GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Practical Report

submitted in partial fulfillment of the requirements for the course
„Advanced Research Training - Applied Computer Science“

Fitting 3D objects to point clouds in GroIMP using Fibonacci spheres

Maurice Müller

Department Ecoinformatics, Biometrics and Forest Growth, Büsgeninstitut
Georg-August University of Göttingen

7. February 2023

Supervisor:
Prof. Dr. Winfried Kurth

Contents

1 Introduction

GroIMP is a software for plant modeling and automatable growth simulation. It is mostly used for scientific research about the correlation between virtualized plants and virtually generated plants. In nature, plants consist of many structural components, like roots, trunk sections, branches, leaves, fruits, and many more details. Like in nature, the plant growth simulation in GroIMP is usually constructed with fundamental 3D objects, like spheres, cylinders, frustums, cones, boxes, and many more. To let the virtual plants grow or „live“ under different circumstances, the parameters of certain objects are then changed or objects are added, replaced or removed in the current plant model. Goal of most of the projects with GroIMP is to find realistic growth behaviors of selected plant species.

Plants in GroIMP can be represented in two different (commonly used) ways. Assume, a plant has been scanned with a 3D laser scanner. The laser scanner outputs files with all 3D positions that have been detected during the scanning process. Depending on the precision and the resolution of the scanner, more or less points are generated and the surfaces of the scanned object are of higher or lower quality. The resulting files can be imported into GroIMP as point clouds. Point clouds are lists with points that can easily be displayed in the 3D view, but do not have any further meaning for the geometry of plants. Due to their simplicity, point clouds can only be used to visualize the outline of the scanned object.

If plants are generated by XL scripts (or built by hand), they are mostly constructed with objects like cylinders or frustums. In this way of structuring plants, each component gets an own biological meaning and an own mathematical object with a position, a direction, a size and maybe a name tag. As we can see, this way of handling plants is much easier than with point clouds.

A goal of GroIMP is to get detailed information about how plants are structured and how they grow in nature. The most efficient way could be to scan real plants, import them into GroIMP as point clouds, and analyze them. But as we have seen, point clouds are relatively useless for automated processing. By default, point clouds do not contain any metadata about which plant components have been scanned and how they relate to each other. The extraction of a mathematically and biologically useful plant model from a point cloud is, of course, a very complicated process.

In the preceding practical report, a density-based algorithm to cluster point clouds has been described and added to GroIMP. With the results of that work, it became possible to import scanned plants into GroIMP as point clouds and extract different modules of the plant. Most of the modules, for example leaves, branches or fruits, can be clustered automatically if the scan result is of a good quality and the parameters are well chosen for the currently handled point cloud. If modules are clustered wrongly, they can be edited by hand afterwards. This set of features was an important milestone on the way to automated analysis of laser scanned plant models.

The next step to reach that goal is to find out which 3D objects would have to be chosen if the objects should represent the clusters of the point cloud generated by the laser scanner and the belonging components of the real plant. Most of the components of real plants have round shapes and can be described as spheres, cylinders, frustums, and cones.

Goal of this practical report and the belonging course is to design, implement and add following functions to the core of GroIMP:

- Fitting spheres to point clouds
- Fitting cylinders to point clouds
- Fitting frustums to point clouds
- Fitting cones to point clouds
- Fitting automatically chosen objects to point clouds

The item „automatically chosen objects“ will be a kind of automatic mode that decides whether a sphere, a cylinder, a frustum, or a cone describes the point cloud best and returns the best fitting object automatically.

As in the added functions of the preceding practical report, all functions of this project will also be available for XL scripts and in the graphical user interface. All functions in the graphical user interface are designed to be used on one or multiple selected point clouds in the 3D view. Also the XL functions are available for single point clouds and for point cloud arrays.

Note: The XL functions can also be used in **headless mode**, because they do not require graphical interaction with the user.

In analogy to the practical report about point cloud clustering, also this report describes the planning, the design, the implementation and the possible results of the new functions in the main chapters. In the implementation part and the conclusion, also experiences, problems, and solutions of the implementation process are explained and discussed. This could be interesting for future developers. In the appendix, an overview about the new functions in XL and in the point cloud menu is given. This could be interesting for users that want to work with laser scanned plants.

2 Preparation

All required installation and setup steps for Eclipse, Java, GroIMP, and other software are described in detail in the preceding document „Installing and Running GroIMP in Eclipse“ (see references). The setup for the development of the new point cloud fitting functions is the same as that for the preceding development of the point cloud clustering functions. To avoid redundancy, the setup process is not described in this report again and can be reused.

Note: GroIMP was updated to Java 11 in the meantime. Depending on the current version, the Java versions and the compiler versions may differ from the installation guide.

The development of the point cloud fitting algorithms for this report is based on the features that have been added with the preceding practical report „Implementation of point cloud tools in GroIMP“ (see references). The code base is reused, improved and extended in this project.

Recommendation: As further reading you can read the practical report „Implementierung von Hüllkörpern in der Plattform GroIMP für Pflanzenmodellierung“ by Jan Tristan Koch (see references). In the report, J. T. Koch introduced a concept to calculate enveloping ellipsoids for other objects by using principal component analysis. The concept of principal component analysis is not directly used in the implementation of the point cloud fitting algorithms, but has led to a provisional algorithm for cylinder fitting by principal component analysis. The algorithm is explained in the chapter about the software design, but not implemented in GroIMP. The reasons why it was not used in the final implementation are also discussed there.

3 Design

The most important goal of the project described in this practical report is to describe and implement an algorithm to fit a cylinder to a point cloud in GroIMP. To understand what such an algorithm should do in detail, it becomes necessary to look at the mathematical differences between point clouds and cylinders. A point cloud is a low level data structure with raw information about measured positions in a three dimensional space. Mostly, point clouds were created by laser scanners and do not contain any information about the scanned object or parts of it. The only practical way to do something with point clouds is to display them on the screen and try to determine some of their properties by hand.

In contrast to point clouds, cylinders provide much more geometrical information. If a cylinder is stored as object in GroIMP, it provides a position, a direction, a length and a radius. This makes it a high level data structure, compared with a point cloud. Of course, high level data structures have many advantages for plant modeling and growth simulations. This is the reason why an algorithm to transform a low level data structure to a high level data structure is in the interest of GroIMP.

Other high level data structures that are fitted in this project are spheres, frustums, and cones. Frustums and cones are like cylinders, but with different radii. Spheres do not have a direction and a length, but can also be categorized to high level data structures because they have a center point and a radius.

Coming back to cylinders, these can be fitted in many different ways. The question is which is the most appropriate way for this project and the implementation in GroIMP. The first idea was to look into published papers that already describe the problem of fitting cylinders to point clouds in detail. After looking into multiple papers, it became clear that this can not be the most appropriate solution. The implementation of algorithms introduced in most of the papers would be too complicated and time-consuming, because the mathematical strategies often contain integrals, derivations, or other complex numeric transformations that are not supported by Java by default. There are external libraries for these algorithms, but including a further library also means an administrative dependency of the software.

3.1 Fitting cylinders with principal component analysis

So the only way was to think about an other way to fit cylinders to point clouds. An efficient approach of fitting ellipsoids to other objects was introduced in the practical report „Implementierung von Hüllkörpern in der Plattform GroIMP für Pflanzenmodellierung“ by J. T. Koch (see references). In the paper, the principal component analysis was introduced to find object directions, based on their longest length in the three dimensional space. In theory, this algorithm could also be used for cylinder fitting. But in practice, there are some reasons why an algorithm based on principal component analysis is not the best approach for cylinders. In nature, there are not only long thin cylindrical shapes. For example, a slice of a tree trunk could have been scanned and analyzed. Then, the longest length would be the diameter of that slice, but not the direction in which the tree trunk has grown.

To prevent wrong decisions based on long thin cylinders and flat wide cylinders, three components can be calculated: The first component based on the longest length, the second component as ortogonal vector to the first component vector, and the third component as cross product (or ortogonal vector) of the first component vector and the second component vector. By finding three components, three different cylinders can be calculated and their goodness can be determined by comparing each of them with the given point cloud. The best one can then be returned.

But this leads to a further problem. While the longest length of an ellipsoid is the length in the direction given by the principal direction vector, a cylinder has its longest length in a diagonal direction, compared to its direction vector. If a principal component analysis would be applied on a cylinder, the diagonal direction from one point on the circular edge of the base surface to the opposite point on the circular edge on the top surface would be returned. Depending on the ratio between the length and the radius of

the cylinder, this diagonal direction vector can not be used to say anything about the cylinder itself.

The principal component analysis will also fail on approximately spherical or cube-shaped point clouds because the found component is based on random inaccuracies and has nothing to do with the typical „round“ shape of a cylinder. A pseudo code of the principal component analysis algorithm can be found in the appendix. Due to the discussed problems, this algorithm is also not implemented in GroIMP.

3.2 Fitting cylinders with Fibonacci sphere analysis

Considering the problems described above, a more appropriate approach for the original goal has to be found. Because mathematically perfect algorithms are too complicated to implement and a principal component analysis can not be applied to cylinders, the next idea is to go an approximate way. The idea with an approximate way is to generate lots of cylinders and decide which one fits best to the given point cloud. Therefore, cylinders have to be generated with parameters that are nearly uniformly distributed in the whole range of possibilities. Of course, this is not the most efficient way, but it is simple to implement and can be very precise with good parameters. Furthermore, it can be improved later by using an intelligent algorithm that filters the range of possibilities in advance to reduce the number of tried cylinders as much as possible.

As we have seen in the discussions above, the most error-prone part of the cylinder fitting is to find the correct direction vector. The optimal solution to solve this in an approximate way is to generate a list of uniformly distributed direction vectors. To get these direction vectors, a Fibonacci sphere (see references) can be used.

The Fibonacci sphere is generated so that the center of the sphere is equal to the center position of the point cloud. The radius of the generated sphere is equal to the distance between the center position of the point cloud and the position of the point with the most far distance to the center position. This ensures that all points of the point cloud are contained in the volume of the Fibonacci sphere and the Fibonacci sphere encloses the point cloud as tight as possible.

