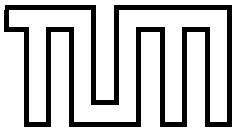
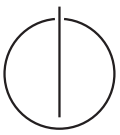


TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK



Intelligent Autonomous Systems Group

Segmentation and Semantic Interpretation of Object Models

Segmentierung und semantische Interpretation von Objektmodellen

Bachelorarbeit in Informatik

Author:	Stefan Profanter
Supervisor:	Prof. Michael Beetz, Ph.D.
Advisor:	Dr. Moritz Tenorth
Submission Date:	15.07.2012

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this bachelor thesis only supported by declared resources.

München, den 15.07.2012

Stefan Profanter

Abstract

In robotics it is more and more essential to understand the surrounding environment. A great advantage is that nowadays almost every object is modelled by a CAD model somewhere in a database. But robots don't yet have the ability to infer properties of these CAD models. So the idea behind this work is to combine these models with understanding: A robot finds an unknown object in the scene and tries to match the scanned point cloud data of the 3D object with a CAD model from a database and analyses the CAD model to get additional information about the object, for example wheter there is a handle or concave parts for carrying fluids.
Fitting CAD models to point cloud data is a mature field of research, so we focus on segmentation and interpretation of CAD models. This bachelor thesis provides some fundamental algorithms and approaches to do segmentation and semantic interpretation of CAD models specially used in household.
Segmentation is achieved by fitting primitives (plane, sphere, cone and cylinder) according to curvature values of each vertex. Afterwards the fitted primitives are analysed to find holes, handles, containers or supporting planes.
Additionally we provide a Prolog interface for querying different properties of these fitted primitives, for example area, volume or direction. We also provide some basic Prolog predicates for querying supporting planes and handles. It is also possible to define new predicates in Prolog to combine properties such as convex part and cylinder with approximately perpendicular direction vectors as object with handle which can carry fluids.

Zusammenfassung

Im Bereich der Robotik wird es immer und immer wichtiger, die Umgebung so gut wie möglich zu verstehen und mit dieser zu interagieren. Ein großer Vorteil dabei ist, dass heutzutage fast jedes Objekt irgendwo als 3D Modell vorhanden bzw. abgespeichert ist. Es existieren bereits verschiedene Ansätze, bei denen Roboter in der Lage sind, diese Modelle in die Umgebung einzupassen. Diese Algorithmen sind jedoch noch nicht in der Lage, semantische Eigenschaften dieser Modelle autonom zu bestimmen. Dieses Problem versuchen wir zu lösen, indem wir dem Roboter die Fähigkeit geben, das dazugehörige Objektmodell automatisch zu suchen und dieses dann zu analysieren, um zusätzliche Informationen wie Griffe oder konkave Teile zu finden, welche z.B. Flüssigkeiten beinhalten könnten.
CAD Modelle in 3D Punktewolken einzupassen wird bereits ausgiebig erforscht. Deshalb legen wir den Schwerpunkt auf die Segmentierung und die semantische Interpretation dieser Modelle. Diese Bachelorarbeit behandelt einige grundlegende Algorithmen und Ansätze, um speziell im Haushalt verwendete Modelle zu segmentieren und zu analysieren.
Die Segmentierung wird dadurch erreicht, indem primitive Objekte (planare Flächen, Kugeln, Kegel, Zylinder) basierend auf die Krümmung des Modells eingepasst werden. Diese primitiven Objekte werden anschließend verwendet, um Löcher, Behälter, Griffe oder gerade Flächen zum Abstellen anderer Objekte zu finden.
Zusätzlich stellen wir eine Schnittstelle zu Prolog zur Verfügung, um verschiedene Eigenschaften der eingepassten primitiven Objekte, wie z.B. die Fläche, das Volumen oder die Richtung abzufragen. Außerdem sind weitere Prädikate definiert, die es erlauben, tragende Flächen oder Griffe zu finden. Es ist auch möglich, neue Prädikate zu definieren, welche verschiedene Eigenschaften kombinieren und dadurch neues Wissen schaffen.

Contents

1 Introduction

Robots find more and more the way into our life. To interact with our environment they have to understand what's around them. If there is a cup, the robot has to know that the cup is a container for fluids and has a handle where you can pick it up. Or if the robot holds something in its hands it has to know where the item can be put down without falling over. A robot should also be able to interact with an unknown environment, for example if the robot is placed in a room where it never was before and needs to cook something, how does it know (besides the recipe and action planning) where different objects in the kitchen can be picked up or which object is for stiring a soup?
These tasks are normally solved by analysing point cloud data from laser scanners. Analysing means segmenting the whole point cloud data in separate objects and try to reason about each object. But point cloud data is noisy and inaccurate especially for small objects and normally you can't be sure that everything is segmented correctly. In addition the range scanner only delivers data on visible parts of an object, the hidden parts remain undiscovered. Therefore you have missing information: maybe it is a cup but the robot sees only the body of the cup and the handle is hidden behind it, so the robot thinks it is a glass or something else.
To improve this task a robot can determine the shape of the object and compare it with already known objects in his database or a database on the world wide web. There is already a huge amount of CAD models from a lot of different sources: the most known 3D model database is from Google called 3D Warehouse [1]. Additional useful databases are: 3D-Net [2], Autodesk 123D [3], 3DModelFree [4], Archive3D [5] or the commercial database with high quality models from TurboSquid [6]. There is also a Robot Operating System (ROS, [Quigley et al., 2009]) module called "ROS household objects" which is a collection of different household objects. Also a lot of companies (for example KARE Design) already provide public databases of their products. These database items are mostly CAD models drawn by humans

[1] http://sketchup.google.com/3dwarehouse/
[2] http://3d-net.org/
[3] http://www.123dapp.com/
[4] http://3dmodelfree.com/
[5] http://archive3d.net/
[6] http://www.turbosquid.com/

and are much more accurate than point cloud data. But finding a match isn't enough, because a CAD model is nothing else than a more accurate and complete representation of the object as point cloud data. So the model doesn't yet provide any additional information how to handle or use the object.
For doing reasoning on such models we first need a simpler representation of these models. So finding CAD models and segmenting them is a means to an end.
There is already a framework for finding the corresponding object of point cloud data from a 3D CAD model database published under the name "3D-Net" [7]. Therefore the methods for finding and fitting models into point clouds aren't part of this work. Instead we try to gather information out of cad models for semantic interpretation. For example we detect the parts of an object or geometric properties and make them available to the robot. We get information of such models by segmenting them into primitive types: sphere, cone, cylinder and plane. Fitting primitives is done by calculating the curvature at each vertex and combine faces by region growing which have same curvature values. The curvature property also indicates which of the primitives the face corresponds to. When all faces are elaborated we try to find for each primitive the properties such as radius or height by least squares fitting. With these properties of each primitive we can do further reasoning, for example calculating the volume or area of an object, finding supporting planes or finding holes in an object.

An example for using these properties is a spoon: the robot mostly isn't able to determine its exact shape with 3D laser scanners. So it can try to find a corresponding CAD model in a database and reason about it. As you can see in Chapter Evaluation, the robot finds a concave part on the spoon and a cylinder which represents the body and handle. With this information a robot knows much more about an object.
Another example may be to find the handle of an object, for example a knife, a bottle or a glass (as shown in Chapter 6 Evaluation) or finding supporting planes of an object to put another object onto it. It is also possible to determine the pose for two objects, a screw and a screw nut, to stick them together by aligning both axes to same direction.
Or another use case is the selection of appropriate objects from a table: If the robot has the task to stir soup and on a table are different tools such as knife, fork and spoon, the robot can analyse the models and finds out that the spoon has the biggest concave part, so it would take the spoon instead the fork. Or if the task is to carry one litre of water to some place, the robot can analyse the volume of each object and take the one, where at least one litre fits in.

[7] http://3d-net.org/

A Prolog interface is provided for getting this information efficiently. Another great advantage is that the robot doesn't need to know what object it is exactly. For example if the model contains a part of a sphere and a cylinder or cone perpendicular to it, the robot can infer that the object can be held at the cylinder and a special amount of fluid can be carried with it. So the robot only needs abstract knowledge about such objects.

At the end of this work we show segmentation results on a few cad models and evaluate our algorithm on a diverse set of CAD models representing different classes of objects commonly encountered by service robots, such as pieces of silverware, cooking utensils, furniture items, as well as mechanical parts like nuts and bolts. Here you can also see the strengths and weaknesses of our approach. Additionally some suggestions are given how this new knowledge can be integrated into robot knowledge processing, in our example within Knowledge Processing for Autonomous Robots (KnowRob [Tenorth, 2011]).

The Implementation of algorithms described in this paper are available as ROS package integrated into KnowRob on http://www.ros.org/wiki/knowrob_mesh_reasoning.

1.1 Reader's Guide

Chapter 1 Here you get an overview for what mesh segmentation and semantic interpretation can be used in field of robotics and automation. We summarize also shortly the content of this elaboration.

Chapter 2 Different approaches in mesh segmentation are discussed, especially segmentation by curvature which we us in our work. Additionally we give a short overview of papers trying to find mesh affordances. There exist also different grasp frameworks for determining the best grasp position of an object.

Chapter 3 A basic algorithm to fit primitives like sphere, cone, cylinder and planes into a CAD model is described in this chapter. For fitting these primitives we use curvature estimation.

Chapter 4 After fitting the primitives, they are used to detect container (e.g. cylinder with a plane on the bottom) or other interesting parts of a CAD model. This information can also be used to get the dimension of a supporting

plane or the volume of a container so that the robot knows how much tee fits in the cup.

Chapter 5 The main part of this work is to integrate the new knowledge into a knowledge representation system for the robot. Here we use "KnowRob" [Tenorth, 2011] in combination with Prolog for reasoning on primitives and getting information out of CAD models.

Chapter 6 As an overview of how accurate or inaccurate our algorithm is, we applied it to different models. Here we discuss why on some of them the algorithm worked quite good and on others not as desired.

Chapter 7 As conclusion we discuss the weaknesses and limitations of our approach and some future improvements for primitive fitting and reasoning. Here you get also an overview which parts of this work were the most challenging.

2 Related Work

In this chapter some related work on mesh segmentation and semantic interpretation of 3D data is presented. A lot of papers already focus on mesh segmentation with different approaches. But in the field of semantic interpretation of 3D data there are quite few publications available. The semantic interpretation builds on the results of mesh segmentation and is therefore dependent on the quality of the segmentation algorithm.
There exist also different Frameworks for finding the best grasping position of objects.

2.1 Mesh segmentation

Mesh segmentation is mostly used in computer graphics with applications in areas such as modelling, compression, 3D shape retrieval, collision detection and a lot more. Therefore many algorithms have already been developed for segmenting 3D data. Segmentation algorithms can be divided into two main classes: *Feature-based detection* and *Direct segmentation* [Attene, Katz, et al., 2006]. These algorithms focus on natural shapes like a hand or bodies of humans or animals. A few of them also deal with segmentation of artificial shapes like CAD models [Attene, Katz, et al., 2006; Bénière, Subsol, Gesquière, Le Breton, and Puech, 2011]. A comprehensive study on mesh segmentations gives [Attene, Falcidieno, and Spagnuolo, 2006] or with additional different approaches [Agathos, Pratikakis, Perantonis, Sapidis, and Azariadis, 2007].

Feature-based detection A feature based detection [Katz, Leifman, and Tal, 2005] tries to find feature lines like folds in triangle meshes (edge where principal curvature exceeds a prescribed threshold). Using this detection mode typically results in gaps at the boundaries of regions and makes it therefore difficult to avoid fuzzy boundaries.

Direct segmentation Instead of finding boundaries, direct methods start with adjacent faces or neighbouring points. Region growing algorithms take some

seed points and expand each region according predefined criteria. Here the main difficulty is to select good initial seeds and growing criteria.
Hierarchical mesh segmentation [Attene, Katz, et al., 2006] doesn't need initial seeds. Here the smallest unit, a triangle face, is taken and adjacent triangles are added hierarchically. Then primitives (plane, sphere, cylinder) are fitted to each new generated face. Another hierarchical based face clustering is presented in [Garland, Willmott, and Heckbert, 2001]. [Bénière et al., 2011] uses curvature values of each vertex to best fit planes, cylinders, cones and spheres into the model. Recovering primitives is also possible from point cloud data [Mehtap, 2010] but is not yet as accurate as using CAD models data. Segmentation of 3D point clouds in combination with learning algorithms is also handled in [Tombari, Di Stefano, and Giardino, 2011; Tombari and Luigi, 2011].
The most recent work on mesh segmentation used for robot manipulation was published in March, 2012: [Lee, Yoo, Kim, and Lee, 2012].

2.1.1 Curvature Estimation

We use curvature estimation for segmenting the mesh into sub meshes and for fitting primitives. The most popular work on curvature estimation is [Goldfeather and Interrante, 2004]. This paper also compares different methods like Normal Curvature Approximation Method, Quadratic Surface Approximation Method and Adjacent-Normal Cubic Approximation Method. [Xiaopeng Zhang, Hongjun Li, and Zhanglin Cheng, 2008] gives a good introduction in calculating curvature from points around a normal vector similar to [Zhihong, Guo, Yanzhao, and Lee, 2011] where the per vertex curvature is calculated by taking a weighted average of per triangle curvature. The problem with most of these algorithms is, that they rely on an equally distributed size of the triangles.
In [Szlivási-Nagy, 2006] the curvature is estimated by drawing up circles around a specific point and finding the minimum and maximum distance to the triangles intersecting this circle.
[Rusinkiewicz, 2004] uses a weighted average of the normal vectors of faces touching a vertex for estimating curvatures and the derivatives. This method is used in this work and will therefore be described in detail in section 3.2.

2.2 Semantic Interpretation

Semantic annotation and interpretation is not as popular as computer graphics but is getting more and more important in the field of robotics and artificial intelligence. Therefore this field is not yet as much explored as the field of mesh segmentation.
Semantic interpretation means that the machine or robot knows the functionality and meaning of an object or a specific part of it. [Attene, Robbiano, Spagnuolo, and Falcidieno, 2009] does a pre segmentation of shapes and needs a human to specify the semantic relationship between the segments.
Finding affordances like containment (the object can contain other objects), liquid-containment (the object can contain liquids), unstable (the stability of the pose is compromised if pushed), stackable-onto (objects can be stacked onto the object) or sittable (an agent can sit on it like a human would do) based on the pose in the scene is described in [Aldoma, Tombari, and Vincze, 2012].

The following publications and frameworks especially focus on grasping objects, which is also part of our work, by finding handles in CAD models:
In [Dag, Atıl, Kalkan, and Sahin, 2010] categorizing objects and their properties is achieved by simple interaction with objects in the environment. Here the agent can categorize objects based on the predicted effects of actions that can be applied on them.
The Grasp project's [1] aim, funded by the European Commission, is the design of a cognitive system capable of performing grasping and manipulation tasks in open-ended environments, dealing with novelty, uncertainty and unforeseen situations.
Another similar grasp pipeline is described in [Ciocarlie et al., 2010] which combines aspects such as scene interpretation from 3D range data, grasp planning, motion planning, grasp failure identification and recovery using tactile sensors.
Grasping based on primitives like cylinders and spheres is presented in [Nieuwenhuisen, Stückler, Berner, Klein, and Behnke, May 2012]. These primitives are detected by approximating surface normals with predetermined parameters. In comparison to our work, they use their inaccurate fitted primitives only for grasping tasks, not for further reasoning such as volume calculation or finding supporting planes.
3D-Net [2] is another framework for grasping objects with integration into Point

[1] http://www.csc.kth.se/grasp/
[2] http://3d-net.org

Cloud Library (PCL [3]). 3D-Net tries to match CAD models from a database into point cloud data and analyses the grasping position on the matched CAD model.

[3] http://pointclouds.org/

3 Segmentation

The basic structure of our application is based on the structure of the Apache Unstructured Information Management Applications (UIMA, [1]) project (see Section 3.1). The segmentation is achieved by analysing the curvature of each vertex (Section 3.2) and creating sub meshes for fitting primitives (Section 3.3).

