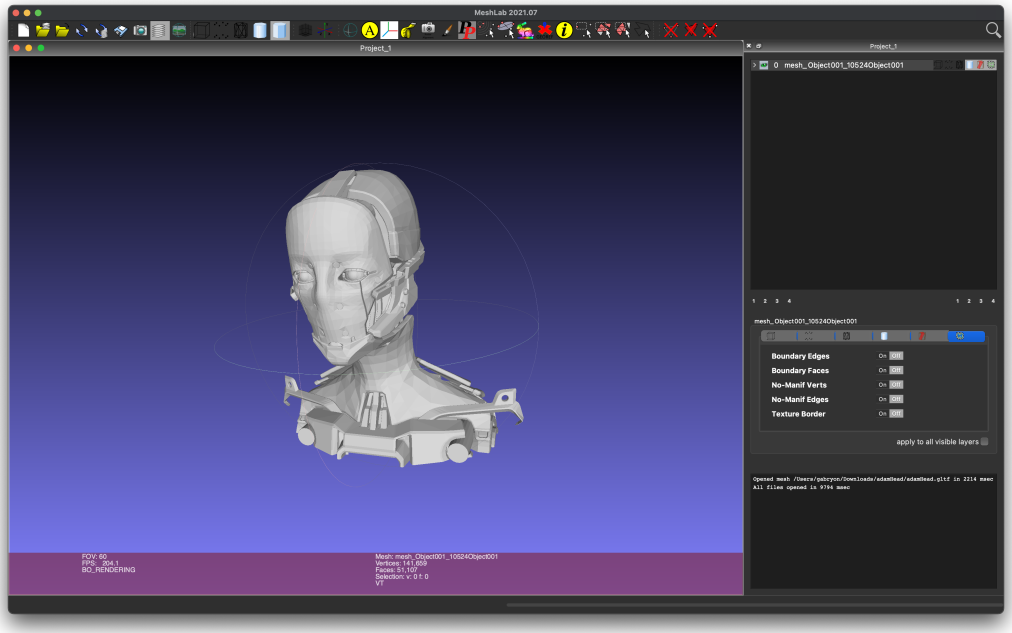


Figure 1.3: *Adam's Head*. This is a glTF file loaded into MeshLab 2021.07. Authors: *Unity Technologies* ©. The screenshots were captured using MeshLab 2021.07.

[1] glTF: *Graphics Language Transmission Format*

IOPlugin access API

Switching to a developer's view point, the Figure 1.2 lists some important methods used for the work of this thesis:

1. `std::list<FileFormat> IOPlugin::exportFormats`: returns a list of all the output files supported by the plugin.
2. `std::list<FileFormat> IOPlugin::importFormats`: returns a list of all the input files supported.
3. `void IOPlugin::save`: is called by the framework every time a mesh has to be saved in a file.
4. `unsingned int IOPlugin::numberMeshesContainedInFile`: returns the number of meshes that the open function is going to load from the file given as parameter. The default return value is one. If the file format can contain just one mesh per file, then there is no need to re-implement the function.
5. `void IOPlugin::open(..., MeshModel& meshModel>)`: is called by the framework every time a mesh is loaded, who would like to extend the file formats has to implement this method. This method is useful when the file to load contains only one mesh.
6. `void IOPlugin::open(..., std::list<MeshModel&> meshModelsList)`: this is an overload for the open function introduced before, it allows opening different meshes from a file containing more than one mesh. This function has been implemented during the work and the reason will be explained in Section 3.4.

The IOPlugin allowed us to achieve an *objective* of this work, that is, the support for a new file format, called *E57*. The plugin will be discussed in Section 3.

1.2.2 FilterPlugin

The second plugin category used for this thesis is the FilterPlugin. Before technically describing the main access APIs of these plugin types, let's show an example. Suppose we have a point cloud (described in Chapter 2) and we would like to perform the *surface reconstruction* (Section 2.2) i.e. building a triangle mesh that approximates correctly this set of isolated points in space. In MeshLab, there are many possibilities for such a problem. One of the most used is the "*Surface Reconstruction: Screened Poisson*" filter, which uses the algorithm proposed by *Michael Kazhdan and Hugues Hoppe* in their paper "**Screened Poisson surface reconstruction**" [11]. The Figure 1.5 shows us the results of this filter plugin. In practice, filter plugins encapsulate the concept of black box processing that takes as input one or more meshes and a set of parameters; after some automatic processing, gives back a well-defined set of meshes as a result. Filter plugins are the core processing power of MeshLab and expose more than two hundred different functionalities for many purposes, like cleaning, measuring, analyzing, checking, remeshing, transforming, and so on.

It is important to underline the black box nature of filter plugins because it is the reason for which it is possible to automatically translate filter actions into Python functions (see Section 1.3).

FilterPlugin access API

When creating a FilterPlugin, the developer defines different *actions* in an *enumerator*, which will be exposed to the MeshLab's user. As for the previous section, the Figure 1.2 lists the main methods used to implement the alignment filter that will be discussed in Chapter 4:

1. `QString FilterPlugin::filterName`: returns a string describing each filtering action defined in the action enumerators;
2. `QString FilterPlugin::filterInfo`: returns a long description for each filtering action.

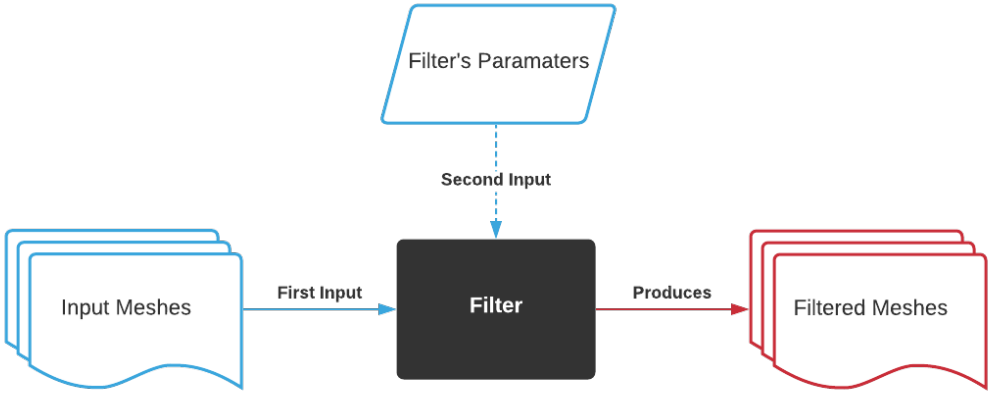
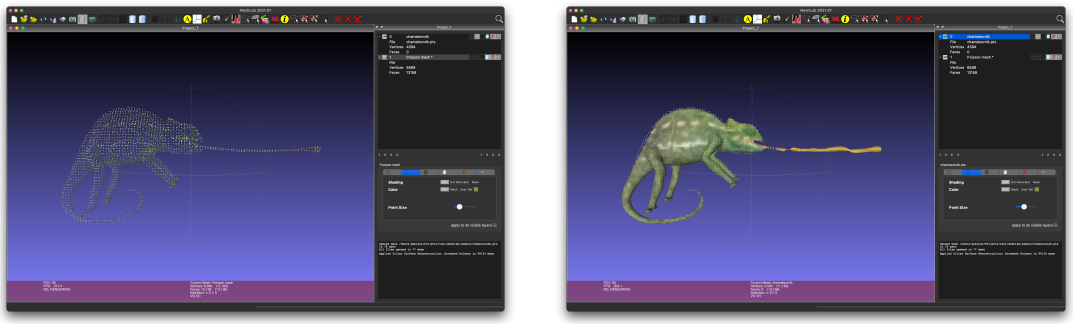


Figure 1.4: Flow diagram showing the principle behind the filter plugins. The user gives as input the meshes to filter and several parameters. Once the filter execution starts, it produces a set of new meshes. The filter acts as a black box because the user cannot see what elaborations are being performed.



(a) Pre-filtering (b) Post-filtering

Figure 1.5: Chameleon. The figures above show a chameleon represented as a point cloud reconstructed in a surface using the filter "*Surface Reconstruction: Screened Poisson*". The screenshot has been captured using MeshLab 2021.07.

3. `RichParamaterList FilterPlugin::initParamaterList`: this function is called to initialize the list of parameters for each action defined in the plugin.
4. `std::map<std::string, QVariant> FilterPlugin::applyFilter`: this function is the filter plugin's main core. It applies the selected action with the already established parameters, and it is called by the plugin framework. The function returns a map of (*string*, *value*) pairs, that will be the output of command-line calls of the filter using PyMeshLab.

1.2.3 EditPlugin

As explained in Section 1.2, the *EditPlugin* are additional components that allow a user to interact directly on a mesh with MeshLab using an *edit tool*, capturing user interface events, such as mouse clicks, keyboard input and other inputs coming from different peripherals. MeshLab has several edit tools: *manipulators* tool, *measuring* tool, *Z-painting* tool, *pick points*, *selection of vertex clusters*, *quality mapper* and so on.

One of them is important for this work, that is, the *align tool* (see Figure 1.7), which its code has been refactored to allow the PyMeshLab users to use the alignment algorithms implemented inside the software, since due to the interactive nature of the plugin, a CLI [2]

[2] *CLI: Command Line Interface*

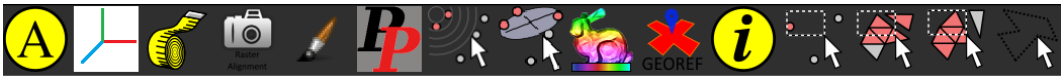


Figure 1.6: A Zoom on the various interactive Edit Tools that are exposed in toolbar of MeshLab. From left to right: *Align*, *Manipulators*, *Measuring*, *Raster alignment*, *Z-painting*, *PickPoints*, *Select Vertex Clusters*, *Select Vertices on Plane*, *Quality Mapper*.

user cannot interact with the tool. The refactoring process will be described in Chapter 4.

Figure 1.7: The dialog of the *Align Tool*. The tool offers several ways to perform the alignment process manually and automatically between the meshes. In the standard pipeline, the user starts by choosing a base mesh and then manually performing a rough alignment with other meshes. Once this manual rough alignment process is done, the user can perform the global alignment routine, clicking the *Process* button. The screenshot has been captured using MeshLab 2021.07.

1.3 The Python Counterpart: PyMeshLab

PyMeshLab [12] is a Python [3] library that exposes most of the MeshLab functionalities. PyMeshLab has been made possible by the well-defined black box nature of filter plugins, in fact, most of PyMeshLab is automatically built from the same code of MeshLab, filters are translated into python functions whose parameters and interfaces are automatically derived from the declarative nature of the MeshLab plugins.

It is possible to download the library using the standard Python package manager: *pip*. Using PyMeshLab is elementary, for example, if a user would like to load a point cloud contained in a E57 file simply types:

```
1 import pymeshlab
2
3 # Initialize the MeshSet as a new Workspace
4 meshSet = pymeshlab.MeshSet()
5
6 # Load the mesh pointCloud.e57 inside the MeshSet
7 meshSet.load_new_mesh('/path/to/mesh/pointCloud.e57')
```

[3] Python is a popular scripting programming language: https://www.python.org

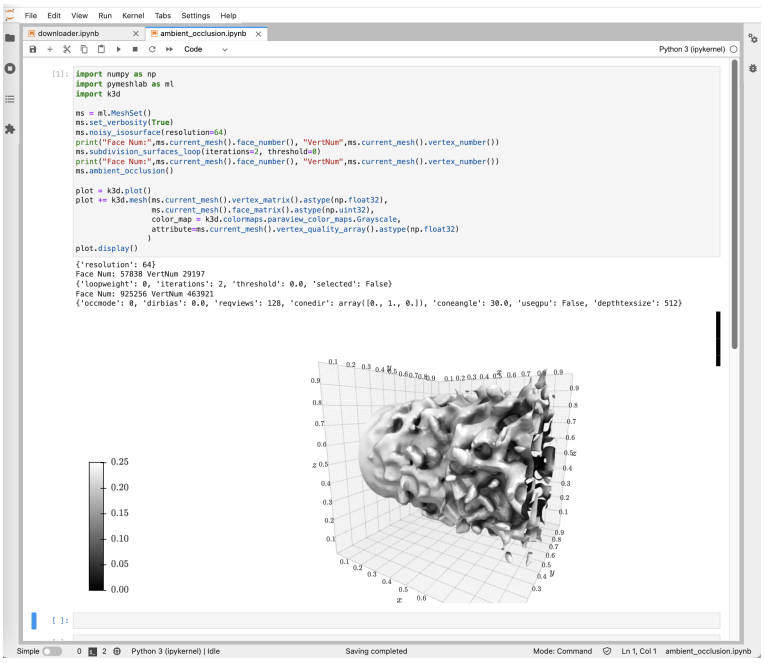


Figure 1.8: Using PyMeshLab inside a JupyterLab notebook. A simple script that use MeshLab functionality to create a test shape, refine the mesh, and compute an ambient occlusion factor over it. The result is shown using the *k3d* Python visualization library, by simply passing the vertex and face arrays provided by MeshLab.