In the next step, a list with all potential direction vectors is created. Each vector is calculated by subtracting the position of one of the points of the Fibonacci sphere from the center position of the point cloud. Now, the points of the Fibonacci sphere and the direction vectors can be used to generate a first version of a list of cylinders. For each cylinder, the radius is adapted so that it is as small as possible, but with all points of the point cloud included. This can be done by comparing the distances of all points to the belonging direction vector of the currently calculated cylinder. The same is done with the length and the position. They are adapted so that the bottom surface and the top surface are moved as close as possible to the point cloud.

After calculating all optimizations for all cylinders (with the uniformly distributed direction vectors), lots of cylinders remain in the list. The last step is to search for the cylinder with the best score.

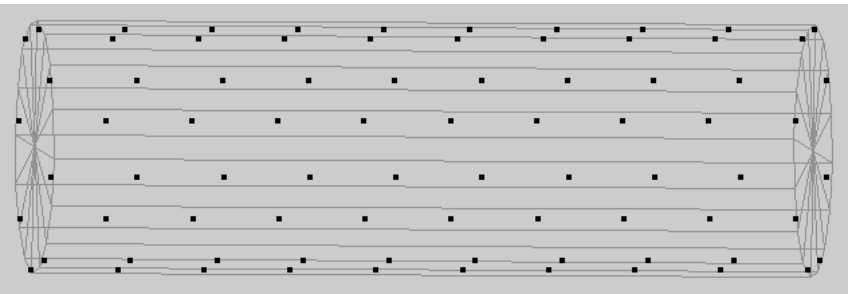

Figure 1: This cylindric point cloud was fitted with the maximum-fitting algorithm. All points are located inside the cylinder and the volume is as minimal as possible.

3.3 Generating a Fibonacci sphere

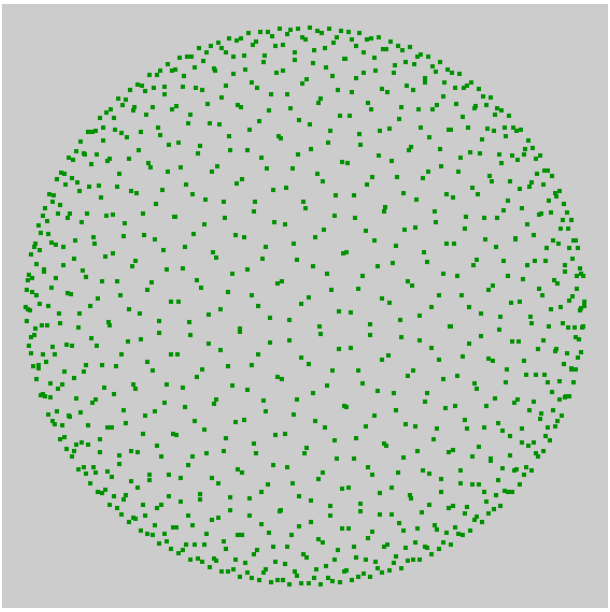

Figure 2: In this image, a Fibonacci sphere is displayed. It was generated around the origin position of the coordinate system, has a radius of one meter and consists of 1000 points.

The algorithm to generate a Fibonacci sphere requires three parameters. The first one is the center position of the sphere, the second one is the radius, and the third one is the number of points. With this information, as many points as requested are generated so that they are uniformly distributed around the surface of such a sphere.

A Fibonacci sphere is generated from the minimum x position to the maximum x position in a linear way. The size of each of the $numberOfPoints$ steps is $diameter/numberOfPoints$, so that the final distance is equal to $2 \cdot radius$. For each x position, a y value and a z value are calculated by getting the current radius (the current x value) and using it with sinus, cosinus and phi for both directions. This algorithm produces uniformly distributed points around the sphere surface because the ratio of points per x distance is equal for each x value. The ratio of points per distance is equal because on the beginning of the sphere and the end of the sphere there are many points due to a „vertical“ surface while in the middle of the sphere there are many points due to a greater radius.

The implementation of the Fibonacci sphere algorithm is also added as pseudo code in the appendix of this document.

3.4 Fitting more object types

Later, in the context of this project, it became apparent that it would be appropriate to also implement fitting algorithms for frustums, cones, and spheres. To fit a sphere to a given point cloud is pretty simple. The algorithm only has to get the center position of the point cloud and the distance between the center position and the most far point. The center position is used as position of the sphere and the most far distance is used as radius. After setting both parameters, the sphere can be returned.

To fit a frustum to a given point cloud is a bit more interesting. It uses the cylinder fitting algorithm to generate a cylinder that looks like the expected frustum, but with two equal radii on the bottom surface and the top surface. The bottom and top surfaces are already correct and can be reused. Technically, these are represented by the position vector and the direction vector of the pregenerated cylinder. The only parameters that have to be recalculated are the radius on the bottom side and the radius on the top side. This is done with a linear regression that uses all points of the point cloud and maps them to a coordinate system in which the direction vector of the frustum represents the x-axis and the plane of the base surface represents the y-axis. This makes it possible to get the x-value from the distance to the base surface and the y-value from the distance to the frustum direction vector for each point in the point cloud. By calculating the linear regression and shifting it to the outer side (increasing the intercept) so that the point with the maximum distance to the x-axis lies on the regression line, the final function for the new radii is produced. By calculating the y-values of this function for the x-values 0 and $lengthOfCylinder$, the new radii for the base surface (0) and the top surface ($lengthOfCylinder$) can be calculated. As an

additional step, the frustum is inverted if the bottom radius is smaller than the top radius.

The cone fitting algorithm is based on the frustum fitting. The first step is to generate a frustum that looks like the resulting cone, but without the tip. In a second step, the tip is added by calculating the missing distance between the top surface of the frustum and the theoretical tip of the expected cone. Then, a cone with the base radius of the frustum and the new length is returned. The remaining distance can easily be calculated because the used frustum is always oriented so that the smaller surface is the top surface and the larger one is the base surface.

As an additional feature, an automatic mode is included in the new point cloud fitting feature of GroIMP. It generates one object of each type (sphere, cylinder, frustum, and cone) and returns the one with the best score, compared with the given point cloud. The automatic mode has two difficulties with frustums. The first one concerns the decision between frustums and cones. In nature, no cone is perfect and the algorithm would always decide to return a frustum, because the mathematically perfect tip of the cone is (normally) never represented by a laser scanned point. Due to this, a frustum would always be smaller than a cone and would always be preferred by the score algorithm. To prevent this, a cone is always returned instead of a frustum if the top radius has a maximum of five percent of the base radius. The other problem concerns the decision between frustums and cylinders. In nature, no „cylinder“ is perfect and the algorithm would always decide to return a frustum, because mathematically perfect cylinders do (normally) not exist in laser scanned point clouds. To prevent this, a cylinder is always returned if the top radius of a frustum has at least 95 percent or more of the base radius of the frustum.

For all types of objects, an average-radius mode and a maximum-radius mode are available. Mode details can be found in the appendix about the algorithms.

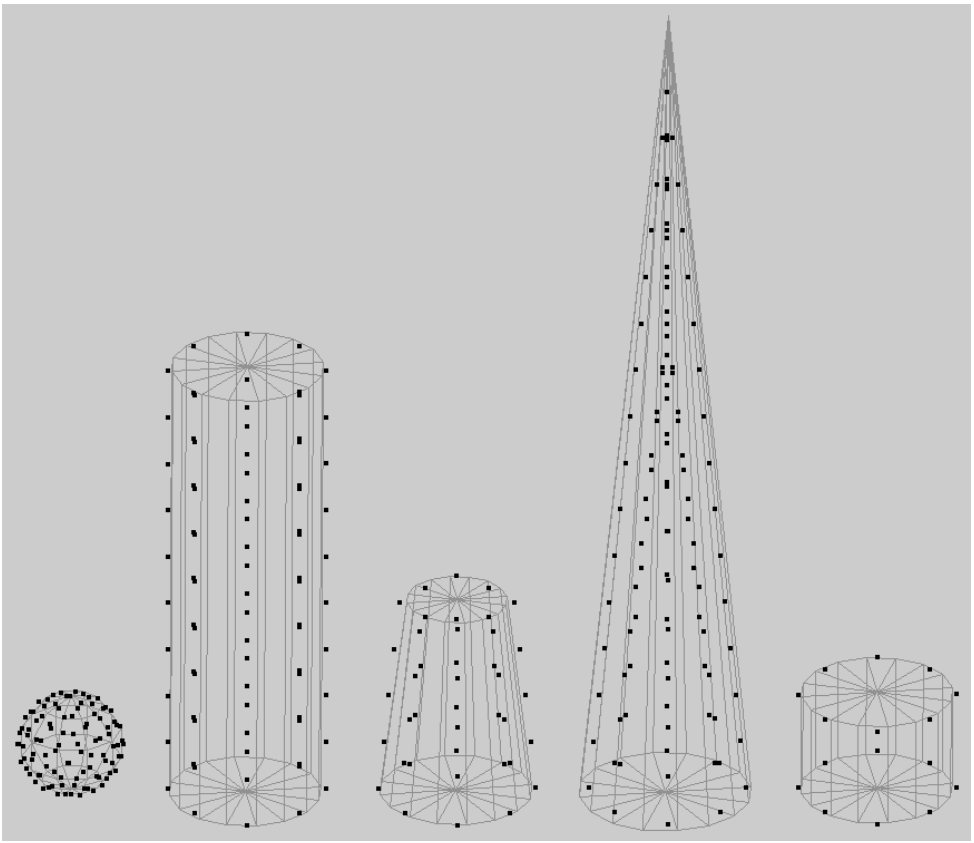

Figure 3: In this image, the result of an automatic fitting is shown. The sphere, the cylinders, and the frustum have been fitted with the average fitting algorithm. The cone was fitted with the maximum algorithm. The types of the 3D objects are guessed by the automatic function, based on the comparison of the minimum volume of all possible object types.