3.1 Application structure

The Apache UIMA project is a collection of frameworks, tools and annotators for analysis of unstructured content such as text, audio and video.
UIMA can be seen as box which holds all the parts for analysing the unstructured data together.

> UIMA is a software architecture which specifies component interfaces, data representations, design patterns and development roles for creating, describing, discovering, composing and deploying multi-modal analysis capabilities. [http://uima.apache.org]

For explaining the basic structure of UIMA we use the following example (for a complete explanation please refer to http://uima.apache.org):
3D mesh data is simply a collection of vertices which, when connected, result in a collection of triangles. Compared to text, a triangle can be a single character. Segmenting these parts is the same as splitting a sentence without spaces like "QuestoÈUnaCasaGrande" into "Questo È Una Casa Grande". This can be done because you know that each word begins with a capital letter. But you don't know yet the meaning of this sentence. The next step is to semantically analyse (translate) the parts of the sentence to get "This is a big house". Additionally you may have a picture of the house. By analysing the picture, an algorithm comes to the same conclusion that the house is big.
In the view of UIMA, the collection of raw data (Italian sentence, image) and

[1] https://uima.apache.org/

the translated sentence/interpreted picture is called *Common Analysis Structure* (CAS). The sentence and image are called document. An *Analysis Engine* (AE) is composed to analyse a single document, so in our example there is a Splitting AE, Translating AE and an Image Processing AE.
Additionally UIMA supports multiple views of a document: e.g. the whole sentence or only a single word. An Analysis Engine analyses one or more views of a document. Therefore these views are called *Subject of Analysis* (SOFA). An Analysis Engine produces *Annotations* on a specific span of the input: each word is an annotation (the annotation contains the start index and end index of the word in the string), the house in the image is also an annotation (the annotation contains the pixels which represent the house). These annotations are stored in the CAS container.
An annotation may be of a specific *Type* like Word or ImagePart. Types have properties called *Features*. So for example LengthOfWord may be a Feature.

This principle is mapped in the `edu.tum.sc.uima` package. The classes within this package are base classes for each analyser and annotation in the package `edu.tum.cs.vis.model.uima.analyser`, `edu.tum.cs.vis.model.uima.annotation` and `edu.tum.cs.vis.model.uima.cas`. `MeshCas.java` is the class definition for the CAS containing a full CAD model and all annotations regarding this model. The different analyser and annotations are explained in the following sections.

3.2 Curvature estimation

The curvature values of a vertex are used to determine the curvature on the surrounding area for segmenting the mesh into flat, convex or concave parts. These parts are then fitted with primitives for further analysis. Different methods exist for estimating the direction and magnitude of the minimum and maximum curvature. In this work we use the method by [Rusinkiewicz, 2004] because it can be used for irregular triangle meshes and approximates the curvature with very small errors. Additionally the user doesn't have to specify the size of neighbourhood which will be analysed for curvature.
The normal curvature k_n of a surface in some direction is the reciprocal of the radius of the circle that best approximates the surface in this direction. The curvatures in minimum and maximum direction are also called principle curvatures k_1 and k_2 with the corresponding principle directions. There exist also methods that

estimate only the mean curvature $H = (k_1 + k_2)/2$ or Gaussian curvature $K = k_1 k_2$.

3.2.1 Finding vertex normals

The first step is to calculate for each triangle vertex the vertex normal (see Figure 3.1).

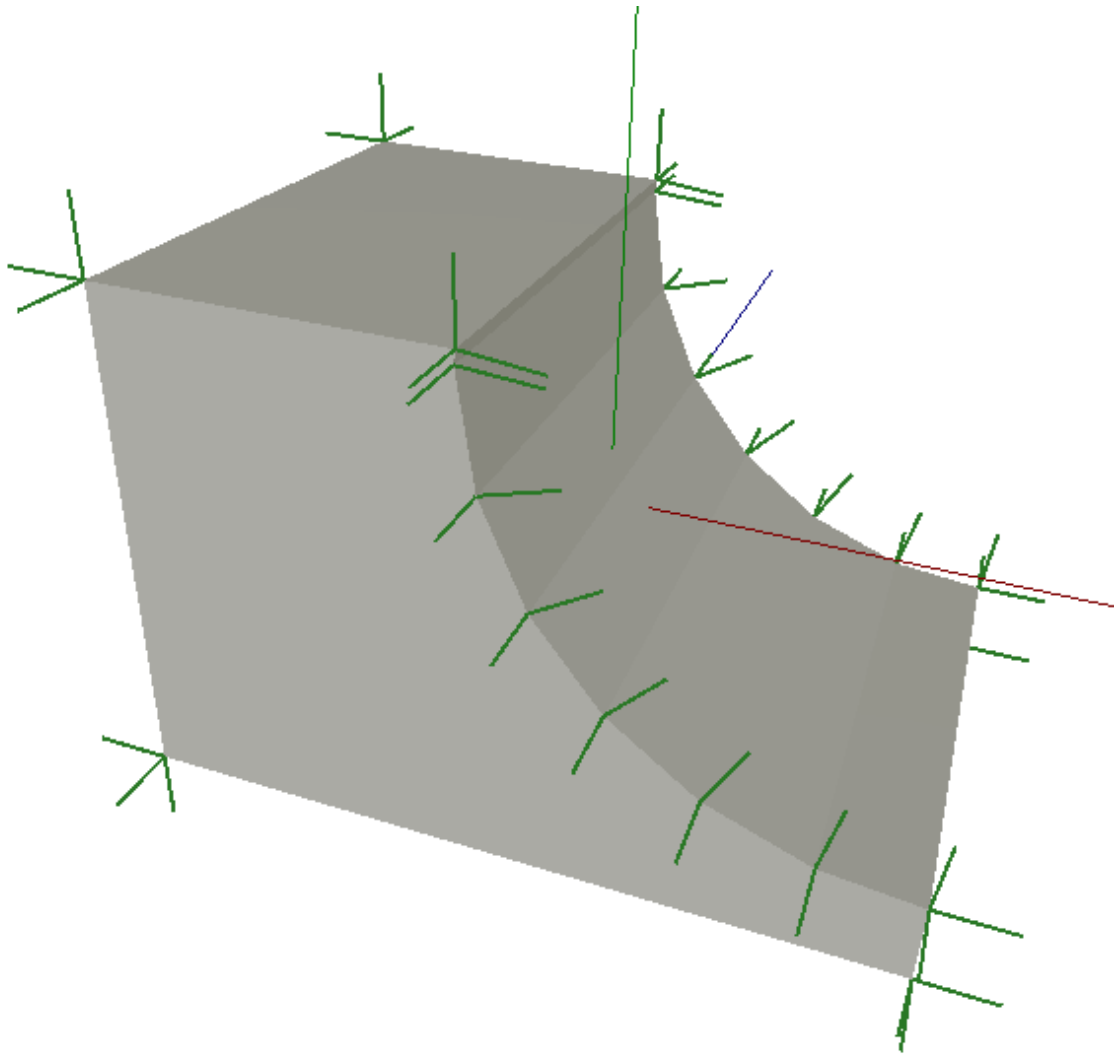


Figure 3.1: *Vertex normals for a rounded model. A specific coordinate may have more than one normal vector because there are multiple vertices at this coordinate but of different faces.*

This is done with Algorithm 1.

Algorithm 1: Calculate vertex normals as shown in figure 3.1

for all triangles in mesh **do**
  $e1 \leftarrow pos1 - pos0$ {calculate triangle edge vectors}
  $e2 \leftarrow pos1 - pos2$
  $e3 \leftarrow pos2 - pos0$
  if length of a vector ei is null **then**
    continue with next triangle
  end if
  $facenormal \leftarrow e2 \times e1$ {Direction of normal vector of face, NOT normalized}
  $normal(pos0) \leftarrow normal(pos0) + facenormal.scale(1/(len(e1) * len(e3)))$
  $normal(pos1) \leftarrow normal(pos1) + facenormal.scale(1/(len(e1) * len(e2)))$
  $normal(pos2) \leftarrow normal(pos2) + facenormal.scale(1/(len(e2) * len(e3)))$
end for
for all vertices in mesh **do**
  normalize vertex normal
end for

3.2.2 Voronoi Area

To calculate the curvature at a specific vertex we need to know how much of the face curvature should be accumulated at each vertex. For this weight we use the area belonging to each vertex, defined as Voronoi Area restricted to the 1-ring of a vertex (see Figure 3.2).
The method and formula for computing the Voronoi Region Area for a vertex is given in the chapter "3.3 Voronoi Region Area" in [Meyer, Desbrun, Schröder, and Barr, 2002].

3.2.3 Principle direction and curvature

The next step is to create an initial orthogonal coordinate system (n, u_p, v_p) for each vector by setting the principle direction vectors at each vertex to the two axes perpendicular to the face normal n.

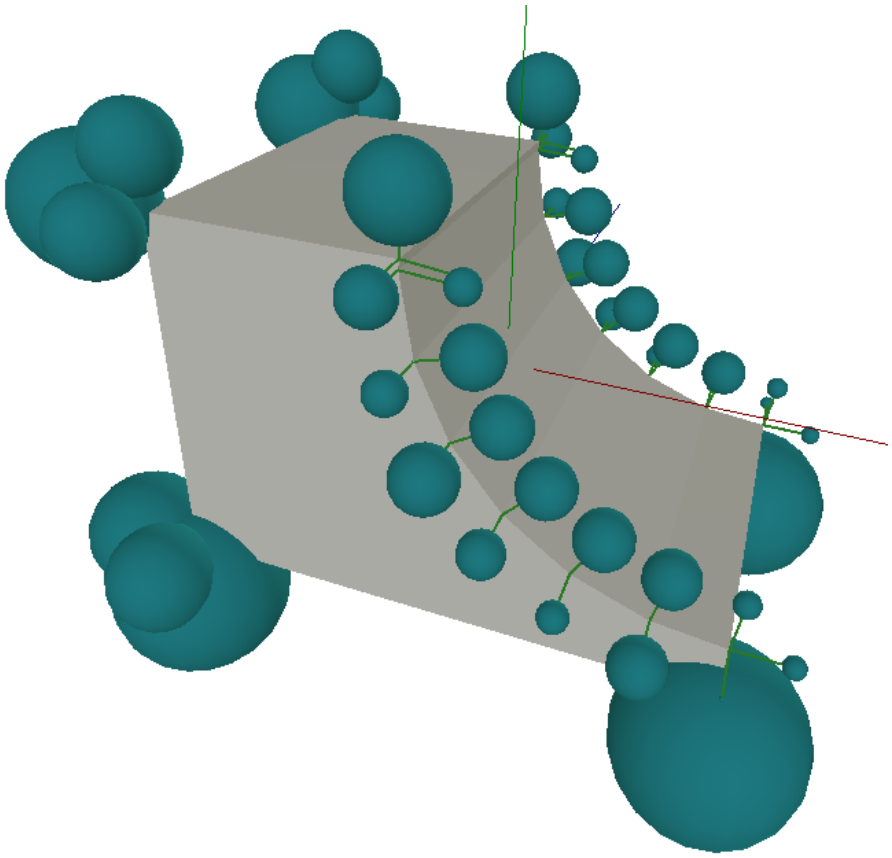


Figure 3.2: *Voronoi area of each vertex. The Voronoi Area is indicated by the size of each sphere at the top of vertex normals (Figure 3.1).*

For getting the curvature of a triangle we need the second fundamental form **II**, which is defined in terms of the directional derivatives [2] as:

$$\mathbf{II} = \begin{pmatrix} D_u n & D_v n \end{pmatrix} = \begin{pmatrix} \frac{\partial n}{\partial u} \cdot u & \frac{\partial n}{\partial v} \cdot u \\ \frac{\partial n}{\partial u} \cdot v & \frac{\partial n}{\partial v} \cdot v \end{pmatrix} \tag{3.1}$$

, where (u,v) are the directions of an orthonormal coordinate system in the tangent frame of the face.
Multiplying **II** by any vector on the tangent plane gives the derivative of the normal in that direction $\mathbf{II}s = D_s n$. This derivative of a normal is itself a vector on the tangent plane.

[2] http://mathworld.wolfram.com/DirectionalDerivative.html

With the second fundamental form we can form a set of linear constraints (see also Figure 3.3)

$$\mathbf{II} = \begin{pmatrix} e_0 u \\ e_0 v \end{pmatrix} = \begin{pmatrix} (n_2 - n_1)u \\ (n_2 - n_1)v \end{pmatrix} \tag{3.2}$$

$$\mathbf{II} = \begin{pmatrix} e_1 u \\ e_1 v \end{pmatrix} = \begin{pmatrix} (n_0 - n_2)u \\ (n_0 - n_2)v \end{pmatrix} \tag{3.3}$$

$$\mathbf{II} = \begin{pmatrix} e_2 u \\ e_2 v \end{pmatrix} = \begin{pmatrix} (n_1 - n_0)u \\ (n_1 - n_0)v \end{pmatrix} \tag{3.4}$$

, which can be solved using least squares for getting the parameters of the second fundamental form and from that the curvature tensor expressed in the coordinate system (u_f,v_f) of the face. This curvature tensor must be averaged

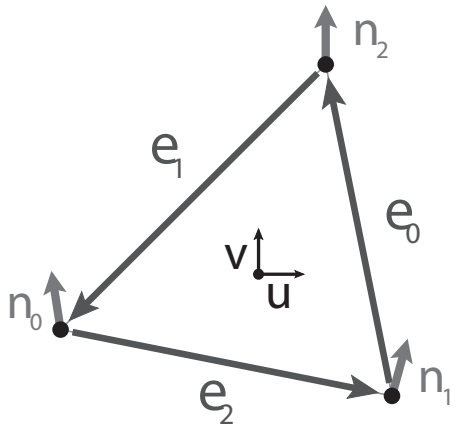


Figure 3.3: *Normal vectors n_i for each vertex and edge vectors e_i of a triangle*

with contributions from adjacent triangles. To do this, we use the coordinate system (u_p,v_p) of each vector (as defined previously) and project the face curvature tensor to each vector by rotating (u_p,v_p) to be coplanar to (u_f,v_f). The projected curvature tensor gets then multiplied with the previously defined Voronoi area weight and added to the minimum and maximum curvature for the corresponding vertex.
After each face is elaborated, for each vertex the accumulated curvature tensor **II** is divided by the sum of the Voronoi Area weights. Out of this curvature tensor the principal directions and curvatures are calculated for the vertex by computing eigenvalues and eigenvectors of **II**.

A summary of the steps for estimating the curvature for each vertex is given in Algorithm 2. The result is shown in Figure 3.4.

Algorithm 2: Calculate the principle curvatures and directions of each vertex

```
Compute per-vertex normals (see Algorithm 1)
Construct an initial orthogonal coordinate system (u_p,v_p)
for all triangles in mesh do
  Compute edge vectors e_i and normal differences Δn
  Solve II using least squares
  for all vertex p of the triangle do
    Project II into (u_p,v_p)
    Add the projected tensor, weighted by w_{f,p} t vertex tensor
  end for
end for
for all vertices in mesh do
  Divide the accumulated vertex tensor II by the sum of weights
  Find principal curvature and directions by computing eigenvalues and
  eigenvectors of II
end for
```

3.2.4 Colouring by curvature

By calculating a hue and saturation for each curvature value we have two advantages: First, we are able to visualize curvature values graphically by colouring each vertex according to it's curvature (see Figure 3.5), Second, we have a compact representation of the curvature which we can use for determining primitive types (see Section 3.3).
For calculating hue and saturation we use the following formulas:

$$H = \frac{max + min}{2} \quad \text{mean curvature} \tag{3.5}$$

$$K = max \cdot min \quad \text{Gaussian curvature} \tag{3.6}$$

$$hue = \frac{4}{3}|atan2(H^2 - K, H^2 \cdot sgn(H))| \quad \text{(Get hue between 0 and 240 degree)} \tag{3.7}$$

$$saturation = \frac{2}{\pi}atan((2H^2 - K) \cdot scale) \quad \text{(Saturation between -1 and 1} \tag{3.8}$$

min and *max* are the minimum and maximum curvature values of each vertex. *scale* can be used to saturate also small curved planes or only vertices with high curvature. It is recommended to use *scale* depending on the total size of the model (eg. use the smallest enclosing ball [Gärtner, 1999]).