1.3.1 Motivations

By default, MeshLab provides a GUI [4] written using the *Qt* framework for easy portability on many platforms. However, while a GUI allows to easily and interactively experiment with the many possibilities of processing meshes in an interactive way, in many cases users need to perform repetitive tasks like applying the same set of filtering actions to a large set of meshes. This kind of advanced scripting activities is missing from the desktop version of MeshLab and is the reason for having a Python library exposing most of the processing capability of MeshLab. In this way, using PyMeshLab, eventually coupled with one of the many Python libraries for rendering 3D meshes and using notebooks (*i.e.*, Jupyter, Figure 1.8), people can experiment with the many algorithms of MeshLab in a more programmatic way.

PyMeshLab offers all the filters contained in MeshLab, plus the capability to create new projects which can be open in MeshLab. For further information, check the documentation at the following link: https://pymeshlab.readthedocs.io.
Let's illustrate two use case scenarios to show the potential of PyMeshLab.

Case 1: Color Processing

Suppose we have a folder containing only gray-scale point clouds. We would like to color the points depending on their position in space, in a sense that near vertices will be painted with similar colors. This filter is called *Perlin Color*. Doing this task inside standard MeshLab can be very long, since a user has to do:

1. Open MeshLab and create a new project.
2. Open the interested meshes.
3. For each mesh, it has to open the filter's window, configure the parameters and then apply the filter and see the results.

[4] GUI: Graphical User Interface

If we had 100, or even more, meshes, this task would take a huge amount of time. With PyMeshLab, the user could write a script that loads the meshes inside a MeshSet and then simply applies the filter. The listing below shows the script:

Listing 1 Perlin Color script.

```
import os
import pymeshlab

meshFolderPath = "/path/to/meshes"
outputFolderPath = meshFolderPath + "/out"

meshSet = pymeshlab.MeshSet()
redColor = pymeshlab.Color(255, 0, 0, 255)
blueColor = pymeshlab.Color(0, 0, 255, 255)

for meshFile in os.listdir(meshFolderPath):

    # Load the mesh inside the MeshSet
    meshSet.load_new_mesh(meshFolderPath + meshFile)

    # Apply the 'Perlin Color' filter.
    meshSet.perlin_color(redColor, blueColor)
    outputFile = outputFolderPath + "pc_" + meshFile

    # Save the edits applied into a new mesh file
    meshSet.save_current_mesh(outputFile)
```

The script imports the `pymeshlab` library and then creates a new *MeshSet*. As described by *Alessandro Muntoni* [12], the MeshSet class represents a set of meshes, each one with its own unique ID, where every mesh corresponds to a layer inside MeshLab. Automatically, when a mesh is loaded it becomes the current one and, on the invoking of a filter, the edits will be applied on it.

Case 2: Faces Cleanup

Here is another use case scenario. Suppose we have a set of scans containing a 3D model of a bunny statue that we would like to align. The scans are dirty because the initial 3D scanner couldn't see behind a certain spot, therefore some faces of the bunny have a long edge (see Figure 1.9). Figure 1.10 shows what we would like to achieve.

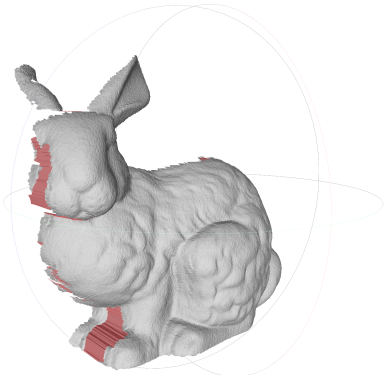
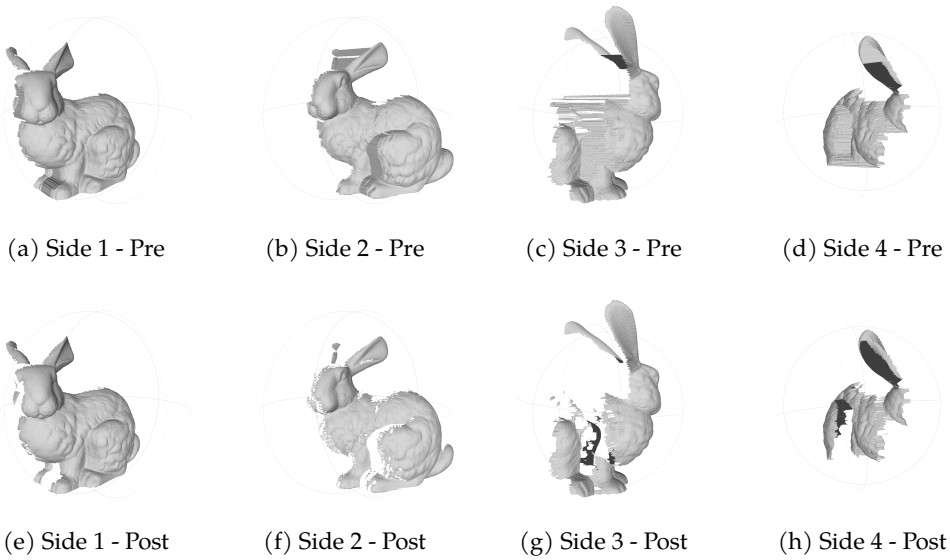


Figure 1.9: The faces we would like to remove are highlighted in red.

If we had to clean all the meshes manually, we would repeat the cycle described in the previous subsection plus additional steps for each mesh:



(a) Side 1 - Pre (b) Side 2 - Pre (c) Side 3 - Pre (d) Side 4 - Pre

(e) Side 1 - Post (f) Side 2 - Post (g) Side 3 - Post (h) Side 4 - Post

Figure 1.10: Result of the script. The first row contains all the images before the script execution, the second row shows the results with the cleaned faces.

1. choose a threshold for the edges;
2. select the faces that match the threshold;
3. invoke the "*Delete Selected Faces*" filter;

Then again, PyMeshLab comes in our hand, and a user could write a script like the one inside the Listing 2 to do all these operations in just one click.

Listing 2 Cleanup script described in Section 1.3.1

```
import os
import pymeshlab

meshFolderPath = "/path/to/meshes"
outputFolderPath = meshFolderPath + "/out"

meshSet = pymeshlab.MeshSet()
for meshFile in os.listdir(meshFolderPath):

    # Load the mesh inside the MeshSet
    meshSet.load_new_mesh(meshFolderPath + meshFile)

    # Select the faces according to a threshold
    meshSet.select_faces_with_edges_longer_than(0.02)
    # Delete the selected faces
    meshSet.delete_selected_faces()

    outputFile = outputFolderPath + "df_" + meshFile
    # Save the edits applied into a new mesh file
    meshSet.save_current_mesh(outputFile)
```

1.3.2 PyMeshLab's internals

PyMeshLab expose the filtering functionality of MeshLab. MeshLab filtering plugins contain a set of actions that correspond to filters in the MeshLab interface (that can be accessed in MeshLab either by the menus structure or using the search functionality). The library's functions are automatically created through the use of *pybind11* [13]. As explained by its

developers: "*PyBind11* is a lightweight header-only library [14] that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code".

PyMeshLab has two main cores: the first one consists of the definition of the binding classes that can be called in Python; the second one, the most important, performs the actual binding with the *MeshLab Common framework* (described in Section 1.2), linking the existing MeshLab's plugins into the Python module. When the module is loaded inside a script, an instance of the Plugin Manager loads all the MeshLab's plugins and, thanks to `pybind11`, a dynamic binding process starts. This routine seeks all the loaded plugins and binds the main plugin methods into the MeshSet class, exposing the plugin functionalities to the Python environment. Finally, when the loading ends, the user can use the scripts as if on MeshLab. PyMeshLab doesn't bind the *EditPlugin* (Section 1.2.3) since they are meant for user interaction activities, and they can only be used into the MeshLab GUI, which is not part of the *MeshLab Common framework*.

The source code of PyMeshLab contains the file `meshset_helper.cpp`, which is of great relevance because it contains the implementation of the functions defined inside the `MeshSet` Python class, previously seen in the first use case in Section 1.3.1. This set of functions contains the `pybind11:dict toPyDict(...)` which is the one that converts the map obtained by applying the filter described in Section 1.2.2 into a Python dictionary. This function allows the access to the computed data by the execution of a filter and allows the parsing process to support different types, such as *int*, *double*, *Box* and *Matrix*. However, thanks to the work of this thesis, we have found a limit of the function itself that will be explained later in Section 4.5.

Chapter 2

Mesh, Point Clouds, and VCGLib

In this chapter, before describing the work done inside MeshLab, we introduce a few concepts, data structures and the library that we used in the current implementation.

2.1 Mesh

Let's introduce the concept of mesh as the way in which we represent digitally the actual shape of a three-dimensional object. There exist several ways to represent these objects, the one which we are interested in, is the approach describing the surface of an object using by discretizing it into a set of polygonal faces. Specifically, we choose the triangle as the only used polygon because it has a compelling descriptive power: we can describe the other polygons as a composition of triangles. Plus, the triangles have some interesting math properties, allowing a better modelling and rending algorithms design. The set of faces takes the name of *mesh*. Now, let's illustrate two approaches to represent a mesh.

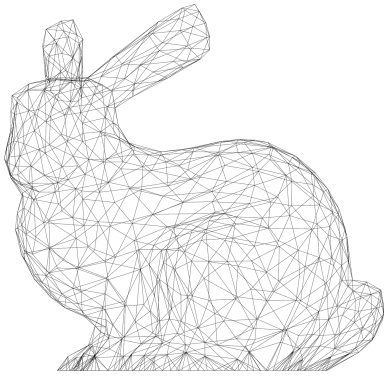


Figure 2.1: The surface of the Stanford's Bunny, represented as a mesh composed by a collection of triangles.

Definition 2.1.1 (Naive Mesh) *A mesh M is a list of faces, each of which makes explicit the coordinates of the vertices that delimit it:*

$$M : T = (t_1, t_2, \ldots, t_n)$$

where $t_i = (p_{i1}, p_{i2}, p_{i3})$ and $p_i = (x_i, y_i, z_i)$.

This first representation is extremely redundant because each vertex is repeated in all the incidents faces [1], furthermore is inefficient because we can't find easily the incidents faces on a defined vertex or find the adjacent faces. Hence, let's try to give another definition.

Definition 2.1.2 (Indexed Mesh) *A mesh M is a tuple made by two sets: the vertices V and the faces T.*

[1] Two faces are incidents if they have at least one element in common.

$$M = (V, T)$$
$$V = \{v_i : v_i = (x_i, y_i, z_i), v_i \in \mathbf{R}^3\}$$
$$T = \{t_i : t_i = (v_j, v_k, v_l), v_{j,k,l} \in V\}$$

This second definition is called *indexed mesh*, in which a face list is being associated with the vertices list, thanks to this approach redundancy reduces.

2.2 3D Scanning

3D scanning is the process of capturing the shape of a physical object and replicating it as a 3D model inside a computer [15]. 3D scanning is not so easy, before to produce a final 3D model the data acquired from a scanner must go under the *3D scanning pipeline*, which is a list of steps to re-create the physical object, we can summarize the pipeline in three main steps:

1. Acquisition of multiple range maps.
2. Alignment and merging of the scans.
3. Applying a surface reconstruction algorithm.

Acquiring a physical object is done in several ways. The tools that scan an object are called 3D Scanners and there exist different types of them, but we are going to focus on the *optical* ones which are divided in:

- *Active*: emit some kind of ray of lights and detect its reflection in order to probe an object (x-ray, ultrasound).
- *Passive*: they do not emit any kind of radiation themselves, but instead rely on detecting the reflection of ambient radiation.

There are two famous optical scanning techniques. The first one, an active, is called **laser scanning**, in which a three-dimensional scanner casts some rays of light onto a surface of an object and the time between the casting and the moment in which the laser reflects off the object is measured. The second, a passive, is called **photogrammetry**, where some special two-dimensional images are cast on a surface of an object to triangulate the points and tracks them in a three-dimensional space.

