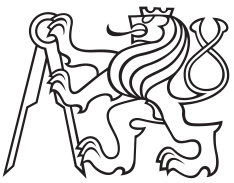
Bachelor's thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Example-based 3D-model synthesis

Adam Papoušek
Software Technologies and Management

May 2014
Supervisor: Ing. Michal Lukáč

Acknowledgement / Declaration

I would like to sincerely thank my supervisor Ing. Michal Lukáč for his guidance and assistance with this work, RNDr. Petr Olšák for providing help with typesetting of the document and my family and friends who always supported me throughout my studies.

I hereby declare that I have completed this work independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on 23/5/2014

.....

Abstrakt / Abstract

Tvorba 3D modelů je pro běžné uživatele obtížnou činností. Proto vznikly nové metody tvořící modely z příkladů, které využívají rostoucího množství modelů v online databázích. Tato práce zkoumá tři metody založené na tomto principu – první využívá geometrické analýzy tvarů (Jain et al. 2012), druhá je založená na prohazování specifických uspořádání částí (Zheng et al. 2013) a třetí modely tvoří pomocí pravděpodobnostního modelu vytvořeného učením z dat (Kalogerakis et al. 2012). Jedna z těchto metod je zvolena pro implementaci, která je hlavní náplní této práce.

Klíčová slova: syntéza modelů, analýza modelů, 3D tvorba založená na datech.

Překlad titulu: Syntéza 3D modelů z příkladů

Shape synthesis is generally a task that is hard to master. New methods using 3D-model synthesis from examples emerged with the growing number of shapes in model repositories. This work reviews and compares three such methods – one using geometry analysis (Jain et al. 2012), second one reshuffling certain part arrangements (Zheng et al. 2013) and third one employing probabilistic model learned from training datasets (Kalogerakis et al. 2012). One of the methods is chosen for implementation, which is the main focus of the work.

Keywords: shape synthesis, shape analysis, data-driven 3D creation.

Contents

Chapter 1
Introduction

Three-dimensional content creation is a challenging, cumbersome task that takes a long time and there are not too many easy-to-use modelling systems that can produce high quality results. Users that need 3D models often resort to collections like Google Warehouse or Turbosquid, but these don't always provide the exact shape the user is looking for.

A common use of 3D models is in virtual reality and video games. In order to create a realistic environment we often need to create many similar but not identical shapes. For example, let's imagine a futuristic millitary video game with fighting robots. If we would like the game to look authentic, it would be necessary to have a wide variety of robot models, possibly tens or hundreads, maybe even more. A standard approach to create that many shapes would be to simply make those with a conventional 3D creation software like Maya or Blender, but such approach requires financial resources that many small game development studios can't afford. It would be better if we could make only a fraction of the required shapes and then we would apply a simple automated process that would synthesize new similar, but still different shapes.

This work introduces automated methods of shape synthesis that would be applicable for such case. The process of shape creation is not an easy challenge. Naïve methods that interchange shape parts easily fail, beacause these methods quickly destroy shape plausibility and function. There are three non-trivial approaches mentioned in this work. One interpolates shapes by combining parts retrieved from objects based on hierarchy analysis [Jain et al. 2012], another is based on identification and reshuffling of certain part arrangements [Zheng et al. 2013] and the last employs a probabilistic model learned from training datasets, which encodes part-based composition. Shape synthesis is then done by sampling such probabilistic models [Kalogerakis et al. 2012].

The goal of this work is to implement one of the methods and describe its process. First in chapter 2 the algorithms are reviewed in further detail. Next, in chapter 3, the overall analysis and design of the proposed application is introduced. Chapter 4 contains detail of the implementation. Testing of the program is described in chapter 5. In the last chapter (6) the results of the work are summarized.

Chapter 2
State of Art

The goal of this chapter is to review articles that describe example-based 3D-model synthesis and choose one system to implement. The articles are these three:

- Exploring Shape Variations by 3D-Model Decomposition and Part-based Recombination by Jain et al. [Jain et al. 2012]
- Smart Variations: Functional Substructures for Part Compatibility by Zheng et al. [Zheng et al. 2013]
- A Probabilistic Model of Component-Based Shape Synthesis by Kalogerakis et al. [Kalogerakis et al. 2012]

The following sections will introduce the algorithms described in each article including their inputs, outputs and what they are best suitable for. The systems described have a common workflow – first the shapes are analysed in an offline stage and then the data is used in an online stage, where new shapes are created. This common trait leads to the possibility of creating a base platform, which would incorporate all three systems in one.

2.1 Exploring Shape Variations by 3D-Model Decomposition and Part-based Recombination

This algorithm takes polygonal meshes as its input. The only preprocessing necessary is the upright orientation of the meshes. The shapes don't need to be from the same category – although the results are better if they are.

The offline phase of the algorithm prepares the shape for the online phase in numerous steps – segmentation, contact analysis, symmetry detection and hierarchy creation. This process is done independently for each shape.

During the online phase a new shape is synthesized from two input ones. First the two shapes' hierarchies are unified. The user then chooses the ratio that describes how much each shape contributes to the final shape. An interpolation between the input shapes based on the ratio between them is then done, which results in a new shape. On this shape a mass-spring system is applied, which enforces contacts between the parts of a shape. The final shape can be then used for further shape synthesis.

The system provides best results for shapes with a great amount of symmetries so it is best suited for man-made shapes. More asymmetric shapes like trees or terrain won't probably yield very useful results.

2.1.1 Shape Analysis

This process is an offline part of the application during which the shapes are processed independently. The input shapes S_i are polygonal meshes. The shapes don't need to be

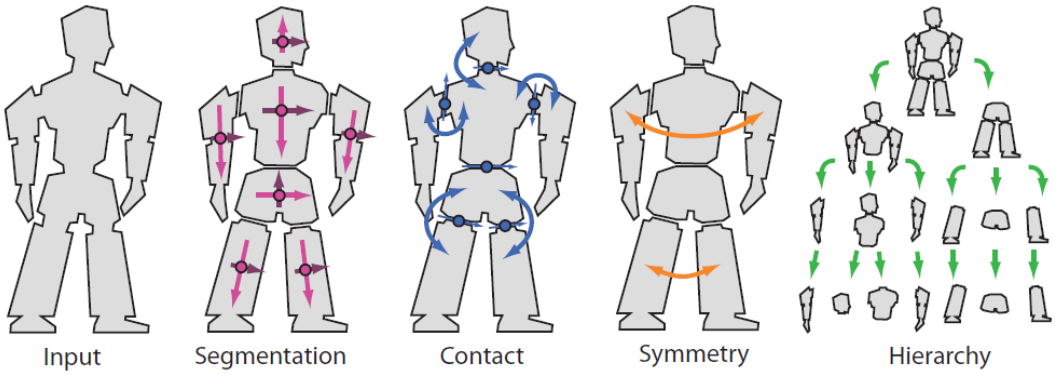


Figure 2.1. The steps of the offline shape analysis. Image taken from [Jain et al. 2012].

segmented, it is only necessary that they have an upright orientation. An illustration of the steps of the analysis can be seen in figure 2.1.

Segmentation. With the use of a region growing algorithm shapes are split into parts $P_{i,j}$, these parts are again polygonal meshes. Every part is then sampled to a point cloud $\bar{P}_{i,j}$, where all the points of each $\bar{P}_{i,j}$ are evenly distributed on the surface so that the distance between them is roughly equal. Principal component analysis (PCA) of $\bar{P}_{i,j}$ is performed, which provides a transformation $T_{i,j}$ from the global into the local coordinate system of the part. A point $\mathbf{p}$ from $\bar{P}_{i,j}$ in the local coordinate system of the part $P_{i,j}$ can be acquired by the formula $\mathbf{p}' = T_{i,j}\mathbf{p} = R_{i,j}S_{i,j}(\mathbf{p} - \mathbf{c_{i,j}})$, where $\mathbf{c_{i,j}}$ is a geometric center of the part, $R_{i,j}$ is a rotation matrix given by the three PCA basis vectors and defines the local rotation axes and scaling matrix $S_{i,j} = diag(1/s_x, 1/s_y, 1/s_z)$ using the three singularvalues $s_{i,j} = (s_x, s_y, s_z)^T$.

Contact analysis. Contacts are found between all parts $P_{i,j}$ using the point clouds $\bar{P}_{i,j}$. A contact $C_{i,j,k}$ is a set of points $\mathbf{p}$ from $\bar{P}_{i,j}$, whose distance between the points from $\bar{P}_{i,k}$ is smaller than 0.1 % of the bounding box diameter of the two parts. The operations so far are visualized in figure 2.2.

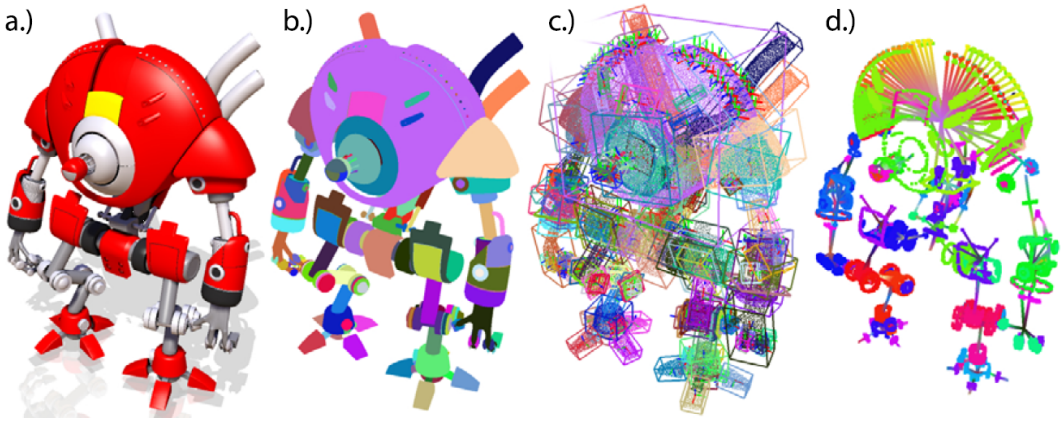


Figure 2.2. a.) shape, b.) segmentation, c.) PCA, d.) contacts. Image taken from [Jain et al. 2012].

Symmetry Detection. In this step the dominant global symmetry transformation H_i is found (a reflection, rotation or translation). In each iteration a transformation candidate K is generated and support $\alpha(K)$ is computed using all parts from of the shape. For each part $P_{i,j}$ its center $\mathbf{c}_{i,j}$ is mapped to $\mathbf{c}'_{i,j} = K\mathbf{c}_{i,j}$ and if a matching part $P_{i,j'}$ is found at $\mathbf{c}'$ the support counter is incremented. Two parts are matching if

their eigenvalues $s_{i,j}$ and $s_{i,j'}$ are similar, which also means that their shape is similar. The process of candidate generation is different for each transformation type. For reflection symmetries two parts $P_{i,j}$ and $P_{i,k}$ are chosen at random and difference vector $\mathbf{d}_{i,j,k} = \mathbf{c}_{i,k} - \mathbf{c}_{i,j}$ is computed, which defines the normal of a reflective symmetry plane and a reference point on the plane is given by $(0.5\mathbf{d}_{i,j,k} + \mathbf{c}_{i,j})$. For translation symmetries $\mathbf{d}_{i,j,k}$ is used as a translation offset. When generating rotation symmetries a third part $P_{i,l}$ is chosen randomly again and a circle is fitted to the center of all three parts defining a rotation around a point by an angle. To choose the right H_i from the candidates the best reflective symmetry with support above threshold 80 % is found. If no such candidate exists translational candidates are generated and if none of these matches the same criteria rotational symmetries are generated. It can happen that no dominant symmetry is found.