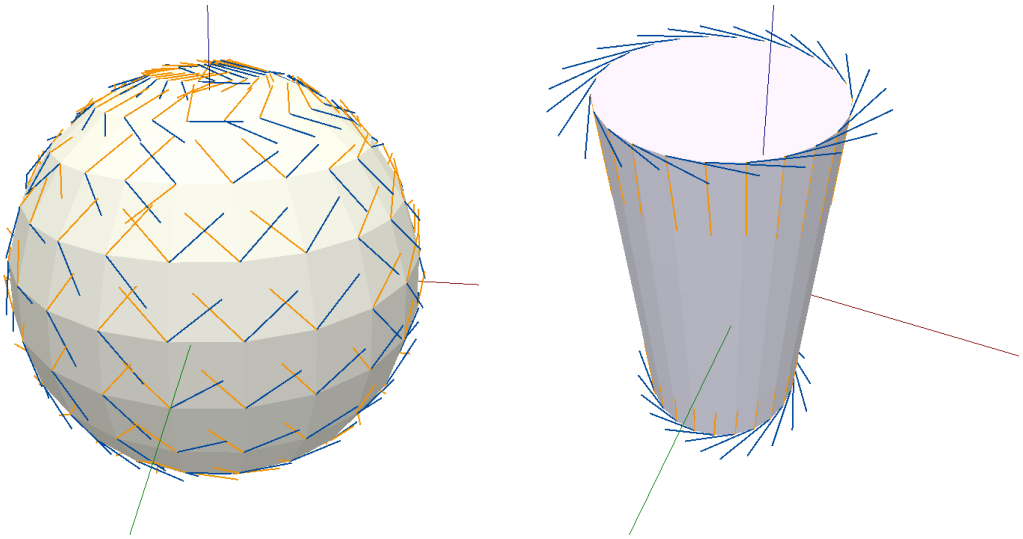


Figure 3.4: *Principle curvature directions on a sphere and cylinder for each vertex. The vectors indicating the direction are scaled by the curvature magnitude and coloured as blue (maximum curvature) and orange (minimum curvature).*

3.3 Fitting primitives

As introduced in the previous section we use estimated curvature values for fitting primitives into an irregular triangle mesh. The main steps are: combine neighbouring triangles with approximately the same curvature values to a new face and fit a primitive to the new face based on the curvature properties. Due to the fact that the curvature is estimated, combining the triangles may be inaccurate. Therefore smoothing is used by including curvature properties of adjacent triangles. So the first step is to find adjacent triangles for each triangle.

3.3.1 Finding neighbours

A neighbour of a triangle is a triangle which has exactly two vertices with identical coordinates. A triangle has three sides so it can have maximum three neighbours. For later elaboration the neighbours are stored as a bidirectional relation for each triangle.

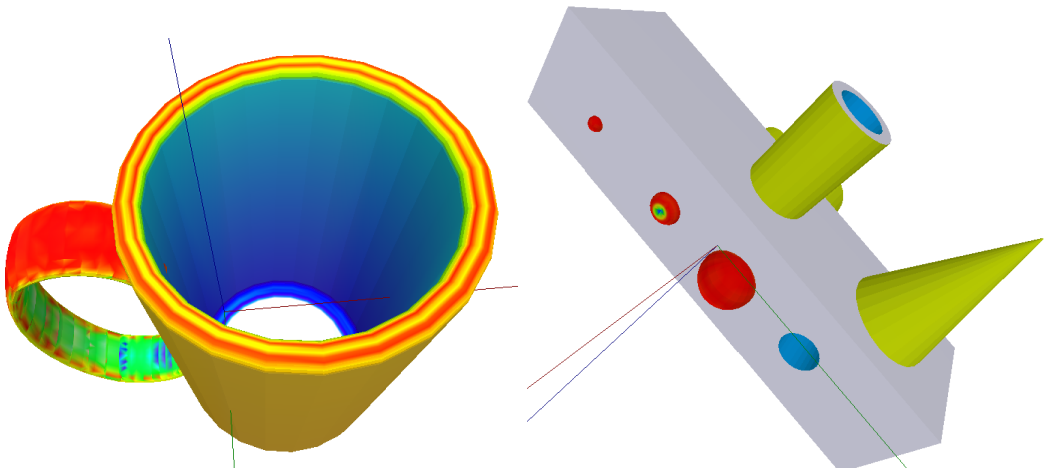


Figure 3.5: *Models coloured by curvature. red: convex sphere, green: concave sphere, yellow: convex cylinder or cone, blue concave cylinder or cone, white: plane or flat surface. The higher the saturation, the higher is the curvature value.*

3.3.2 Combining faces with same curvature

For fitting primitives, faces with approximately the same curvature need to be grouped into a bigger face. Grouping means to create a new annotation which has all of the child triangles stored in an array. To decide if a triangle is part of the same face as the neighbour, first all the vertices of the mesh are analysed and according to its curvature features one of the following properties is assigned: plane, sphere convex, sphere concave, cone convex, cone concave. A special property for cylinder isn't needed because a cylinder is a special form of a frustum cone where both radii are equal.
The colour values of hue and saturation (see Subsection 3.2.4) are a good indicator for the curvature property. Therefore we use the following table to assign to each vertex a curvature property (remember that the saturation is between -1 and 1, hue is between 0 and 240 degree):

plane	$saturation < 0.45$	
sphere convex	$hue < 35°$	(red)
cone convex	$35° \leq hue < 75°$	(yellow)
sphere concave	$75° \leq hue < 150°$ or $hue \geq 230°$	(green)
cone concave	$150° \leq hue < 230°$	(blue)

Now for each triangle the curvature property is assigned by taking the property which most often occurs on the three vertices. If all of the vertices have different properties the order of decision is `plane`,`sphere convex`, `sphere concave`, `cone`

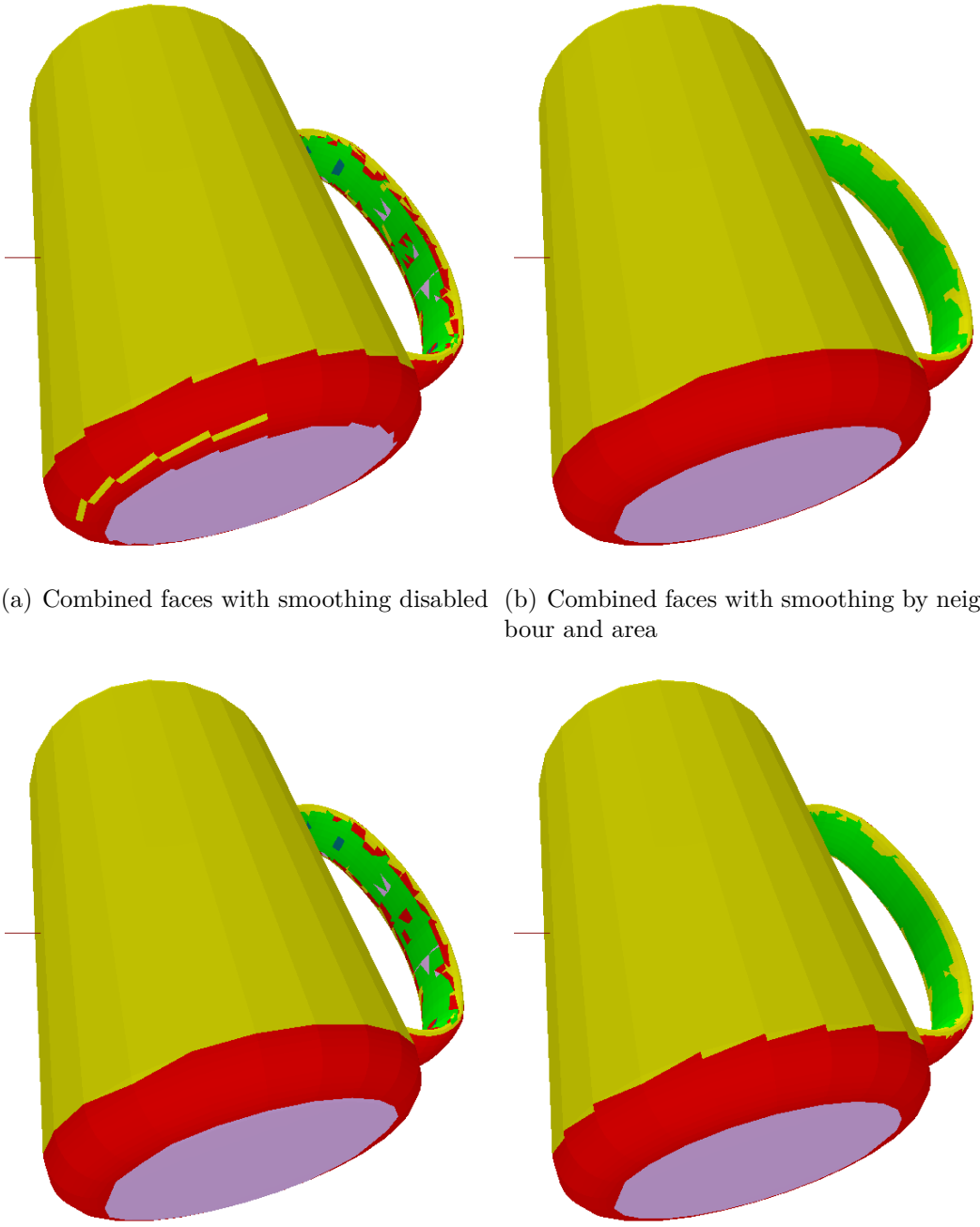
`convex`, `cone concave`. We choose this order because a face is most likely a plane, then a sphere or if none of them a cone. The resulting property assignment is shown in Figure 3.6(a). Now all the triangles which are direct neighbours (see Section 3.3.1) and have the same curvature property assigned, will be combined into bigger faces which are then called primitive annotations. This process is also often called region growing. Each primitive annotation contains a list of triangles and the according primitive type. As it can be seen in Figure 3.6(a), the result contains a lot of small faces of different type (e.g. on the handle of the cup).
These small errors can be smoothed by iterating over each primitive annotation and comparing the area of the current annotation to its direct neighbours. If the current area is smaller than 5% the current annotation is merged into the bigger one. This type of smoothing eliminates small annotations which should be part of bigger ones (see Figure 3.6(d)).
Additionally it is possible to smooth the borders of each annotation as shown in Figure 3.6(c) by checking which annotation type the direct neighbour triangles have. For this we use a counter for each primitive type (5 counters). First each vertex of the current triangle is checked and the corresponding counter is incremented. Then for each of the maximum three neighbouring triangles the corresponding counter is incremented for the curvature property of the triangle. So afterwards the sum over all counters is between 3 and 6. The counter with the biggest value indicates now the primitive type of the triangle. If two counters are equal, the order of decision is the same as above.
Combining both smoothing algorithms the final result is shown in Figure 3.6(b).

3.3.3 Fitting primitives to faces

After the faces are combined as described in previous section each primitive annotation needs to be approximated by the corresponding primitive type. Approximating means fitting a plane, sphere or cone into the face and determining the features (width, height, radius, ...) of it.

3.3.3.1 Fitting plane

A plane is defined as a rectangle lying in 3D space with a normal vector and two values for the length of each side with the corresponding four corner coordinates.



(a) Combined faces with smoothing disabled

(b) Combined faces with smoothing by neighbour and area

(c) Combined faces with smoothing by neighbour

(d) Combined faces with smoothing by area

Figure 3.6: *Combined faces with different smoothing approaches. plane (white), sphere convex(red), cone convex (yellow), sphere concave (green), cone concave (blue)*

Normal vector First the plane normal vector is needed. It is estimated by the following least squares method [3]:
The formula for an arbitrary 3D plane which goes through $(0, 0, 0)$ is $a \cdot x + b \cdot y + c \cdot z + d = 0$. We wish to minimize the distance from each point to the plane. This is noted as finding a, b, c and d which minimizes the equation:

$$f(a, b, c, d) = \sum \frac{(a \cdot x_i + b \cdot y_i + c \cdot z_i + d)^2}{a^2 + b^2 + c^2} \tag{3.9}$$

The partial derivative of equation 3.9 with respect to d set equal to zero results in

$$d = -(a \cdot x_0 + b \cdot y_0 + c \cdot z_0) \tag{3.10}$$

where (x_0, y_0, z_0) is the centroid of data. Substitute this back into 3.9 and you get:

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0 \tag{3.11}$$

So $f(a, b, c, d)$ can be rewritten like this:

$$f(a, b, c) = \sum \frac{(a(x_i - x_0) + b(y_i - y_0) + c(z_i - z_0))^2}{a^2 + b^2 + c^2} \tag{3.12}$$

or in matrix form:

$$v = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \qquad M = \begin{pmatrix} x_1 - x_0 & y_1 - y_0 & z_1 - z_0 \\ x_2 - x_0 & y_2 - y_0 & z_2 - z_0 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ x_n - x_0 & y_n - y_0 & z_n - z_0 \end{pmatrix} \tag{3.13}$$

because

$$f(v) = (v^T M^T)(Mv)/(v^T v) \tag{3.14}$$

$$= v^T (M^T M)v/(v^T v) \tag{3.15}$$

To compute eigenvectors of $A = M^T M$ we use the singular value decomposition (SVD) of $M = UWV^T$. The column of V where W has its biggest singular value is then the normal vector of the plane. To avoid that small triangles, which deviate from the real normal vector, affect the estimated normal vector too much we use weighted SVD:

$$M_w = \begin{pmatrix} (x_1 - x_0) \cdot w_1 & (y_1 - y_0) \cdot w_1 & (z_1 - z_0) \cdot w_1 \\ (x_2 - x_0) \cdot w_2 & (y_2 - y_0) \cdot w_2 & (z_2 - z_0) \cdot w_2 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ (x_n - x_0) \cdot w_n & (y_n - y_0) \cdot w_n & (z_n - z_0) \cdot w_n \end{pmatrix} \tag{3.16}$$

where w_i is the area of triangle i divided by 3 to distribute the area to each vertex.

[3] http://mathforum.org/library/drmath/view/63765.html

2D Bounding box Additionally to the normal vector the dimension and rotation of the rectangle (2D bounding box of vertices) is needed. To fit a rectangle to 2D data, we need to project 3D vertices of each triangle to a 2D plane perpendicular to the normal vector. The orthonormal basis (N, U, V) for the 3D plane is calculated with:

$$tmp = \begin{cases} (0,1,0), & \text{if } N.x > N.y, \\ (1,0,0), & \text{otherwise} \end{cases} \tag{3.17}$$

$$V = normalized(tmp \times N) \tag{3.18}$$

$$U = N \times V \tag{3.19}$$

Each vertex v_i can now be classified by a simple dot product within this orthonormal basis as a 2D point p (where $c = (x_0, y_0, z_0)$ is the centroid of all 3D points and $\cdot$ is the dot product):

$$p_i = \begin{pmatrix} U \cdot (v_i - c) \\ V \cdot (v_i - c) \end{pmatrix} \tag{3.20}$$

The minimum area enclosing rectangle for these points is now calculated with the algorithm as described at http://www.mathworks.com/matlabcentral/fileexchange/31126-2d-minimal-bounding-box/content/minBoundingBox.m. This algorithm first calculates the convex hull for all the points. We use the Graham Scan algorithm for calculating the convex hull. With the edges of this convex hull, rotation matrices for the 2D coordinate systems are created. With this collection of coordinate systems the area for each enclosing rectangle aligned to the rotated axes is calculated. The rectangle with minimal area and its corresponding coordinate system is now our resulting rectangle (see Figure 3.7).