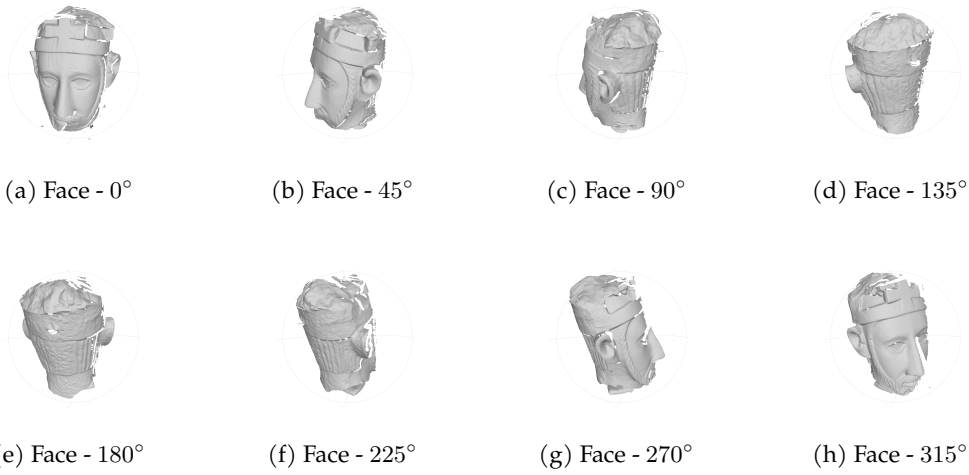
Almost all optical scanner uses a camera as input device. What is recovered after a single shot is a depth value for each pixel in its sensor, which is converted in a 3D point. Therefore, from the scanner's perspective, all the 3D points are on a regular grid, that is promptly triangulated using this intrinsic regularity. This is possible because of the limited Z-span.

Figure 2.2: *Xbox Kinect v2*. The *Kinect* is an "*active structured light*" (active) 3D scanner, primarily thought to be used with the *Xbox* game console. Nowadays, *Microsoft* does not make any more game for the peripheral, but in the latest years a community of 3D hobbyists, that reevaluated the hardware, is born due to the device's lower cost.

The result of a single scan is generally called a **range map**. A range map is an incomplete 3D model, that's because the map refers only to a point of view, and for 3D objects we need more point of views to reconstruct the entire object.
The next step of the pipeline is the **alignment** (often called *registration* in literature). Because each scan is generated in a different shop, the partial object acquired has its coordinate system, this means that if we render the scans in a 3D software such as MeshLab, each scan will be in the wrong place. So, our goal is to bring these scans to a common reference system. We could do this using this two alignment techniques:

1. *Rough alignment*: a user manually position the various pieces in an "approximately correct" position.
2. *Fine alignment*: the computer execute an algorithm to refine the results using the shared area between the range maps.

These two techniques combined allow users to align a set of scans in only one mesh. MeshLab does perform both the alignments using a combination formed by the *Iterative Closest Point* algorithm and a global optimization strategy. The alignment step is crucial in the pipeline. We cannot assume that the scans are *error-free*, so we introduce two types of errors: *acquisition* and *alignment* error (cannot be $\leq$ *acquisition error*, $\leq$*acquisition resolution* / 2). Their sum upper bound the final error produced in the registration process. The alignment topic is the core of the Chapter 4.



(a) Face - 0° (b) Face - 45° (c) Face - 90° (d) Face - 135°

(e) Face - 180° (f) Face - 225° (g) Face - 270° (h) Face - 315°

Figure 2.3: These are some range maps (scans) of a stone head sculpture. The 3D scanner acquired the meshes looking at the object from different angles, and only the portion of surface visible from that direction is present in each range map. To obtain a complete object, all these scans have to be aligned and merged.

One of the results produced by a 3D scanner are the point clouds, which will be discussed in the next Section.

The latest step in the pipeline is the **surface reconstruction**: a procedure aiming for the reconstruction of a real object into a 3D model which can be used in different projects, such as the restoration of a sculpture or being used inside CAD software programs. A reconstructed surface will be available in the tests done in Section 4.6.

2.3 Point Clouds

A **point cloud** is one of the simplest existing three-dimensional object. It's a set of points inside a space defined by a coordinate system (see Figure 2.4).

We can formally define the point clouds with the following definition.

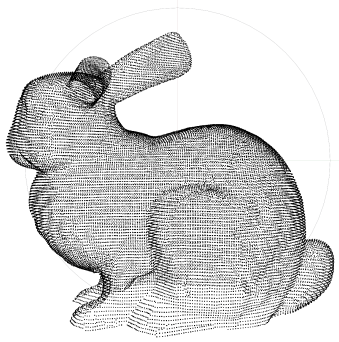


Figure 2.4: A point cloud representing the Stanford's bunny.

Definition 2.3.1 (Point Cloud) *A Point Cloud P is an indexed mesh (2.1.2) with the faces set T equals to $\emptyset$.*

$$P = (V, \emptyset)$$

Usually in a three-dimensional space, a point cloud defines the shape of a real physical object. In addition to the coordinates, a point cloud could have associated metadata, such as the color of the point, for example.

2.4 The VCG library

The *VCG Library* [16] [17] is a C++ template-based library for mesh processing, and it is the core of MeshLab. The library contains many algorithms for:

- *Sampling*: a variety of algorithms for distributing points over the surface of a mesh.
- *Cleaning*: a variety of tools to correct small annoying things, like the ones seen in the Section 1.3.1
- *Color Processing*: the library provides tools for converting from a representation to another one.
- *Measuring*: there are algorithms to measure the distance between surfaces and many geometric elements.
- *Smoothing*: sophisticated noise removals tools.
- *Texturing*: provides algorithms to texture the mesh's surface.
- *Remeshing*: subdivision surfaces, ball pivoting surface reconstruction, clustering simplification and marching cubes.
- *Spatial Indexing*: uniform grids, *K-d trees* (good for point clouds) [2] and hierarchies of Bounding Volumes.
- *File Format*: the library provides importer and exporter for several file formats (PLY, STL, OBJ, 3DS and so on...).

[2] A **K-d tree** [18] is a data structure for organizing points in a k-dimensional space.

2.4.1 Encoding a Mesh

How to declare a mesh

As said in the introduction of this section, the VCGLib is "templated" based, allows to encode a mesh in several ways, allowing the developers to create its definition of mesh. The standard way to encode a view is the one described in Definition 2.1.2. Let's make a basic example of how to declare a new type of mesh.

Suppose we want to encode a point cloud from scratch, without using sophisticated data structure such as the K-d trees. To accomplish that, let's define new classes as shown in the Listing 3.

Listing 3 How to declare a Point Cloud mesh using the VCGLib.

```
/* Forward declaration, used to define new classes. */
class PointCloudVertex;

/**
* Define the types for the new mesh. This is a syntactic sugar
* in order to define correctly a new mesh for the VCGLib. Since
* in C++ there are no constraints on template's inheritance,
* this struct allows to us to force the types of template
* types in the next definitions.
*/
class PCUsedTypes : public vcg::UsedTypes<
    vcg::Use<MyVePointCloudVertexrtex>::AsVertexType
>{};

/**
* Define a point cloud vertex storing: three geometric coordinates,
* three normal coordinates,  color in four channel (Red,
* Green, Blue & Alpha), a quality, and a bitset for flags.
*/
class PointCloudVertex : vcg::Vertex<
    PCUsedTypes,
    vcg::vertex::Coord3f,
    vcg::vertex::Normal3f,
    cg::vertex::Qualityf,
    vcg::vertex::Color4b,
    vcg::vertex::BitFlags
>{};

/**
* In the end we can define our point cloud, like the one
* defined in Definition 2.3.1, that is, a mesh with the
* face set equals to none.
*/
class PointCloud : vcg::tri::TriMesh<
    std::vector<PointCloudVertex>
>{};
```

That's it. To define a new mesh, a developer needs only to derive from `vcg::tri::TriMesh` class and provides the type of containers (in the example we used the STL's vector) of the elements used to encode the mesh. The VCGLib offers a *great flexibility* and a pretty elegant way to define *custom* mesh types. There are many other components inside the library (i.e. faces, edges, ...). During the work of this thesis, the refactor process that will be discussed in Chapter 4 has introduced two new classes for alignment that are templated on mesh type as well.

A list of samples can be found inside the VCGLib:
https://github.com/cnr-isti-vclab/vcglib/tree/master/apps/sample.

Allocating and de-allocating a Mesh

The VCGLib library has several ways to encode a mesh. The most common is a vector of vertices and vector of triangles, that is, the *Indexed Mesh* described previously. This type of mesh is called `vcg::tri::TriMesh` and it is inheritable to create new subclasses. The creation of a mesh in memory is managed through the use of the `tri::Allocators`, for example, to create a mesh:

Listing 4 Snippet illustrating how to allocate memory for a mesh.

```
1   // Declare a new mesh to initialize.
2   MyMesh m;
3
4   // Allocate memory for 3 vertices and returns
5   // a pointer pointing to the first of them.
6   MyMesh::VertexIterator vi =
7       vcg::tri::Allocator<MyMesh>::AddVertices(m,3);
8
9   // Same thing for the faces.
10  MyMesh::FaceIterator fi =
11      vcg::tri::Allocator<MyMesh>::AddFaces(m,1);
```

To free the memory used by the created mesh, the library adopts a *Lazy Deletion Strategy*, or in other words, the elements in the vector that are deleted are only **flagged**, but they are still there. To get rid of the flagged elements, the user has to call two garbage collecting functions: `CompactFaceVector` and `CompactVertexVector`.

Topological Relations

In many cases, it is useful to consider how the faces of a mesh are connected to each other without considering the specific particular position of the vertices in space. This kind of information is usually called the adjacency or "topological" aspect of the mesh. These relations are useful for example to consider local portion of meshes (e.g. finding the faces that are connected to a single face) or to traverse in continuous way the surface of the mesh.
An intuitive convention to name topological/adjacency relations is to use an ordered pair of letters denoting the involved entities of the mesh: Face Edge and Vertices. There are three mains topological relations for triangle meshes: *FF* (Face-Face, edge adjacency between triangular faces), *FV* (from Faces to Vertices, that is, the vertices composing a face) and *VF* (Vertex-Face, from a vertex to a triangle, that is, the triangles incident on a vertex). The other relationships derive from those threes.

The VCGLib does not have a single way to encode topology relationships between triangles and edges. In fact, thanks to the C++ templates, it's possible to define the relations in the mesh class definition (just as shown in Listing 3) by just enumerating the adjacency components needed. Let's take as example the FF adjacency, which is represented by the class `vcg::face::FFAdj`. This relationship encodes the adjacency between faces through edges, so the class contains three pointers to the adjacent faces (just three because a triangular face can only have three adjacent triangles). MeshLab, that is based over the VCGLib, adopts a mesh class that has FF and VF adjacencies stored using a lazy/on-demand approach. They are computed and stored only when an algorithm needs them.

Chapter 3

IOPlugin: E57 file format

This chapter will discuss the E57 file format and how the MeshLab's new IOPlugin, handling the format, has been created and what are the problems we faced during the implementation.

3.1 Why do we need a standard file format for Point Clouds

The point clouds didn't have a standard file format. That meant each hardware vendor produced its own closed-source file format to rely on, preventing the open-source programs to implement their libraries to read files, accessing the data contained or moreover exchanging data between users. Hence, there was a critical need to create a new open-sourced file format for the 3D image industry that allows interoperability among 3D hardware and software manufactures. At the end of the 2000s, some committee members and associations began to create new standards. According to the members, the new file format had to follow these five principles:

1. *Reliable interoperability*: the data should be easy to exchange between vendors.
2. *Open-Source*: so anyone could contribute to improve the file format, implement it easily, contribute to address bugs and issues.
3. *Low price point*: should be affordable for the developers.
4. *Minimalist design*: the file format must be as simple as possible.
5. *Extensibility and Flexibility*: so the standard can be extended with new features without breaking the previous versions.

3.2 Current file format standards: LAS vs E57

At the time this document is written, there exist two main standard file formats: **LAS** and **E57**.

LASer (known as *ASPR LAS* [19]) is a special purpose, public, file format created by the *American Society for Photogrammetry and Remote Sharing* to exchange three-dimensional point clouds by different users. The ASPR thought of this file format to become the standard for the acquisitions created by a LIDAR Scanner, which is an active scanner (Section 2.2). LAS are binary files, inside which there are different metadata about the scan acquired by the hardware scanner. The information contained inside the metadata regards: colors in RGB format, acquisition date, the software which generated the file and a description of the content itself, furthermore, they contain X, Y and Z coordinates for the points contained inside the cloud. This document will not go into more detail to analyze the binary structure of the file.

The second file format is **ASTM E57** [20], developed by *David Huber* of the robotics department of the Carnegie Mellon University and the ASTM (*American Society for Testing and Materials*) E57 Committee on 3D Imaging Systems. As described by the Huber's paper, the

file format can store data from a point cloud created by a laser scanner (or other hardware) and it allows saving 2D images shot during the acquisition. E57 is designed by the committee to be used for a general purpose use instead of the LAS file format.