Hierarchy Creation. The shape is now organized into a hierarchical structure, where each node $N_{i,j,k}$ contains parts, an eigen-transform $T_{i,j,k}$ calculated by the PCA and symmetry transformation $H_{i,j,k}$ if it exists. The hierarchy is constructed in a coarse-to-fine manner, which means that on the coarsest level all parts are included in one node and on the finest level there is a multitude of nodes into which the parts are split. A root node $N_{i,0,0}$ in depth $k = 0$, which contains all shape parts, eigen-transform $T_{i,0,0}$ of the whole shape and the dominant global symmetry transformation H_i is generated first. When going from a coarser level $(k - 1)$ to a finer level for each parent node $N_{i,j,k-1}$ usually two child nodes are generated into which the parts of the parent node are split. If the parent node contains a symmetry transform, the parts are split into two sets using the symmetry plane and are then added to each node. A third child node is added if there are parts whose center $\mathbf{c}_{i,j}$ is close to the splitting plane. The splitting process is successful if at least two nodes, where each contains at least one part, are generated. If it isn't successful or if no symmetry exists a plane $x = 0$ in the local coordinate system of the parent defined by parent's eigen-transform $T_{i,j,k-1}$ is used. Planes $y = 0$ and $z = 0$ are used similarly if no previous attempt yields success and if even these aren't successful a single node is created, which containst all the parts of the parent. For each node transforms $T_{i,j,k}$ (using the PCA) and $H_{i,j,k}$ (using the same procedure as in the previous paragraph) are generated. An example of a generated hierarchy is shown in figure 2.3.

Figure 2.3. An illustration of the hierarchy creation. The coarsest level of the hierarchy is shown top left, the initial part segmentation at the bottom right. Image taken from [Jain et al. 2012].

Enforcing Nodes without Disconnected Parts. The hierarchy generated so far doesn't take contacts into account, which can cause that a node can contain parts that are disconnected. After the creation of each level k an algorithm that enforces only non-disconnected parts is used. The algorithm has a *separating* and a *merging* step, let's assume that each level contains n_k^N nodes after its creation. In the *separating* step a region growing algorithm is used for splitting of the set of the parts into smaller sets, where parts aren't disconnected, for each new set a node is created. The *merging* step is used after the *separating* step was used on each node of the level. Its goal is to merge the nodes back so that the number of nodes in the level is again n_k^N. The nodes are sorted by size and n_k^N nodes are kept. The remaining nodes are merged with kept nodes with which they share at least one contact. If more than one such kept node exists the smallest is used. When merging, all parts of the remaining node are added to the kept node and the remaining node is subsequently deleted. A level with n_k^N nodes without disconnected parts is achieved this way. There is an exception – when nodes smaller than kept nodes have a similar size (95% threshold) these nodes are also used as kept nodes, which makes the number of nodes in a level higher than n_k^N. All the data generated so far is serialized to the disk for later use.

2.1.2 Shape Synthesis

The synthesis is done interactively in the online stage using the data from the offline stage. The process of new shape creation is achieved in three steps – shape matching, shape interpolation and contact enforcement. The operations in the offline phase were applied on a single shape, from now on the actions will be done with two shapes S_1 and S_2.

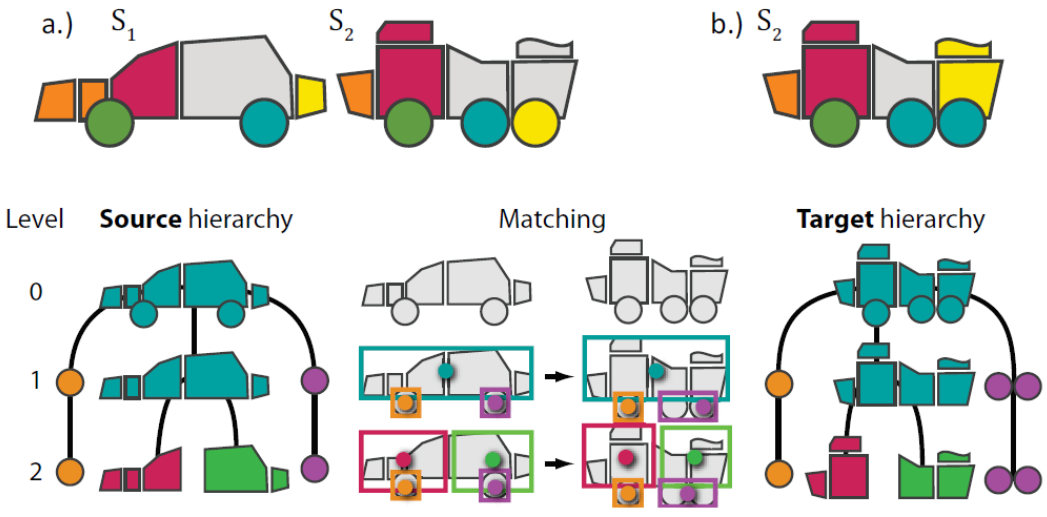


Figure 2.4. a.) Source and target shapes b.) matched target shape. The process of shape matching (bottom). Image taken from [Jain et al. 2012].

Shape Matching. Hierarchy structures and part counts are very likely to be different from each other. Because of this a four step shape matching process between source shape S_1 and target shape S_2 is necessary. In the **first step** the hierarchy of the target shape is created anew. New empty root node is created with eigen-transform $T_{2,0,0}$. The algorithm iterates over all the levels of source shape hierarchy and creates hierarchy of the target shape that is identical to heirarchy of the source shape. The nodes in S_2 are now empty and have references to the nodes of S_1. The **second step** redistributes

parts into nodes based on nearest neighbour matching. $T_{2,p(j),k-1}$ is an eigen-transform of target parent node, $\mathbf{c}_{2,j}$ is center of the part. Center in the local coordinate system of the parent node is $\mathbf{c'}_{2,j} = T_{2,p(j),k-1}\mathbf{c}_{2,j}$. Similarly, $T_{2,p(m),k-1}$ is an eigen-transform of source parent node of child node $N_{1,m,k}$, whose center $\mathbf{c}_{1,m}$ in the local coordinate system of the parent node is $\mathbf{c'}_{1,m} = T_{2,p(m),k-1}\mathbf{c}_{1,m}$. The approach adds the target parts into the target nodes that have the smallest distance between $\mathbf{c'}_{2,j}$ and $\mathbf{c'}_{1,m}$. This is possible due to the references between the nodes of the source shape and the target shape. When all parts are added an eigen-transform is calculated for the newly created node. If there are no parts added, the node is deleted in the **third step**. To keep the same number of nodes in each hierarchy the corresponding source node is merged with nearest child node of its parent node, whose corresponding target node contains at least one part. The final **fourth step** applies the procces of non-disconnected nodes enforcement explained before with the change that the number of nodes in each level must remain the same. The shape matching process is illustrated in figure 2.4.

Shape Interpolation. New shape $S(w)$ is created based on weight parameter w from interval $[0; 1]$, naturally $S(0) = S_1$ and $S(0) = S_2$. In the other cases the nodes in the finest level are used. The process so far ensured that there is equal number of nodes and the nodes have references between each other. In the level there are n_l^N nodes. From shape S_1 first wn_l^N nodes are used and from S_2 the rest, which is $1 - wn_l^N$ nodes. If nodes are symmetrical it must be ensured that only both or none of the nodes are used in the final shape. The final result is highly dependent on the way the nodes were sorted before interpolation. Best results are achieved when nodes are sorted by size in a way that the biggest nodes are in the middle ($w = 0.5$) and the smallest are at the edges ($w = 0.0, w = 1.0$). There can be multiple other sorting criteria (e.g. random order). The final shape can be reused again for another interpolation.

Figure 2.5. Finding contacts for interpolated shapes: a.) the contacts are known for the source and the target shape; b.) an instantiated node should connect to its contact partners from that originated from the same shape, but if a contact partner isn't available, the node connects to the node of the other shape with which the original contact partner forms a match. Image taken from [Jain et al. 2012].

Contact Enforcement. The positions of parts in interpolated shapes still need to be adjusted based on the contacts between them in the source and the target shapes. Nodes, which were in contact in both the source and the target shapes should remain connected in the final shape as well. Based on the relation between nodes unidirectional (if nodes were in contact only in source or target shape) and bi-directional (if nodes were in contact in both shapes) connections are created. For better illustration see figure 2.5. Unidirectional contacts will be used only if they are not in conflict with another bi-directional contact. The algorithm uses a mass-spring system to optimize the positions of parts. Masses are created in the centers of nodes and springs are for each node created between masses and contact points and between contact points mutually. The contact points of each node with connected node are calculated by averaging all contacts of the two parts in contact – there is usually multiple points where two parts intersect. Springs inside the node will always tend to keep their length, because a node is a cluster of parts that should keep its shape. Springs between nodes are added, which

tend to make their length zero, making the nodes stick together and create a compact shape.

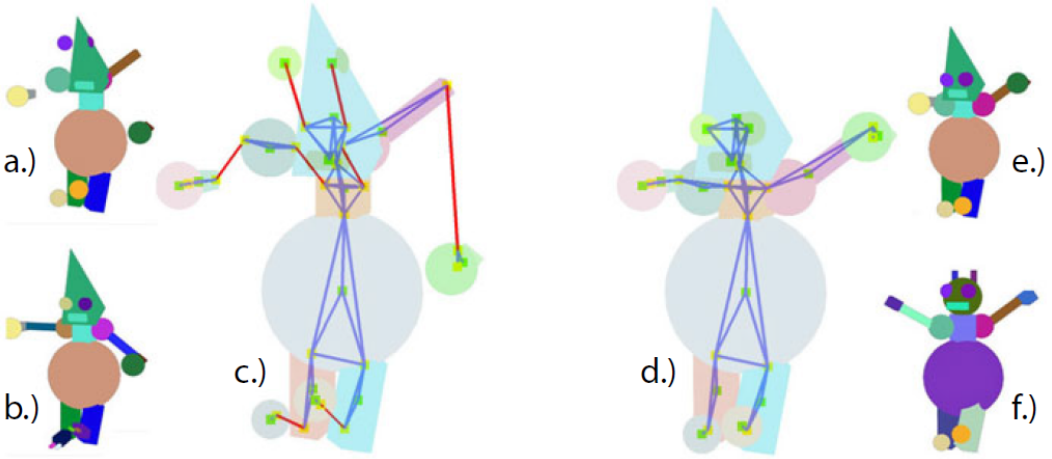


Figure 2.6. The mass-spring system: a.) interpolated shape without mass-springs b.) source shape c.) mass-spring system d.) solved mass-spring system e.) solved shape f.) target shape. Image taken from [Jain et al. 2012].

2.2 Smart Variations: Functional Substructures for Part Compatibility

The input of this system are polygonal meshes pre-segmented into meaningful parts, which don't need to be labeled. This can be achieved using automated methods [Sidi et al. 2011, Huang et al. 2011]. The shapes can be from different categories.

The key feature of the algorithm are *symmetric functional arrangements*, *sFarr*s. An *sFarr* is a triplet of parts, where two parts are symmetrical and third one connects them. There are three types of *sFarr*s (placement, embed, support) and also three functionalities (stable, unstable, coaxial).

Each shape is processed by the offline phase. The offline phase of the system consists of these steps:

- The creation of the spatial relation graph, where parts of the shape are nodes and relations between them are edges,
- identification of *sFarr*s and clusters of *sFarr*s (*sFarr* type may change if they share a part with another *sFarr*) and
- rectification of orientation (front facing vs. back facing).

Within the online phase the system creates new shapes from multiple input ones. First, compatible *sFarr*s are indentified – *sFarr*s are compatible if their type and functionality are the same. Next, the order of *sFarr* replacement between compatible *sFarr*s is determined based on geometry compatibility. As the last step, *sFarr*s are replaced and positioned.