3.3.3.2 Fitting sphere

Fitting a sphere into given vertices is based on [Eberly, 2008].
A sphere is represented by $(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$ where (a, b, c) is the sphere centre and r is the sphere radius. A precondition for this algorithm is, that not all points are coplanar which is given in our case because otherwise the vertices wouldn't be recognized as part of a sphere.
The energy function to minimize the error of the sphere an the vertices is

$$E(a, b, c, r) = \sum_{i=1}^{m} (L_i - r)^2 \tag{3.21}$$

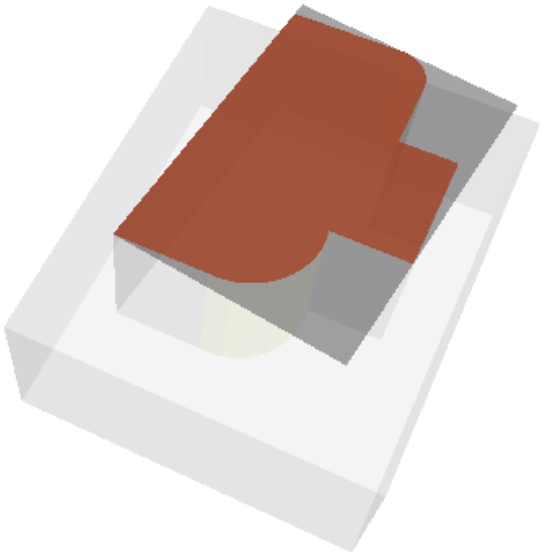


Figure 3.7: *Fitted rectangle bounding box (dark grey) to an arbitrary plane face (red)*

m is the number of vertices and $L_i = \sqrt{(x_i - a)^2 + (y_i - b)^2 + (z_i - c)^2}$. The partial derivative of 3.21 with respect to r set equal to zero yields:

$$r = \frac{1}{m} \sum_{i=0}^{m} L_i \tag{3.22}$$

Partial derivative of 3.21 with respect to a set equal to zero:

$$a = \frac{1}{m} \sum_{i=1}^{m} x_i + r \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L_i}{\partial a} \tag{3.23}$$

Partial derivative of 3.21 with respect to b set equal to zero:

$$b = \frac{1}{m} \sum_{i=1}^{m} y_i + r \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L_i}{\partial b} \tag{3.24}$$

Partial derivative of 3.21 with respect to c set equal to zero:

$$c = \frac{1}{m} \sum_{i=1}^{m} z_i + r \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L_i}{\partial z} \tag{3.25}$$

This can be simplified as:

$$a = \overline{x} + \overline{LL_a} =: F(a, b, c) \tag{3.26a}$$
$$b = \overline{y} + \overline{LL_b} =: G(a, b, c) \tag{3.26b}$$
$$c = \overline{z} + \overline{LL_c} =: H(a, b, c) \tag{3.26c}$$

where

$$\overline{x} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3.27a}$$

$$\overline{y} = \frac{1}{m}\sum_{i=1}^{m} y_i \tag{3.27b}$$

$$\overline{z} = \frac{1}{m}\sum_{i=1}^{m} z_i \tag{3.27c}$$

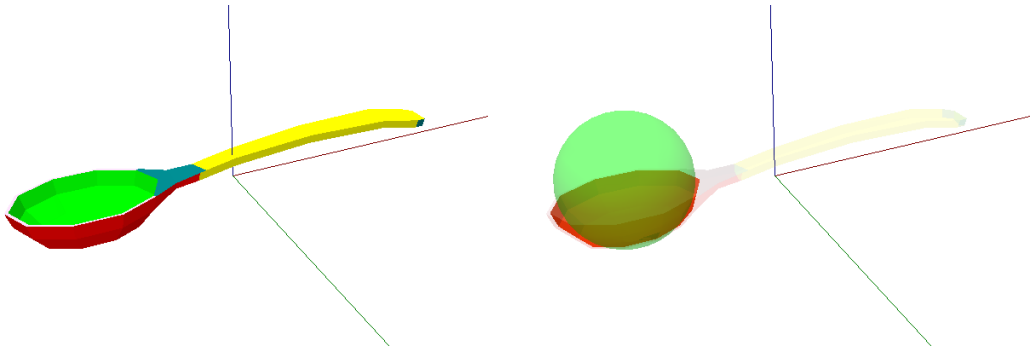
$$\overline{L} = \frac{1}{m}\sum_{i=1}^{m} L_i \tag{3.27d}$$

$$\overline{L}_a = \frac{1}{m}\sum_{i=1}^{m} \frac{a - x_i}{L_i} \tag{3.27e}$$

$$\overline{L}_b = \frac{1}{m}\sum_{i=1}^{m} \frac{b - y_i}{L_i} \tag{3.27f}$$

$$\overline{L}_c = \frac{1}{m}\sum_{i=1}^{m} \frac{c - z_i}{L_i} \tag{3.27g}$$

By using fixed point iteration we can solve these equations. The start conditions are: $a_0 = \overline{x}$, $b_0 = \overline{y}$ and $c_0 = \overline{z}$. Iteration is done over $a_{i+1} = F(a_i, b_i, c_i)$, $b_{i+1} = G(a_i, b_i, c_i)$, $c_{i+1} = H(a_i, b_i, c_i)$ for $i \geq 0$ until $|a_i - a_{i-1}| + |b_i - b_{i-1}| + |c_i - c_{i-1}| = 0$. To ensure that the iteration stops we use a maximum iteration count of 500.
An example of a fitted sphere is shown in Figure 3.8(b)



(a) Original spoon segmented

(b) A sphere (green) fit into the concave area of the spoon

Figure 3.8: *A simple spoon coloured by primitive annotations: plane (grey), sphere convex (red), sphere concave (green), cone convex (yellow), cone concave (blue)*

3.3.3.3 Fitting cone

A cylinder is a special form of a cone where bottom radius and top radius are equal. So the following algorithm is the same for cone and cylinder. Additionally we don't need to distinguish between convex and concave because the cone has always the same shape, only a Boolean indicates if it is convex (outside is visible) or concave (inside is visible).
A cone consists of a generating line which goes through the cone centre, a height and two radii for bottom and top radius.

Find generating line The generating line is the line where all surface points perpendicular to it have the same distance. So in ideal case it goes through the centre of the bottom plane, through the centroid and through the centre of top radius (see Figure 3.10(c)). To get this generating line first we need points which define this line. These points are the intersection points of the inverse vertex normal scaled by the reciprocal of the curvature ($\frac{1}{curvature} = radius$). Due to some imprecision it may be that these lines don't intersect but meet close to each other. To get this point with shortest intersection route we use the following algorithm [4]: A point on

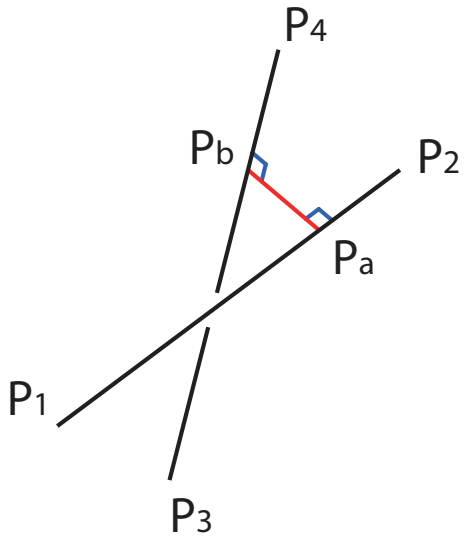


Figure 3.9: *Finding shortest line segment (P_a,P_b) between two lines (P_1,P_2) and(P_3,P_4)*

line a respectively b is defined as

$$P_a = P_1 + mu_a(P_2 - P_1) \tag{3.28a}$$
$$P_b = P_3 + mu_b(P_4 - P_3) \tag{3.28b}$$

[4] http://local.wasp.uwa.edu.au/~pbourke/geometry/lineline3d/

where mu_a and mu_b range from negative to positive infinity and indicate the position of P_a and P_b on each line. If these points lie on the line, mu_a and mu_b are between 0 and 1.
Due to the fact that the line segment is perpendicular to the two lines we have the following two conditions ($\cdot$ = dot product):

$$(P_a - P_b) \cdot (P_2 - P_1) = 0 \tag{3.29a}$$
$$(P_a - P_b) \cdot (P_4 - P_3) = 0 \tag{3.29b}$$

Substituting 3.28 into 3.29 gives:

$$(P_1 - P_3 + mu_a(P_2 - P_1) - mu_b(P_4 - P_3)) \cdot (P_2 - P_1) = 0 \tag{3.30a}$$
$$(P_1 - P_3 + mu_a(P_2 - P_1) - mu_b(P_4 - P_3)) \cdot (P_4 - P_3) = 0 \tag{3.30b}$$

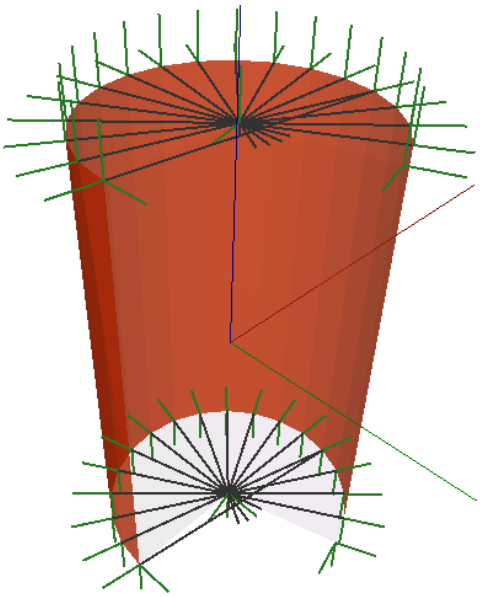
expanded to (x, y, z) coordinates:

$$d_{1321} + mu_a d_{2121} - mu_b d_{4321} = 0 \tag{3.31a}$$
$$d_{1343} + mu_a d_{4321} - mu_b d_{4343} = 0 \tag{3.31b}$$

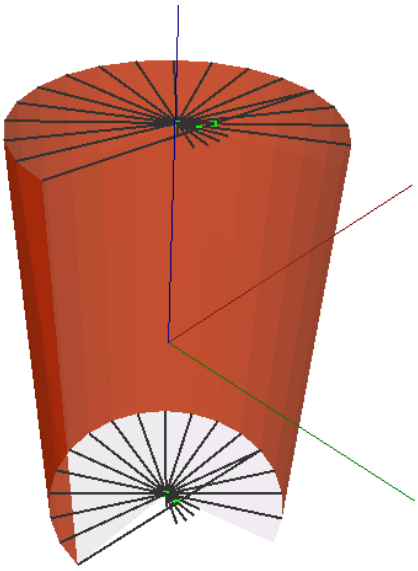
with $d_{mnop} = (x_m - x_n)(x_o - x_p) + (y_m - y_n)(y_o - y_p) + (z_m - z_n)(z_o - z_p)$. 3.31 solving for mu_a and mu_b gives the final result:

$$mu_a = (d_{1343}d_{4321} - d_{1321}d_{4343})/(d_{2121}d_{4343} - d_{4321}d_{4321}) \tag{3.32a}$$
$$mu_b = (d_{1343} + mu_a d_{4321})/d_{4343} \tag{3.32b}$$

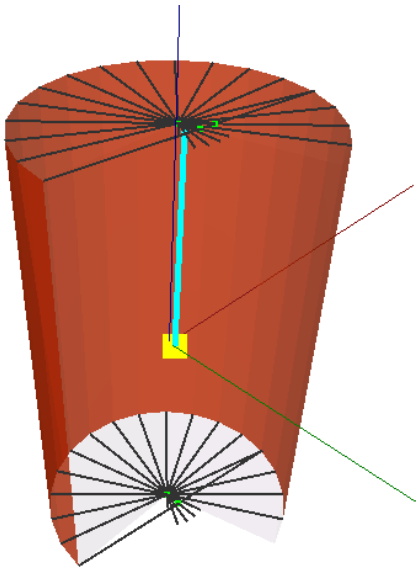
With mu_a and mu_b we can determine the two points P_a and P_b which are the needed intersection points (see Figure 3.10(b)). This is repeated for each combination of previously defined lines to get all intersection points. If no intersection points are found (if all lines are parallel) we use the endpoints of each line as intersection points.
Through these points we now try to best fit a line, which is our generating line, by least squares fitting and SVD. We have already discussed in Section 3.3.3.1 fitting a plane into 3d points. We know that SVD $M = UWV^T$ has an orthogonal base in V. For fitting a plane we took the column of V with biggest singular value. Due to the orthogonal basis the direction vector of best fit line (our generating axis) is the column in V with smallest singular value. This time we don't use weights because intersection points of smaller triangles should be the same as intersection points of bigger triangles. The resulting line already set to the correct length (see following) is shown in Figure 3.10(c).
On very short cones it may happen that the best fitting line through the intersection points is perpendicular to the correct generating line. Therefore we additionally take the vector with biggest singular value and calculate for both directions the variance of the perpendicular distance between each vertex to the generating line (which is our radius) and finally take the direction with smallest variance.



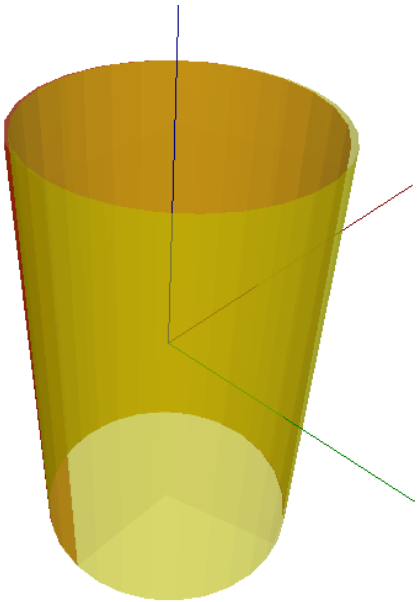
(a) Lines for intersection points (dark grey) are in opposite direction of vertex normals (dark green)



(b) Intersection points (green) near generating line



(c) generating line (turquoise) and centroid (yellow)



(d) Fitted cylinder (yellow)

Figure 3.10: *Part of a cylinder where a pie is cut out, describing fitting of cone annotation.*

Getting radius and height The radius r from a vertex v to the generating axis D is the perpendicular distance from this axis to the vertex. Additionally we need this point P_p projected on the generating axis for calculating the total height of the cone:

$$dot = (v - c) \cdot D \tag{3.33}$$
$$P_p = c + dot\ D \tag{3.34}$$
$$r = ||P_p - v|| \tag{3.35}$$

Then we use following Algorithm 3 for weighted radius and height calculation. Weight w_v is the area of triangle i divided by 3 to distribute the area to each vertex v. The bottom height may be different to top height because the centroid may be shifted up or downwards in direction of generating axis for example on the cup model in previous sections. This is because the cup model contains a lot more points on top. This error will be corrected by taking arithmetic average of bottom and top height and shifting the centroid up or downwards on the generating line.

The weighted radius and height explains why on Figure 3.11 the cone (yellow) hasn't the full length of selected red part. This is a desired behaviour to avoid that small faces with different direction have too much impact on the resulting cone direction and its radius.