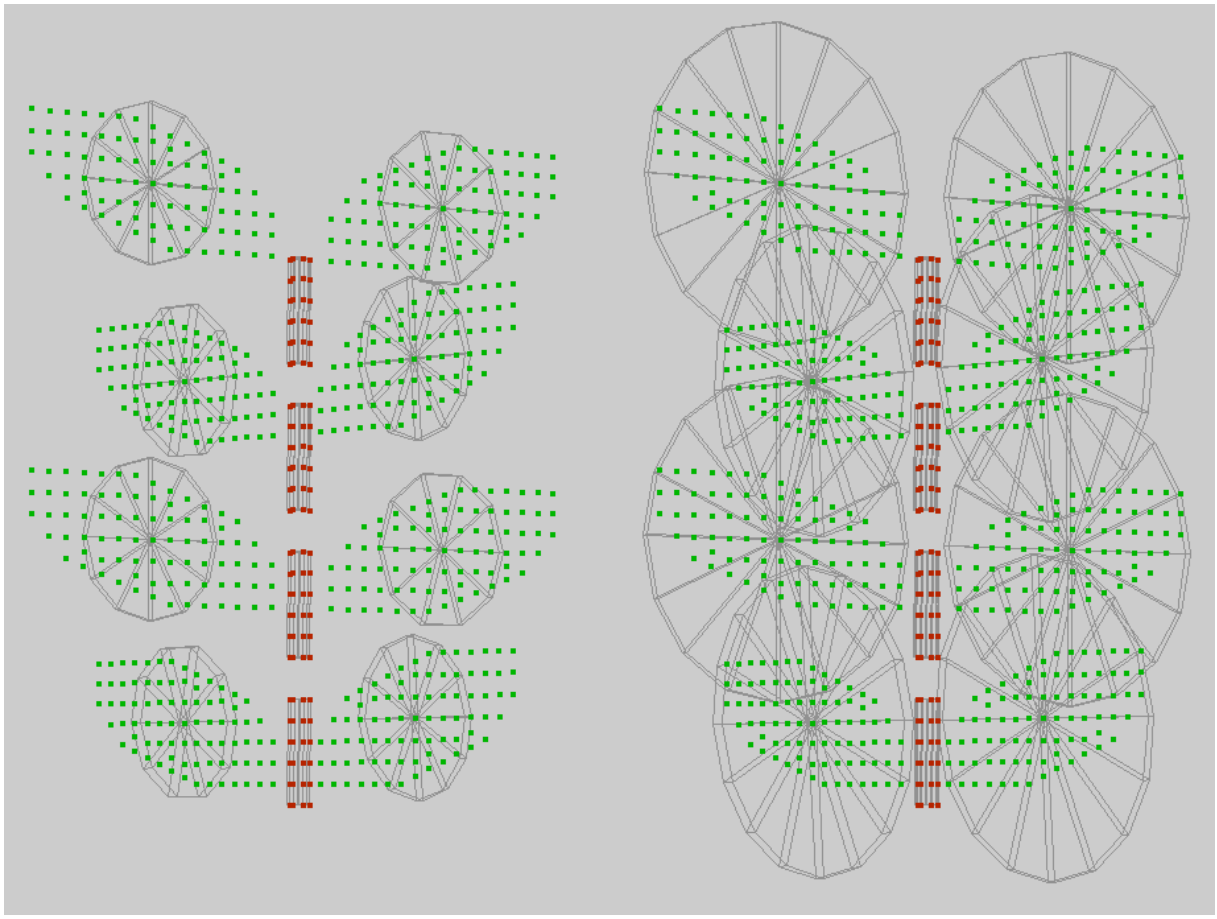

Figure 4: The clusters of the plant on the left side were cylinder-fitted with the average mode. Normally, the points in each leaf of the example plant are on a plane and the cylinder fitting algorithm would return a cylinder with a length of 0 centimeters. Because objects with a size of 0 are not displayed in GroIMP, the length is automatically set to one centimeter if it is 0. The clusters of the plant on the right side were cylinder-fitted with the maximum mode.

3.5 Calculating a score for objects

The score of fitted objects is always calculated in relation to the given point cloud. To get a score, the average distance between all points and the objects surface is calculated (see mathematical details in appendix). To prevent the algorithm from preferring objects where all points lie on one part of the objects surface and the object is much larger than the point cloud, the volume and the surface area of the generated object are added to the score. Because all objects are generated so that the points are always contained in the volume of the object, the smallest object is also the best fitting one.

4 Implementation

In the preceding practical report, the implementation chapter was the most important part since it contained lots of detailed explanations around all implemented algorithms. In contrast to that the mathematical and logical topics are outsourced to their own appendices in this report. Therefore, this chapter will only describe some design and implementation decisions that have been made during the development of the new point cloud fitting features.

A special peculiarity of the added `pointcloud` package is that some classes are redundant to the already existing java classes in the `objects` package. Normally, such a redundancy should be avoided. But for the implementation of the fitting algorithms, the objects (spheres, cylinders, frustums, and cones) needed some more fundamental properties that did not fit to the structure how the already existing objects are stored and used. To avoid a collision between the new fitting functions and the already existing functions, the most justifiable solution was to create new data structures that are fully independent from the existing ones, but can then be converted to them when the fitting algorithm terminates.

Another noticeable peculiarity of the `PointCloudTools` class is that it also contains the point cloud fitting functions, especially if there is also a `PointCloudFittingTools` class. This makes it easier to use the functions in XL with the same import statement.

All implemented functions work equally in XL and in the graphical menu. In both environments, all five fitting functions (sphere, cylinder, frustum, cone, and automatic fitting) are implemented so that they can be used with one or with multiple point clouds at the same time. The automatic mode has the special feature that the returned objects can have different types. This is useful in the graphical mode because lots of point clouds can be selected, for example after using the clustering function, and be processed directly. In XL, the automatic fitting function returns an array of graph node objects. This makes it possible to handle different objects at the same time and distinguish between the types later. The usage of the XL functions is explained in detail in the appendix of this document and in the example project „Advanced Point Cloud Fitting Tools“.

In the final implementation of the preceding point cloud tools project there was a strange behavior of point clouds. All point clouds had their 3D marker (the red-green-blue arrows to move the object around) on the origin position of the coordinate system. This has been fixed in the implementation of this project. The issue was that the points in a point cloud had their absolute position inside the point cloud, but the point cloud itself had no position (0/0/0). Now, there is an additional optimization function that translates the point cloud to the position of its own points and moves the internal positions of the points in the opposite direction. This leads to the effect that the 3D marker is no longer displayed anywhere in the 3D view, but directly next to the concerning point cloud.

The 3D rotation of point clouds is ignored in the current version of all newly implemented point cloud tools, but can be added later if it becomes necessary for future projects. The non-implementation has multiple reasons. Normally, point clouds are imported from files that are generated by laser scanners. Laser scanners do not produce rotated point clouds, because they do not know whether the scanned object is rotated or not. The rotation is then represented by deviating point positions in the returned point cloud. If point clouds are split, clustered or merged with the functions from the preceding project, also no rotated point clouds are generated. Another reason is that point clouds are normally not used in RGG trees or automatically generated plants.

All texts that can occur in the graphical user interface are translatable. In the package `de.grogra.imp3d.pointcloud`, there is a text file `Resources.properties` with a set of key-value pairs. Each of the keys is used in the code as placeholder for a finally displayed text. The value is the english text that should appear in that situation. By adding a file with a language identifier contained in the file name, a file with translations can be added for another language. The language depends on the users system language or the JVM parameter `-Duser.language=zh`. In this example, a file named `Resources_zh.properties` could be added and can then contain the chinese translations.

5 Conclusion

5.1 Difficulties and solutions

In order to succeed in this project, lots of difficulties had to be solved. The most important of them are listed here:

- GroIMP is a very large and old software and was not developed and managed by professional software engineers all the time. This means that GroIMP is very complicated to understand nowadays because lots of different people have worked on it with their own ideas and skills. This makes it very difficult to understand larger concepts and to add new features so that they work reliably. With a lot of experience and patience it becomes possible to solve this issue.
- The current navigation in the 3D view is complicated to use and some preferences have to be done each time after the software was restarted. This leads to a very time-consuming procedure of testing newly implemented features. Probably, this can be solved soon in an other current project that is focused on improving the 3D navigation.
- It could have been simple to use or implement an already existing algorithm to fit cylinders or other objects to point clouds in GroIMP. But the problem with most of the „mathematically perfect“ algorithms is that they are very difficult to implement in a programming language like Java because lots of mathematical concepts (like integration or derivation) are not supported by default and dependencies on third-party software should rather be avoided in order to be able to maintain the software more easily. An own approach with an approximative algorithm has lead to an applicable solution.
- The score calculation did not work as explained in the design chapter. To only compare the volumes is much easier and works for simple objects like spheres, cylinders, frustums, or cones.
- The 3D rotation and Euler matrices are difficult to use and to debug in general. In combination with the time-consuming restart procedure of GroIMP it becomes an imposition to implement complex features that rely on 3D rotation.

5.2 Available results

With the completion of this project, multiple results can be reused by interested people in the future:

- The java code that contains the whole point cloud feature implementation is located in the package `IMP-3D/de.grogra.imp3d.pointcloud`. Some entries have been added in the files `IMP-3D/plugin.xml`, `IMP-3D/plugin.properties` and in the list of example projects. The source code can be found in the official GroIMP repository, see references. Because the code is further developed, a point cloud tag will be added (unfortunately after the submission of this report) in the repository.
- The complete code is documented with JavaDoc. In a new version of the automatically generated java documentation, the whole package will be documented with descriptive comments.
- Of course, the features are usable directly in GroIMP. The XL functions are introduced in the belonging example project „Advanced Point Cloud Fitting Tools“. The graphical functions are designed to be intuitive with graphical information boxes, input dialogs and error output boxes.
- In addition to this practical report, also the previous one „Implementation of point cloud tools in GroIMP“ and the installation guide are recommended if one is interested in the development of further point cloud features in GroIMP.