3.3 The E57 file format

An E57 file is a "hybrid" binary file containing data and metadata about point clouds. The E57 file format adapts a particular data file format design. As said by *Huber*, there are two main designs: the first one has fixed sized fields and records, the second one is more flexible, and it uses flexible, self documenting structures with variable length and structure. The E57 file format is a combination of the best aspects of these two designs. An E57 file can be divided in three main parts (see Figure 3.1): the **Header**, the **binary data** and at last the **XML** [1] **section**.

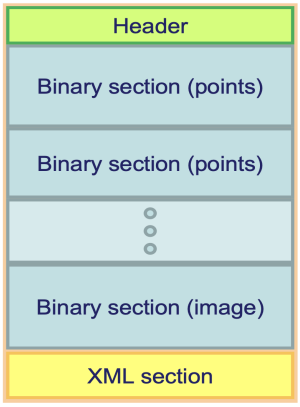


Figure 3.1: E57 file structure as illustrated by *Huber*.

The *Header* section is a small data structure, 48 bytes large, containing all the critical information about the file's structure, like the version file version number and the beginning of the XML section will be explained later.

As the name suggests, the *Binary Data* area contains coordinates and colors of the points, plus the encoded 2D images.

Finally, the *XML Section* is a hierarchical tree, as said before. As you can see in the Listing 5, there is a root node (`<e57Root>`) which contains all the nodes. The nodes refer to metadata about the acquisition (i.e. the time of the scan) and to two special tags:

- `<data3D>`: which contains the translation vector and the quaternions of the point cloud, plus the `<points>` tag used to get the `fileOffset` and the `recordCount` to read the binary data section.
- `<images2D>`: similar to the first one but has metadata about the images contained in the file (such as the encoding, and the representation).

3.4 Implementation of libE57 inside MeshLab

The E57 file format is supported by MeshLab thanks to the **libE57** library. This library is used by an IOPlugin (see Section 1.2) called `E57IOPlugin`. The plugin uses the E57 Simple API, which is wrapped around the E57 Foundation API [21]. The latter API, is a collection of functions that helps to implement reading and writing of a E57 file.

[1] XML stands for eXtensible Markup Language, a language used to define the encoding of a document.

Listing 5 Snippet of the XML Section of an E57 file. Some XML nodes are omitted.

```
<?xml version="1.0" encoding="UTF-8"?>
<e57Root type="Structure"
  xmlns="http://www.astm.org/COMMIT/E57/2010-e57-v1.0">
  <formatName type="String">
    <![CDATA[ASTM E57 3D Imaging Data File]]>
  </formatName>
  <guid type="String">
    <![CDATA[{56D8F874-3656-4CA8-BA17-8BF9757063D0}]]>
  </guid>
  <versionMajor type="Integer">1</versionMajor>
  <creationDateTime type="Structure">
    <dateTimeValue type="Float">9.6758299097398019e+008</dateTimeValue>
    <isAtomicClockReferenced type="Integer"/>
  </creationDateTime>
  <data3D type="Vector" allowHeterogeneousChildren="1">
    ...
    <points fileOffset="40" recordCount="1345856">
    </points>
  </data3D>
  <images2D type="Vector" allowHeterogeneousChildren="1">
    ...
  </images2D>
</e57Root>
```

3.4.1 E57IOPlugin

This subsection will explain the main work done for the new IOPlugin and how the meshes from an E57 file are mapped onto a MeshLab's ones.

Main Methods

The plugin has two main methods to handle an E57 file: `void E57IOPlugin::open` and `void E57IOPlugin::save`. The first one loads an E57 file and parses its content by reading it using the `e57::Reader` helper class. The parse process creates new instances of `MeshModel` class, which is the abstraction to represent a single mesh (plus some metadata) inside MeshLab. The `save` method allows to export the mesh loaded inside MeshLab to a new E57 file thanks to the `e57::Writer` helper class.

Mapping an E57 file to VCG Library's Mesh and vice versa

When the `E57IOPlugin::open` is invoked, then for each view, the `E57IOPlugin::loadMesh` is called. The method takes as input the `MeshModel` and the `e57::Reader` to read the data from the binary section contained inside the file.

Before reading the raw data a class called `E57Data3DPoints` is instantiated, it contains several buffers, allocated in the heap, used by the data reader. The helper object created follows the **RAII** [2] idiom, which is used in the best practices of modern C++. Since C++ standard implementation doesn't have a *Garbage Collector* [3], **RAII** is used to tie resources allocation to objects lifetime, making it easier to avoid resource leaks.

Once the data is read from the binary section, several checks will be done for each point contained in the cloud, such as:

1. the availability of *coordinates*: if they are in Cartesian or in spherical, in the latter case, they will convert in the former geometric reference system.
2. the availability of *colors*: which are in the RGB format.

[2] RAII: *Resource acquisition is initialization*

[3] An automatic technique to reclaim the allocated memory to the operating system.

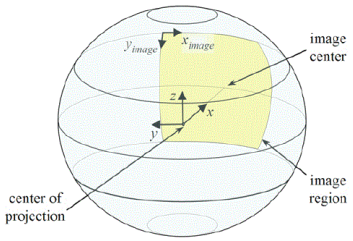
3. the availability of surface *normals* and *intensity*.

When the checks are done, using the VCGLib a new vertex will be allocated inside the mesh to be added and the values will be inserted according to the library requirements. If the colors are not available, then the vertex quality (or intensity) will be used to compute the colors as a scale of gray. Once the function completes its execution, the flow control is returned to the caller, which will check if there are other points clouds contained inside the file and eventually re-invoke the `loadMesh` method.

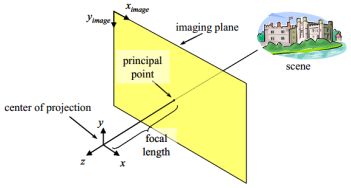
Whenever the user will save a mesh in E57 file causing the invocation of `E57IOPlugin::save` method, a similar process to the load will be executed, except that the save function passes only one mesh model. Therefore, it's not possible to save multiple views inside the E57 file.

3.4.2 Encoded 2D Images: what went wrong

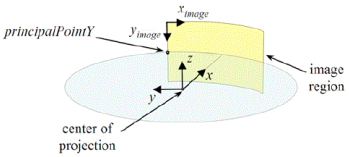
As said before, E57 files contain encoded 2D images within, in PNG or JPEG format according to the file format specifications. The images are treated differently inside the file because they contain the geometric data inside the XML Section. In the Listing 6 is shown two main XML nodes: the *pose* and the *representation*. The first one contains info about the rotation and the translation of the image inside the three-dimensional space, the second one contains metadata about the image representation, according to the standard can be: Spherical, Pinhole and Cylindrical.



(a) Spherical Representation



(b) Pinhole Representation



(c) CylindricalrRepresentation

There are some E57 files which do not contain colors for the points, so we thought to map the missing colors by using the encoded 2D images. But, due to the lack of details in the libE57 documentation, we weren't able to achieve this goal, thus for the mesh without colors we used a gray-scale coloring, like the one shown in the Figure 3.4.

(a) Layer 1

(b) Layer 2

(c) Layer 3

(d) Layer 4

(e) Layer 5

Figure 3.2: These are all the images encoded inside the '*manitou.e57*' file. The images were captured using a spherical projection camera model.

3.5 E57 files tested

Several E57 files have been tested thanks to the samples provided by the website http://www.libe57.org in "Test Data" section. In this section, there are some point clouds read and displayed by MeshLab. All the captions comes from the authors of the images.

Listing 6 XML Section of a 2D image encoded in a E57 file. The nodes contain all the information about the size and rotation of a 2D image.

```
<pose type="Structure">
    <rotation type="Structure">
        <w type="Float">4.1495323938135464e-001</w>
        <x type="Float">-3.2966867867298849e-003</x>
        <y type="Float">1.0011244071749291e-003</y>
        <z type="Float">-9.0983621533387571e-001</z>
    </rotation>
    <translation type="Structure">
        <x type="Float">-1.0437926491522425e+002</x>
        <y type="Float">7.1935301367989l4e+001</y>
        <z type="Float">-3.9486336466587668e-001</z>
    </translation>
</pose>
<sphericalRepresentation type="Structure">
    <jpegImage type="Blob" fileOffset="21807" length="1372"/>
    <imageHeight type="Integer">2169</imageHeight>
    <imageWidth type="Integer">9892</imageWidth>
    <pixelHeight type="Float">
        6.2860821192160918e-004
    </pixelHeight>
    <pixelWidth type="Float">
        6.2793742364311957e-004
    </pixelWidth>
</sphericalRepresentation>
```

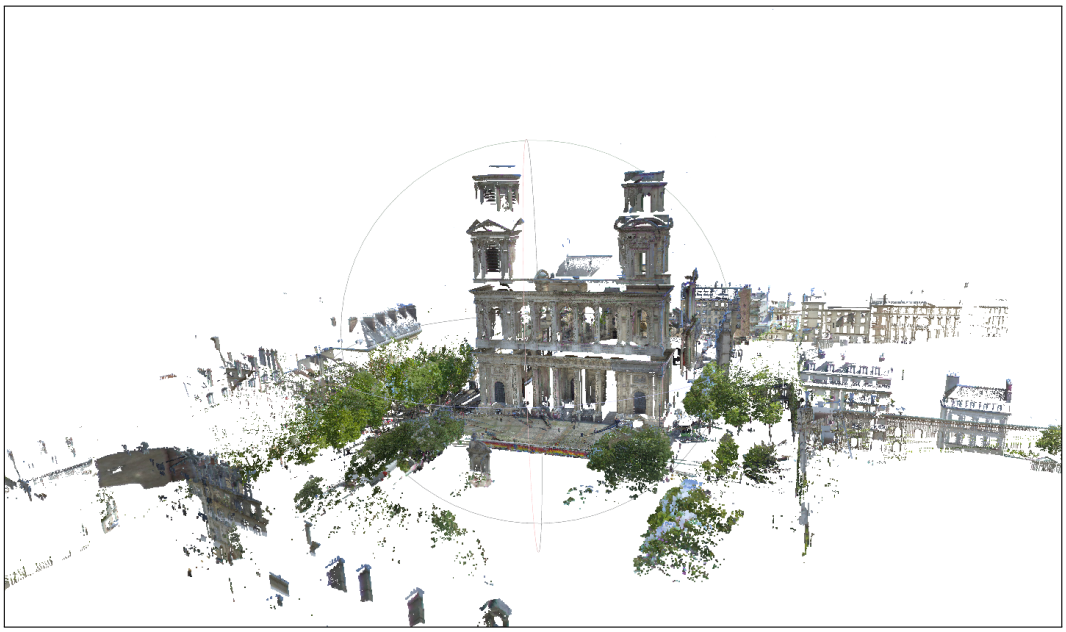


Figure 3.3: *Trimble Data - Multiple Scans point cloud*. This scene was scanned in Paris using a Trimble TX8 for project Terra Mobilita. Color has been captured with a Nikon 7100 using a fish eye Sigma 10 mm. Colorization and export to e57 has been done using Trimble RealWorks 8.1. The project contains one multi-station file (5 scans down-sampled at 50 mm going down from 650 M point to 8 M point) and two files containing gridded scans (down-sampled from 120 M point to 2 M point each). Gridded files are stored as spherical coordinates, and non-gridded ones use XYZ. Screenshot captured in MeshLab 2021.07 during the test phase.

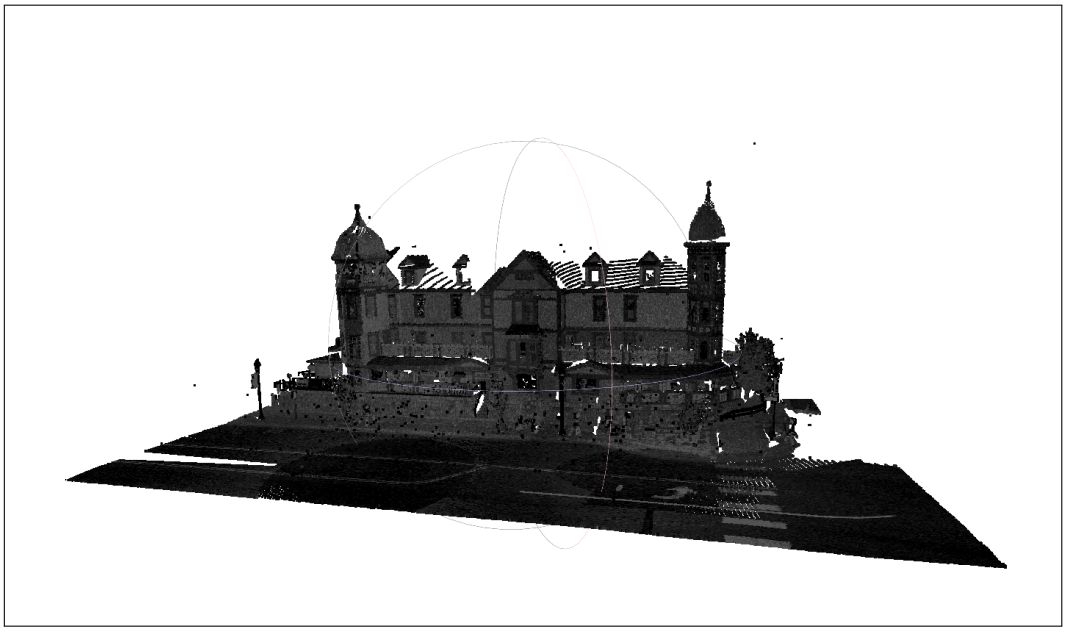


Figure 3.4: *Manitou*. This scene has 5 scans and 5 JPG images using the spherical projection camera model. It was scanned using a Reigl Z420I scanner with an LD3 Texel Camera on top in July 2007. It was converted to E57 using LD3 Studio V5.1. Screenshot captured in MeshLab 2021.07 during the test phase.