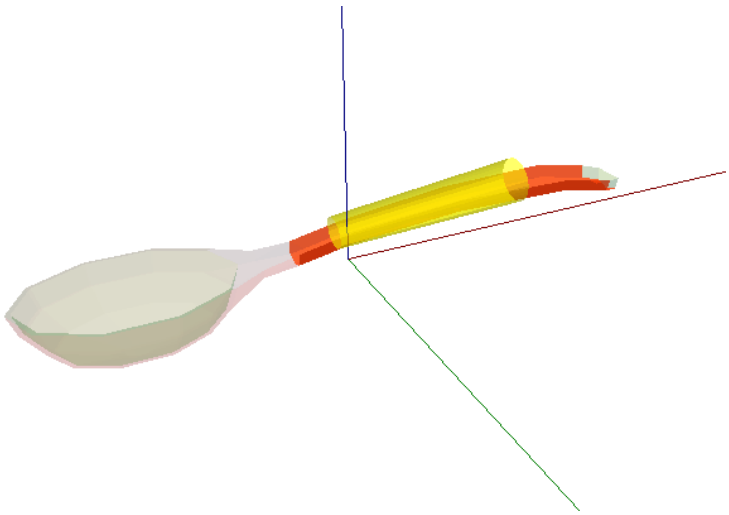


Figure 3.11: *Spoon (same as in Figure 3.8(a)) where handle is fitted by a cone.*

Algorithm 3: Calculate top and bottom radius and height

radiusTop ← 0
radiusTopWeight ← 0
heightTop ← 0
radiusBottom ← 0
radiusBottomWeight ← 0
heightBottom ← 0
for all vertices in cone annotation **do**
  Compute *dot* and r for vertex v
  if dot < 0 **then**
    {vertex is bottom}
    radiusBottom ← radiusBottom + $r\ w_v$
    radiusBottomWeight ← radiusBottomWeight + w_v
    heightBottom ← heightBottom + $|dot|\ w_v$
  else
    {vertex is top}
    radiusTop ← radiusTop + $r\ w_v$
    radiusTopWeight ← radiusTopWeight + w_v
    heightTop ← heightTop + $|dot|\ w_v$
  end if
end for
heightBottom ← heightBottom / radiusBottomWeight
radiusBottom ← radiusBottom / radiusBottomWeight
heightTop ← heightTop / radiusTopWeight
radiusTop ← radiusTop / radiusTopWeight
c ← $c + D\frac{heightTop - heightBottom}{2}$ {fix position of centroid}
D ← $D\frac{heightTop + heightBottom}{2}$ {Set length of generating axis to average height}

4 Interpretation

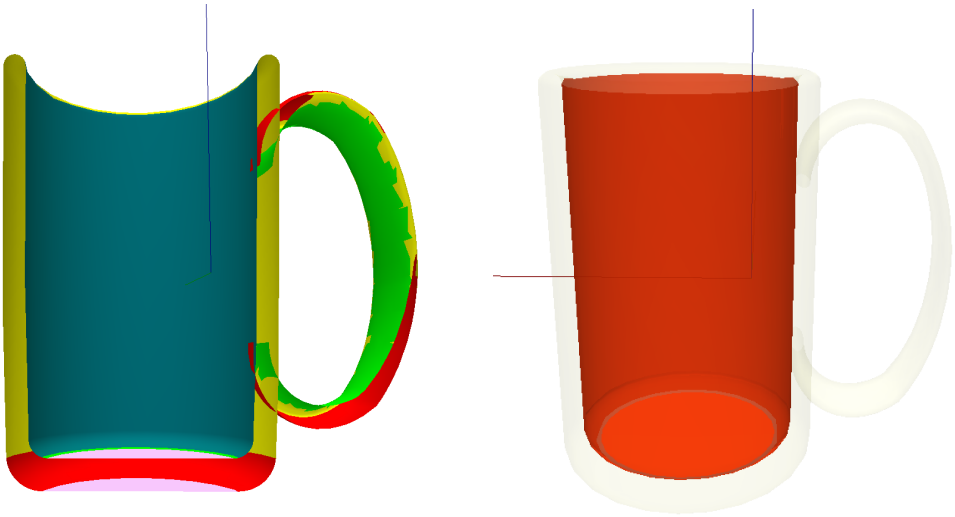
The segmented mesh and primitives alone don't provide yet much useful information for the robot. Therefore we have to interpret the meaning of primitives for the model so that the robot gets additional information for manipulating objects. In this work we use two different interpretation methods: directly in Java or with Prolog. In Java we do more complex reasoning where heavy interaction with underlying data structure is needed. There is also the possibility to start the Java application with JPL Prolog Interface (delivered with SWI Prolog) which allows calling Java methods from Prolog or vice versa. With this interface it is also possible to do simple (or even complex) interpretation directly in Prolog.
In the following sections we present examples for interpreting in Java and Prolog. For additional interpreting possibilities please refer to Section 7.

4.1 Java

In Java we implemented simple container detection. A container is a construct where some kind of wall exists and a bottom plate closes this construct. In our implementation we only consider cones, where a plane surface is at the bottom or top of the cone, which is the case for cups or buckets, as containers. Additional container types are described in Section 7.
So first we need all primitive annotations which are cone annotations and only those cones which are concave. Because containers can only be on the inside of an object and therefore the cone has to be concave.
The next step is to check for each cone found, if it has a top and/or a bottom cap. A cap is defined as a PlaneAnnotation which lies perpendicular to the generating line of the cylinder and has approximately the area to cover the end of the cone. Additionally the position of the plane must be near the end of the cylinder to avoid wrong detections.
So we iterate over each PlaneAnnotation and check if the generating line intersects with the plane. If it does we know that the plane lies at least somewhere in the direction of the generating line. Next we check if the angle between generating line and plane is approximately 90 degrees. If this matches, we already have the

intersection point from intersection checking which is the point where the generating line intersects with the plane. This intersection point is then used to determine the distance between the end of the cone and the plane by subtracting the cone centroid which results in a vector where the length is exactly the distance from centroid to plane. We defined a plane as a cap if it lies between 70% and 130% of cone height. If the distance is in this predefined range we know that the plane is a cap.
The last step is to determine if the cap is on bottom or top of the cone. This can be achieved by using the dot product of the distance vector between centroid and intersection point and the direction vector of the cylinder. If the result is negative, the plane forms a bottom cap, otherwise a top cap.
But a cone may have a bottom and a top cap. If this happens it isn't a container anymore because then it is a solid object. Therefore we have to continue iteration over all planes to check if the other side of the cone is open or covered by another plane.
The resulting container for the cup is shown in Figure 4.1(b). In this case the volume of the container equals the volume of the cone. With this knowledge the robot can for example find out how much fluid fits in the cup or if it knows that there is already fluid in the cup it knows the maximum amount of how much it may be.
Another use case could be the following: In a room there are two buckets and a sink with water. The task for the robot is to get a bucket with 4 litres of water. By fitting CAD models to the buckets and analysing them, the robot finds out that one bucket can hold 3 litres, the other one 5 litres. Assuming the robot has the appropriate knowledge, it can solve the problem with following steps:

1. fill 5L bucket
2. fill 3L bucket with water from 5L bucket. So in 5L bucket remain 2L
3. pour the 3L away
4. fill the remaining 2L from 5L bucket to 3L Bucket.
5. fill 5L bucket again
6. fill 3L bucket until it is full which means 1L from 5L bucket to 3L bucket.
7. now the 5L bucket contains 4L of water



(a) Lateral cut of a cup. The container is a combination of concave cone (blue) and bottom plate (purple).

(b) The resulting container in the cup

Figure 4.1: *On a cup the container is composed of a concave cylinder and a plane.*

4.2 Prolog

We use the rosprolog [1] package for ROS which is a wrapper around SWI Prolog and has the advantage to automatically load Prolog initialization scripts from ROS packages. This package in combination with rosjava_jni [2] allows us to execute our Java application in ROS environment and additionally provides the full power of Prolog for reasoning.
In Prolog we implemented some basic operators for executing the application with and without GUI and analysing models directly by file path or URL (`mesh_reasoning_path`) or by an identifier (`mesh_reasoning`) from KnowRob knowledge database (see also Section 5). Additionally there are operators for getting a list of specific annotation types, for example a list of all plane annotations. There is also the possibility to highlight one or more annotations (`mesh_reasoning_highlight`) in the GUI from within Prolog (see Figure 4.2) and of course it is possible to clear all highlighted annotations.

[1] http://www.ros.org/wiki/rosprolog
[2] http://www.ros.org/wiki/rosjava_jni

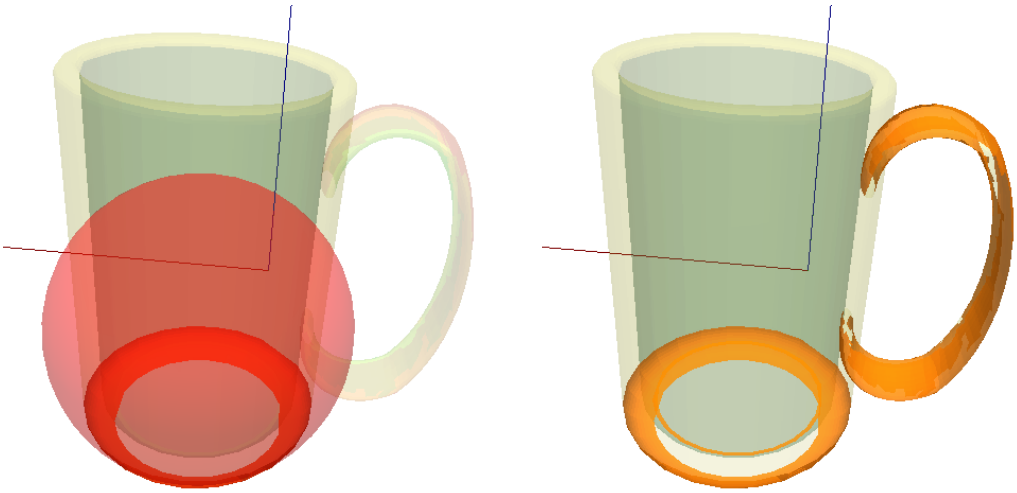


Figure 4.2: *Left side: single highlighted sphere annotation from within Prolog. Right side: all sphere annotations highlighted from within Prolog.*

4.2.1 Supporting planes

One of the reasoning tasks we implemented in Prolog is searching all supporting planes. A supporting plane is a flat or nearly flat surface whose normal vector is approximately in the opposite direction of earth's gravitational force which is our z-axis. For calculating the angle between normal vector and z-axis we take the absolute value of arccosine of z component of the normal vector: $angle = |arccos(n.z)|$. If *angle* is in the tolerance of 10 degrees the plane is a supporting plane.
Here you also have to consider the current pose of an object if it is available. Storing and getting the current pose of an object is implemented in KnowRob. So for additionally considering the pose of an object to determine if a plane is a supporting plane we check if a pose is available. If yes, then each plane normal is rotated by the pose rotation matrix before calculating its angles.
The supporting planes allow a robot to search for a place where it can put down an object with a specific size.
With commands shown in Listing 4.1 and 4.2 a model is parsed and all supporting planes are highlighted as shown in Figure 4.3.

```
roscd knowrob_mesh_reasoning/
rosrun rosprolog rosprolog knowrob_mesh_reasoning
```

Listing 4.1: *Starting java application* `knowrob_mesh_reasoning` *from command line*

```
?- mesh_reasoning_path('/path/to/model.kmz',Mr),
   mesh_find_annotations(Mr,'Plane',Ann).
```

```
Mr = @'J#0000000000046510392',
Ann = @'J#0000140426307191384'.

?- jpl_set_element($Ann,P), mesh_is_supporting_plane(P),
    mesh_reasoning_highlight($Mr,P).
P = @'J#0000000000047006640' ;
P = @'J#0000140425700603912' ;
P = @'J#0000140425700603832' ;
P = @'J#0000140426307191480' ;
P = @'J#0000140426307191464' ;
P = @'J#0000140426036122152' ;
P = @'J#0000140426036122120' ;
P = @'J#0000140426036122072' ;
P = @'J#0000140426036122056' ;
P = @'J#0000140426036122024' ;
P = @'J#0000140426036121976' ;
false.
?-
```

Listing 4.2: *Searching all supporting plane annotations and highlighting them.*

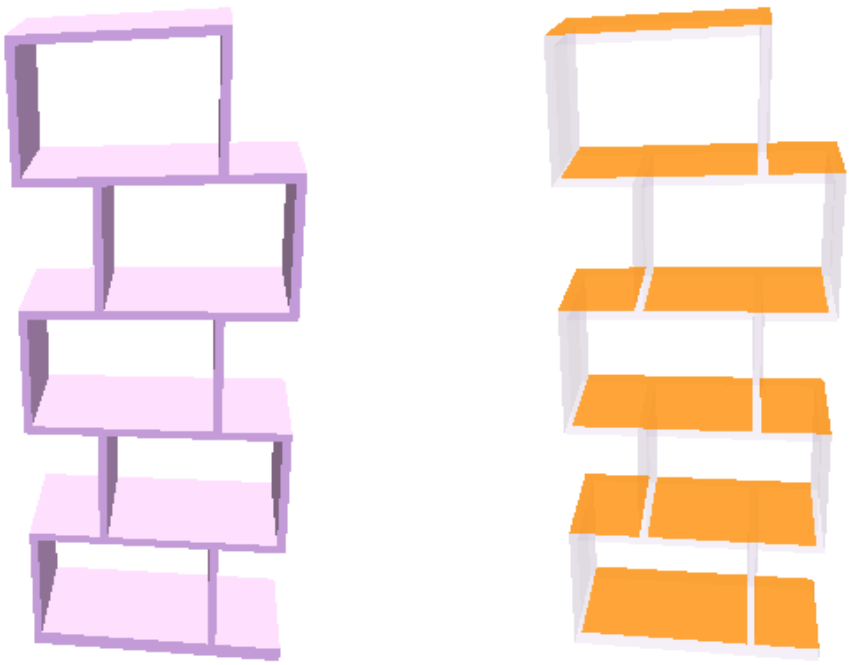


Figure 4.3: *A shelf and all supporting planes highlighted by Prolog interface.*

4.2.2 Handle

Another reasoning task implemented in Prolog is getting the handle(s) of an object (`mesh_find_handle`). A handle is a part of the object where it should most likely be carried. Currently we consider a handle as a cone and most times the handle is the longest cone (or cylinder) of an object. So we try to find the longest cone of the object. To do this, we first create a list with all cone annotations. This list is then sorted by the probability for each cone that it is a handle. This probability is calculated in our case by using area coverage, cone height and optionally minimum and maximum radius.
Area coverage is a value bigger than 0 and indicates how exactly the fitted primitive covers the area of all containing triangles. So if area coverage is 1 (which would be an optimal fitted cone), the fitted primitive has exactly the same surface area as all including triangles together, or if the area coverage is 0.5 it means that the triangles form a lateral cut cone. We defined a threshold of 0.6: if area coverage is below this threshold it is more likely that it isn't a handle. Additionally it is more likely that longer cones are handles.
Minimum and maximum radius can be used to define the radius grippable for the hand to avoid cones with radius bigger than the hand can grip declared as handles.
The compare function is shown in Algorithm 4. As a future improvement a function of all the variables (coverage, height, minimum and maximum radius) could be used to improve the order for example to avoid hard boundaries for minum and maximum radius.
Two examples for found handles are shown in Figure 4.4.
The item at first position of the list is then our handle. We use this approach of a sorted list to allow a higher decision maker to choose also another handle if for example the first one failed to grab or isn't reachable.