5.3 Future possibilities

Of course, the completion of this practical course does not mean that the possibilities of processing point clouds in GroIMP are exhausted. To fit spheres, cylinders, frustums, and cones to point clouds is only one of many scopes of applications. Here is a list with further ideas that could be implemented in GroIMP in the future.

- The complexity of the Fibonacci sphere based fitting algorithm is really high, because the algorithm calculates as many potential cylinders as declared by the precision parameter. Until now, all potential cylinders are generated and checked, regardless of how good or bad they are. By checking the potential cylinders in an other order and looking for maximum scores around the Fibonacci sphere, it could be possible to reduce the runtime of the fitting algorithm a lot.
- For now, only the object translation is considered during the fitting, clustering, merging, and splitting of point clouds. In practice, this is enough because point clouds are nearly always generated by laser scanners or the clustering algorithm and these tools do not use rotated point clouds. But from mathematical point of view, the rotation would be necessary to always have results as expected.
- The score system is good, but not optimal. In the current implementation, the cylinders and other shapes are generated so that they surround the given point cloud. By using the objects volume as score, the one with the minimum volume can be used and the results are sufficient for most of the point clouds. If this result is not precise enough, a new score system should be implemented in the future as explained in the design chapter and as shown in the appendix. A better score system should consider the volume, the surface, and the matching of the generated object and the given point cloud.
- Goal of this project was to fit round objects (spheres, cylinders, frustums, and cones) to point clouds by searching for a principal axis for the generated object. To be able to fit other kinds of objects, for example boxes, other fitting strategies have to be used and could be an interesting project for an other practical course.
- Due to special requirements, the import and export functions for point clouds are located in the „Edit Point Clouds“ menu. For a better user experience, it would make sense to integrate these functions to the general „Import file“ and „Export file“ option. But one thing has to be considered: There are lots of conventions to store point clouds and there are no standardized file name endings. With the new import and export functions, point cloud files can be processed independently from their file name endings. This behavior was necessary during the development and could be very difficult or impossible to integrate into the general file import and export function of GroIMP.
- Currently, the „Edit Point Clouds“ menu is located in the „Objects“ menu. Depending on further discussions, it can make sense to move it into the „Edit“ menu, because point clouds are edited in the clustering, merging, and splitting function. On the other hand, new objects are created if the old ones are kept or objects are fitted to point clouds. However, there are two reasons why the menu is located in the „Objects“ menu. First reason: The „Objects“ menu is only loaded if a 3D project is opened. The point cloud tools should also only be available when a 3D project is opened. Second reason: The „Edit“ menu is located in the IMP module and has no access to the IMP-3D module. The point cloud features have to be located in the IMP-3D module to be able to operate with 3D objects.
- To have more possibilities to handle large sets of point clouds and other objects, a better interactive selection system for the 3D view is required. At the time of completion of this practical report, an other project has already been started to partially solve this issue.

6 References

The screenshots, the sketches (cylinder, frustum, and cone in the formulary), and the test data sets have been created with GroIMP, GeoGebra, and GIMP by Maurice Müller.

First version of the installation guide for GroIMP in Eclipse:
Title: GroIMP Eclipse Install Notes
Author: Michael Henke

Second version of the installation guide for GroIMP in Eclipse:
Title: Installing and Running GroIMP in Eclipse
Author: Maurice Müller

Current version of the installation guide for GroIMP in Eclipse:
Author: Maurice Müller
Url: `https://gitlab.com/grogra/groimp/-/wikis/Developer%20Guide`

Preceding practical report about point cloud clustering:
Title: Implementation of point cloud tools in GroIMP
Author: Maurice Müller
Published: Georg-August-Universität Göttingen, 2021

Original repository of GroIMP:
SourceForge: `https://sourceforge.net/projects/groimp/files/groimp/1.5/`
Gitlab: `https://gitlab.com/grogra/groimp/-/tree/master`
Java documentation: `https://wwwuser.gwdg.de/~groimp/api/overview-summary.html`

Doctoral thesis as documentation of GroIMP:
Title: Design and Implementation of a Graph Grammar Based Language for Functional–Structural Plant Modelling
Author: Ole Kniemeyer
Published: Brandenburgische Technische Universität Cottbus, 2008
Url: `https://opus4.kobv.de/opus4-btu/frontdoor/deliver/index/docId/462/file/thesis.pdf`

Fibonacci sphere algorithm:
Title: A Comparison of Popular Point Configurations on $\mathbb{S}^2$
Authors: D.P. Hardin, T. Michaels, E.B. Saff
Published: Center for Constructive Approximation, Vanderbilt University, Nashville, USA, 2016
Url: `https://arxiv.org/pdf/1607.04590.pdf`

Calculation of enveloping ellipsoids for different 3D objects:
Title: Implementierung von Hüllkörpern in der Plattform GroIMP für Pflanzenmodellierung
Author: Jan Tristan Koch
Published: Georg-August-Universität Göttingen, 2021

Clustering software ORIPUCA and example point clouds:
Title: Object recognition in point clouds using a density–based clustering algorithm
Author: Jörg Krause
Published: Brandenburgische Technische Universität Cottbus, 2007

Density-Based Clustering Algorithm:
Title: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise
Authors: Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu
Published: Ludwig-Maximilians-Universität München, 1996
Url: `https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf`

Appendix A: Available java functions in XL

This appendix contains the list of available java functions. They can be used in the java code (by future developers) and in XL code (by GroIMP users). All functions can be used when the class `de.grogra.imp3d.pointcloud.PointCloudTools` is imported with the following java/XL command:

```
import de.grogra.imp3d.pointcloud.PointCloudTools;
```

All functions are **static** functions in the `PointCloudTools` class and **not** available as object based functions like `pointcloud.function()`.

The given point clouds and returned 3D objects are always of the following types and fully compatible to the other objects in the 3D view:

- `de.grogra.imp3d.objects.PointCloud`
- `de.grogra.imp3d.objects.Sphere`
- `de.grogra.imp3d.objects.Cylinder`
- `de.grogra.imp3d.objects.Frustum`
- `de.grogra.imp3d.objects.Cone`
- `de.grogra.graph.impl.Node`

All objects that are created and returned by java functions and called in XL scripts, are **not** automatically added to the RGG graph. They only exist as objects in the code context and can be added to the graph manually if required.

The java functions that can be called in XL scripts exist twice. Each function can be used to fit one point cloud to one object or to fit an array of point clouds to an array of objects. All of them are listed below:

Parameters and return values are not described in detail here. More information about how to choose the right parameter values is given in the design chapter.

In **headless mode**, all functions introduced in this appendix can be used. All parameters and return values can be used as specified in the belonging description.

Fitting spheres to point clouds

With the following two functions, spheres can be fitted to point clouds. If the first one is used, one point cloud is required and one sphere is returned. By using the second function, an array of point clouds must be provided and an array of spheres is returned.

Declaration in PointCloudTools.java:

```
1   // Fitting one sphere to one point cloud
2   public static Sphere fitSphereToPointCloud(PointCloud pointCloud, boolean average) {
3       ...
4       return sphere;
5   }
6   // Fitting multiple spheres to multiple point clouds
7   public static Sphere[] fitSpheresToPointClouds(PointCloud[] pointClouds, boolean average) {
8       ...
9       return spheres;
10  }
```

XL code example:

```
boolean average = true;
// Fitting one sphere to one point cloud
PointCloud pointCloud = ...;
Sphere sphere = PointCloudTools.fitSphereToPointCloud(pointCloud, average);
// Fitting multiple spheres to multiple point clouds
PointCloud[] pointClouds = ...;
Sphere[] spheres = PointCloudTools.fitSpheresToPointClouds(pointClouds, average);
```

Fitting cylinders to point clouds

With the following two functions, cylinders can be fitted to point clouds. If the first one is used, one point cloud is required and one cylinder is returned. By using the second function, an array of point clouds must be provided and an array of cylinders is returned.

Declaration in PointCloudTools.java:

```
// Fitting one cylinder to one point cloud
public static Cylinder fitCylinderToPointCloud(PointCloud pointCloud, boolean average, int precision) {
    ...
    return cylinder;
}
// Fitting multiple cylinders to multiple point clouds
public static Cylinder[] fitCylindersToPointClouds(PointCloud[] pointClouds, boolean average, int
     precision) {
    ...
    return cylinders;
}
```

XL code example:

```
boolean average = true;
int precision = 1000;
// Fitting one cylinder to one point cloud
PointCloud pointCloud = ...;
Cylinder cylinder = PointCloudTools.fitCylinderToPointCloud(pointCloud, average, precision);
// Fitting multiple cylinders to multiple point clouds
PointCloud[] pointClouds = ...;
Cylinder[] cylinders = PointCloudTools.fitCylindersToPointClouds(pointClouds, average, precision);
```

Fitting frustums to point clouds

With the following two functions, frustums can be fitted to point clouds. If the first one is used, one point cloud is required and one frustum is returned. By using the second function, an array of point clouds must be provided and an array of frustums is returned.

Declaration in PointCloudTools.java:

```
// Fitting one frustum to one point cloud
public static Frustum fitFrustumToPointCloud(PointCloud pointCloud, boolean average, int precision) {
    ...
    return frustum;
}
// Fitting multiple frustums to multiple point clouds
public static Frustum[] fitFrustumsToPointClouds(PointCloud[] pointClouds, boolean average, int
     precision) {
    ...
    return frustums;
}
```

XL code example:

```
boolean average = true;
int precision = 1000;
// Fitting one frustum to one point cloud
PointCloud pointCloud = ...;
Frustum frustum = PointCloudTools.fitFrustumToPointCloud(pointCloud, average, precision);
// Fitting multiple frustums to multiple point clouds
PointCloud[] pointClouds = ...;
Frustum[] frustums = PointCloudTools.fitFrustumsToPointClouds(pointClouds, average, precision);
```

Fitting cones to point clouds

With the following two functions, cones can be fitted to point clouds. If the first one is used, one point cloud is required and one cone is returned. By using the second function, an array of point clouds must be provided and an array of cones is returned.