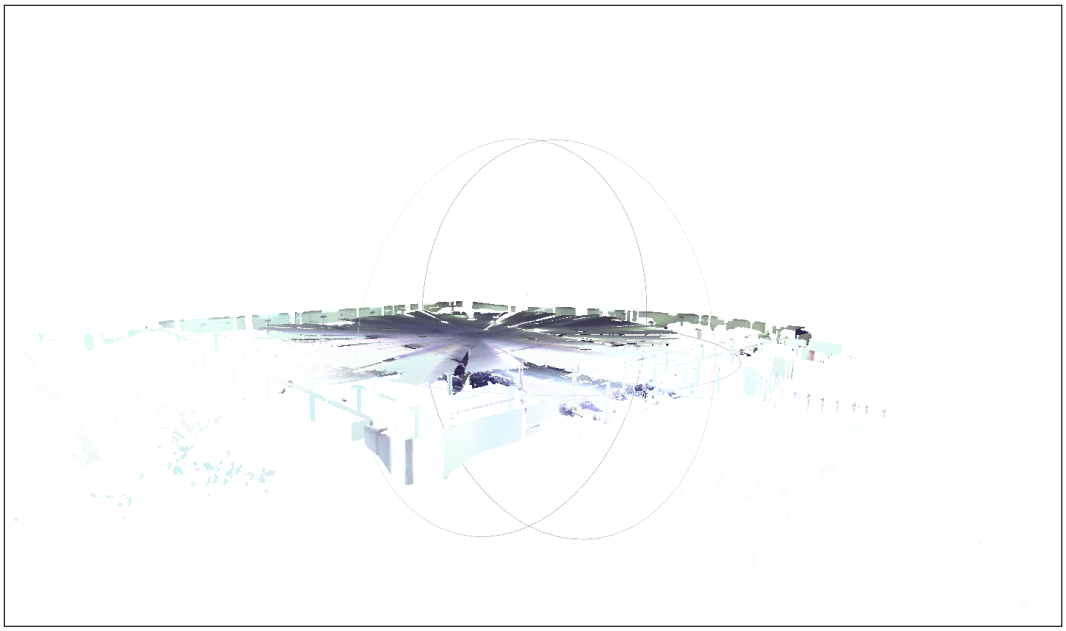


Figure 3.5: *Garage*. This scene has a large 27M point 360 degree scans and a PNG using the spherical projection camera model. It was scanned using a new Faro Focus 3D scanner.Screenshot captured in MeshLab 2021.07 during the test phase.

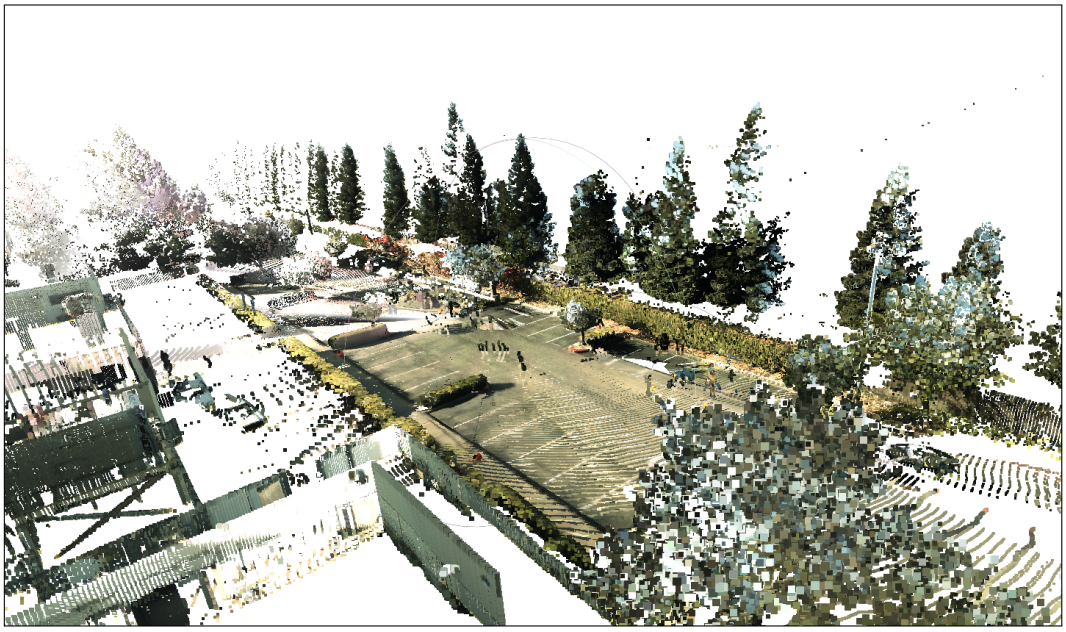


Figure 3.6: *Leica Data - Parking*. This scene has three 360 degree scans and multiple JPG images using the pinhole projection camera model. It was scanned using a Leica ScanStation 2 and converted to E57 using Leica's Cyclone release. WARNING: There are no scan names given, the image names have an illegal file name character(:), and the camera orientation is Z forward and Y down. (E57 standard is Z backward and Y up). Screenshot captured in MeshLab 2021.07 during the test phase.

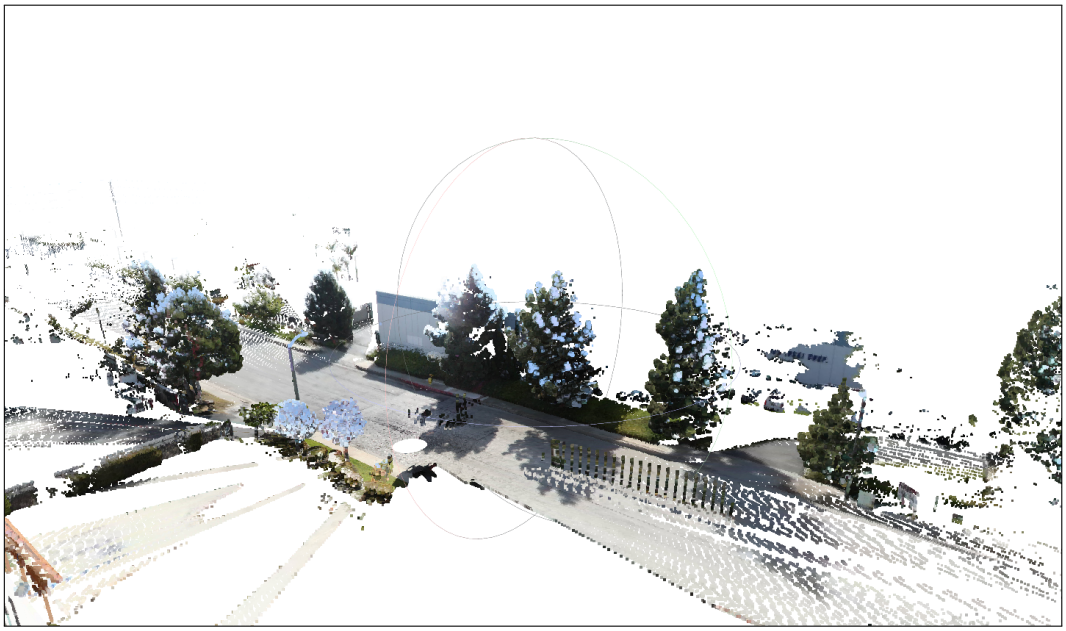


Figure 3.7: *Station*. This is a single scan taken with a Leica C10 scanner and Nodel Ninja Panoramic camera. The 271 MB ptx file and 36 MB JPG was reduced to an 117 MB e57 file. Further, zipping the e57 file only produced a 103 MB zip file. It was converted to E57 using LD3 Studio V5.1. Screenshot captured in MeshLab 2021.07 during the test phase.

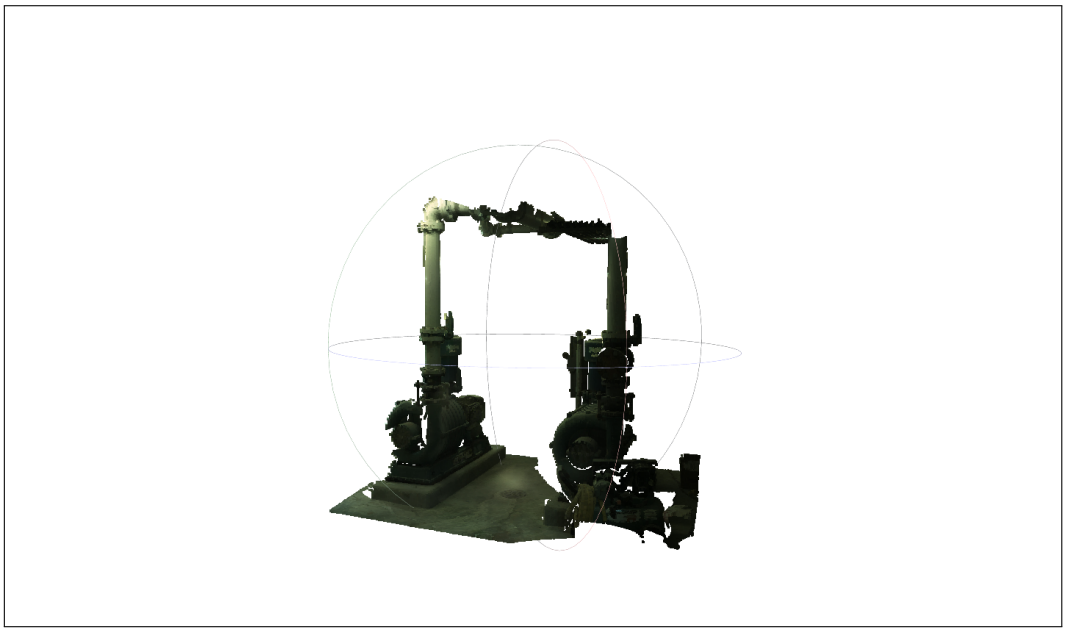


Figure 3.8: *Pump*. This is a single structured scan stored as cartesian and spherical points. Data includes intensity and RGB color. Screenshot captured in MeshLab 2021.07 during the test phase.

Chapter 4

FilterPlugin: ICP, Overlapping Meshes & Global Alignment

Chapter 2 introduced point clouds and the process of surface reconstruction (Section 2.2). This chapter will describe a common problem regarding point alignment, known as the *point set registration problem*, and how the existing code for the alignment process has been refactored between the *VCGLib* and MeshLab. Thanks to the refactoring we succeeded to implement a new FilterPlugin, allowing PyMeshLab users to perform alignment between meshes, see overlapping meshes and execute global alignment of a set mesh.

4.1 What is the Point Set Registration problem

The **Point Set Registration** is a process used to find a transformation matrix that allows two meshes to be aligned, i.e. it searches the transformation that minimize the distance between portions of the meshes where they overlap. This is used as the core problem of registering together different scans of the same object. In practice, the problem is solved by an iterative approach that solves a specific case of the least-squares problem at each step, and it can be formalized in this way.

Point Set Registration Problem Let be R, S two transformation matrices, belonging to the *Reference Mesh* and *Source Mesh*, in a three-dimensional space. The problem is finding a transformation matrix, called T, to be applied to the *Source Mesh* such that the geometrical distance between itself and the *Reference Mesh* is *minimum*.

$$T^* = \min_T \text{dist}(R, T \cdot S) = \min_T ||R - (T \cdot S)||_2 \tag{4.1}$$

There are algorithms that use different approaches to solve the problem, such as the "*Correspondence-based Methods*" and the "*Simultaneous Pose and Correspondence Methods*". The latter approach will describe the upcoming algorithm, and it is used inside MeshLab: the *Iterative Closest Point Algorithm*.