In most cases (85%, based on the result of user study included in the article) this system produces plausible results, but it can produce failures as well. This of course applies to shapes representing man-made objects and in other cases the plausability won't be as good.

2.2.1 Shape Analysis

***sFarr*s.** As mentioned in the introduction an *sFarr* is defined as a triplet of parts, where two parts are symmetric and the third connects them. Each *sFarr* has its type and functionality.

The three types of *sFarr*s are support, where the connecting part is above the symmetrical parts, embed, where the connecting part is between the symmetrical parts and placement, where the connecting part is below the symmetrical parts. For a picture of the different types see figure 2.7.

Figure 2.7. *sFarr* types. Image taken from [Zheng et al. 2013].

An additional property of an *sFarr* is its functionality. Together with the type of an *sFarr* functionality defines the compatibility of two *sFarr*s. There are three types of *sFarr*s – stable, unstable and coaxial. Put simply an *sFarr* is stable if it doesn't topple and it is unstable if it does. To be more precise an *sFarr* is stable, if the centre of mass of the *sFarr* projected on the ground falls into the convex hull formed by its ground-touching vertices. A vertex touches the ground, if it's the vertex with the lowest position in the shape. Coaxial *sFarr*s have all the part centres on a same axis and their symetrical parts have a cylindrical shape – this can be determined using primitive fitting. The different *sFarr* functionalities can be seen in figure 2.8.

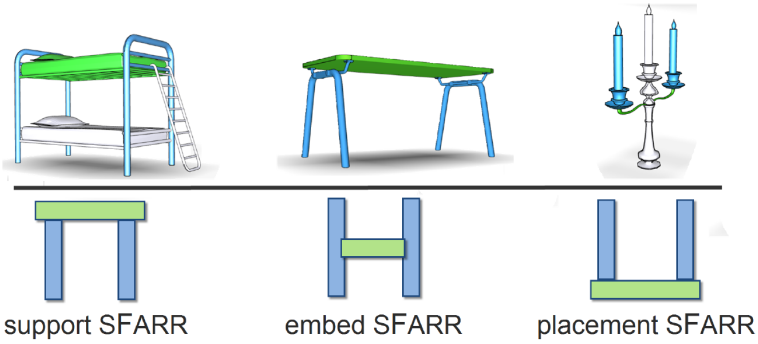


Figure 2.8. *sFarr* functionalities. Image taken from [Zheng et al. 2013].

Pre-processing. The input shapes must be segmented into meaningful parts. The segmentation can be done manually using an application like ShapeAnnotator [1]) or via an implementation of automated algorithms [Sidi et al. 2011, Huang et al. 2011]. To be able to determine which vertices are ground-touching, the shapes need to be upright oriented. This can be again automated [Fu et al. 2008]. The flip problem (front-facing vs. back-facing orientation) is resolved as a part of the system.

The Spatial Relation Graph. In the graph structure shape parts are nodes and the relations between pairs of parts are edges. Symmetries are detected using the method

[1]) `http://shapeannotator.sourceforge.net/`

of [Mitra et al. 2006]. The edges are oriented based on the type of a relation. If one part supports another, the edge will then be directed towards the part that is supported. If two parts are from an embed *sFarr*, their relation is then bidirectional. Nodes are marked as ground-touching if a part is the lowest from the shape. Examples of spatial relation graphs can be seen in figure 2.9.

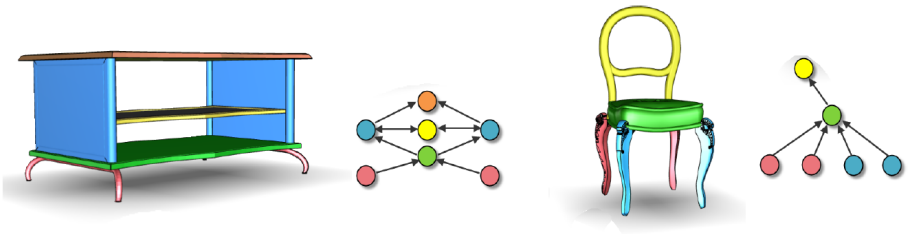


Figure 2.9. The spatial relation graph. Each part represents a node. Graph edges are either supportive or embedding. Symmetric nodes are in the same color. Image taken from [Zheng et al. 2013].

Group *sFarr*s. If there are *sFarr*s that share nodes with other *sFarr*s, it is necessary to add extra compatibility criteria to improve reshuffling results. If the two *sFarr*s are a placement and a support, both *sFarr*s should then be considered as placement and support. If they are embed and support, they should be regarded as embed and support. If the *sFarr*s are both ground-touching, a support attribtue is added. In other cases no new attributes are added. To better understand *sFarr* grouping, see figure 2.10.

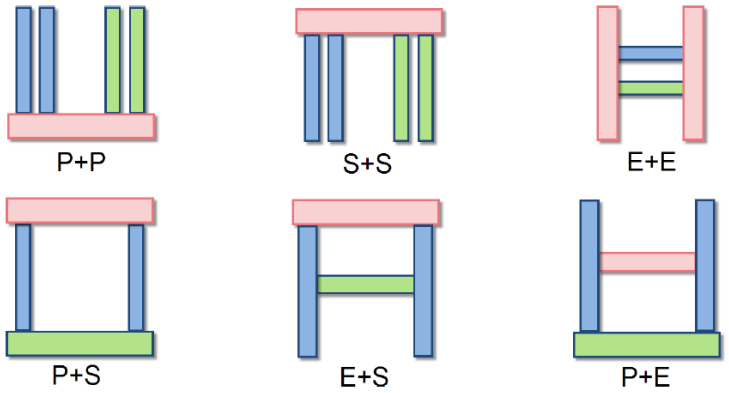


Figure 2.10. Group *sFarr* combinations. Image taken from [Zheng et al. 2013].

Graph-based Orientation Rectification. It is necessary to make sure that the shapes are either front or back facing during the reshuffle phase. This is done for each category separately (the user needs to unify the orientation, when using multiple categories). The best alignment is found as the minimum value of pairwise matching cost using a Markov Random Field formulation. A graph assignment problem is solved using the Hungarian algorithm based on the Euclidean distances between the centroids of nodes of the spatial relation graphs.

2.2.2 Shape Synthesis

***sFarr* Compatibility.** Two *sFarr*s are compatible if they are of the same type (support, embed, placement) and functionality (stable, unstable, coaxial). Let's have two *sFarr*s n_1 and n_2, where at least one is from an *sFarr* cluster, and sets N1, N2 that contain

their attributes. An *sFarr* n_1 can be substituted by n_2 if $N1 \subseteq N2$ or if $(N1 \setminus N2) \cap \{support, placement\} = \varnothing$. Both *sFarr*s also need to have the same number of contact slots.

***sFarr* Replacement.** Given a graph G_i of a shape from the set of all graphs all *sFarr*s are found in the rest of the graphs that are compatible with the *sFarr*s from G_i, denote the set of such *sFarr*s as Ω. Then all *sFarr* clusters are found in G_i and they are sorted by size, denote this set as Ξ. The *sFarr*s from each cluster in Ξ are then sorted by the compatibility with the *sFarr*s in Ω. If there are two *sFarr*s t_k and t_l their geometric compatibility can be determined using a function $\Upsilon = (t_k, t_l)$ that is based on relative sizes, angles and distances. The size of the clusters together with the value of Υ define the order of reshuffling. The threshold for the value of Υ is set to 10 by default, higher values may yield more interesting results.

***sFarr* Positioning.** An *sFarr* is replaced one part at a time. First the parts that will stay unchanged or these that were already changed are fixed, denote this set as Λ. The part is then replaced based on the contact slots, part size and the relations between parts in Λ. As soon as a part is replaced it is added into Λ and the proces is repeated for the remaining parts.

2.3 A Probabilistic Model of Component-Based Shape Synthesis

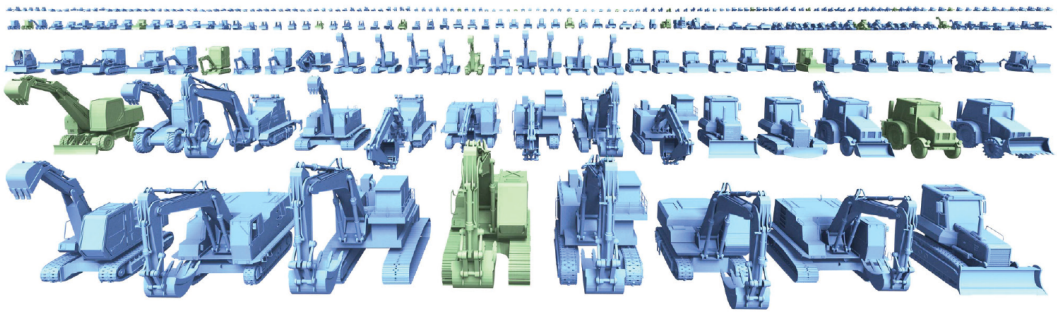


Figure 2.11. The results of the methoid. Given 22 construction vehicles (green), the probabilistic model synthesizes 253 new vehicles (blue). Image taken from [Kalogerakis et al. 2012].

Again, the input of the system are polygonal meshes, but this time they need to be pre-segmented into meaningful parts and these parts need to be uniformly labeled. As with the system by Zheng et al. automated methods can be used [Sidi et al. 2011, Huang et al. 2011]. The shapes also must be from the same category.

The steps of the offline phase are model creation and learning. The core of this approach is a probabilistic model, which is created for each category and contains multiple random variables, where some are observed (computed directly from the shapes and include mostly geometric properties) and some are latent (computed during the learning phase).

In the online stage multiple new shapes are created using the probabilistic model from the offline stage. Using a greedy algorithm new shapes with the best score are created. The newly created shapes are then geometrically optimized based on contacts and symmetries. The number of output shapes is usually bigger than the number of input shapes by an order of magnitude.

Due to the input strictness of the system, the system is able to produce a great amount of plausible shapes, but the geometrical optimization may sometimes fail to yield plausible results.

2.3.1 Shape Analysis

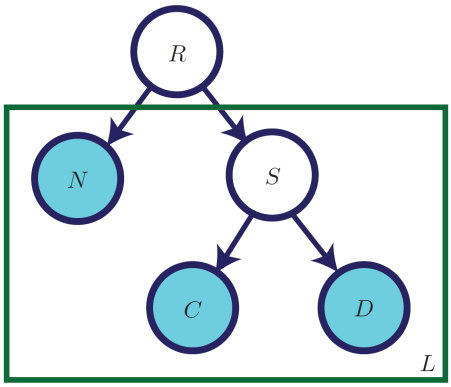


Figure 2.12. The probabilistic model. The outlined part is repeated L times for each component. The shaded variables are observed. Image taken from [Kalogerakis et al. 2012].

Probabilistic Model. The model is composed of several random variables, which are organized into a hierarchical structure, see figure 2.12. R is a variable that represents shape type from a category (e.g. a table), l is a component category (e.g. table top or leg), S_l represents a component style from a particular category (e.g. round or square table top), N_l is a number of components from category l for each style (tables can have one or two table tops or a different number of legs). C_l is a continuous geometry vector for components from category l, which includes a curvature histogram, shape diameter histogram, scale parameters, spin images, PCA-based descriptors and lightfield descriptors. D_l is a discrete geometric feature vector for components from category l, especially adjacency information. The random variables R and S_l are latent, which means that they can't be observed directly and need to be calculated during the learning process. The other random variables N_l, C_l and D_l are observed directly from the input shapes. For an illustrative example of the model, see figure 2.13.