Declaration in PointCloudTools.java:

```
// Fitting one cone to one point cloud
public static Cone fitConeToPointCloud(PointCloud pointCloud, boolean average, int precision) {
    ...
    return cone;
}
// Fitting multiple cones to multiple point clouds
public static Cone[] fitConesToPointClouds(PointCloud[] pointClouds, boolean average, int precision) {
    ...
    return cones;
}
```

XL code example:

```
boolean average = true;
int precision = 1000;
// Fitting one cone to one point cloud
PointCloud pointCloud = ...;
Cone cone = PointCloudTools.fitConeToPointCloud(pointCloud, average, precision);
// Fitting multiple cones to multiple point clouds
PointCloud[] pointClouds = ...;
Cone[] cones = PointCloudTools.fitConesToPointClouds(pointClouds, average, precision);
```

Fitting automatically detected objects to point clouds

With the following two functions, automatically selected objects can be fitted to point clouds. If the first one is used, one point cloud is required and one node object is returned. By using the second function, an array of point clouds must be provided and an array of node objects is returned.

Note: These functions return `Node` objects. `Node` is the super class of `Sphere`, `Cylinder`, `Frustum`, and `Cone` (and all other 3D objects in XL, but that is not important here). This means that a `Node` object is like a „category“ type for the other types and the returned node is of one of the other types implicitly. The algorithm decides automatically which of these fits best to the given point cloud. If multiple point clouds are given to the automatic function, different types of objects can be contained in the resulting array of `Node` objects.

Declaration in PointCloudTools.java:

```
// Fitting one object to one point cloud
public static Node fitAutomaticObjectToPointCloud(PointCloud pointCloud, boolean average, int precision
    ) {
    ...
    return node;
}
// Fitting multiple objects to multiple point clouds
public static Node[] fitAutomaticObjectsToPointClouds(PointCloud[] pointClouds, boolean average, int
    precision) {
    ...
    return nodes;
}
```

XL code example:

```
boolean average = true;
int precision = 1000;
// Fitting one object to one point cloud
PointCloud pointCloud = ...;
Node node = PointCloudTools.fitAutomaticObjectToPointCloud(pointCloud, average, precision);
// Fitting multiple objects to multiple point clouds
PointCloud[] pointClouds = ...;
Node[] nodes = PointCloudTools.fitAutomaticObjectsToPointClouds(pointClouds, average, precision);
```

The following code example shows how the type can be detected in XL.

XL code example:

```
PointCloud pointCloud = ...;
boolean average = true;
int precision = 1000;
Node node = PointCloudTools.fitAutomaticObjectToPointCloud(pointCloud, average, precision);
if (node instanceof Sphere) {
    Sphere sphere = (Sphere)(node);
    // Do anything with 'sphere'
} else if (node instanceof Cylinder) {
    Cylinder cylinder = (Cylinder)(node);
    // Do anything with 'cylinder'
} else if (node instanceof Frustum) {
    Frustum frustum = (Frustum)(node);
    // Do anything with 'frustum'
} else if (node instanceof Cone) {
    Cone cone = (Cone)(node);
    // Do anything with 'cone'
} else {
    // This case can not happen. 'node' is never null and never of an unknown type.
}
```

This also works with multiple point clouds.

XL code example:

```
PointCloud[] pointClouds = ...;
boolean average = true;
int precision = 1000;
Node[] nodes = PointCloudTools.fitAutomaticObjectsToPointClouds(pointCloud, average, precision);
int index = 0;
while (index < nodes.length) {
    if (nodes[index] instanceof Sphere) {
        // Do the same distinction as in the example above, but with 'nodes[index]'
    }
    // ...
    index++;
}
```

Appendix B: Point cloud tools in the objects menu

All features in this appendix are available when a 3D project or an RGG project is opened in GroIMP. The functions work independently from XL scripts and are only based on the objects that are currently available in the 3D view.

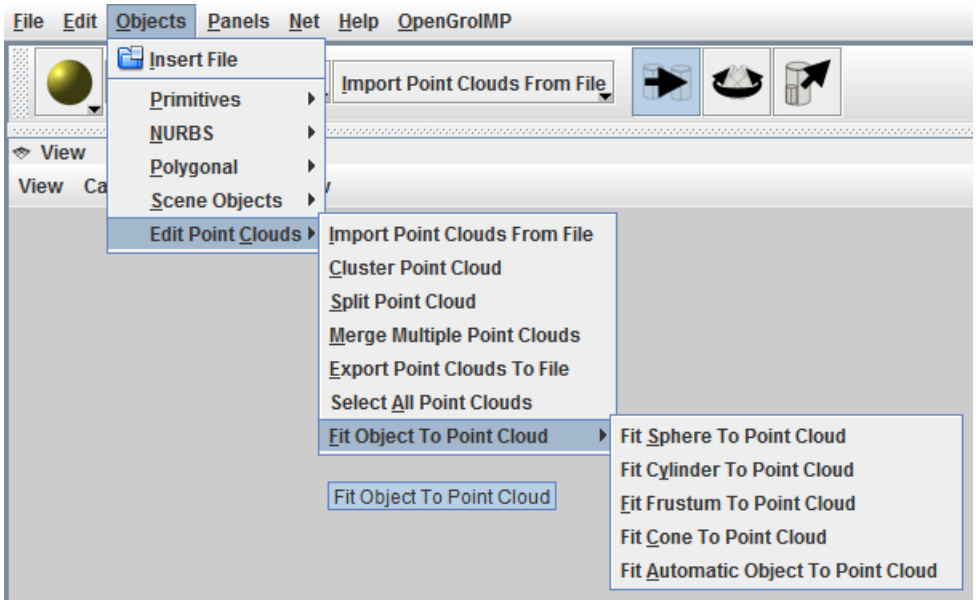

Figure 5: The point cloud fitting menu is located inside the objects menu and inside the point cloud edit menu. The floating blue box „Fit Object To Point Cloud“ is the tooltip text. Unfortunately, the mouse pointer was not captured by the screenshot software.

Warning: In the current version of GroIMP, the shortcut Alt+O does not work. This is due to the fact that 'O' is used twice and the wrong menu entry is focused by default. If the objects menu is already open, the other shortcuts (→ C → [custom key]) work as described.

Overview

Feature	Shortcut	Selection before	Selection after
Fit sphere to point cloud	Alt+O → C → F → S	Multiple point clouds	Multiple spheres
Fit cylinder to point cloud	Alt+O → C → F → Y	Multiple point clouds	Multiple cylinders
Fit frustum to point cloud	Alt+O → C → F → F	Multiple point clouds	Multiple frustums
Fit cone to point cloud	Alt+O → C → F → C	Multiple point clouds	Multiple cones
Fit any object to point cloud	Alt+O → C → F → A	Multiple point clouds	Multiple objects

Detailed description

All functions described here require one or multiple point clouds selected in the current 3D view. Otherwise, an error dialog is shown. If point clouds are selected, a dialog box asks for the fitting mode. In case of a cylinder, a frustum, a cone, and the automatic mode also the precision parameter is requested. Another dialog box asks whether the selected point clouds should be kept or removed. After confirming one of the options, the objects are created, displayed in the 3D view and get selected.

The differences are as follows:

- The **fit sphere to point cloud** function does not request for the precision parameter. The precision parameter is not used since the sphere fitting does not use the Fibonacci sphere fitting algorithm. After the fitting was executed, spheres are added to the 3D view.
- The **fit cylinder to point cloud** function adds cylinders to the 3D view.
- The **fit frustum to point cloud** function adds frustums to the 3D view.
- The **fit cone to point cloud** function adds cones to the 3D view.
- The **fit automatic object to point cloud** function adds automatically selected objects to the 3D view. If multiple point clouds are fitted, different objects can be created.

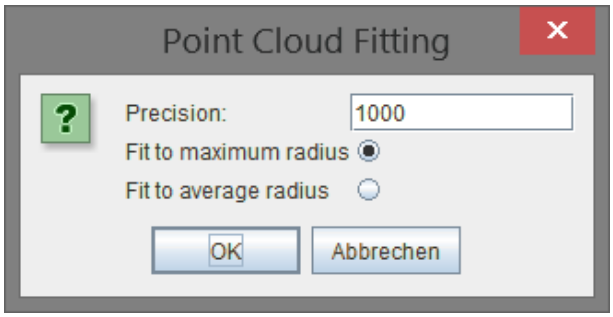

Figure 6: In this dialog box, the fitting parameters can be set. For cylinders, frustums, and cones, the precision and the fitting mode can be selected. The precision must be a positive integer value. It is used as the number of points in the internally used Fibonacci sphere. The points of a Fibonacci sphere are then used as possible direction vectors for the fitted object. Spheres do not need a direction vector and do not request the precision parameter. The fitting mode can be selected to either „average“, or „maximum“. It is available for all types of fittable objects and is used to set its radius. In case of a sphere, the radius is determined by the average or maximum distance of all points in the point cloud to the center position of the fitted sphere. In case of a cylinder, frustum, or cone, the radius is determined by the average or maximum distance of all points in the point cloud to the principal axis of the respective object.

Note: All texts in the user interface are provided in English by default. They can be translated with language depending resource files. Some of the dialog boxes of Java are translated automatically and can differ from the GroIMP language.

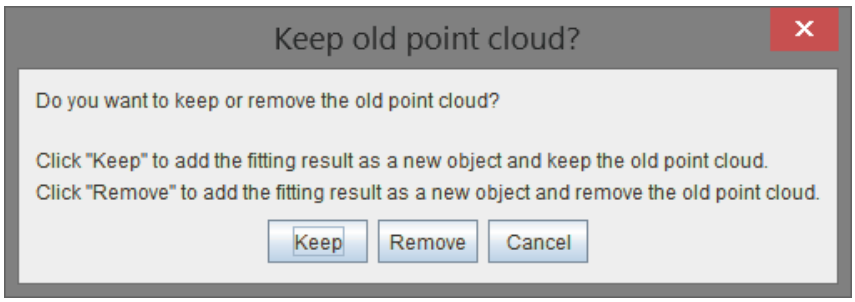

Figure 7: This dialog box is always shown after the parameters have been entered successfully. Here, the user is asked whether the point cloud should be removed during the fitting process. This can be useful in some cases, but it can also be disturbing in some other cases. If the fitting function was selected accidentally, it can also be canceled here. With the function in the graphical user interface, it is possible to fit one or more point clouds at the same time. If only one point cloud is selected (and fitted later), the texts are written in singular in this dialog box. Otherwise, plural is used.