4.1.1 The Iterative Closest Point Algorithm

The ICP algorithm [22] was invented by *Yang Chen* and *Gérard Medioni*. This algorithm can be summarized in a few steps.

Input Two parameters are taken in input: the Reference Mesh and Source Mesh to align, plus some other optional parameters to adjust the result.

Output The transformation matrix, when applied to the Source Mesh, will align it to the Reference Mesh.

Steps The algorithm steps are:

1. For each point in the source mesh, pair the closest point in the reference point cloud.
2. Estimate the combinations of rotation and translation using a minimization technique which will best align to its match found in the previous step.
3. Transform the source points using the computed transformation.
4. Loop the previous steps until a certain threshold is reached.

The ICP algorithm, and its various variants, are used for fine aligning meshes pair-wise.

4.1.2 ICP MeshLab's Implementation

MeshLab's ICP algorithm implementation uses the formulation proposed by *Szymon Rusinkiewicz* and *Marc Levoy* in their paper "*Efficient Variants of the ICP Algorithm*" [23]. According to the research of the two authors, there exist several variants of the ICP algorithm, which can be classified mainly in six points:

1. **Selection** of some points set in one or both the meshes.
2. **Matching** these points to samples in the other mesh.
3. **Weighting** the corresponding pairs suitably.
4. **Rejecting** some pairs based on looking at each pair individually or globally.
5. Assigning an **Error Metric** based on these pairs.
6. **Minimizing** the error metric.

Rusinkiewicz et al. at the end of their paper proposed an efficient algorithm born from the combinations they studied. According to the tests done by the authors, the main stage influencing the algorithm convergence is the *matching* phase. The authors propose to use a projection-based algorithm to generate the point correspondences. They combine the matching algorithm with a point-to-plane error metric and the standard "select-match-minimize" proposed by *Chen* and *Medioni*. For the other stages of the ICP process, they suggest using the simplest approaches such as random sampling (*Masuda* [24]) for the selection phase, constant weight to assign to each corresponding point pairs chosen and for the rejecting phase to use a distance threshold.

MeshLab implements the ICP algorithm using the `vcg::AlignPair` class provided by the VCGLib library.

The ICP algorithm implemented in MeshLab returns an instance of `vcg::AlignPair::Result` class, which holds the computed transformation matrix to apply onto the moving mesh to be aligned with the reference one. There are also data contained in the structure, which are the chosen points on the reference mesh and the chosen points on the moving mesh before the algorithm's execution. Plus, the result contains an error code to check if anything went wrong during the ICP process and statistical data.

4.2 Multiview Registration

When more than two meshes have to be aligned, a problem called *Multiview Registration* comes up. For more than two meshes, Chen and Medioni proposed in their paper to:

1. Align two meshes of the set in a single one, called "*metaview*".
2. Align the *metaview* with another mesh.
3. Repeat the second step until there are no more meshes to align.

Using this approach, as intuitive as it may seem, is not a good idea because it can produce a final mesh misaligned due to a terrible distribution error. *Kari Pulli* [25] came up with a better solution to avoid this problem. *Pulli* suggests performing a pairwise alignment between every mesh and each of its overlapping meshes once. Instead of using all the meshes to process the alignment, the algorithm uses only the pairwise constraint that should be always satisfied. The constraints are great for space efficiency because they use less memory than the original data, allowing the algorithm to handle large data sets at the same time without worrying. Then, the algorithm incrementally enforces these constraints to obtain a global multiview alignment, using a method that does not depend on the initial registration transformations. This method is less likely to get stuck into a local minimum, moreover distributes the alignment error between meshes.

The algorithm proposed by *Pulli* can be read in the Listing 7. The multiview alignment begins by choosing the mesh with the most connections or overlapping areas, putting it inside the *active set*, and finally inserting the remaining meshes inside the *dormant set* except for the one chosen previously. Then the algorithm adds the meshes in the dormant set into the active set at the time. At each iteration, the mesh with the most connections coming from the dormant set is chosen and a queue of still moving meshes is initialized with the current view. This queue will be processed until it becomes empty by removing the top element and aligning it with the neighbors who are in the active set. If the *alignment error* is small enough, then the algorithm will merge the current mesh's neighbors into the queue. So, the error gets distributed evenly among the mesh pairs in the queue loop. A view is aligned to another using the ICP algorithm.

Listing 7 Code for the multiview alignment algorithm by Kari Pulli.

```
1   activeSet = {}
2   dormantSet = views
3   curr = mostLinks(dormantSet, dormantSet)
4   activeSet.add(curr)
5   dormantSet.remove(curr)
6
7   while (!dormantSet.empty()) {
8
9       queue = {}
10      curr = mostLinks(dormantSet, activeSet)
11      activeSet.add(curr)
12      dormantSet.remove(curr)
13      queue.push(curr)
14
15      while (!queue.empty()) {
16
17          curr = queue.pop()
18          neighbors = activeSet.neighbors(curr)
19          relativeChange = align(curr, neighbors)
20
21          if (relativeChange > tolerance) {
22              queue.merge(neighbors)
23          }
24      }
25  }
```

4.3 Refactoring

Previously in MeshLab, the code used for the alignment was tightly coupled to the **Align Tool**, see the Figure 1.7. Thus, a refactoring has been done to relax the coupling of the code, porting it to the VCGLib.

Refactoring is not about moving code from one codebase to another or simply cleaning up

the code, on the contrary, this process requires a lot of consideration. As said by the software developer *Martin Fowler* in its book "*Refactoring: Improving the Design of Existing Code*" [26]:

"*A **refactoring** is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*"

Therefore, this operation has to be done meticulously, otherwise, there is the risk to break already working code and the program flow control may be altered, especially if the classes being considered are *legacies*. Also, the refactoring process allowed us discovering new bugs never found in the previous code implementation.

4.3.1 "Decoupling" the MeshTree and the OccupancyGrid classes

As said in the introduction of this section, the previous code for the alignment was tightly coupled to MeshLab. The main classes who needed to be refactored were: the `MeshTree` and the `OccupancyGrid`. These two classes have been abstracted, templatized and moved inside the VCG Library to be independent from MeshLab and usable over generic types of meshes instead of being customized to the specific types of MeshLab.

The `MeshTree` class, as the name suggests, is used to build a tree data structure of meshes that will be aligned. The second class is an abstraction of the overlapping of the views to understand the connections between them.

Dependencies from Qt

Both the classes had dependencies to the *Qt* framework, so the first step done during the refactoring was to remove all the references to the framework. The main classes used were the `QtList` and `QStringList`. Fortunately, all the modern C++ implementations contain the *STL* [1], so the `QtList` was replaced with the `std::list` and thanks to the template mechanism of C++ the `QtStringList` was converted into `std::list<std::string>`. Also, all the loops iterators were replaced with the *range-based for loop* available since *C++11*, so if in the future we would like to change the class implementation of a data structure there won't be the necessity to modify the entire source code.

Cleaning code

During the refactoring some type declarations were redefined to avoiding *naming collision*, so for example the variables declared as `vector` were changed into `std::vector`. Moreover, the code contained pointer variables initialized with zeros, so the values have been replaced with the `nullptr` keyword, according to the C++ best practices.

Bug founds

Once the refactoring was completed, we began to make some tests to check if the operation broke the existing code. In a test that we will show later in the Section 4.6, we spot a bug contained in the previous code version that has been fixed. If during the global alignment process a mesh had no connected components, then MeshLab would crash due to a failed assertion, so we add just a simple check to fix the bug.

4.4 Implementation of the filters

The new FilterPlugin has three filters that will be explained as follows.

[1] *Standard Template Library*: a library containing general-purpose data structures and algorithms.

4.4.1 First Filter: local alignment between two meshes

The "*Local Alignment Between Two Meshes*" filter just applies the ICP algorithm, previously described, over two elements: the *reference* and the *source* meshes. The first mesh is kept fixed during the ICP process, the latter will be rotated and translated to be aligned with the other mesh.

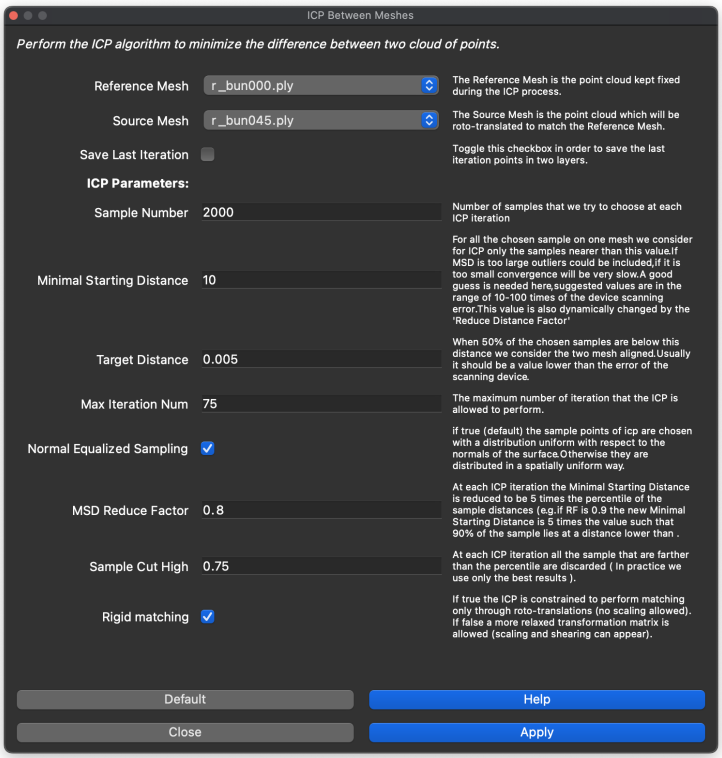


Figure 4.1: The local alignment filter. Screenshot captured in *MeshLab2021.09d_nightly*.

The filter has parameters to apply to the ICP algorithm coming from the Align Tool, plus a new checkbox, *Save Last Iteration*, which if it is flagged, it allows to use the final alignment's result of the two meshes to create two new layers defined:

- *Chosen Source Points*: containing the selected points for the alignment of the Source Mesh prior to the filter application.
- *Corresponding Chosen Points*: containing the corresponding points from the Reference Mesh which will be aligned to Source Mesh.

The Figure 4.2 below shows the generated points by the applying of the new ICP filter.

4.4.2 Second Filter: overlapping meshes

This refactoring process shows a good example of code reusing, in fact, from the refactoring process we were able to uncouple these classes and then create a new piece of software, that is, a new filter. The "*Overlapping Meshes*" filter helps to understand which meshes overlap between themselves. This was possible due to the `OccupancyGrid` class that now is no more highly coupled with the `MeshTree` class, relaxing the coupling lead us to a more code reusability inside the codebase.

The filter returns to PyMeshLab a map containing an array of integer tuples, where each tuple has two IDs, respectively, for the source and target mesh. Each ID is associated by MeshLab when loading the mesh in the document. The Listing 8 shows a text output after the filter's execution.

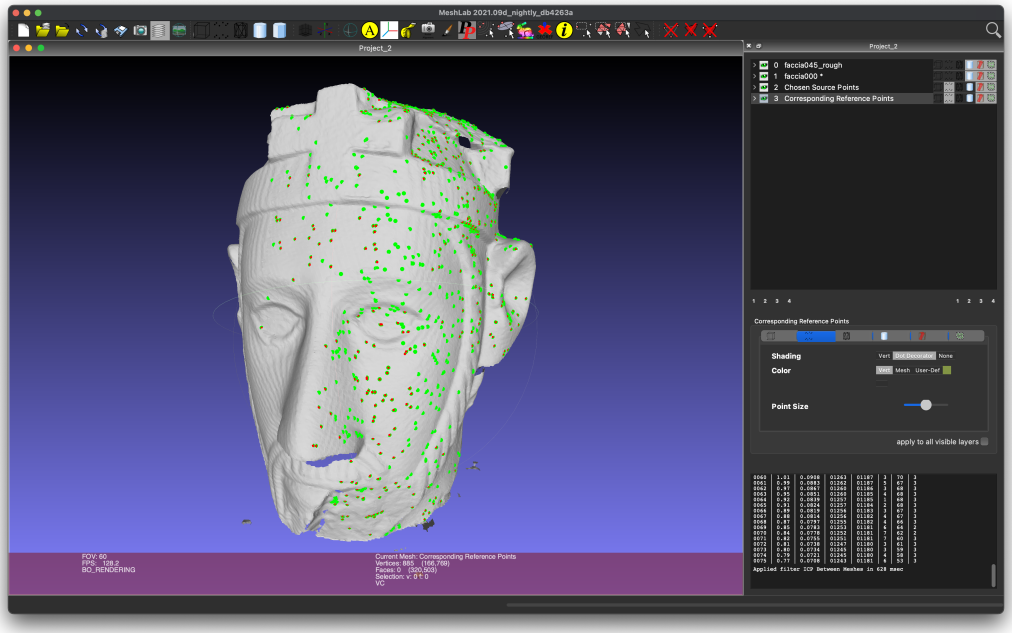


Figure 4.2: This figure shows the two new layers generated by the "ICP Between Meshes" filter. The green points are the "*Chosen Source Points*", the red points are "*Corresponding Reference Points*". Screenshot captured in *MeshLab2021.09d_nightly*.