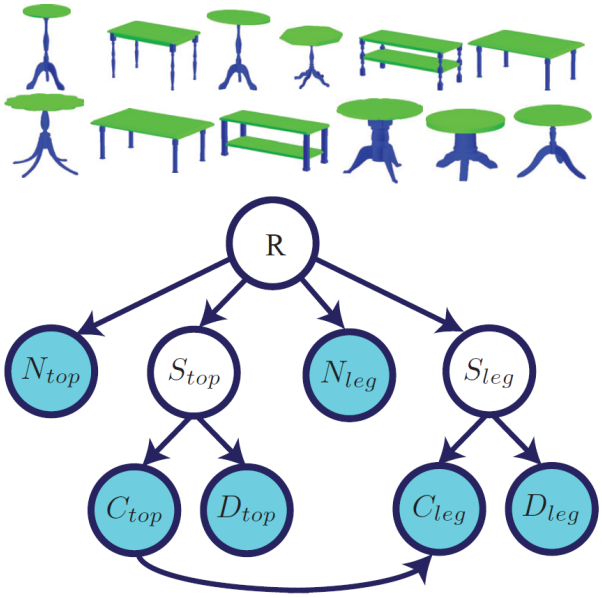


Figure 2.13. A dataset of input shapes (top) and the probabilistic model learned (bottom). Image taken from [Kalogerakis et al. 2012].

Learning. Based on the set $O = N, C, D$ the goal is to get a structure G, which has the highest probability score. By Bayes' rule this probability can be expressed as

$$P(G|O) = \frac{P(O|G)P(G)}{P(O)}.$$

To make the process computationally tractable, it is approximated using Cheeseman-Stutz score [Cheeseman and Stutz 1996]. A greedy algorithm is used. There is R with the size of one and S_l with one component. The size of S_l is increased and if the probability score decreases, the last size is saved and the process is repeated for a different category. After it has been iterated over all categories, the size of R is increased and the procedure is repeated. The algorithm is finished if a local maximum is reached that doesn't improve for ten interations.

2.3.2 Shape Synthesis

Synthesizing Components. Based on the probability model created in the offline phase new shapes are created. First the nodes of the model are sorted into a list. A tree with an empty root is created and all possible assignments are added into R based on the sorted list. Assignments that have probability lower than 10^{-12} are removed, leaving us with a set of components that can be used for shape creation.

Component Placement Optimization. During the model creation the contact points, where two components are in contact, are found. Each point contains a designation of a component that belongs to it as well as a symmetry that allows for a better placement (e.g. four symmetrical legs of a chair). A few other placement and scaling optimisations are applied.

2.4 Comparison of the Systems

This section describes the main differences between the systems. A table summarizing the differences can be found at the end of the section.

Shape Restrictions. The system [Jain et al. 2012] requires no pre-segmentation as it should be a part of the system itself. The other two systems [Zheng et al. 2013, Kalogerakis et al. 2012] require that the shapes are segmented into meaningful parts and the system [Kalogerakis et al. 2012] even need the parts to be labeled. For both of the systems an automated approach can be applied [Huang et al. 2011, Sidi et al. 2011].

Category Restrictions. The systems [Jain et al. 2012, Zheng et al. 2013] allow the synthesis of shapes from two different categories, while the system [Zheng et al. 2013] requires that the two categories are facing in the same direction (front/back facing), which needs to be made the same for both categories by the user. The system [Kalogerakis et al. 2012] doesn't allow synthesis of parts from multiple categories.

Number Of Shapes. The system [Jain et al. 2012] takes two shapes on the input and creates a third one. To create more than one shape, the intermediate results with different value of the weight parameter can be used or a repeated synthesis between a multitude of shapes must be done. The other two methods [Zheng et al. 2013, Kalogerakis et al. 2012] can take multiple shapes on the input and create multiple new shapes.

	Jain et al.	Zheng et al.	Kalogerakis et al.
requires pre-segmented input shapes	no	yes	yes
requires labeled parts of shapes	no	no	yes
number of input/output shapes	2/1	multiple/multiple	multiple/multiple
shapes must be from the same category	no	no	yes

Table 2.1. Comparison of the reviewed systems.

Chapter 3
Analysis and Design

The system chosen for implementation is [Jain et al. 2012]. The main reason for that is its independency on the input shapes as it doesn't need any pre-processing like segmentation or part labelling as well as its ease of use – user only chooses two shapes and then adjusts a slider to achieve a desired result. Also the fact that the models can be from different categories is a big advantage as it allows for an even more interesting outcome. More information about the system can be found in section 2.1.

3.1 Programming Language

The wide range of libraries available for C++ made it almost the only choice. Other languages like C# or Java were considered but some of the libraries needed exist only in C++ and programming of e.g. shape segmentation from scratch would require too much time and effort.

The platform used is Windows PC. The native compiler of the platform is used – Microsoft Visual C++ Compiler 12.0 (2013). It was necessary to use this version as it is the only version supported by some of the libraries (VCGLib, cereal), due to their reliance on C++11 template metaprogramming features.

3.2 External Libraries and Frameworks

One of the biggest strengths of C++ is the amount of libraries that are available, which helps the focus on the algorithm itself instead of recreation of code that has already been written and is available in open source form.

Mesh Manipulation. The backbone of the application is the VCG Library[1]). It offers import and export, point cloud sampling, transformation and displaying features for meshes. The code was downloaded directly from the SVN repository[2]) of the authors, revision used is 5197.

User Interface. The VCG Library has a direct support for Qt GUI framework [3]). It was found that it is easy to use and provides all the features necessary - OpenGL 3D displaying, standard window support like keyboard and mouse use, layout arrangement and GUI elements (buttons, sliders, combo boxes, etc.). The signal/slot mechanism the framework is based on has proven to be very intuitive and easy to manage. The version used in the application is the latest build of Qt4 at the time that supports Visual Studio 2013 compiler, version 4.8.6-rc1. The reason Qt5 wasn't used is to ensure compatibility with VCGLib.

Math. Eigen[4]) is used for all the linear algebra needs. The features used are Principal Component Analysis, basic vector and matrix operations and transformations. Latest version 3.2.1 is used.

[1]) `http://www.vcg.isti.cnr.it/~cignoni/newvcglib/html/`
[2]) `svn://svn.code.sf.net/p/vcg/code/trunk/vcglib`
[3]) `http://www.qt-project.org/`
[4]) `http://www.eigen.tuxfamily.org`

Shape Segmentation. The most recent shape segmentation with freely available source code is the method using Shape Diameter Function [Shapira et al. 2008]. The source code is freely available[1]) without any licensing information. Permission of one of the authors was granted and the implementation was integrated into the application but during the testing it was found that the system wasn't very reliable. It wouldn't segment all the parts and would leave a big chunk of the mesh unsegmented, so it couldn't be used. Luckily, an implementation of the same method was released at the beginning of April in the CGAL library[2]), version 4.4. One downside of this implementation is that it segments a triangulated polyhedron, which is an orientable 2-manifold. A process of mesh repairing is employed but it is not always successful, which results in some meshes being unusable.

Mesh Repairing. The JMeshLib[3]) in version 1.2 is used to run its repairing procedures. It implements cutting-edge technologies to process 3D geometry and it is the same as in the ReMESH tool [Attene and Falcidieno 2006]. Unfortunately some low quality meshes can't be repaired even by this library and thus can't be used.

Serialization. C++11 library cereal[4]) version 1.0 is used to save the data generated during the offline phase. The library provides serialization for all standard C++ library types as well as simple user defined types. The library is easily extensible so support for other external types such as `Eigen::Matrix` or VCGLib mesh structure wasn't very hard to implement. The library doesn't support raw pointers or references but the application uses smart pointers such as `std::shared_ptr` and `std::unique_ptr` instead, which are fully supported even including `std::enable_shared_from_this`.

3.3 Application Classes

This section describes all the created classes, especially their data structure and general purpose. The `GLArea` and `MainWindow` parts also contain some additional information about how the GUI works. To see the processes in which the classes are involved see figure 3.1.

Shape. `Shape` is a class, which represents the whole shape. It contains all the parts of the shape, a root `Node` and the hierarchy levels in an `std::vector` of `std::vector`s containing `Node`s of each level. All the important operations like segmentation, contact detection, building of a hierarchy, shape matching and interpolation are initiated by this class. The class also contains methods that convert the structure of the shape into a `VCGMesh` that represents some of the visualizations of the analysed shape.

Part. `Part` class represents a part of the mesh. It is a simple data structure containing an `std::vector` of `Contact` objects, a `PointCloud` object and a `VCGMesh` object. The `VCGMesh` object represents the mesh of this part and the `PointCloud` is a point cloud sampled from the mesh. Apart from some standard get/set methods it contains a `findConnected` method, which retrieves all the `Parts` this `part` is in contact with, even through other part contacts.

Node. `Node` objects are the building blocks of the shape hierarchy. Each `Node` contains an `std::vector` of `Parts`, an `std::vector` of children `Node` objects, a parent `Node`, a corresponding `Node` during shape matching, a `PointCloud` object as a joined point cloud of its `Parts` and a `Symmetry` object that determines how children `Node`s should be split from this `Node`. The `PointCloud` is computed only if it

[1]) `http://www.liors.net/shape-diameter-function`
[2]) `https://www.cgal.org/`
[3]) `http://sourceforge.net/projects/jmeshlib/`
[4]) `http://uscilab.github.io/cereal/`

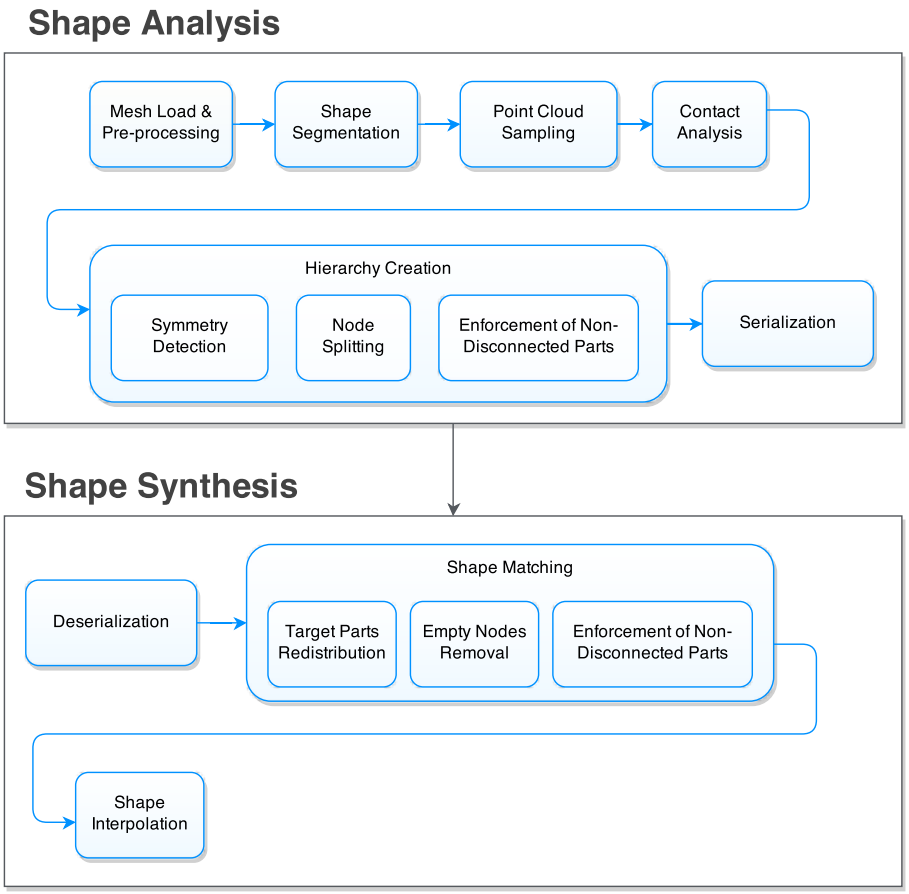


Figure 3.1. An overview of the application pipeline. *Shape Segmentation* creates new `Part` objects for a `Shape`. *Point Cloud Sampling* gives the `Parts` `PointCloud` objects as properties and *Contact Analysis* adds `Contacts`. During the *Hierarchy Creation* `Parts` of a `Shape` are put into a `Node` structure that forms the hierarchy of the `Shape`. *Shape Analysis* then works with the `Node` hierarchy of two `Shapes`.

is necessary, a boolean `pointCloudNeedsUpdate` variable is changed, whenever new parts are removed or added. This variable is checked in the getter to make sure the `PointCloud` is computed only once, when the `Node` isn't changed. The class also contains a `getConnectedSets` method, which separates its `Parts` into sets of connected parts. This method is used during the separating step.