Algorithm 4: Comparison of two handles for their probability. $W1$ and $W2$ are the two annotations to compare, *Comp* is the result for calling method to reorder the items

```
Cov1 ← AreaCoverage(W1)
Cov2 ← AreaCoverage(W2)
Rad1 ← AreaRadius(W1)
Rad2 ← AreaRadius(W2)
if MinRad and MaxRad are available then
  Rad1Ok ← Rad1 > MinRad and Rad1 < MaxRad
  Rad2Ok ← Rad2 > MinRad and Rad2 < MaxRad
end if
if Cov1 < 0.6 and Cov2 ≥ 0.6 then
  Comp ← >
else if Cov1 ≥ 0.6 and Cov2 < 0.6 then
  Comp ← <
else if ¬ Rad1Ok and Rad2Ok then
  Comp ← >
else if Rad1Ok and ¬ Rad2Ok then
  Comp ← <
else
  H1 ← Height(W1)
  H2 ← Height(W2)
  if H1 < H2 then
    Comp ← >
  else
    Comp ← <
  end if
end if
```

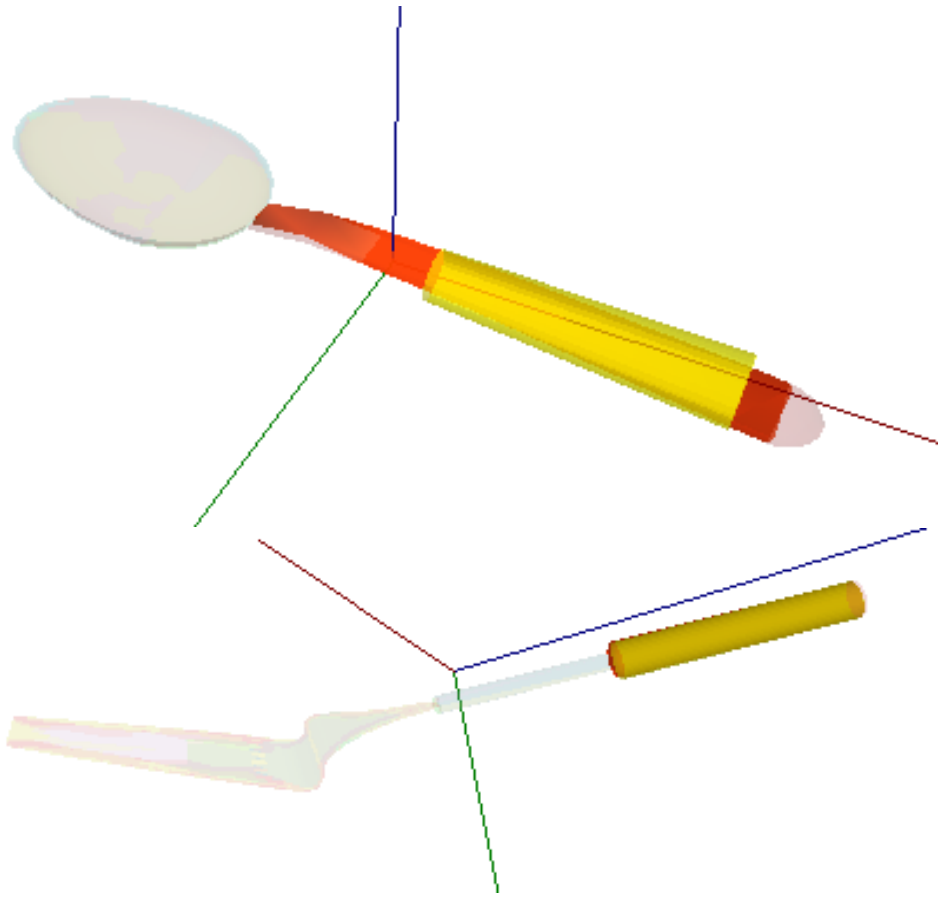


Figure 4.4: *Top: Handle found on a spoon, Bottom: Handle found on a spatula*

5 Integrating into Knowlege-Base

How the semantic knowledge on CAD models can be integrated into knowledge bases has already been addressed a bit in Section 4.2. In this chapter we will expand on this integration. We focus primarily on KnowRob, but the basic ideas should be easy transferable to other knowledge bases.

5.1 KnowRob integration

We used in our KnowRob knowledge base a combination of ROS, Prolog and Java to provide reasoning tasks to the robot. The basic structure of ROS is divided into executable nodes, which provide messages to a specific topic. Another node can subscribe to such topic and receive all messages. With this concept one or more robots can communicate with a node at a totally different location.
Our application can either be executed directly as a Java application (where Prolog predicates wouldn't be available) or as a ROS node in combination with Prolog (rosprolog package). We chose rosprolog because KnowRob already uses rosprolog and so we can guarantee a flawless integration into the existing knowledge base. To execute Java from within rosprolog we use the JPL library delivered with SWI Prolog. With this combination we can start our Java application from Prolog and call all available method members of a Java class: for example start parsing and segmentation and getting all cones or spheres for further reasoning in Prolog.

The knowledge base in KnowRob is mainly stored as OWL structured files where different classes for object types exist and each object instance and class has a unique identifier. We have added the possibility to assign to each class or instance a CAD model by the property `pathToCadModel`. As in Section 7.1 described a future improvement could be the integration of 3D Net into the knowledge base. 3D Net provides the possibility to find the best matching CAD Model out of a database of different models to an object scanned with 3D range scanners. So if 3D-Net finds a model it could set the pathToCadModel property and it would make no difference for our workflow.

So if a robot needs information about a CAD model it starts our reasoning application with either the class identifier or instance identifier. With this identifier we can automatically determine the path to the CAD model and start the segmentation and semantic interpretation task. After this task is finished the robot or another reasoning task can access this information by the Prolog interface or execute additional reasoning predicates as described in Section 4.2.

6 Evaluation

In this chapter we show some example segmentations and semantic interpretations on different types of CAD Models from different databases, most of them are from 3D-Net or Google 3D Warehouse. At the end we discuss the results and show where the algorithm may have problems on segmentation and interpretation.
The evaluation is divided in multiple sections for each group of models.

Here is a description for different colours in the evaluation:

Purple Plane annotation

Yellow Cone or cylinder, convex

Blue Cone or cylinder, concave

Red Sphere, convex

Green Sphere, concave

Dark green Container annotation

Orange line Bounding box

Smoothed colours Colour by curvature

Please note that there is a small problem with vertex normals in inverse direction and therefore convex and concave is sometimes flipped (see Section 6.2.1).

6.1 Examples

6.1.1 Bottles

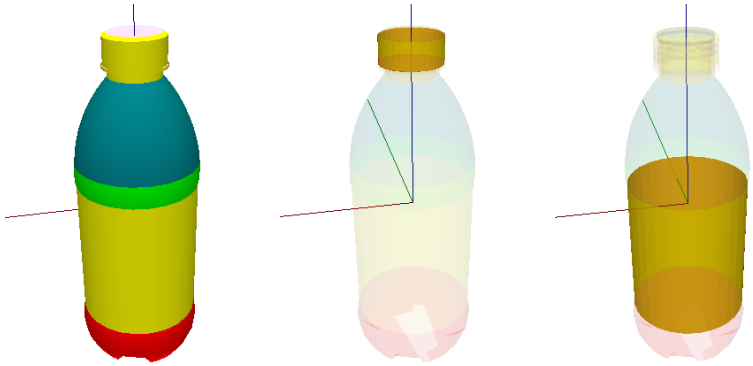


Figure 6.1: *A plastic bottle segmented, the cap as cone annotation, the handle selected by Prolog*

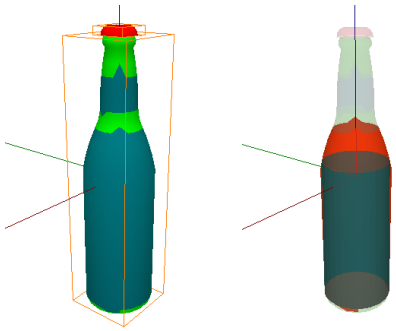


Figure 6.2: *Beer bottle segmented with bounding boxes, the handle selected with Prolog. Here the vertex normals are inverted. As you may see, the bounding boxes of both groups provide an additional segmentation into bottle and cap.*

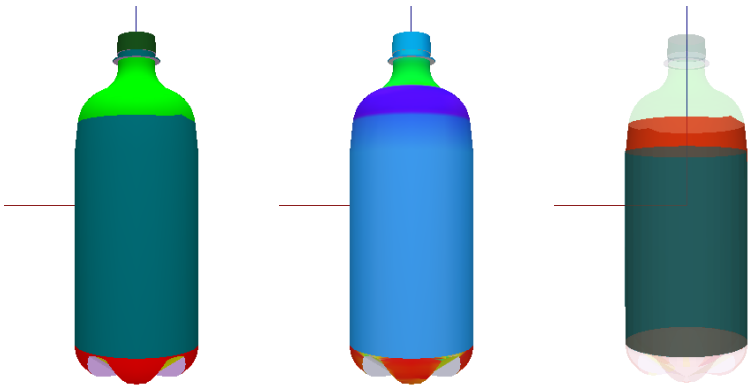


Figure 6.3: *Another plastic bottle segmented, curvature colours, the handle selected by Prolog*

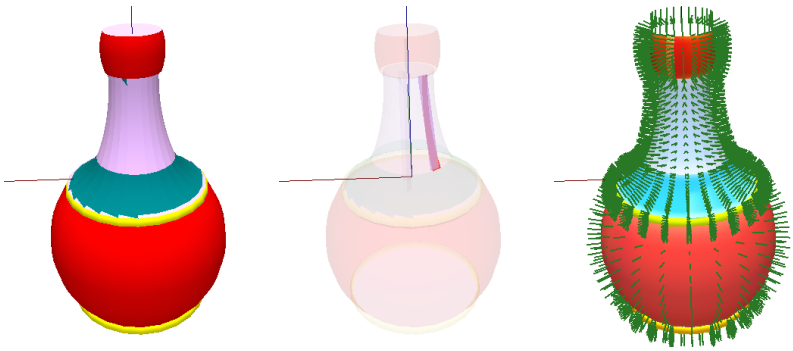


Figure 6.4: *A sphere shaped bottle segmented. As you can see in the second image, the bottle neck is segmented into planes instead of a cone. This is because at the bottle neck the vertices aren't shared (see 6.2.2)*

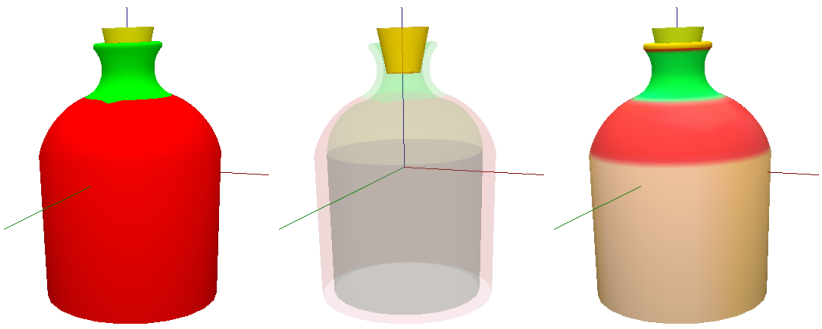


Figure 6.5: *Some kind of cognac bottle segmented. As you can see in the third image, the curvature of the body is more red than yellow and therefore recognized as sphere instead of a cylinder. An improvement to the algorithm could be to refine the margin between cone and sphere.*

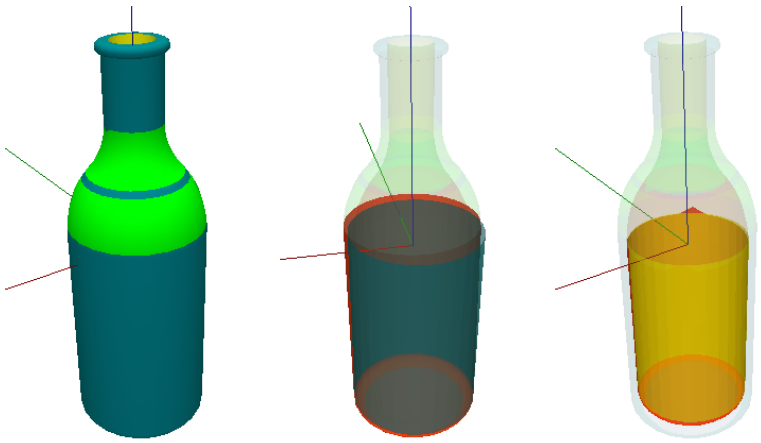


Figure 6.6: *A wine bottle with inner and outer side segmented. The second images shows the handle found by Prolog, the third image shows the inner cone of the bottle. (normal vectors inverted)*

6.1.2 Bowls

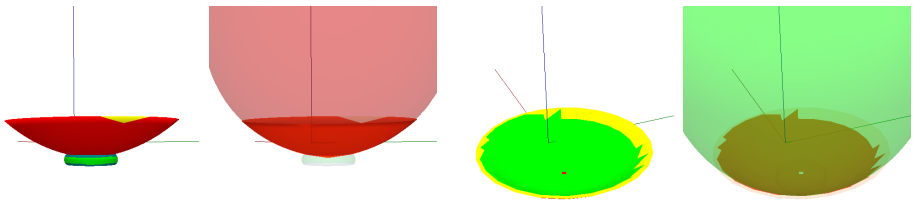


Figure 6.7: *A bowl with a small socket. In the second and fourth image you see the fitted spheres at outer and inner side.*

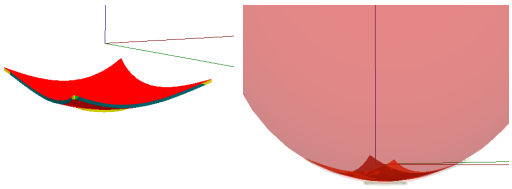


Figure 6.8: *Bowl in shape of a square.*

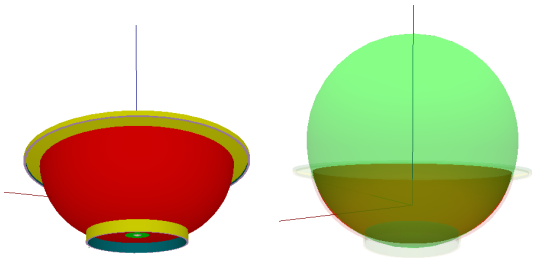


Figure 6.9: *Bowl with frame and fitted sphere inside.*

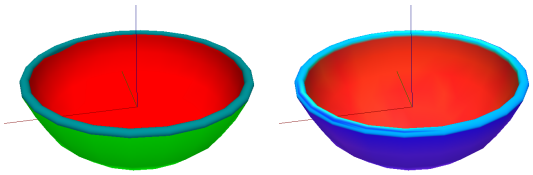


Figure 6.10: *Bowl with rounded edge. Left side: segmented, right side: curvature colours. (Normal vectors inverted)*

6.1.3 Cups and Glasses

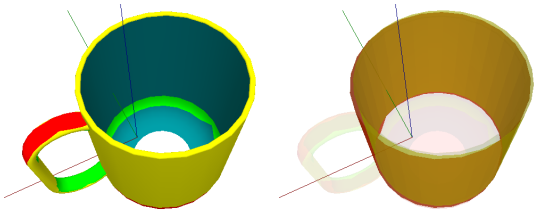


Figure 6.11: *Cup with rounded edges. Here not all containers are found because the bottom plate is too small. On the right side the fitted cylinder is shown.*

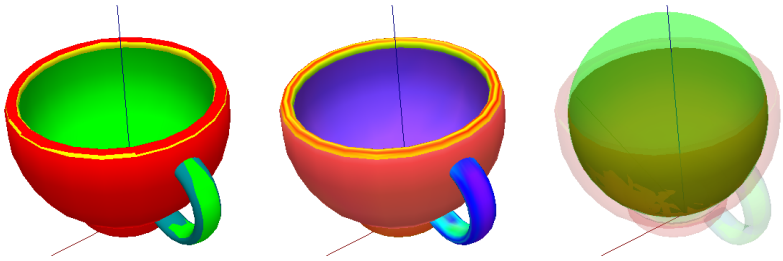


Figure 6.12: *Cup in sphere form. Middle: coloured by curvature, right side: sphere fit inside*

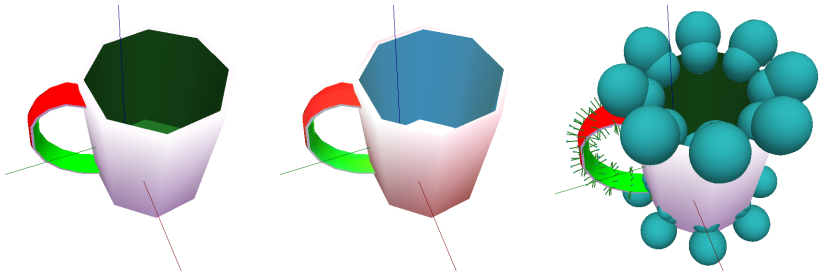


Figure 6.13: *Container found inside the cup. The outside isn't recognized as cone because the curvature indicates that there are planes (middle image, white colour). Last image shows the Voronoi Area for each vertex normal by the size of the sphere.*

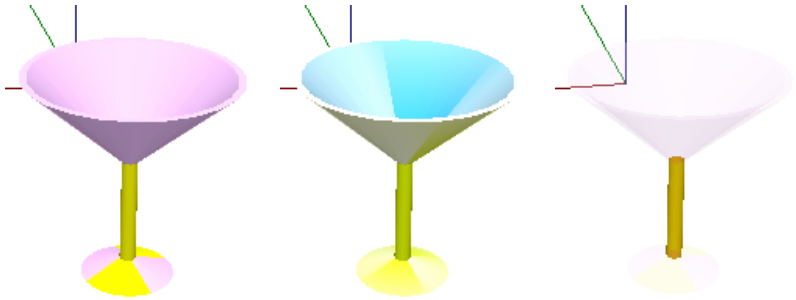


Figure 6.14: *It seems that the cone shape of this glass is too flat to recognize it as a cone. At least the curvature colour is very bright. But the most important part, the handle is found (last image).*