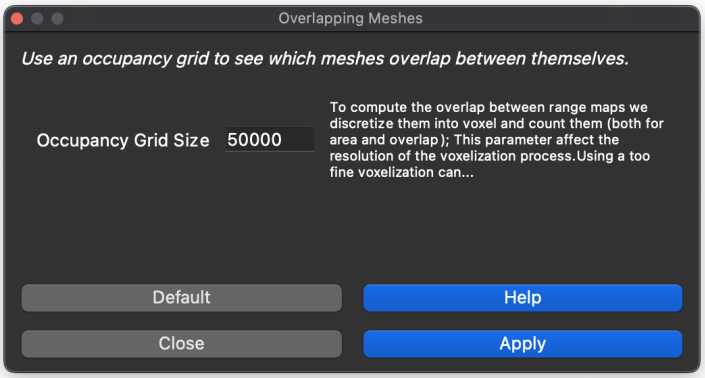


Figure 4.3: The overlapping meshes filter. Screenshot captured in *MeshLab2021.09d_nightly*.

4.4.3 Third Filter: global alignment between meshes

The main classes used by the Align Tool, to perform the global alignment, are the `MeshTree` and the `OccupancyGrid` previously explained in Section 4.3.1.

The "*Global Alignment Between Meshes*" filters apply the alignment process as described in the Section 4.2. The filter, in addition to the ICP parameters, has a category for the *Arc Creation Parameters*, which were inherited by the Align Tool.

Listing 8 Here is the output of the filter's execution. Each pair is represented by the first string of each line, the first element indicates the source mesh and the second one the target. The arrow (→) is read as "*overlaps with*".

```
[4 → 5]: Mesh "bun045.ply" overlaps with "bun000.ply".
[0 → 5]: Mesh "bun315.ply" overlaps with "bun000.ply".
[3 → 4]: Mesh "bun090.ply" overlaps with "bun045.ply".
[0 → 4]: Mesh "bun315.ply" overlaps with "bun045.ply".
[3 → 5]: Mesh "bun090.ply" overlaps with "bun000.ply".
[2 → 3]: Mesh "bun180.ply" overlaps with "bun090.ply".
[0 → 1]: Mesh "bun315.ply" overlaps with "bun270.ply".
[1 → 5]: Mesh "bun270.ply" overlaps with "bun000.ply".
[1 → 2]: Mesh "bun270.ply" overlaps with "bun180.ply".
[1 → 4]: Mesh "bun270.ply" overlaps with "bun045.ply".
[0 → 3]: Mesh "bun315.ply" overlaps with "bun090.ply".
[2 → 4]: Mesh "bun180.ply" overlaps with "bun045.ply".
[1 → 3]: Mesh "bun270.ply" overlaps with "bun090.ply".
[2 → 5]: Mesh "bun180.ply" overlaps with "bun000.ply".
[0 → 2]: Mesh "bun315.ply" overlaps with "bun180.ply".
```

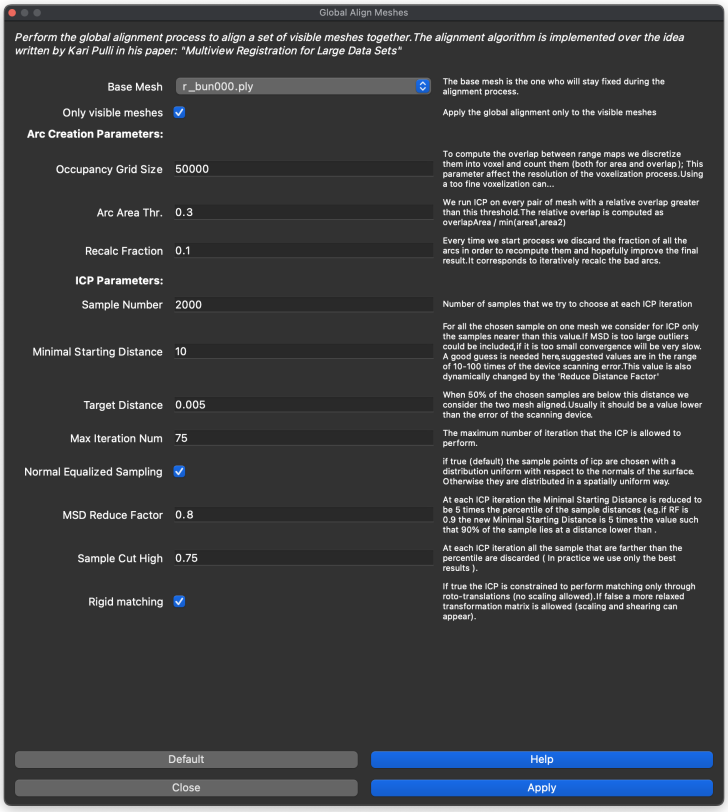


Figure 4.4: The global alignment filter. Screenshot captured in *MeshLab2021.09d_nightly*.

4.5 Some limitations in PyMeshLab

The new filter allows aligning two meshes, and beside the 4x4 matrix encoding the transformation that align the first mesh to the second, it returns a collection of statistical data to understand the performances of the ICP algorithm. However, during the testing of the filter in PyMeshLab, we discovered a problem. The map generated by the filter is an associative array of a list of doubles, like the one shown below.

Listing 9 The function's body is used to parse the map returned by the filter execution in PyMeshLab. It's reported only a tiny snippet of the true code. The complete code is contained in the file linked to this GitHub page.

```
pybind11::dict
toPyDict(const std::map<std::string, QVariant>& qVariantMap) {

pybind11::dict outDict;

for (const auto& p : qVariantMap){
        if (std::string(p.second.typeName()) == "int"){
                outDict[p.first.c_str()] = p.second.toInt();
        }
        else if (std::string(p.second.typeName()) == "double"){
                outDict[p.first.c_str()] = p.second.toDouble();
        }
        else if (std::string(p.second.typeName()) == "float"){
                outDict[p.first.c_str()] = p.second.toFloat();
        }
         ....
        else {
                std::cerr << "Warning: type "
                << p.second.typeName()
                << " still not supported for py::dict conversion\n"
                << "Please open an issue on GitHub about this.";
        }
}
return outDict;

}
```

```
std::map<std::string, QVariant> {
    {"min_dist_abs",        QVariant::fromValue(minDistAbs)},
    {"pcl_50",              QVariant::fromValue(pcl50)},
    {"sample_tested",       QVariant::fromValue(sampleTested)},
    {"sample_used",         QVariant::fromValue(sampleUsed)},
    {"distance_discarded",  QVariant::fromValue(distancedDiscarded)},
    {"border_discarded",    QVariant::fromValue(borderDiscarded)},
    {"angle_discarded",     QVariant::fromValue(angleDiscarded)},
};
```

When executing the filter function in PyMeshLab, it returns an object of type `None` and not a dictionary containing the list of doubles shown above. Using the verbosity mode in PyMeshLab, a log has been printed into the terminal window:

> *"Warning: type std::list still not supported for py::dict conversion"*

Hence, we discovered that PyMeshLab doesn't support the STL containers as values for the keys. This error is thrown by the `pybind11::dict toPyDict` function discussed in the Section 1.3.2. The function invoked acts as a simple parser, and since there are no branches that handle the STL containers, then the parse fails. The fix is almost straightforward to apply, we just need to add the containers to the function, and we are done. The code for the `toPyDict` function is shown in the Listing 9.

4.6 Alignment Tests

To test the refactored code, we used two datasets: the first one is a set of 3D scanned sculpture's head and the second one is the well-known Stanford's Bunny that helped us find a bug in the previous alignment code. To understand the tests, we compared them with the old MeshLab stable release, both in single-precision and double-precision mode. The testing results differs for a non-significant digit, since in the ICP algorithm there is a little randomness factor used to align the meshes, see Figure 4.5.

```
Starting Processing of 6 glued meshes out of 6 meshes
Computing Overlaps 6 glued meshes...
Arc with good overlap      6 (on      15)
      0 preserved 0 Recalc
(  1/  6)  4 ->  5 Aligned AvgErr dd=0.000420 -> dd=0.000412
(  2/  6)  0 ->  5 Aligned AvgErr dd=0.000997 -> dd=0.000794
(  3/  6)  3 ->  4 Aligned AvgErr dd=0.001035 -> dd=0.000950
(  4/  6)  0 ->  4 Aligned AvgErr dd=0.000910 -> dd=0.000755
(  5/  6)  3 ->  5 Aligned AvgErr dd=0.001128 -> dd=0.000922
(  6/  6)  2 ->  3 Aligned AvgErr dd=0.001563 -> dd=0.001448
Completed Mesh-Mesh Alignment: Avg Err 0.001; Median 0.001; 90% 0.001
Completed Global Alignment (error bound 0.0010)
```

(a) Old Align Tool before refactoring

```
Starting Processing of 6 glued meshes out of 6 meshes
Computing Overlaps 6 glued meshes...
Arc with good overlap      6 (on      15)
      0 preserved 0 Recalc
(  1/  6)  4 ->  5 Aligned AvgErr dd=0.000394 -> dd=0.000389
(  2/  6)  0 ->  5 Aligned AvgErr dd=0.000922 -> dd=0.000708
(  3/  6)  3 ->  4 Aligned AvgErr dd=0.001074 -> dd=0.000990
(  4/  6)  0 ->  4 Aligned AvgErr dd=0.001055 -> dd=0.000799
(  5/  6)  3 ->  5 Aligned AvgErr dd=0.001150 -> dd=0.000948
(  6/  6)  2 ->  3 Aligned AvgErr dd=0.001379 -> dd=0.001311
Completed Mesh-Mesh Alignment: Avg Err 0.001; Median 0.001; 90% 0.002
Global Alignment...Completed Global Alignment (error bound 0.0010)
```

(b) New Align Tool after refactoring

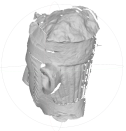
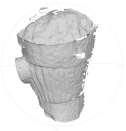
Figure 4.5: The *old* Align tool vs the *new* one results. These screenshots are captured from *MeshLab2021.09d_nightly* (before and after refactoring) running in single-precision mode.

The following subsection will show the datasets used during the test phase.

First dataset

This dataset has eight different rangemaps. The Table 4.1 shows all the scans of a king's sculpture. Each rangemap is acquired by a different angle. The scans have been loaded inside MeshLab and aligned using the Align Tool. Once the rough alignment is finished, global set registration performs and align the meshes globally. The results of the global alignment are shown in Table 4.2.

Table 4.1: First dataset

First dataset				
Mesh	Filename	Vertices	Faces	Size
	faccia000.ply	85.849	166.259	5.2 MB
	faccia045.ply	79.150	154.244	4.8 MB
	faccia090.ply	77.538	150.170	4.7 MB
	faccia135.ply	74.207	144.557	4.5 MB

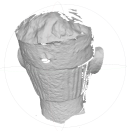
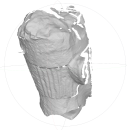
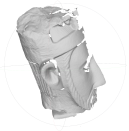
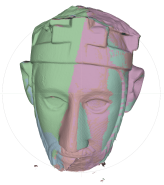
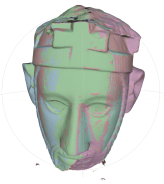
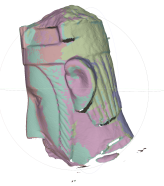
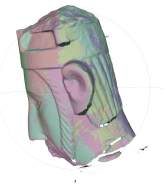
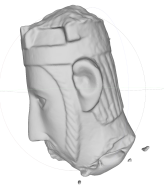
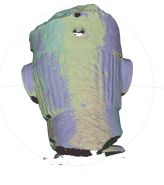
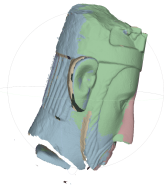
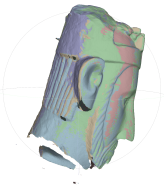
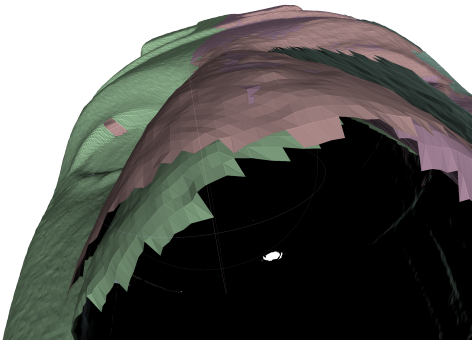
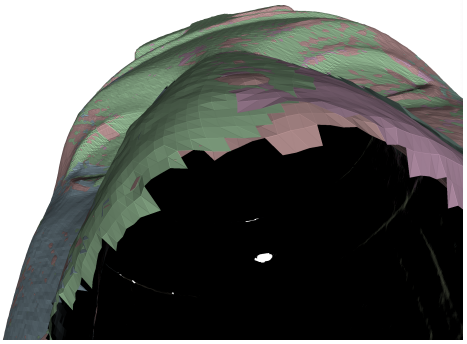
Continuation of Table 4.1				
Mesh	Filename	Vertices	Faces	Size
	faccia180.ply	79.752	154.807	4.8 MB
	faccia225.ply	73.853	143.021	4.5 MB
	faccia270.ply	79.110	153.721	4.8 MB
	faccia315.ply	80.281	156.662	5.9 MB

Table 4.2: First dataset results

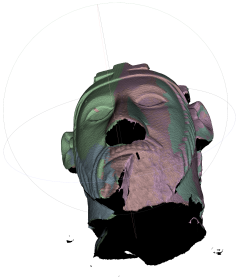
First dataset		
Rough Alignment	Fine Alignment	Reconstruction
		
		
		
		



(a) Rough Aligment - 1



(b) Fine Alignment - 2



(c) Rough Aligment - 1

(d) Fine Alignment - 2

Figure 4.6: After a global fine alignment step, the range maps are all finely aligned together. The different scans (in different color) before the fine alignment did not match perfectly, (see for example the nose in the top left figure) while after the process there are no discontinuities between the various scans.