PointCloud. `PointCloud` is a class that represents a point cloud of a `Part` or an `Node`. The actual point cloud is represented by an `Eigen::MatrixXd` variable `cloud`, which is simply a matrix, where the rows are the coordinates of the points. The class samples the point cloud data from a `VCGMesh`, which is then used by a `Part` object or it can take the `cloud` data of other `PointCloud` objects to create a union of these input point clouds, which is used by an `Node`. The main purpose of the class is to calculate the the eigen-transform of the part or node and the data is used for during the contact detection procedure.

Contact. `Contact` objects contain a set of `Eigen::Vector3d` points and a smart pointer to the `Part` these points are in contact with for a given `Part`. Imagine that the contact structure is a graph, the `Part` objects would be nodes and `Contact` objects edges that connect them.

Symmetry. `Symmetry` is a class that is used for the splitting of nodes when shape hierarchy is being built. Its objects are either generated by the `SymmetryGenerator` class or they can represent a splitting plane $x = 0$, $y = 0$ or $z = 0$ in the local coordinate system of a node. Every `Symmetry` contains an `Eigen::Vector3d` called `referencePoint`, which represents a point on the symmetry plane, and another `Eigen::Vector3d` called `normal`, which represents a normal of the plane. These two variables are used in the splitting process of nodes. If a `Symmetry` was generated by the `SymmetryGenerator` class it also contains a an `Eigen::MatrixXd`, which represents the actual symmetry transformation. A `Symmetry` has a `support` value, which is used during quality computation.

SymmetryGenerator. `SymmetryGenerator` is used to generate a number of candidate `Symmetry` objects for a set of parts of a node, which are then evaluated and the dominant symmetry is found.

Segmentation. `Segmentation` is similarly to the `MeshRepair` class just a class that accesses the CGAL library classes and methods used to triangulate an input mesh, run Shape Diameter Function segmentation on it and save the parts as OFF mesh files.

MeshRepair. `MeshRepair` is a simple class that is used to run the JMeshLib mesh repairing operations.

GLArea. `GLArea` is together with the MainWindow class the backbone of the GUI and displaying part of the application. The code is based on the example Qt application located in `apps/sample/trimesh_QT/` in the VCGLib files. The class inherits from the Qt `QGLWidget` class and reimplements `initializeGL`, `resizeGL` and `paintGL` methods. It takes care of the higher level operations compared to the MainWindow class.

The VCGLib objects `vcg::GlTrimesh` and `vcg::Trackball` objects take care of the user input/output commands for view like resizing or moving and the actual displaying of the different visualizations of the analysed shape (point clouds, contacts, segmentation, etc.). When the `paintGL` method is called the `vcg::Trackball` object is used to orient the view and the `vcg::GlTrimesh` calls its `Draw` method that draws the mesh based on the `DrawMode` enumeration value.

Two enumerations store the current state for the viewer. First one is called `DrawMode` with values `SMOOTH`, `POINTS`, `WIRE`, `FLATWIRE`, `HIDDEN` and `FLAT`, which represent the methods the mesh is drawn. For example when `WIRE` is set, the mesh is drawn as a wireframe, when `POINTS` is set, only the vertices of the mesh are drawn. The other one is called `ShapeMode` and stores the type of visualization to be shown by the application represented by the values `SEGMENTATION`, `CONTACTS`, `CLOUDS`, `TRANSFORMS`, `CONTACT_PARTS`, `HIERARCHY_LEVELS`, `HIERARCHY_TREE`. Most of these correspond to the operations done in the shape analysis phase. `TRANSFORMS` shows the eigen-transformation visualization a box of size 1 transformed into the local coordinate system of its corresponding part. `CONTACT_PARTS` displays parts that have at least one contact as green and contactless parts as red. `HIERARCHY_LEVELS` shows the different levels of the shape hierarchy, `HIERARCHY_TREE` displays the whole hierarchy.

MainWindow. `MainWindow` inherits from the `QMainWindow` class and manages the lower level operations of the Qt GUI – slots and signals connections[1]). Every action in Qt can have a reaction that is done as a response. A connection needs to be established first for example:

```
connect(sender, SIGNAL(somethingHappened()),
receiver, SLOT(doSomethingAsReaction()))
```

[1]) `http://qt-project.org/doc/qt-5/signalsandslots.html`

establishes a connection between the `sender` and the `receiver` object. If the signal `somethingHappened()` is emitted a slot method `doSomethingAsReaction()` is subsequently called.

VCGMesh. `VCGMesh` is a class used to define a triangle mesh to be used with the VCGLib. Each component of the mesh (vertex, face) needs to be enabled as well as the properties of these components (color, position, normal, etc.). More information about this can be found in a tutorial by the creators of the library[1]). The class inherits from the `vcg::tri::TriMesh` with `VCGVertex` and `VCGFace` as the template parameters:

```
class VCGMesh : public vcg::tri::TriMesh
    <std::vector < VCGVertex >, std::vector < VCGFace >> {...};
```

These two types also inherit from their respective `vcg::tri::` classes, which have standard components enabled via the template parameters:

```
 class VCGVertex : public vcg::Vertex
    <VCGUsedTypes, vcg::vertex::VFAdj, vcg::vertex::Coord3f,
     vcg::vertex::Normal3f, vcg::vertex::Color4b,
     vcg::vertex::BitFlags> {};
```

```
class VCGFace : public vcg::Face
    <VCGUsedTypes, vcg::face::VFAdj, vcg::face::VertexRef,
     vcg::face::Normal3f, vcg::face::Color4b, vcg::face::BitFlags> {};
```

The `vcg::vertex::VFAdj` property enables that a reference between adjacent vertices and faces is stored. This is necessary for the point cloud samplig. The `VCGUsedTypes` inherits from the `vcg::UsedTypes` class and declares which are the types invoved in the definition of the mesh:

```
class VCGUsedTypes : public vcg::UsedTypes
    <vcg::Use<VCGVertex>::AsVertexType,
     vcg::Use<VCGFace>::AsFaceType> {};
```

The class has a `surfaceArea` method, which computes the area of all faces of the mesh and is used later for point cloud sampling, it also contains some convenience methods for loading the mesh from file (`loadOBJ`, `loadOFF` and so on)

[1]) `http://vcg.isti.cnr.it/~cignoni/newvcglib/html/basic_concepts.html`

Chapter 4
Implementation

4.1 Shape Analysis

The shape analysis is an offline part of the method that is done on a single input mesh. First the mesh is segmented into parts, which are sampled to point clouds. Using these point clouds contacts are detected. Hierarchy used later for the synthesis of new shapes is created using the contact information and symmetries that need to be detected. The following subsections describe how all these steps were implemented.

4.1.1 Mesh Segmentation

Before the actual segmentation of the mesh a couple of operations are carried out. The mesh is first loaded as a VCGMesh and it is uniformly rescaled by the average of its width, height and depth, so that every shape loaded has a similar size. The width, height and depth are the absolute values of the differences between maximum and minimum values of the point cloud of the whole mesh in the x, y and z dimensions. The rescaled shape is exported to an OFF file that is used for mesh repairing.

The `MeshRepair` object's `run` method is then used, which simply creates an instance of JMeshLib `Triangulation` class, loads the OFF file and `checkAndRepair` method of the `Triangulation` object is ran to repair the mesh, which is then saved again as OFF that is going to be segmented by CGAL.

`Segmentation` object is created and loads a repaired mesh as a polyhedron, which must be an orientable 2-manifold. If it is not the segmentation process can't be ran and the shape can't be used. If the shape is correct CGAL's `triangulate_polyhedron` method is applied and the segmentation can begin. First the `CGAL::sdf_values` method is ran, which computes the SDF values using the default parameters. Afterwards, the `CGAL::segmentation_from_sdf_values` method is ran to do the segmentation of the mesh using parameters `number_of_clusters = 7` and `smoothing_lambda = 0.15`. These values have proven to give the best results for a wide variety of shapes. The method fills a map object, where the keys are `Facets` and the values are part ids. Using this map the parts of the mesh are built with the use of `CGAL::Polyhedron_incremental_builder_3` and each part is then exported as an OFF file. These files are then loaded into `VCGMesh` objects, which are used in the `Part` objects of a `Shape`. All the newly created files are then deleted as they are no longer needed.

4.1.2 Contact Analysis

When a `Part` object is being constructed, its mesh is sampled as a point cloud. The method used for this is the `vcg::tri::PoissonSampling` that uses the Poisson-Disk sampling method to sample the mesh to a point cloud, where the distance between points is roughly similar. The number of samples, which needs to be specified, is

determined using the formula, which ensures that the sampling is done uniformly for all parts and shapes:

$$\frac{A_{part} \cdot \mu}{\varphi} + \omega$$

The part area (A_{part}) is computed by iterating over all faces of the mesh and computing their areas, the area of the part is then the sum of all face area values. The area of a face is given by $\frac{1}{2}|\vec{a} \times \vec{b}|$, where $\vec{a}$ and $\vec{b}$ are the vectors from vectices A to B and A to C respectively. The sample count multiplier (μ) determines the density of the sampling. To make sure the sampling is uniform the diameter of the whole shape (φ) is also used. It was found that it gives best results for a value of 100 000. The minimal sample count (ω) variable ensures that even the smallest parts are sampled to at least 200 points, which is the value it is set to. The sampled points are then put into an `Eigen::MatrixXd` with rows as individual points, let's call it a *data* matrix. The coordinates of the points are homogeneous, because a translation is required to compute the transformation into the local coordinate system of the part (eigen-transform) and it is convenient that the eigen-transform is a single matrix.

Once the point cloud is sampled, the transformation into the local coordinate system of the part eigen-transform is computed. The center c of a part is calculated as an average of all rows of the point cloud values. Using the center a *translation* that moves the part into the center is given as:

$$T = \begin{pmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Next the rotation matrix is computed. To do this, `Eigen::SelfAdjointEigenSolver` is used, which takes a covariance matrix of the data on the input. The covariance matrix is calculated from the *data* matrix. First the center c is subtracted from each row, let's call this matrix C. The covariance matrix is given as:

$$cov = \frac{C^T \cdot C}{N_{rows} - 1}$$

The three eigenvectors $\vec{e_1}$, $\vec{e_2}$, $\vec{e_3}$ from the eigen solver form a *rotation* matrix:

$$R = \begin{pmatrix} e_{1_x} & e_{1_y} & e_{1_z} & 0 \\ e_{2_x} & e_{2_y} & e_{2_z} & 0 \\ e_{3_x} & e_{3_y} & e_{3_z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For the *scaling* matrix it would be a good assumption that the eigenvalues computed by the eigen solver could be used, but unfortunately these don't seem to give a correct result. Instead *data* matrix is transformed by the *rotation* and then the singular values s_x, s_y and s_z are computed as the absolute value of the difference between the extremes of the corresponding dimension. The *scaling* matrix is then:

$$S = \begin{pmatrix} \frac{1}{s_x} & 0 & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 & 0 \\ 0 & 0 & \frac{1}{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Once these three matrices are known, the eigen-transform (ET) is simply:

$$ET = S \cdot R \cdot T$$

During the construction of a `Part` a bounding box is set as a `vcg::Box3` created from the minimum and maximum values of the point cloud. The bounding box is enlarged uniformly for each `Part` by adding 1 % of the diameter of the whole shape, so that when two bounding boxes are right next to each other, but don't actually collide, they are still considered as colliding.