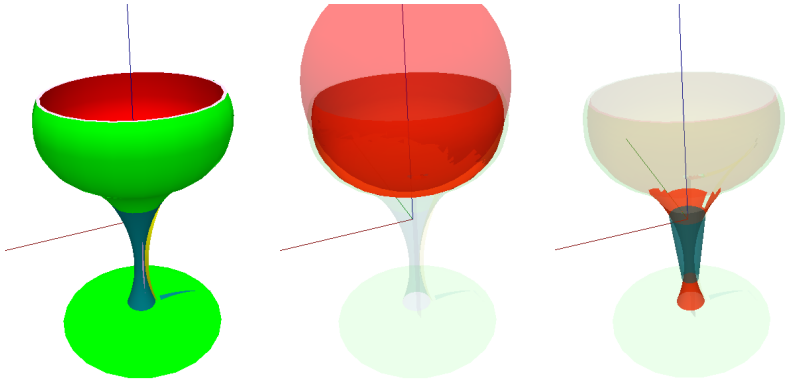


Figure 6.15: *Glass in shape of a sphere. Here again the vertex normals are inverted, but the handle is found anyway (last image).*

6.1.4 Kitchen utensils

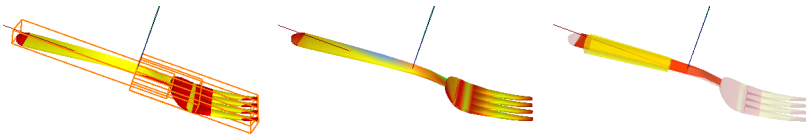


Figure 6.16: *Fork segmented into cones and spheres with bounding boxes for each group. The creator of this model already segmented the handle form the body. Second image is coloured by curvature. Last image shows the handle by Prolog.*

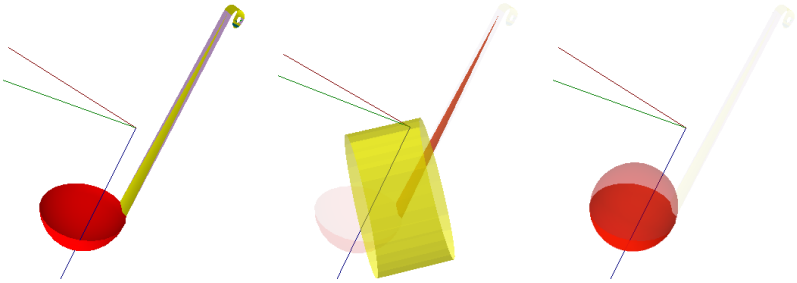


Figure 6.17: *The fit cone for this dipper isn't very accurate. This is due the fact that the reference for the cone is only one triangle.*

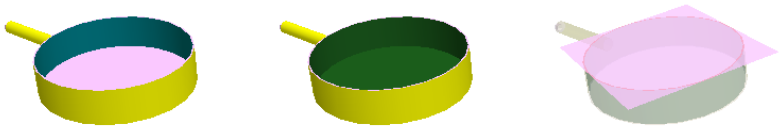


Figure 6.18: *A cooking pot segmented into primitives. The second image shows the container marked as dark green. The last image shows the fit plane for upper edge.*

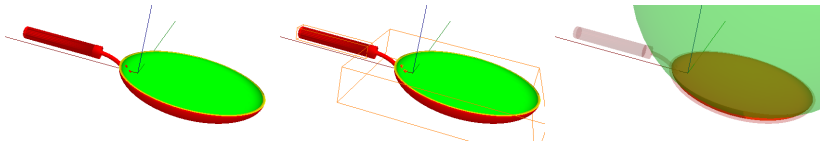


Figure 6.19: *A sphere shaped pan with bounding boxes and fit sphere. Here the handle isn't recognized as a cone because there is too much noise due to the detailed structure.*

6.1.5 Tools

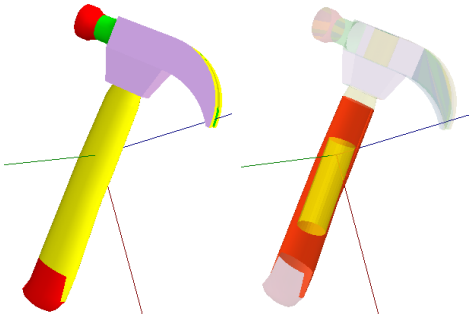


Figure 6.20: *A hammer segmented and handle selected by Prolog*

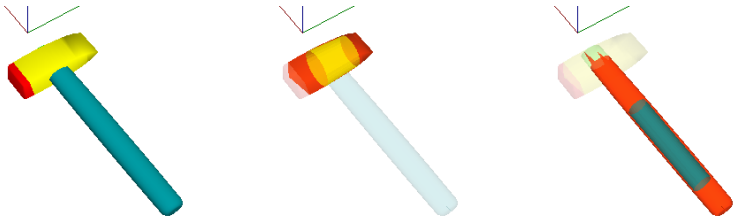


Figure 6.21: *Another hammer segmented. Here the vertex normals on the handle are inversed. But the handle is found anyway.*

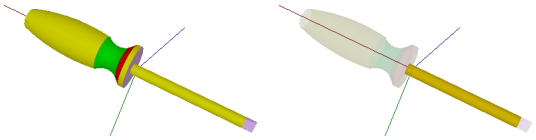


Figure 6.22: *With this screwdriver, finding the handle didn't work because the shank is longer than the real handle.*

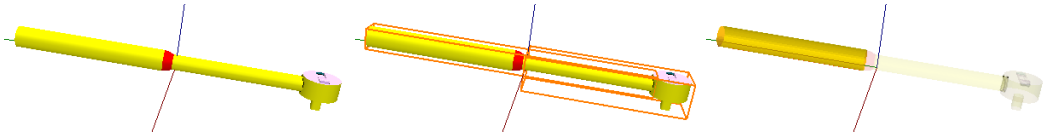


Figure 6.23: *The bounding boxes already segment this socket wrench into handle and body. Here finding the Handle with Prolog also worked (last image).*

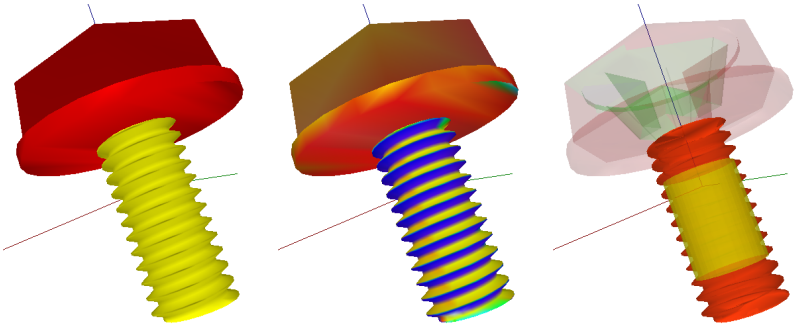


Figure 6.24: *M3 screw with screw thread modelled on the shank. The second image shows curvature colours, the third shows the cone on the shank.*

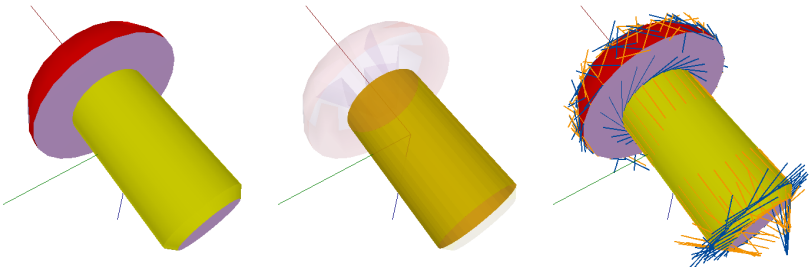


Figure 6.25: *Another screw without screw thread. The last image shows the principle directions of the curvature. Blue is Maximum curvature, orange Minimum curvature.*

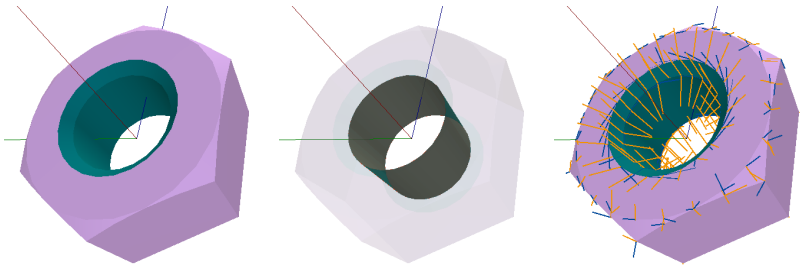


Figure 6.26: *A simple screw nut. Second image shows the fit cone, the last one the principle directions of the curvature. Blue is Maximum curvature, orange Minimum curvature.*

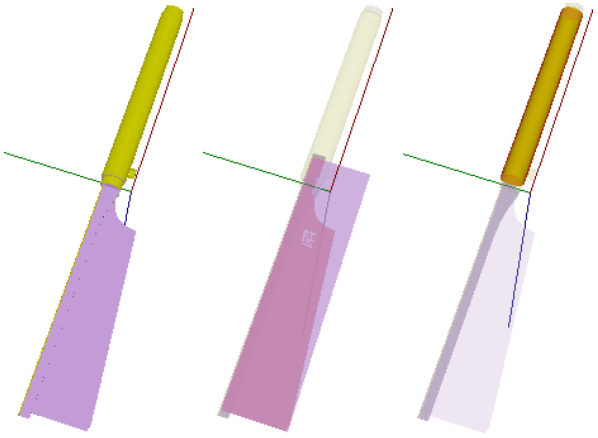


Figure 6.27: *A Kataba (Japanese wood saw) with selected handle and the fitted plane on the cutting face.*

6.1.6 Furniture

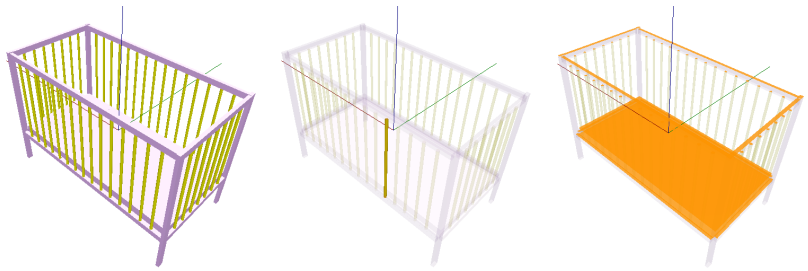


Figure 6.28: *A baby bed with a lot of cones and planes. The second image shows a fit cone. The last image all supporting planes from Prolog.*

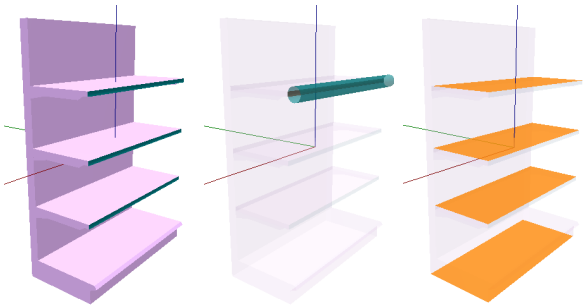


Figure 6.29: *A book shelf with rounded front faces. The last image shows all supporting planes by Prolog.*

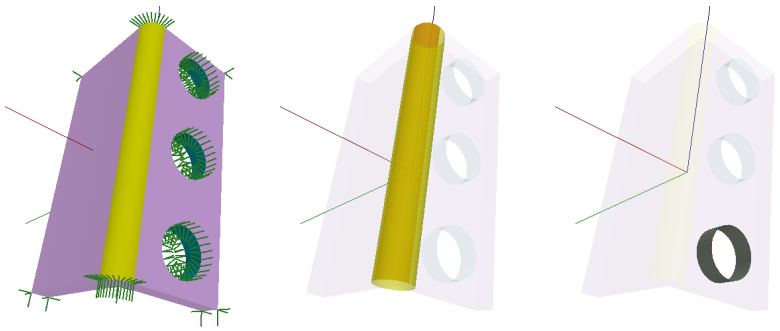


Figure 6.30: *A simple profiled metal with holes and rounded edge. The first image additionally shows vertex normals.*

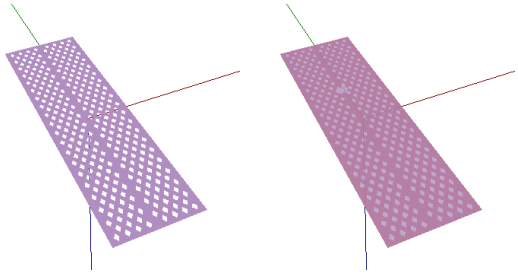


Figure 6.31: *A plate with holes. In the second image you can see the fit plane for the plate.*

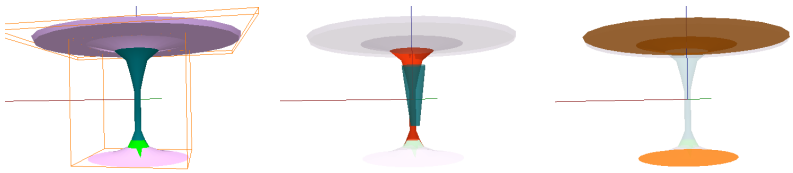


Figure 6.32: *A rounded table with bounding boxes for each group defined by the designer. The second image shows the handle detected by Prolog, the last one all supporting planes which are on the wrong side because vertex normals are inverted.*

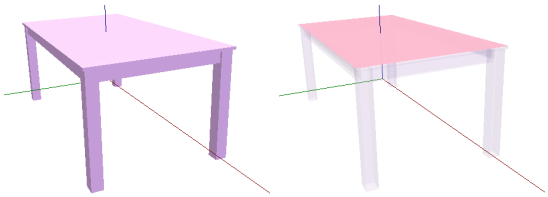


Figure 6.33: *A simple table with only plane annotations. The last image shows the single found supporting plane.*

6.2 Discussion

As it can be seen in previous section, some of the models are nearly perfect segmented and interpreted, and others have some small or even bigger errors. This depends heavily on the complexity of each model and the detail of modelling. The biggest problem is to determine the correct vertex normal vectors and out of them the correct curvature. In Figure 6.34 you can see some examples for failed segmentation. The first image shows a cup where the heart isn't drawn with texture, instead the modeller used triangles to draw it. Additionally this cup isn't very smooth and therefore our algorithm didn't find any cone for the body of the cup. The second image, the chair, is another model where our algorithm had difficulties to find all of the correct parts. Additionally here is the problem of inverted vertex normals (see Section 6.2.1).
Currently there is also sometimes a problem with finding the best fit cone for a cone annotation. It happens sometimes that the cone is in the perpendicular direction if the cone annotation has a very unregularly shape or a lot of small triangles with some noise.
But on most of the models the algorithm delivered for the most interesting parts, the handles and supporting planes, quite accurate results. Even on some complex models such as Figure 6.24 (Screw with screw thread) the algorithm hasn't been disturbed by the screw thread and found a cone on the shank.
In summary it depends on the level of detail and cleanness of how good models are segmented and primitive annotations can be fit into them.