Appendix C: Formulary

This appendix contains some important formula that has been used for the sphere, cylinder, frustum, and cone fitting. All points, vectors, and other objects are assumed to be used in a 3D context and always have an x, a y, and a z component.

Conversion from 3D vector to x, y, and z angles

The components (R_x, R_y, and R_z) of a 3D rotation R can be calculated from a 3D vector V as following. r is the pythagorean length of V.

$$\begin{aligned} r &= \sqrt{(V_x)^2 + (V_y)^2 + (V_z)^2} \\ R_x &= 0 \\ R_y &= acos\left(\frac{V_z}{r}\right) \\ R_z &= atan2\left(\frac{V_y}{V_x}\right) \end{aligned}$$

Note: $atan()$ can not be used to calculate R_z, because $atan()$ only provides results in the upper half of the sphere. With $atan2()$, the result works with the whole sphere.

Distance between point and other point, line, and plane

The distance D_{PQ} between the two points P and Q is:

$$D_{PQ} = \sqrt{(Q_x - P_x)^2 + (Q_y - P_y)^2 + (Q_z - P_z)^2}$$

The distance D_{PL} between a point P and a line $L = L_T + L_D \cdot x$ with L_T as translation vector and L_D as direction vector can be calculated as follows:

$$D_{PL} = \frac{|(P - L_T) \times L_D|}{|L_D|}$$

The distance D_{PE} between a point P and a plane E with E_T as translation vector and E_N as normal vector can be calculated as follows:

$$D_{PE} = \frac{|(P - E_T) \cdot E_N|}{|E_N|}$$

Distance between point and cylinder, frustum, cone, and sphere surface

Distance between point and cylinder surface

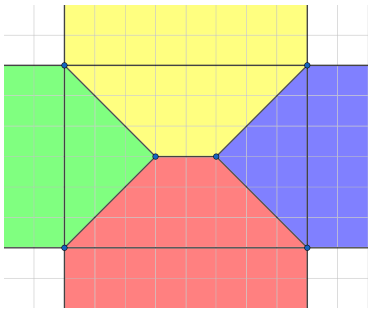

Figure 8: This image shows a cylinder from the side. The horizontal line in the yellow area is the top surface, the horizontal line in the red area is the bottom surface, and the vertical lines in the green and blue areas are the visible edges of the lateral surface.

The surface of a cylinder consists of a base surface, a top surface, and a lateral surface. Depending on where the point is located in the environment of the cylinder, one of the areas shown in the image has to be selected and the distance to that surface has to be calculated.

- The distance of points in the red area is calculated with the distance to the base surface.
- The distance of points in the white area below the cylinder is calculated with the distance to the base surface (a circle with the radius of the cylinder).
- The distance of points in the green or blue area is calculated with the distance to the cylinder axis minus the radius. This is possible because the lateral surface is parallel to the direction vector.
- The distance of points in the yellow area is calculated with the distance to the top surface. The top surface is similar to the base plane, but with the direction vector added to the position vector.
- The distance of points in the white area above the cylinder is calculated with the distance to the top surface (a circle with the radius of the cylinder).

Distance between point and frustum surface

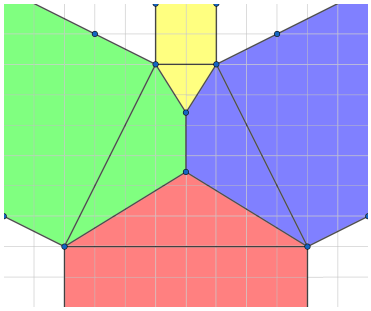

Figure 9: This image shows a frustum from the side. The horizontal line in the yellow area is the top surface, the horizontal line in the red area is the bottom surface, and the diagonal lines in the green and blue areas are the visible edges of the lateral surface.

The surface of a frustum consists of a base surface, a top surface, and a lateral surface. Depending on where the point is located in the environment of the frustum, one of the areas shown in the image has to be selected and the distance to that surface has to be calculated.

- The distance of points in the red area is calculated with the distance to the base surface.
- The distance of points in the white area below the frustum is calculated with the distance to the base surface (a circle with the base radius of the frustum).
- The distance of points in the green or blue area is calculated with the distance to the lateral surface. The distance has to be calculated with the normal vector to that surface. The normal vector has its base point on the lateral surface and directs to the point of interest.
- The distance of points in the yellow area is calculated with the distance to the top surface. The top surface is similar to the base plane, but with the direction vector added to the position vector.
- The distance of points in the white area above the frustum is calculated with the distance to the top surface (a circle with the top radius of the frustum).

Distance between point and cone surface

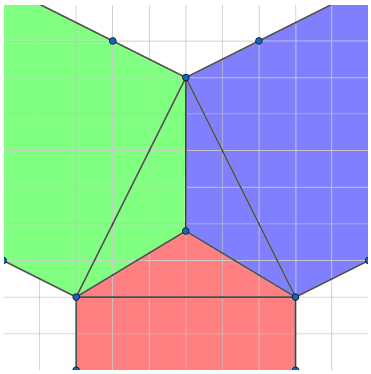

Figure 10: This image shows a cone from the side. The horizontal line in the red area is the bottom surface and the diagonal lines in the green and blue areas are the visible edges of the lateral surface.

The surface of a cone consists of a base surface and a lateral surface. Depending on where the point is located in the environment of the cone, one of the areas shown in the image has to be selected and the distance to that surface has to be calculated.

- The distance of points in the red area is calculated with the distance to the base surface.
- The distance of points in the white area below the cone is calculated with the distance to the base surface (a circle with the base radius of the cone).
- The distance of points in the green or blue area is calculated with the distance to the lateral surface. The distance has to be calculated with the normal vector to that surface. The normal vector has its base point on the lateral surface and directs to the point of interest.
- The distance of points in the white area above the cone is calculated with the pythagorean distance between the point and the tip of the cone.

Distance between point and sphere surface

The distance D_{PS} between a point P and the surface of a sphere S is calculated with the difference between the radius r of the sphere and the pythagorean distance between the point and the center c of the sphere:

$$D_{PS} = \sqrt{(c_x - P_x)^2 + (c_y - P_y)^2 + (c_z - P_z)^2} - r$$

The distance D_{PS} is > 0 if the point is outside the sphere volume and < 0 if the point is inside the sphere volume.

Volume and surface of sphere, cylinder, frustum, and cone

In the following table, the volumes and surface areas of a sphere, a cylinder, a frustum and a cone are listed. r is the radius and l is the length of the concerning object. In case of a frustum, R is the base radius and r is the top radius with $r < R$.

Object	Volume	Surface
Sphere	$V = \frac{4}{3} \cdot \pi \cdot r^2$	$S = 4 \cdot \pi \cdot r^2$
Cylinder	$V = \pi \cdot r^2 \cdot l$	$S = 2 \cdot \pi \cdot r \cdot (r + l)$
Frustum	$V = \frac{1}{3} \cdot \pi \cdot l \cdot \left(R^2 + R \cdot r + r^2\right)$	$S = \pi \cdot R^2 + \pi \cdot (R + r) \cdot \sqrt{l^2 + (R - r)^2} + \pi \cdot r^2$
Cone	$V = \frac{1}{3} \cdot \pi \cdot r^2 \cdot l$	$S = \pi \cdot r \cdot \left(r + \sqrt{r^2 + l^2}\right)$

Appendix D: Algorithms

This appendix contains the most important algorithms that have been used for the object fitting features. Here, the algorithms are written in pseudocode. The java implementation of each algorithm can be found in the belonging java code (project IMP-3D, class `de.grogra.imp3d.pointcloud.PointCloudFittingTools.java`).

In the pseudocode examples, some further functions are used:

The function `GET_CENTER_POINT` requires a list of 3D points and returns a new point with the average position of all given points.

The function `GET_POINT_POINT_DISTANCE` requires two 3D points and returns the pythagorean distance between these two points.

The function `GET_LINE_POINT_DISTANCE` requires a position vector of a 3D line, a direction vector of a 3D line, and a 3D point. It returns the minimum distance between the point and the line.

The function `GET_PLANE_POINT_DISTANCE` requires a position vector of a 3D plane, a normal vector of a 3D plane, and a 3D point. It returns the minimum distance between the point and the plane.

The function `GET_SURFACE_POINT_DISTANCE` requires a 3D object (a sphere, a cylinder, a frustum or a cone), and a 3D point. It returns the minimum distance between the point and the objects surface.

The function `SET_VECTOR_LENGTH` requires a vector and a numeric value. It multiplies all components of the vector, so that the pythagorean length of the vector is as long as the given numeric value.

The function `LENGTH` requires a vector and returns the pythagorean length of the vector.

The function `CREATE_POINT` requires three values (x, y, and z value) and returns a point object with the given coordinates.

The function `CREATE_SPHERE` requires a position vector and a radius. It returns a sphere object.

The function `CREATE_CYLINDER` requires a position vector, a direction vector, a radius, and a length. It returns a cylinder object.

The function `CREATE_FRUSTUM` requires a position vector, a direction vector, a base radius, a top radius, and a length. It returns a frustum object.

The function `CREATE_CONE` requires a position vector, a direction vector, a base radius, and a length. It returns a cone object.

The function `SIZE` requires an array and returns its number of elements.

The function `SUM` requires an array and returns the sum of all values in that array.

The function `VOLUME` requires a sphere, a cylinder, a frustum, or a cone and returns its volume.

The function `AREA` requires a sphere, a cylinder, a frustum, or a cone and returns its surface area.

The functions `SQRT`, `ABS`, `SIN` and `COS` require a numeric value and return the square root, the absolute value, the sinus value, and the cosinus value of the parameter.