When finding the contacts, each possible pair of parts is checked. First the bounding boxes of the parts are tested for a collision using the `vcg::Box3::Collide` method. If the parts collide then for each point pair of the two parts a distance between the two points is computed and if it is smaller than 0.1 % of the shape's diameter the two points are added into the `Contact` object that is constructed afterwards. Of course it is ensured that all the points in the contact set of points are unique, because a point can be in contact with multiple other points.

4.1.3 Symmetry Detection

The symmetries are detected by the `SymmetryGenerator` class for a set of `Parts` from a `Node` object. Each type of symmetry is detected separately, reflection candidates first, then translation candidates and roatation candidates last. At the start $2 \cdot N_{parts} + 10$ candidates are generated, where N_{parts} parts is the number of parts in a `Node`.

The generating process starts by choosing two (for reflection, translation) to three (rotation) randomly selected parts, whose centers are used for the creation of a symmetry matrix. The parts aren't chosen fully randomly as it is always checked if the part sizes are similar as mentioned in [Wang et al. 2010]. This greatly improves the performance of the algorithm, because high quality symmetries are always created from similar parts anyway. The parts are considered similar if for each two parts from the chosen parts the difference between their diameters is smaller than 5 % of the average of these diameters. If the parts aren't similar, it is sought for another ones that are until a maximum of attempts is reached, which is $N_{parts}^2 + 10$. It can happen, that a smaller number of candidates is generated than initially requested, but not by a dramatic margin.

From the part centers symmetries are created. For the reflection symmetries there are two centers c_1 and c_2 used. Normal $\vec{n}$ of the reflection is the difference between the centers. A reference point ref on the reflection is given by $\frac{1}{2}\vec{n} + c_1$. This point is translated to origin of the coordinate system by matrix T:

$$T = \begin{pmatrix} 1 & 0 & 0 & -ref_x \\ 0 & 1 & 0 & -ref_y \\ 0 & 0 & 1 & -ref_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The normal vector is rotated to reflection plane at origin until it is coincident with positive z axis by matrix R_{xy}, where $\lambda = \sqrt{\vec{n_y}^2 \vec{n_z}^2}$:

$$R_{xy} = \begin{pmatrix} \frac{\lambda}{|\vec{n}|} & \frac{-\vec{n}_x \vec{n}_y}{\lambda |\vec{n}|} & \frac{-\vec{n}_x \vec{n}_z}{\lambda |\vec{n}|} & 0 \\ 0 & \frac{\vec{n}_z}{\lambda} & \frac{\vec{n}_y}{|\vec{n}|} & 0 \\ \frac{\vec{n}_x}{|\vec{n}|} & \frac{\vec{n}_y}{\lambda} & \frac{\vec{n}_z}{|\vec{n}|} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The object is then reflected throught xy plane by matrix R_l:

$$R_l = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Inverse transformations are applied and the final reflection matrix (M_{refl}) is given by:

$$M_{refl} = T^{-1} \cdot R_{xy}^{-1} \cdot R_l \cdot R_{xy} \cdot T$$

The translation symmetries (M_{transl}) are again created using two part centers c_1 and c_2. The normal $\vec{n}$ and reference point ref of the translation are calculated the same way as with reflection. The translation symmetry is simply a translation by $\vec{n}$:

$$M_{transl} = \begin{pmatrix} 1 & 0 & 0 & \vec{n}_x \\ 0 & 1 & 0 & \vec{n}_y \\ 0 & 0 & 1 & \vec{n}_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

To get a rotation symmetry three part centers c_1, c_2 and c_3 are used. It is necessary to find the center c_{rot} of the circumscribed circle of the the part centers. The center c_{rot} is given by the linear combination:

$$c_{rot} = \alpha c_1 + \beta c_2 + \gamma c_3$$

To get α, β and γ dot and cross products are used:

$$\alpha = \frac{|c_2 - c_3|^2 (c_1 - c_2) \cdot (c_1 - c_3)}{2|(c_1 - c_2) \times (c_2 - c_3)|^2}$$

$$\beta = \frac{|c_1 - c_3|^2 (c_2 - c_1) \cdot (c_2 - c_3)}{2|(c_1 - c_2) \times (c_2 - c_3)|^2}$$

$$\gamma = \frac{|c_1 - c_2|^2 (c_3 - c_1) \cdot (c_3 - c_2)}{2|(c_1 - c_2) \times (c_2 - c_3)|^2}$$

Normal $\vec{n}$ is computed as a cross product of vectors between the part centers:

$$\vec{n} = (c_3 - c_2) \times (c_2 - c_1)$$

Angle δ between the vectors from centers c_1 and c_2 to c_{rot} is given by the formula:

$$\delta = acos\left(\frac{(c_1 - c_{rot}) \cdot (c_2 - c_{rot})}{|c_1 - c_{rot}||c_2 - c_{rot}|}\right)$$

Rotation matrix R is then retrieved using an `Eigen::AngleAxis` object constructed using the normal $\vec{n}$ and angle δ, which is then changed into a 4×4 matrix. This matrix is not the symmetry matrix yet as it is necessary to translate the centers into origin of the coordinate system using a translation matrix:

$$T = \begin{pmatrix} 1 & 0 & 0 & -c_{rot_x} \\ 0 & 1 & 0 & -c_{rot_y} \\ 0 & 0 & 1 & -c_{rot_z} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The final rotation symmetry matrix (M_{rot}) is then a simple multiplication of the matrices mentioned above:

$$M_{rot} = T^{-1} \cdot R \cdot T$$

After the symmetry candidates have been generated their quality needs to be rated. Each part of the node is taken and its point cloud is transformed by the symmetry candidate. The closest center to the center of the transformed point cloud is then found and if the singular values of both point clouds are similar the support value (κ) of the candidate is increased. The singulare values are considered similar if their difference in each dimension is lower than 0.025. The quality of the candidate is simply $\frac{\kappa}{N_{parts}} \cdot 100\ \%$. The whole process is first done for reflections and if there exists a candidate with best quality greater or equal to 80 % it is used as the dominant symmetry of the node. If there isn't such candidate, translation candidates are generated and tested same way and if even these don't match the criteria rotation candidates are attempted. It can happen that no symmetry is found.

4.1.4 Hierarchy Creation

Because the paper [Jain et al. 2012] doesn't mention how to work with parts without any contacts at all, such parts are excluded from the whole hierarchy building process. Contactless parts are a rare occurance but they can spoil the process of hierarchy creation severely.

The hierarchy of a `Shape` is a simple tree structure, where nodes have smart pointers to a parent node and children nodes. The number of children is not limited to a specific number as a node can be split into multiple parts when all steps of the hierarchy creation are applied.

The hierarchy creation starts with a root `Node`, where all parts of the mesh are inserted. This node is added into the first level of the hierarchy, which is represented as an `std::vector` of `std::vector`s containing `std::shared_ptr`s of `Node` objects, `typedef`'d as `NodePtr`. To make this more clear – The "outer" `std::vector` represents the levels of a hierarchy and the "inner" `std::vector`s contain the `Node`s of the level.

Several local variables are created – `std::vector<NodePtr> currentLevel`, which represents the level being currently created, `int lastLevel = 0`, which keeps track of the number of the last created level and `boolean levelSplitSuccess = true`, which stores whether the level got split, if this is `false` the last level wasn't split and the hierarchy building process ends.

Node Splitting. While the `levelSplitSuccess` variable holds true the following iterative process is done, which is started by clearing the `currentLevel` to get rid of the `Node`s from the last iteration. For each `Node` from the `lastLevel`th level of the hierarchy a symmetry is found as described in the previous subsection. Now for each `Node` splitting is attempted according to the symmetry found, if no symmetry was found or if the splitting among it is unsuccessful planes $x = 0$, $y = 0$, $z = 0$ are used until the splitting is successful. If the splitting isn't successful for any `Node` of the level the whole process is finished, the process continues for another iteration otherwise.

The actual splitting process is done using a reference point on the splitting plane and a normal. Symmetries generated by the `SymmetryGenerator` object have these properties set during their creation. When it is necessary to use the non-generated splitting planes ($x = 0$, $y = 0$, $z = 0$), these two properties need to be obtained. The reference point is acquired by multiplying the origin of the coordinate system ($o = (0, 0, 0)$) with the inverse eigen-transform of the `Node` that is about to be split. The normal is a vector $\vec{n}$

multiplied by the same transformation. For plane $x = 0$ the vector is $\vec{n_x} = (1, 0, 0)$, for $y = 0$ $\vec{n_y} = (0, 1, 0)$ and for $z = 0$ $\vec{n_z} = (0, 0, 1)$.

A `Node` is split using a simple method for each of its `Parts`. The distance between the center of the part and the reference point is computed. The center is then moved by a 1000th of the normalized normal vector and the same distance is computed again. If the distance before was greater than the distance after, the part is added into `std::vector<PartPtr> left` and into the `std::vector<PartPtr> right` otherwise. If the difference between the distances is smaller than 0.0001 the part is considered to be close to the splitting plane and it is added into the `std::vector<PartPtr> middle`. From these three `std::vectors` new Nodes are created, with their parents set to the `Nodes` they were split from and they are added into the `std::vector<NodePtr> current` for further processing, namely the enforcement of nodes with non-disconnected parts. Results of the spliting can be seen in 4.1.

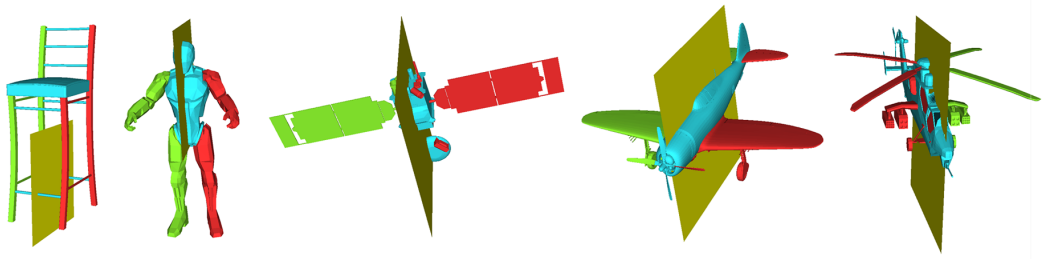


Figure 4.1. The dominant symmetries (yellow) of whole meshes from different categories.

Enforcement of Non-disconnected Nodes. To make sure the parts in the nodes are connected with contacts together a two step process that enforces this is applied. The first step, called *separating step*, separates the nodes into sets of parts, where the parts in one sets are connected by contacts together. The algorithm first remembers the number of nodes in an `int numberOfNodes` variable, which is the size of the `currentLevel` collection.

The *separating step* is started and for each `Node` the sets of connected `Parts` in a `Node` are found. To do this a method `Node::getConnectedSets` is called, which creates an `std::vector` of `std::vectors` containing `std::shared_ptrs` of `Part` objects, simply `std::vector<std::vector<PartPtr>> sets` and `std::vector<PartPtr> remainingParts`, where all parts of the `Node` are put.

While the `remainingParts` vector contains any elements the sets of connected parts are found. For the first part of `remainingParts` all connected `Parts` are found using a `Part::findConnected` method. These parts are then put into the `sets` variable and are deleted from the `remainingParts` vector. From the `sets` variable new `NodePtrs` are made to reflect the separation.