Second dataset

This dataset contains ten scans of the Stanford's Bunny (Table 4.3), which is a standard dataset used as a reference. Like for the statue, the bunny has been acquired by different angles and prospective, once the scan is complete, the rangemaps have been aligned with a manual point base alignment, and then we applied the global alignment process (Table 4.4).

Table 4.3: Second dataset

Second dataset				
Mesh	Filename	Vertices	Faces	Size
	bun000.ply	40.256	79.013	2.0MB
	bun045.ply	40.097	78.686	2.0MB

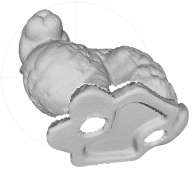
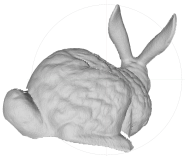
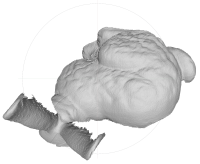
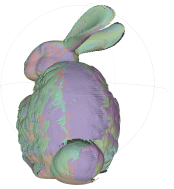
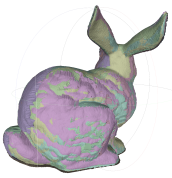
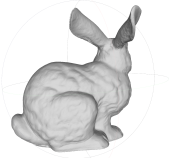
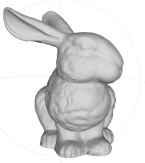
Continuation of Table 4.3				
Mesh	Filename	Vertices	Faces	Size
	bun090.ply	30.379	58.865	1.7MB
	bun180.ply	40.251	79.119	2.0MB
	bun270.ply	31.701	61.244	1.7MB
	bun315.ply	35.336	68.934	1.8MB
	chin.ply	37.738	73.602	1.9MB
	ear_back.ply	32.193	62.639	1.7MB
	top2.ply	38.298	75.100	1.9MB
	top3.ply	36.023	70.640	1.8MB

Table 4.4: Second dataset results

Second dataset results		
Rough Alignment	Fine Alignment	Reconstruction
		
		
		
		

Chapter 5

Conclusions

Thanks to this work now the MeshLab and PyMeshLab users have more features to use for their daily usage in the context of the processing of 3D scanned data. The creation of the new E57 plugin will allow users to import scanned data in his format directly inside MeshLab, where they can subsequently process. Thanks to the refactoring of the alignment core, the PyMeshLab users can now access to the set of functionality for registration that before were possible only inside the desktop version of MeshLab, this allows for example to use mesh alignment automatically in their script to create new automatic processing pipelines, boosting their productivity to even higher level.

5.1 Contributions

All the contributions made to MeshLab and VCGLib codebases can be found on GitHub. Each contribution has been made through a pull request using git. To see the pull requests quickly, here are the links:

- "*Adding IOPlugin for E57*": https://github.com/cnr-isti-vclab/meshlab/pull/1036.
- "*Adding ICP FilterPlugin*": https://github.com/cnr-isti-vclab/meshlab/pull/1087.
- "*Adding Overlapping Meshes*": https://github.com/cnr-isti-vclab/meshlab/pull/1101.
- "*Add MeshTree and OccupancyGrid*": https://github.com/cnr-isti-vclab/vcglib/pull/177.

5.2 Future developments

Right now, MeshLab can only export E57 files single-layer, in the future we would like to give the capability to export several multi layers-files. It would be great to extend the support for others metadata contained in a E57 file, for example loading the contained 2D images to be able to see the photo took during the acquisition by the 3D scanner directly inside MeshLab.

Furthermore, as already discussed in Section 4.5, at the moment PyMeshLab supports only a small subset of C++ types, thus each time we would like to add a new type we have to edit the source code in order to be able to parse and to bind the new type into the Python environment. So, using the pybind11's conversion mechanisms, we could obtain a dynamic type inference without compiling PyMeshLab from the source and removing the dependency from `QVariant` class.

5.3 What has been learned

Before this thesis, I did not know how meshes and other 3D graphics element were implemented in a high-end graphics library such as the VCGLib, also I had never put my hands into a large codebase like MeshLab. Thanks to this work, I begin to understand what are some problems of 3D modelling, designing and building complex interactive applications

that works in this domain.

Developing MeshLab plugins allowed me to learn the basis of mesh processing, from the data structures used for meshes and how the 3D graphics is handled by standard libraries. Also, I learned the power of the bindings in Python and how they can help to reuse existing code to help people's productivity.

Furthermore, from a professional working competences point of view, I saw for the first time a CI/CD (*Continuous Integration / Continuous Development*) pipeline using GitHub actions. For each operating systems, there exists a workflow that compiles and deploys MeshLab in the two versions: the single-precision and the double-precision. It sure was exhausting waiting for the completion of the workflows because MeshLab is not a small program, but it was worth it because now I begin to understand some common practices in professional development of large codebases.

Chapter 6

Acknowledgements

Small premise: these are personal thoughts, the reader is free to ignore them. I am not so fantastic at thanks, but I will try my best.

At the end I made it. After just three years I finished my first, unforgettable, academic journey. I met so many friends, so many wonderful teachers who taught me how to be a "*junior*" computer scientist! Sure, it wasn't all fun and games, but that didn't discourage me because when I have a goal on my mind I am really stubborn. Every time I fell down, I learn a new lesson, that is make people humans, the ability to learn from mistakes and I strongly believe in this so-called "power".

I would like to thank my friends, they were always present in difficult times, also I would like to say thanks to my university colleagues that spent plenty of time with me to study and to have fun. I would like to thank my part-time job colleagues because they taught me how to behave inside a professional environment such a software house. I would like to say a huge thanks to the professor Paolo Cignoni and Alessandro Muntoni for their availability to make this thesis. Furthermore, I would like to thank my girlfriend, Alessia, for the considerable amount of patience and love she gave to me. And, at last but not for importance, I would like to thank my family, I have no words to describe my gratitude to them.

To summarize: thanks to everyone, I'm glad to have people like you by my side. I hope to grow up more and more and learn new things always. I can't wait to see what waits for me next.

By the "Junior" Computer Scientist:
Gabriele Pappalardo

Bibliography

[1] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia, "MeshLab: an Open-Source Mesh Processing Tool," in *Eurographics Italian Chapter Conference*, V. Scarano, R. D. Chiara, and U. Erra, Eds. The Eurographics Association, 2008.

[2] P. Kyriakou and S. Hermon, "Building a dynamically generated virtual museum using a game engine." in *Digital Heritage* (*1*), 2013, p. 443.

[3] G. Constantinou, G. Wilson, S. Sadeghi-Esfahlani, and M. Cirstea, "An effective approach to the use of 3d scanning technology which shortens the development time of 3d models," in *2017 International Conference on Optimization of Electrical and Electronic Equipment* (*OPTIM*) *2017 Intl Aegean Conference on Electrical Machines and Power Electronics* (*ACEMP*), 2017, pp. 1083–1088.

[4] M. M. Shashkov, C. S. Nguyen, M. Yepez, M. Hess-Flores, and K. I. Joy, "Semi-autonomous digitization of real-world environments," in *2014 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games* (*CGAMES*), 2014, pp. 1–4.

[5] M. Koopaie and S. Kolahdouz, "Three-dimensional simulation of human teeth and its application in dental education and research," *Medical journal of the Islamic Republic of Iran*, vol. 30, p. 461, 2016.

[6] B. Bradley, A. D. Chan, and M. Hayes, "A 3d scanning system for biomedical purposes," *International journal of advanced media and communication*, vol. 3, no. 1-2, pp. 35–54, 2009.

[7] L. Frizziero, G. M. Santi, A. Liverani, F. Napolitano, P. Papaleo, E. Maredi, G. L. D. Gennaro, P. Zarantonello, S. Stallone, S. Stilli, and G. Trisolino, "Computer-aided surgical simulation for correcting complex limb deformities in children," *Applied Sciences*, vol. 10, no. 15, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/15/5181

[8] J. R. Jastifer and P. A. Gustafson, "Three-dimensional printing and surgical simulation for preoperative planning of deformity correction in foot and ankle surgery," *The Journal of Foot and Ankle Surgery*, vol. 56, no. 1, pp. 191–195, 2017.

[9] Wikipedia, "Unified Modeling Language — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Unified%20Modeling%20Language&oldid=1044783151, 2021, [Online; accessed 22-September-2021].

[10] F. Robinet, R. Arnaud, T. Parisi, and P. Cozzi, "gltf: Designing an open-standard runtime asset format," *GPU Pro*, vol. 5, pp. 375–392, 2014.

[11] M. Kazhdan and H. Hoppe, "Screened poisson surface reconstruction," *ACM Transactions on Graphics* (*ToG*), vol. 32, no. 3, pp. 1–13, 2013.

[12] A. Muntoni and P. Cignoni, "PyMeshLab," Jan. 2021.

[13] pybind, "pybind11." [Online]. Available: https://pybind11.readthedocs.io/en/latest/

[14] M. Wilson, *Imperfect C++ : practical solutions for real-life programming*. Addison-Wesley, 2009.

[15] B. Curless, "From range scans to 3d models," *SIGGRAPH Comput. Graph.*, vol. 33, no. 4, p. 38–41, Nov. 1999. [Online]. Available: https://doi.org/10.1145/345370.345399

[16] P. Cignoni, F. Ganovelli, N. Pietroni, F. Ponchio, G. Ranzuglia, M. Corsini, M. D. Benedetto, L. Malomo, A. Muntoni, M. Tarini, T. Alderighi, G. Palma, M. Callieri, G. Marcias, M. Dellepiane, A. Maggiordomo, J. Senneker, N. Wenzel, I. Manolas, and M. Potenziani, "Vcglib 2021.07," Jul. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5113906

[17] P. Cignoni, "Introduction to vcg library: a mesh processing library."

[18] L. M. Rocca and L. Serrano, *Chapter 9*. Manning, 2021.

[19] ASPRS, "Laser (las) file format exchange activities." [Online]. Available: https://www.asprs.org/divisions-committees/lidar-division/laser-las-file-format-exchange-activities

[20] D. Huber, "The astm e57 file format for 3d imaging data exchange," in *Proceedings of SPIE Electronics Imaging Science and Technology Conference (IS&T), 3D Imaging Metrology*, vol. 7864, January 2011.

[21] "Software tools for managing e57 files." [Online]. Available: http://www.libe57.org/

[22] Y. Chen and G. Medioni, "Object modeling by registration of multiple range images," *Image Vision Comput.*, vol. 10, pp. 145–155, 01 1992.

[23] S. Rusinkiewicz and M. Levoy, "Efficient variants of the icp algorithm," in *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*, 2001, pp. 145–152.

[24] T. Masuda, K. Sakaue, and N. Yokoya, "Registration and integration of multiple range images for 3-d model construction," vol. 1, 09 1996, pp. 879 – 883 vol.1.

[25] K. Pulli, "Multiview registration for large data sets," in *Second international conference on 3-d digital imaging and modeling (cat. no. pr00062)*. IEEE, 1999, pp. 160–168.

[26] M. Fowler, *Refactoring: Improving the design of existing code*. Addison-Wesley, 2019.