Generating a Fibonacci sphere

A Fibonacci sphere is a list of points, distributed around a sphere surface as evenly as possible. This function is used to create such a sphere. The center position, the radius and the number of points must be provided as parameter. With more points, the sphere gets more precise and the result of the object fitting gets more accurate. On the other hand, a higher precision requires a longer calculation time.

```
01 FUNCTION CREATE_FIBONACCI_SPHERE(center, radius, number)
02     points := [number]
03     phi := PI * (3 - SQRT(5))
04     index := 0
05     WHILE (index < number)
06         y := radius - (index / (number - 1)) * 2 * radius
07         temporaryRadius := SQRT(radius^2 - y^2)
08         theta := phi * index
09         x := COS(theta) * temporaryRadius
10         z := SIN(theta) * temporaryRadius
11         points[index] := CREATE_POINT(center_x + x, center_y + y, center_z + z)
12         index := index + 1
13     END
14     RETURN points
15 END
```

Calculating a score to compare objects

The score function is one of the most important parts of the fitting algorithm. The object fitting works with creating lots of potential objects and comparing them with the original point cloud. Due to this, the quality of the result of the fitting algorithm is mostly based on this score function. To calculate a score for an object and a point cloud, the average distance of all points to the objects surface is calculated first. The distance is always handled as absolute value. Points with a large distance to the object lead to a worse score and points in the center of the object do also lead to a worse score. In practice, large point clouds can result in a nearly perfect score, if all points are near to the theoretical object surface, but most parts of the surface are not used. This phenomena can happen if a small point cloud has only points on a surface of a much larger object. To prevent this wrong result from being good, the volume and the surface area of the object are added to the score. This has the effect that objects with a small volume and a small surface area are preferred in contrast to huge ones with only some point cloud points in the middle of the surface.

```
01 FUNCTION GET_SCORE(points[], object)
02     score := 0
03     FOR (point :  points)
04         distance := GET_SURFACE_POINT_DISTANCE(object, point)
05         score := score + ABS(distance)
06     END
07     RETURN score / SIZE(points) + VOLUME(object) + AREA(object)
08 END
```

Fitting a cylinder with principal component analysis

The principal component analysis was the first attempt (in this practical course) to fit a cylinder to a point cloud. It expects a point cloud (or a list of points) and calculates the center position of the point cloud as well as the distance between the two most distant points.

The longest distance is used as direction vector (`firstDirection`) and principal component of the analyzed point cloud. With this direction vector, the second direction (`secondDirection`) is calculated so that it is orthogonal to the first direction vector and targets the point with the most far distance to the first direction vector. With the cross product of the first direction vector and the second direction vector, the third direction vector (`thirdDirection`) is calculated.

For all three direction vectors, a cylinder is calculated so that it directs into the respective direction. For each of the cylinders, a score is calculated. The cylinder with the least score is returned.

```
01 FUNCTION FIT_CYLINDER_WITH_PRINCIPAL_COMPONENT_ANALYSIS(points[])
02     center := GET_CENTER_POINT(points)
03     mostFarPoint := NULL
04     mostFarDistance := 0
05     FOR (point :  points)
06         distance := GET_POINT_POINT_DISTANCE(point, center)
07         IF (distance > mostFarDistance)
08             mostFarPoint := point
09             mostFarDistance := distance
10         END
11     END
12     firstDirection := mostFarPoint - center
13     orthogonalMostFarPoint := NULL
14     orthogonalMostFarDistance := 0
15     FOR (point :  points)
16         distance := GET_LINE_POINT_DISTANCE(center, firstDirection, point)
17         IF (distance > orthogonalMostFarDistance)
18             orthogonalMostFarPoint := point
19             orthogonalMostFarDistance := distance
20         END
21     END
22     diagonalVector := orthogonalMostFarPoint - center
23     diagonalDistance := LENGTH(diagonalVector)
24     distanceOnLine := SQRT(diagonalDistance^2 - orthogonalMostFarDistance^2)
25     negative := firstDirection * diagonalVector
26     SET_VECTOR_LENGTH(newVector, (negative < 0 ?  -1 :  1) * distanceOnLine)
27     footPoint := center + newVector
28     secondDirection := orthogonalMostFarPoint - footPoint
29     thirdDirection := firstDirection × secondDirection
30     thirdMostFarDistance := 0
31     FOR (point :  points)
32         distance := GET_PLANE_POINT_DISTANCE(center, thirdDirection, point)
33         IF (distance > thirdMostFarDistance)
34             thirdMostFarDistance := distance
35         END
36     END
37     SET_VECTOR_LENGTH(thirdDirection, thirdMostFarDistance)
    ...
```

```
    ...
38      xLength := LENGTH(firstDirection)
39      yLength := LENGTH(secondDirection)
40      zLength := LENGTH(thirdDirection)
41      xPosition := center - firstDirection
42      yPosition := center - secondDirection
43      zPosition := center - thirdDirection
44      xAverage := (yLength + zLength) / 2
45      yAverage := (xLength + zLength) / 2
46      zAverage := (xLength + yLength) / 2
47      xCylinder := CREATE_CYLINDER(xPosition, firstDirection, xAverage, 2 * xLength)
48      yCylinder := CREATE_CYLINDER(yPosition, secondDirection, yAverage, 2 * yLength)
49      zCylinder := CREATE_CYLINDER(zPosition, thirdDirection, zAverage, 2 * zLength)
50      xScore := GET_SCORE(points, xCylinder)
51      yScore := GET_SCORE(points, yCylinder)
52      zScore := GET_SCORE(points, zCylinder)
53      IF (xScore < yScore AND xScore < zScore)
54          RETURN xCylinder
55      ELSE IF (yScore < zScore)
56          RETURN yCylinder
57      ELSE
58          RETURN zCylinder
59      END
60  END
```

Fitting a cylinder with Fibonacci sphere analysis

The cylinder fitting algorithm with Fibonacci sphere analysis is the final solution for the object fitting feature in GroIMP. It is split into two algorithms. The main algorithm creates a Fibonacci sphere around the point cloud. The Fibonacci sphere has a specific number of points (= precision). The algorithm creates a cylinder for each point of the Fibonacci sphere and calculates the score for each cylinder. Finally, the scores are compared and the cylinder with the best score is returned.

The cylinder fitting function is later used for the frustum fitting and the cone fitting. This function provides the most basic information about the point cloud. It returns the position vector and the direction vector. Later, for the frustums, two different radii can be calculated, based on the cylinder returned by this function. The cone fitting is based on frustum fitting, but with a custom base radius and an adapted length. By reusing this algorithm for the other kinds of objects, lots of redundant implementation can be avoided.

```
01 FUNCTION FIT_CYLINDER_WITH_FIBONACCI_SPHERE_ANALYSIS(points[], precision)
02     center := GET_CENTER_POINT(points)
03     mostFarPoint := NULL
04     mostFarDistance := 0
05     FOR (point :  points)
06         distance := GET_POINT_POINT_DISTANCE(point, center)
07         IF (distance > mostFarDistance)
08             mostFarPoint := point
09             mostFarDistance := distance
10         END
11     END
12     sphere := CREATE_FIBONACCI_SPHERE(center, mostFarDistance, precision
13     score := INFINITY
14     bestCylinder := NULL
15     FOR (point :  sphere)
16         direction := point - center
17         cylinder := FIT_CYLINDER_TO_POINT_WITH_GIVEN_DIRECTION(points, direction)
18         cylinderScore := GET_SCORE(cylinder)
19         IF (cylinderScore < score)
20             score := cylinderScore
21             bestCylinder := cylinder
22         END
23     END
24     RETURN bestCylinder
25 END
```

The main function of the algorithm to fit a cylinder to a point cloud, using the Fibonacci sphere, is also using this function to create a cylinder for a given point cloud and a given direction vector. First, this function requests the center point of the point cloud and uses it for the calculation of the position vector of the returned cylinder. After that, the radius and the length for the cylinder can be calculated with the given direction vector. The returned cylinder is not the finally returned cylinder of the fitting algorithm. It is only used for the test case that tests how good the cylinder is.

```
01 FUNCTION FIT_CYLINDER_TO_POINT_WITH_GIVEN_DIRECTION(points[], direction)
02     center := GET_CENTER_POINT(points)
03     length := 0
04     radius := 0
05     FOR (point :  points)
06         distanceToPlane := GET_LINE_POINT_DISTANCE(center, direction, point)
07         distanceToNormal := GET_PLANE_POINT_DISTANCE(center, direction, point)
08         IF (distanceToPlane < length)
09             length := distanceToPlane
10         END
11         IF (distanceToNormal < radius)
12             radius := distanceToNormal
13         END
14     END
15     lengthVector := direction
16     SET_VECTOR_LENGTH(lengthVector, length)
17     position := center - lengthVector
18     RETURN CREATE_CYLINDER(position, direction, radius, 2 * length)
19 END
```

Calculating a linear regression for frustums and cones

The linear regression function is used to calculate the angle of the side surface of a frustum, using a given cylinder. The idea of this algorithm is to use the positions of the points in the point cloud as data set and the already fitted cylinder as coordinate system. The direction vector of the cylinder represents the x-axis, while the base surface (in all directions around the direction vector) represents the y-axis. This has the advantage that the distance of each point in the point cloud to the direction vector of the cylinder can be calculated and used as y-value. It is then independent from the angle around the direction vector of the cylinder. The distance to the base surface is then used as x-value. This function returns an array with two values. The first value is the slope of the linear regression, the second one is the intercept.

```
01 FUNCTION LINEAR_REGRESSION(cylinder, points[])
02     position := cylinder.position
03     direction := cylinder.direction
04     positions := [SIZE(points)]
05     values := [SIZE(points)]
06     index := 0
07     FOR (point :  points)
08         positions[index] := GET_PLANE_POINT_DISTANCE(position, direction, point)
09         values[index] := GET_LINE_POINT_DISTANCE(position, direction, point)
10         index := index + 1
11     END
12     RETURN LINEAR_REGRESSION(positions, values)
13 END
```