The method `Part::findConnected` finds all the connected parts using breadth-first search. It takes `std::vector<PartPtr> nodeParts` and an `std::unordered_set` of `Parts foundParts` on the input. The `nodeParts` variable represents the parts of the node and `foundNodes` is where the found connected parts are put. First the parts this `Part` is in contact with are found and the contact parts which aren't among `nodeParts` are removed. The current `Part` is added into `foundParts` and for each contact part that is in `nodeParts`, but isn't in `foundParts` yet the `findConnected` method is ran.

To ensure the number of nodes in `currentLevel` is similar to the number of nodes before the *separating step* the second step called *merging step* is ran aftwerwards. The

nodes in `currentLevel` are first sorted by their diameter in descending order. An `std::vectors` of `NodePtrs` called `kept` and another collection of `NodePtr` pairs called `remaining` are created. A smallest kept diameter is set to the diameter of the last node in `kept`. The first `numberOfNodes` from `currentLevel` are put into `kept`. The other nodes of `current` are put into `kept` if they have a diameter greater or equal to 95 % of the smallest kept diameter or into `remaining` as the first `NodePtr` of the pair otherwise.

The `remaining` nodes are now merged with the `kept` nodes. While `remaining` isn't empty or its size isn't shrinking a loop is executed, which takes all the `remaining` nodes to merge them with `kept` nodes with which they share at least one contact. The smallest of the contact `kept` nodes is chosen and set as the second `NodePtr` of the pair. Once all `remaining` nodes have been processed, they are merged with the `kept` nodes (the second `NodePtr` of the pair) and they are deleted from `remaining`. The merging is done after the `kept` nodes are found for a `remaining` node so that a `kept` node doesn't change its size for whole iteration of the process. If the size of `remaining` doesn't change after the iteration, the `remaining` nodes are put into `kept` as there is nothing else to do with them. The paper [Jain et al. 2012] unfortunately isn't very specific when it comes to the *merging step* so some of the procedures mentioned in this paragraph might not be fully correct. The enforcement steps are illustrated in 4.2.

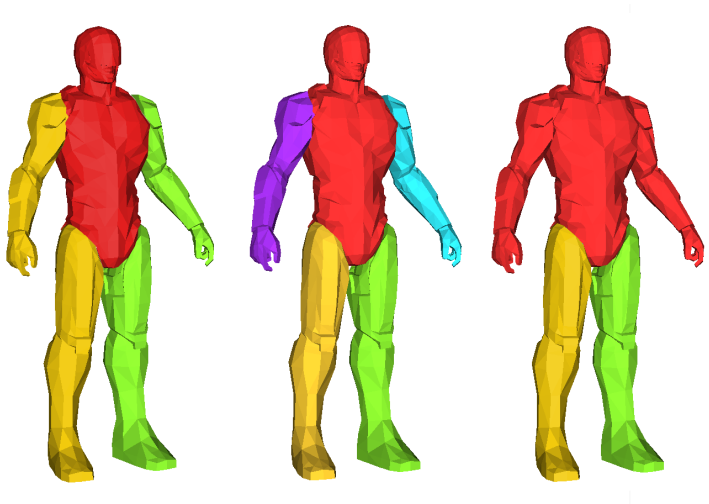


Figure 4.2. The mesh after split (left), the mesh after *separating step* (middle), the mesh after both steps (right). It can be seen that the smallest nodes (arms) were merged with the adjacent (torso) node.

Final Steps. After it is ensured that `Nodes` in `currentLevel` contain only connected `Parts` the `Nodes` in the level are added into their parents as children. This wasn't done until now, because the structure could change during the enforcement of non-disconnected parts in nodes. the `currentLevel` is added into the `levels` hierarchy of the `Shape` and if any splitting occured the whole process is repeated for the last created level.

When the hierarchy is built the `Shape` is ready to be serialized to the disk using the `cereal` library. The `Shape` is stored in a binary archive with the extension ".pcs". The library supports all the types from the standard C++ library used (`std::shared_ptr`, `std::vector`, etc.), so when classes with these objects are serialized, simple `serialize` method is used for saving and loading. The only two exceptions are the objects from the external libraries, namely `Eigen::Matrix` and `VCGMesh`. For `Eigen::Matrix` a new `save` and `load` methods had to be created in an external `cereal/types/matrix.hpp`

file, because it isn't desirable to edit the `Eigen` library files directly. The `VCGMesh` is templated as a user defined type so there is an access to the code itself. To avoid working with the whole structure of a `VCGLib` mesh object a simple workaround is used. While loading the mesh file it is saved as an OBJ file, which is then parsed as an `std::string`. This string is then saved in the `save` function and from its contents the mesh data is loaded in the `load` function using `vcg::tri::io::ImporterOBJ`.

4.2 Shape Synthesis

The shape synthesis is an online phase of the method done after the shapes have been analysed. The synthesis is done using two precalculated analysed meshes called "source" and "target" chosen by the user. The shapes are deserialized from the binary serialization files into the memory and the shape matchin procedure can start.

4.2.1 Shape Matching

The hierarchy of the target shape is cleared and a root node containing all the parts of the target shape are added. During the process of matching every target `Node` keeps a smart pointer to a corresponding node from source shape and vice versa. This relation is later used for `Node` recombination. The correspondency is set for the root node.

There is again an `std::vector` of `Node`s called `currentLevel`, where `Node`s of the currently processed target level are put. For each level of the source shape the following process is done. The `currentLevel` collection is cleaned to wipe the data from processing the last level. Pairs of `Eigen::Vector4d` centers and respective `Node`s called `sourceNodePairs` are created. According to the paper [Jain et al. 2012] the center should be multiplied by an eigen-transform of parent node, but the matching didn't seem to work using this method. Only nodes position is used as another option mentioned in [Jain et al. 2012].

Target Parts Redistribution. New nodes of the target shape are then created. For each `Node` from the source shape level a `Node` is created and the correspondency relationship is set. These newly created `Node`s are then put into the `currentLevel` collection. Parts of the mesh are then redistributed into the new target nodes. For each `Node` from the previous level of target, where the parent `Node`s of the currently processed ones reside, all `Part`s of the parent `Node` are taken and are redistributed using nearest neighbour matching of the center of the `Part` and the precomputed center from the `sourceNodePairs` collection. Again the center of the part should have been transformed by the eigen-transform of the parent `Node`, but this was abandoned as described in the previous paragraph. Only the `sourceNodePairs` `Node`s are used, which are the children of the corresponding source `Node` of the redistributed parent target `Node` to ensure that `Part`s are redistributed only into child `Node`s of the parent source `Node`. The closest source shape `Node` is then used for redistribution and the `Part` is inserted into the corresponding target `Node` of the closest source `Node`.

Empty Nodes Removal. In case that no parts are redistributed into a `Node` on the current level this `Node`'s corresponding source `Node` is merged with the closest sibling, whose corresponding target `Node` has at least one part. During merging all parts of the merged `Node` are added into the `Node` it is merged with and the children `Node`s of the merged `Node` have their parent changed also to the `Node` the merged `Node` is being meged with. The merged `Node` and its corresponding target `Node`s are then deleted from the hierarchies.

The enforcement of non-disconnected parts in nodes is then applied on the `currentLevel` collection with a couple of differences from the one used during shape analysis. The article [Jain et al. 2012] says this process is same as during shape analysis (except that the number of nodes after the enforcement must be always the same as before), but this is in fact not the case as the correspondency relationship can possibly be broken during this process. When the separating step is applied and the node is being split into the connected nodes, the corresponding `Node` of the split `Node` has its corresponding `Node` set to `NULL` and the `Node`s created using the splitting have their corresponding `Node`s set to the corresponding `Node` of the `Node` they were created from. In the merging step, when a `Node` is being merged and its corresponding `Node` isn't set to `NULL`, its corresponding source `Node`s corresponding `Node` is set to `NULL` to remove the relationship with the `Node` being merged.

The correspondecy relationships are then reestablished. For each source node from the currently processed level if the `Node`s corresponding `Node` is set to `NULL`. A node from target that has this source `Node` set as corresponding is found and the correspondency relationship is set. Even after this it can happen that a source node has no corresponding node. It is then checked if corresponding relationship is mutual for each target node and if it isn't the correspondency is incorrect and the target node has the correspondency relationship set with the source node, whose corresponding node is set to `NULL`.

Each `Node` in `currentLevel` is now added into their parent node to keep the parent-child relationship. This is done after all the other steps, because this relationship could be change when they are applied. The `currentLevel` collection is added into the matched target shape and the process is then applied to all the other levels. After the execution of the matching on all levels the two shapes are matched and they are ready to be used in the shape interpolation.

4.2.2 Shape Interpolation

With the use of the matched source and target shapes the interpolated shape can now be created as a recombination of the two shapes. The finest levels of the matched source and target shapes are sorted by the cojoined diameter of the nodes and their corresponding nodes in the other shape. The nodes on the coarsest level are then reordered so that the nodes with the smallest diameter are at the beginning and the end of the collection and the biggest node is in the middle. The weight parameter w in the range 0–100, which represents the percentage of the nodes used from the source shape is determined by the user. The recombined shape is then composed of $w \cdot N_{nodes}$ source nodes and $1 - w \cdot N_{nodes}$ target nodes, where N_{nodes} is the number of nodes on the finest level of the source as well as the target shapes. The result shape can then be saved as OBJ, OFF, PLY or VRML file.

Chapter 5
Testing

Testing was done for the different steps of the shape analysis, some performance data are also included for the record. This chapter also includes the results of the shape matching for various categories. The computer used for the testing was a Windows 7 PC with an Intel Core i7 870 processor, 8 GB of DDR3 RAM and an NVIDIA GeForce GTX 560 Ti graphics card.

5.1 Shape Analysis

Because it is hard prove that the steps of the analysis are correct, simple illustrations with commentary are provided. Although the shape analysis seems to be very robust and gives correct results, it remains for the reader to decide whether it really is the case.

Segmentation. The CGAL segmentation provides two main parameters to be set `number_of_clusters` and `smoothing_lambda`. The `number_of_clusters` represents the amount of sets of facets with similar SDF values. As mentioned in [Yaz and Loriot 2000] the number of clusters doesn't necessarily result in a higher number of segmented parts, but it actually mostly is the case as can be seen in figure 5.1.

Figure 5.1. Segmentation of a shape with different values of the `number_of_clusters` variable (1, 3, 5, 7 and 9 from left to right). The `smoothing_lambda` was set to the value used – 0.15.

The value of `number_of_clusters` was set to 7 as it segments the mesh into a good amount of detail, while still keeping the parts meaningful. The `smoothing_lambda` parameter on the other hand does always ensure that the number of parts will change as it changes. The number of parts gets bigger as the parameter gets closer to 0, where no smoothing is applied. A value of 0.15 was used for similar reasons as with the `number_of_clusters` parameter. A multitude of segmentations with different `smoothing_lambda` values can be seem in figure 5.2

Figure 5.2. Segmentation of a shape using different values of the `smoothing_lambda` variable (0.0, 0.05, 0.1, 0.15, 0.2 and 0.5 from top left to bottom right). The `number_of_clusters` variable was set to 7, which is the value used.

In general a slightly more segmented mesh is better than a less segmented one, because the procedures used later will merge the parts into more bigger groups. Figure 5.3 shows segmentations of shapes from different categories.

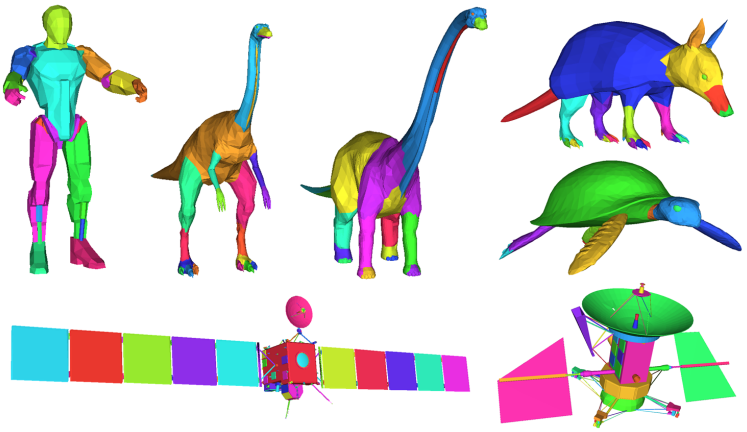


Figure 5.3. Segmentations of shapes from various categories.