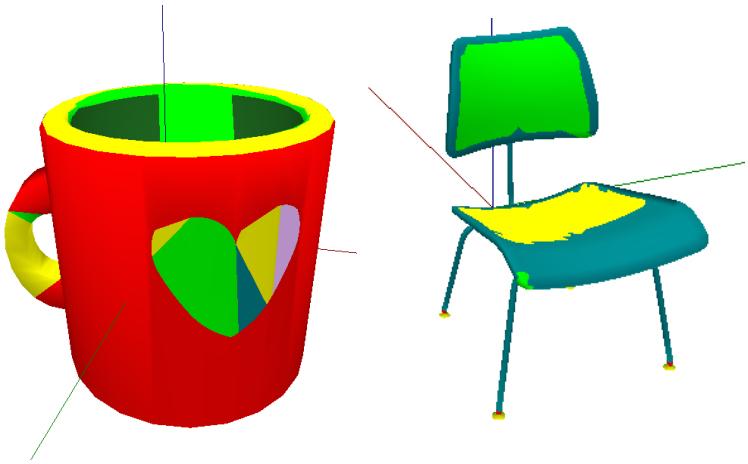


Figure 6.34: *Some examples where curvature calculation failed.*

6.2.1 Vector inverted

If no vertex normals are given for triangulation data, we have to calculate those ourselves (see Section 3.2.1). Here it is difficult to decide in which direction the vertex normal should point, which means to decide which side of the face is visible and which one is hidden. In our algorithm we only calculate the vertex normal vector without taking into account the face direction. Most times this results in the correct vector direction, but sometimes this may also fail as you can see on different previos evaluation examples.

6.2.2 Single vertex or shared?

Normally 3D Data is stored in the following format: first you define the number of vertices and number of faces, then an indexed list of all vertex coordinates is given and then there is the list of faces which reference to vertex indices. Here it is possible to use a compact representation by sharing vertices, for example reference vertex i on all neighbouring faces which have a vertex at exactly the same position, or use a representation, where vertices aren't shared. No shared vertices means, that if you have some triangles with a vertex at the same position each triangle has its own vertex referenced. So you may have in the vertex list duplicate entries of the same position.
The problem here is that if we calculate the vertex normal for each vector and they aren't shared we get wrong results, because the resulting vertex normal isn't an average value of adjacent faces, but is only the direction for the single adjacent face. You can find an example for this problem in Figure 6.35 and Figure 6.36.
It would be possible to programmatically search for identical vertices and combine them in our algorithm, which would rise another problem that you can't distinguish between important parts where planes and cones or legs have the same coordinates but have a different meaning (see Figure 6.37).

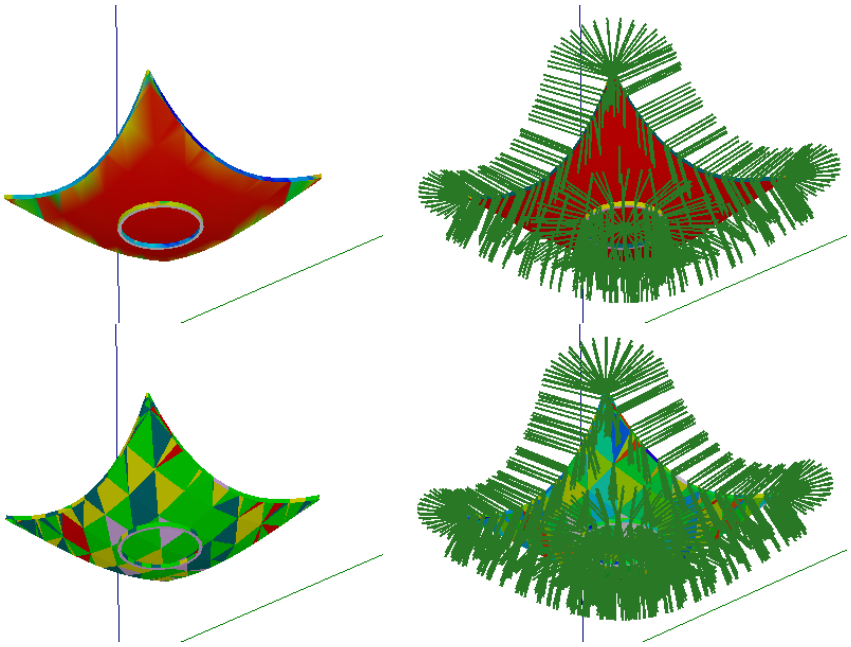


Figure 6.35: *Red bowl: shared vertices. Curvature calculation is correct. Green bowl: no shared vertices, you can see at each vertex more than one green line indicating the vertex normal.*

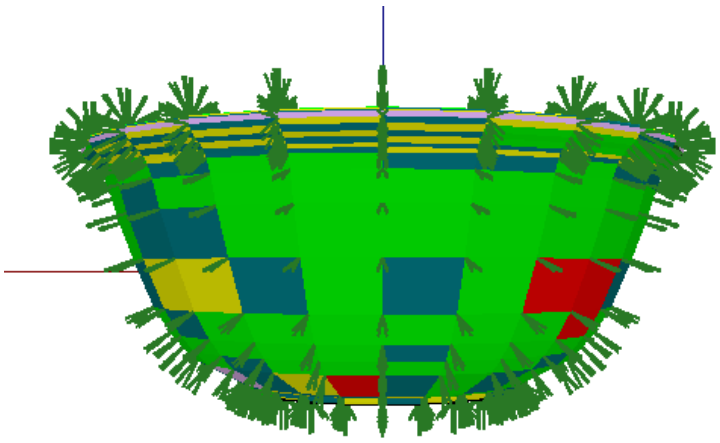


Figure 6.36: *Another example for not shared vertices. The curvature calculation totally fails.*

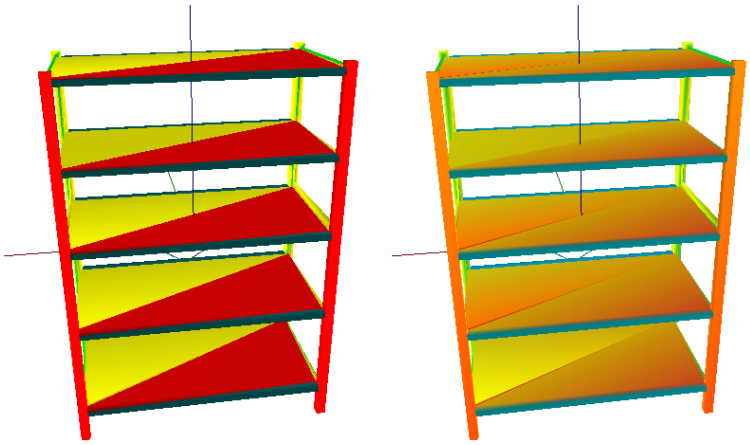


Figure 6.37: *Model where all vertices are shared, even those of different meaning. Here the algorithm sets vertex normals for the shared vertices which impacts also the vertex normals of those triangles which are planes. So in the second image coloured by curvature you can see, that the planes are colored between yellow and red instead of white.*

7 Conclusions

In this work we have shown that using CAD models as an additional knowledge source could be a big advantage for a robot. Instead of trying to infer properties of objects directly in 3D point cloud data, it is more efficient and accurate to search the corresponding CAD model and analyse this instead.
By segmenting these models and fitting primitives we got a base structure for further semantic interpretation. We provided semantic interpretation for some interesting tasks as finding containers, supporting planes or handles. But with these primitives much more and also more complex reasoning is possible.
This new knowledge gives the robot additional capabilities, for example selecting a vessel with appropriate size or inferring, where objects can be grasped optimally and where they can be put down.
Probably everyone remembers his childhood where there was a toy in form of a plate with different shaped holes: round, triangle, star and rectangle. Additionally you got wooden blocks in the same shape and had to stick them through the correct hole. Even this task the robot is now able to solve by analysing the objects.

In the evaluation you can see that the segmentation and fitting primitives is working for different types of objects and also for different model complexity. But there are some special cases where the segmentation and curvature estimation didn't work as expected. Here additional improvement is necessary.

7.1 Future Improvements

Here we provide some thoughts about future improvements for our application, which would make our application better structured or adding new features or improves the segmentation process.
The most important feature is the integration of 3D-Net to automate CAD model search. 3D-Net is used to find the correct CAD model for an object in 3D point cloud data. Without this feature the robot wouldn't be able to know which model

it has to analyse.

To make the application easier expandable, a complete migration to the Apache UIMA project could be an option. We structured our application by some concepts of UIMA, but aren't currently using it directly. The complete migration would also allow simple multi cluster analysis or simply considering different information sources such as annotations by human and CAD models and combine them into a final result.
Manually created annotations by human on CAD models could be another useful extension to annotate information on CAD models which can't be gathered by simply analysing the CAD model. For example it may be useful to annotate on the models in Figure 6.25 (screw) and Figure 6.26 (screw nut) that the cone actually isn't flat, but has a screw thread. This would allow the robot to infer that the screw nut has to be rotated around its axis to stick it on the screw. Or if you have a model of a portable boiling plate you can annotate which region of the plate gets hot, or for a plier you could annotate that there is an axis which allows opening and closing it.
All this information can then be integrated in robot motion and action planning to automatically detect subtasks: If the robot has the task description "Put the nut on the screw" and additionally knows that both screw threads should be joined, it would infer that both axes should match and the screw has to be rotated on contact with the nut.

Most CAD modellers do already some kind of segmentation by grouping special parts of an object into subgroups (see some examples in Section 6.1). This information isn't yet used by our algorithm, but should be integrated for even better segmentation. Another unconsciously given segmentation information is the colour or texture of an object. Different colours or texture types normally mark different parts of an object. This could also improve our algorithm.

To avoid repeatedly reanalysing of already analysed models it would be useful to store the segmentation and interpretation result somewhere in the knowledge base. For example the variables, such as centre, radius, height and dimension for all the spheres, cones and planes could be stored to reuse it, when the robot sees the model again.
Another improvement regarding the knowledge base would be, to define with OWL restrictions special properties. For example a spoon could be described by a long cone and a part of a sphere attached to it. With this OWL restrictions the robot has the capability to identify unknown objects or CAD models for example as a spoon. OWL restrictions would also be useful in the opposite direction: if some

tools are placed on a table and the robot needs a spoon, it can analyse all the models and decide that the second object is the needed spoon.

Currently we have for cones and sphere only a percentage value for area coverage. But if the area coverage is for example 25% (only a quarter of a cylinder) we don't store the information where this quarter is exactly located, we only store its radius, height and centroid. The same problem is with spheres. Here an improvement would be to describe with four angles from where to where the area of the fitted cone or sphere is covered, by using the spherical coordinate system [1]. Two angles indicate the minimum and maximum θ angles, the other two minimum and maximum φ angles of the vertices on the sphere. This concept can also be simply mapped to cones, where you use instead of two θ angles the distances between the equator and vertices on both sides of the equator.

To improve container detection, additional feature detection has to be implemented to detect open squared boxes or squared boxes with rounded corner or even containers which are combinations of cones, planes and spheres.

7.2 Challenge

The most challenging part of this work was the segmentation. It took more than 2/3 of the time to get satisfying results for curvature calculation and then fitting the primitives. Therefore the interpretation part is a bit shorter, but with this segmentation we have some useful information for expanding the whole interpretation part in future work.
For human beings it seems very easy to say that the model consists of a cone and a sphere, but machines yet don't have this capability. The trained eye of the human immediately recognizes curves and corners. But machines have to analyse each detail to determine the shape of an object. If there is additionally a detailed structure on the object surface it gets even more complex.
Also the semantic interpretation isn't an easy task. Containers or handles may have a lot of different shapes. For us humans, our experience tells us where to grasp specific objects and which shapes a container may have. But robots don't yet have this knowledge and therefore have to rely on some standardized shapes which aren't flexible enough for all the possibilities.
But I am sure the human being will even solve this problem in near future.

[1] https://en.wikipedia.org/wiki/Spherical_coordinate_system

Bibliography

Agathos, A., Pratikakis, I., Perantonis, S., Sapidis, N., & Azariadis, P. (2007). 3d mesh segmentation methodologies for cad applications. *Computer-Aided Design & Applications*, *4*(6), 827–841.

Aldoma, A., Tombari, F., & Vincze, M. (Eds.). (2012). Supervised learning of hidden and non-hidden 0-order affordances and detection in real scenes.

Attene, M., Falcidieno, B., & Spagnuolo, M. (2006). Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer*, *22*(3), 181–193. doi:10.1007/s00371-006-0375-x

Attene, M., Robbiano, F., Spagnuolo, M., & Falcidieno, B. (2009). Characterization of 3d shape parts for semantic annotation. *Computer-Aided Design*, *41*(10), 756–763. doi:10.1016/j.cad.2009.01.003

Attene, M., Katz, S., Mortara, M., Patané, G., Spagnuolo, M., & Tal, A. (Eds.). (2006). Mesh segmentation – a comparative study.

Bénière, R., Subsol, G., Gesquière, G., Le Breton, F., & Puech, W. (Eds.). (2011). Recovering primitives in 3d cad meshes.

Ciocarlie, M., Hsiao, K., Jones, G., Chitta, S., Rusu, R. B., & Sucan, I. A. (Eds.). (2010). Towards reliable grasping and manipulation in household environments.

Dag, N., Atıl, I., Kalkan, S., & Sahin, E. (Eds.). (2010). Learning affordances for categorizing objects and their properties. Retrieved from http://www.kovan.ceng.metu.edu.tr/~erol/publications/pdf/Dag-ICPR-2010.pdf

Eberly, D. (2008). Least squares fitting of data. Retrieved from http://www.geometrictools.com/Documentation/LeastSquaresFitting.pdf

Garland, M., Willmott, A., & Heckbert, P. S. (Eds.). (2001). Hierarchical face clustering on polygonal surfaces.

Gärtner, B. (Ed.). (1999). Fast and robust smallest enclosing balls, *Berlin [etc.]* Springer. Retrieved from http://www.inf.ethz.ch/personal/gaertner/texts/own_work/esa99_final.pdf

Goldfeather, J., & Interrante, V. (2004). A novel cubic-order algorithm for approximating principal direction vectors. *ACM Transactions on Graphics*, *23*(1), 45–63. doi:10.1145/966131.966134

Katz, S., Leifman, G., & Tal, A. (2005). Mesh segmentation using feature point and core extraction. *The Visual Computer*, *21*(8-10), 649–658. doi:10.1007/s00371-005-0344-9

Lee, S. K., Yoo, K. D., Kim, J. W., & Lee, M. J. (2012). Surface patch primitive based object modeling from cad data. *Applied Mechanics and Materials*, *162*, 179–183. doi:10.4028/www.scientific.net/AMM.162.179

Mehtap, A. (2010). 3d raumdarstellung und rekonstruktion geometrischer primitive mittels eines 2d laserscanners. Retrieved from http://tams-www.informatik.uni-hamburg.de/publications/2010/DA_Arli.pdf

Meyer, M., Desbrun, M., Schröder, P., & Barr, A. H. (2002). Discrete differential-geometry operators for triangulated 2-manifolds. Retrieved from http://www.multires.caltech.edu/pubs/diffGeoOps.pdf

Nieuwenhuisen, M., Stückler, J., Berner, A., Klein, R., & Behnke, S. (Eds.). (May 2012). Shape-primitive based object recognition and grasping. Retrieved from http://www.ais.uni-bonn.de/papers/robotik2012_nieuwenhuisen_grasping.pdf

Quigley, M., Gerkeyy, B., Conleyy, K., Fausty, J., Footey, T., Leibsz, J., ... Ng, A. (2009). Ros: an open-source robot operating system. Retrieved from http://ai.stanford.edu/~ang/papers/icraoss09-ROS.pdf

Rusinkiewicz, S. (2004). Estimating curvatures and their derivatives on triangle meshes. *3DPVT '04 Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, *2nd International Symposium*, 486–493.

Szlivási-Nagy, M. (Ed.). (2006). About curvatures on triangle meshes.

Tenorth, M. (2011). Knowledge processing for autonomous robots. Retrieved from http://ias.cs.tum.edu/~tenorth/thesis.pdf

Tombari, F., Di Stefano, L., & Giardino, S. (Eds.). (2011). Online learning for automatic segmentation of 3d data.

Tombari, F., & Luigi, D. S. (Eds.). (2011). Automatic semantic segmentation of 3d urban scenes.

Xiaopeng Zhang, Hongjun Li, & Zhanglin Cheng (Eds.). (2008). Curvature estimation of 3d point cloud surfaces through the fitting of normal section curvatures.

Zhihong, M., Guo, C., Yanzhao, M., & Lee, K. (2011). Curvature estimation for meshes based on vertex normal triangles. *Computer-Aided Design*, *43*(12), 1561–1566. doi:10.1016/j.cad.2011.06.006