Internally, the linear regression is done with raw x- and y-values. They are extracted from the given cylinder and the given point cloud and provided to this function. For the linear regression, the center x-y-center position of the data set is calculated. Then, the average slope is calculated by analyzing the local slopes between each data set. Finally, the intercept can be calculated and the regression parameters can be returned. The returned value is an array that contains the slope in the first field and the intercept in the second field.

```
01 FUNCTION LINEAR_REGRESSION(positions[], values[])
02     sumX := SUM(positions)
03     sumY := SUM(values)
04     averageX := sumX / SIZE(positions)
05     averageY := sumY / SIZE(values)
06     sumX := 0
07     sumY := 0
08     index := 0
09     number := SIZE(positions)
10     WHILE (index < number)
11         difference := positions[index] - averageX
12         sumX := sumX + difference * (positions[index] - averageX)
13         sumY := sumY + difference * (values[index] - averageY)
14         index := index + 1
15     END
16     slope := sumY / sumX
17     intercept := averageY - slope * averageX
18     RETURN slope, intercept
19 END
```

Fitting spheres to point clouds

A sphere with a maximum radius is fitted to a point cloud in two steps. The first step is to calculate the center position of the point cloud. In the second step, the point with the maximum distance to the center position is choosed and stored. The returned sphere is then located at the center position and has a radius, specified by the maximum distance.

```
01 FUNCTION FIT_SPHERE_MAXIMUM(points[])
02     center := GET_CENTER_POINT(points)
03     radius := 0
04     FOR (point :  points)
05         distance := GET_POINT_POINT_DISTANCE(center, point)
06         IF (distance > radius)
07             radius := distance
08         END
09     END
10     RETURN CREATE_SPHERE(center, radius)
11 END
```

A sphere with an average radius is also fitted to a point cloud in two steps. The first step is to calculate the center position of the point cloud. In the second step, the average distance of all points to the center position is calculated and stored. The returned sphere is then located at the center position and has a radius, specified by the average distance.

```
01 FUNCTION FIT_SPHERE_AVERAGE(points[])
02     center := GET_CENTER_POINT(points)
03     sum := 0
04     FOR (point :  points)
05         sum := sum + GET_POINT_POINT_DISTANCE(center, point)
06     END
07     radius := sum / SIZE(points)
08     RETURN CREATE_SPHERE(center, radius)
09 END
```

Fitting cylinders to point clouds

The cylinder fitting with maximum radius is already done by the algorithm that fits a cylinder by using the Fibonacci sphere analysis. This function is only a mapping function (for completeness in this overview).

```
01 FUNCTION FIT_CYLINDER_MAXIMUM(points[], precision)
02     RETURN FIT_CYLINDER_WITH_FIBONACCI_SPHERE_ANALYSIS(points, precision)
03 END
```

The cylinder fitting with average radius is a bit more interesting. It first generates a cylinder with maximum radius. After that, it uses the direction vector of the cylinder as x-axis and the distance of each point of the point cloud to the direction vector as y-value and calculates the average radius. Finally, the radius of the found cylinder is changed to the average radius and the cylinder is returned.

```
01 FUNCTION FIT_CYLINDER_AVERAGE(points[], precision)
02     cylinder := FIT_CYLINDER_MAXIMUM(points, precision)
03     position := cylinder.position
04     direction := cylinder.direction
05     radiusSum := 0
06     FOR (point :  points)
07         radiusSum = radiusSum + GET_LINE_POINT_DISTANCE(position, direction, point)
08     END
09     radius := radiusSum / SIZE(points)
10     RETURN CREATE_CYLINDER(position, direction, radius, cylinder.length)
11 END
```

Fitting frustums to point clouds

To fit a frustum with average radius is a bit more complex. The first step is to generate a cylinder. The position vector, the direction vector, and the length are then extracted from the cylinder and reused for the returned frustum. To get the base radius and the top radius, a linear regression is executed with the points of the point cloud, oriented on the direction vector of the cylinder. To get the radii, the y-values of the linear regressions are calculated for the x-values 0 (base surface) and *length* (top surface). The resulting y-values are then used as base radius and top radius of the returned frustum. In a second step, the frustum is flipped if the base radius is smaller than the top radius. This ensures that frustums do always have a logically deterministic direction.

```
01 FUNCTION FIT_FRUSTUM_AVERAGE(points[], precision)
02     cylinder := FIT_CYLINDER_AVERAGE(points, precision)
03     regression := LINEAR_REGRESSION(cylinder, points)
04     slope := regression[0]
05     intercept := regression[1]
06     position := cylinder.position
07     direction := cylinder.direction
08     length := cylinder.length
09     baseRadius := slope * 0 + intercept
10     topRadius := slope * length + intercept
11     IF (baseRadius < topRadius)
12         vector := direction
13         SET_VECTOR_LENGTH(vector, length)
14         position := position + vector
15         direction := -1 * direction
16         temporary := topRadius
17         topRadius := baseRadius
18         baseRadius := temporary
19     END
20     RETURN CREATE_FRUSTUM(position, direction, baseRadius, topRadius, length)
21 END
```

The frustum fitting with maximum radius is based on the respective fitting algorithm with average radius. This algorithm searches for the point in the point cloud with the maximum distance to the local position that would have been calculated with linear regression. By doing this, an additional value for the radius can be calculated and added to the existing radius. It is then added to the base radius and to the top radius to get a frustum with the same angle as before, but with the surface on the most-outside point.

```
01 FUNCTION FIT_FRUSTUM_MAXIMUM(points[], precision)
02     frustum := FIT_FRUSTUM_AVERAGE(points, precision)
03     position := cylinder.position
04     direction := cylinder.direction
05     maximumDistance := 0
06     FOR (point :  points)
07         distanceToPrincipalAxis := GET_LINE_POINT_DISTANCE(position, direction,
point)
08         distanceToBasePlane := GET_PLANE_POINT_DISTANCE(position, direction, point)
09         distance := distanceToPrincipalAxis - (slope * distanceToBasePlane +
intercept)
10         IF (distance > maximumDistance)
11             maximumDistance := distance
12         END
13     END
14     frustum.baseRadius := frustum.baseRadius + maximumDistance
15     frustum.topRadius := frustum.topRadius + maximumDistance
16     RETURN frustum
17 END
```

Fitting cones to point clouds

The calculation of a cone with average radius is based on the calculation of a frustum with average radius. This function generates a frustum and scales the length so that the angle of the frustum side surface is kept.

```
01 FUNCTION FIT_CONE_AVERAGE(points[], precision)
02     frustum := FIT_FRUSTUM_AVERAGE(points, precision)
03     ratio := frustum.baseRadius / frustum.topRadius
04     additionalLength := frustum.length / (ratio - 1)
05     coneLength := additionalLength + frustum.length
06     radius := frustum.baseRadius
07     RETURN CREATE_CONE(frustum.position, frustum.direction, radius, coneLength)
08 END
```

The calculation of a cone with maximum radius is also based on the calculation of a frustum with maximum radius. This function also generates a frustum and scales the length so that the angle of the frustum side surface is kept.

```
01 FUNCTION FIT_CONE_MAXIMUM(points[], precision)
02     frustum := FIT_FRUSTUM_MAXIMUM(points, precision)
03     ratio := frustum.baseRadius / frustum.topRadius
04     additionalLength := frustum.length / (ratio - 1)
05     coneLength := additionalLength + frustum.length
06     radius := frustum.baseRadius
07     RETURN CREATE_CONE(frustum.position, frustum.direction, radius, coneLength)
08 END
```

Fitting automatic objects to point clouds

The automatic fitting function is the most powerful function in the set of point cloud fitting algorithms. It generates one object of each type (a sphere, a cylinder, a frustum, and a cone) and compares them by their score. The object with the best score is returned. This function also considers the average/maximum mode.

Two special cases have to be considered for frustums. In some cylindric or conical test point clouds, only frustums were returned during the debugging phase. This effect appears because no objects are perfect in nature.

Cylinders are not perfect. Their radius on the top end and on the bottom end are slightly different. This brings the algorithm to always return a frustum for natural cylinders. To avoid this, a frustum is only returned, if the radii differ by at least 5%. For smaller differences, a cylinder is returned.

Cones are also not perfect. The algorithm will only return a cone, if the tip is represented by a point. Due to floating point unprecision, this is nearly impossible for natural cones. To avoid this effect, a cone is returned if the top radius of the frustum has 5% of the base radius or less. If the top radius is larger, a frustum is returned.

```
01 FUNCTION FIT_AUTOMATIC_OBJECT(points[], precision, average)
02     IF (average)
03         sphereObject := FIT_SPHERE_AVERAGE(points)
04         cylinderObject := FIT_CYLINDER_AVERAGE(points, precision)
05         frustumObject := FIT_FRUSTUM_AVERAGE(points, precision)
06         coneObject := FIT_CONE_AVERAGE(points, precision)
07     ELSE
08         sphereObject := FIT_SPHERE_MAXIMUM(points)
09         cylinderObject := FIT_CYLINDER_MAXIMUM(points, precision)
10         frustumObject := FIT_FRUSTUM_MAXIMUM(points, precision)
11         coneObject := FIT_CONE_MAXIMUM(points, precision)
12     END
13     sphere := GET_SCORE(sphereObject, points)
14     cylinder := GET_SCORE(cylinderObject, points)
15     frustum := GET_SCORE(frustumObject, points)
16     cone := GET_SCORE(coneObject, points)
17     IF (sphere < cylinder && sphere < frustum && sphere < cone)
18         RETURN sphereObject
19     ELSE IF (cylinder < frustum && cylinder < cone)
20         RETURN cylinderObject
21     ELSE IF (frustum < cone)
22         IF (frustum.topRadius < frustum.baseRadius * 0.1)
23             RETURN coneObject
24         ELSE IF (frustum.topRadius > frustum.baseRadius * 0.9)
25             RETURN cylinderObject
26         ELSE
27             RETURN frustumObject
28         END
29     ELSE
30         RETURN coneObject
31     END
32 END
```