Point Cloud Sampling. Sampling of the parts into point clouds with evenly distrubuted points is essential for the correctness of the contact analysis as well as symmetry detection. Figure 5.4 shows that the point clouds have correct distributionm with the exception of smaller parts, where the minimal sample count variable ω is applied, which has proven to be necessary for correct contact detection, when small parts are involved.

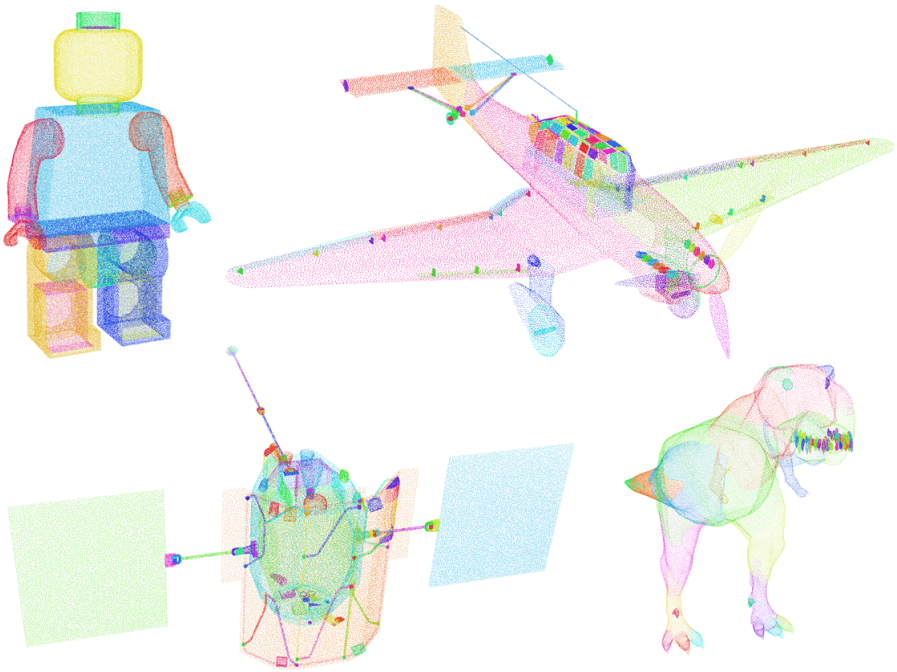


Figure 5.4. Shapes from different categories sampled to point clouds.

The most important parameter is the sample count multiplier μ, which determines the number of samples (points) of the point cloud. When set to $\mu = 75\ 000$, satisfying results are already achieved, but to make sure the processes depending on the sampling are running smoothly a value of 100 000 is used. Figure 5.5 shows the contact graph visualizations for diferent values of μ.

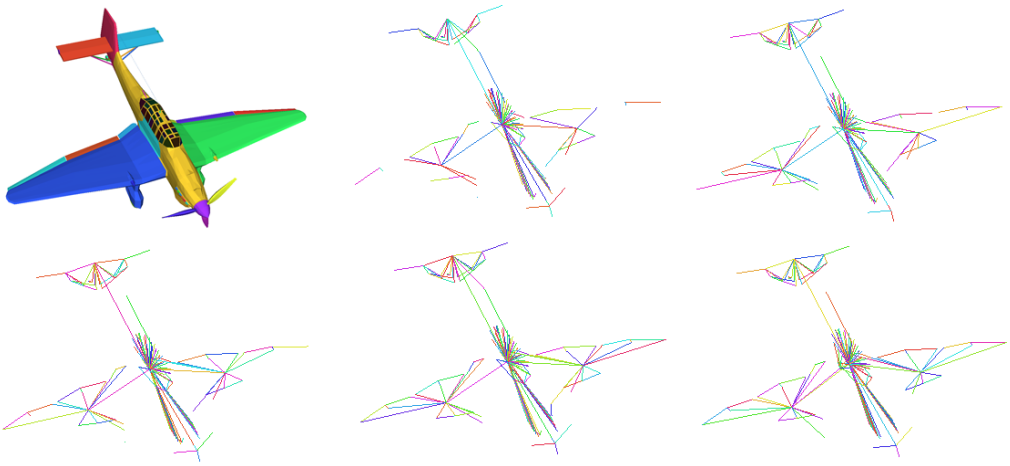


Figure 5.5. Graphs of shape's contacts using different values of the sample count multiplier μ (from left to right – segmented mesh, graphs using μ with a value of 25 000, 50 000, 75 000, 100 000 and 200 000).

Symmetry Detection. As it was shown in 4.1 in a previous chapter, the symmetry detection process produces reasonable results for most of the shapes. There are, however, some shapes, where the symmetries are not easy to detect as they can be rather ambiguous. Examples of these cases can be seen in figure 5.6.

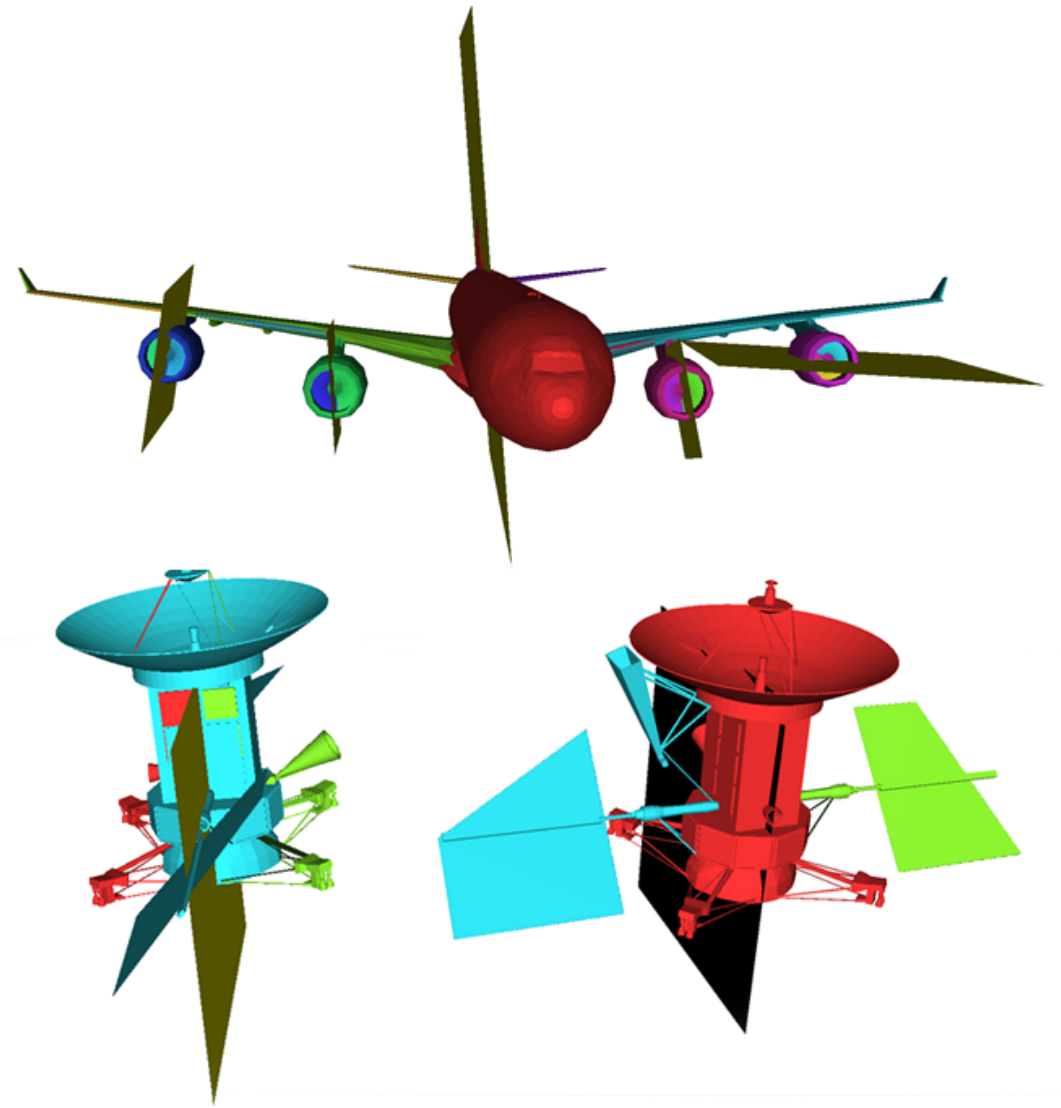


Figure 5.6. Shapes with ambiguous symmetries. The blades of an airplane engine are symmetric in multiple ways (top). The Magellan space probe (bottom) is symmetric in two ways, while only one is desirable in the context of the category as it separates the solar panels (bottom right).

Shape Hierarchy. There isn't a simple way of making sure the hierarchies are correct, but it can be seen in figure 5.7 that the most important features work as intented. The nodes that are split appear on the lower levels of the hierarchy and when no splitting is done, the node remains on the next level. The enforcement of non-disconnected parts works well too as can be seen in figure 4.2, which is comparable to 2.3 at least for the coarser levels.

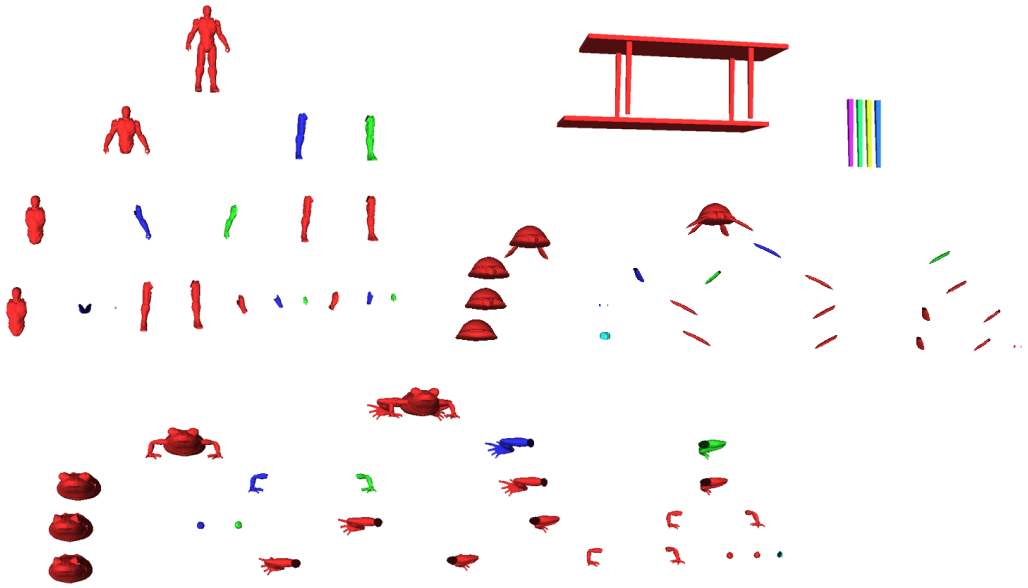


Figure 5.7. Hierarchies of shapes from various categories. The first child node is always in red and the other ones sharing the same parent are in other colors.

Performance. The table below shows the durations of different operations done for shapes of varying faces count and from various categories. The results show that a great portion of the time is spent during the segmentation that is done by the external CGAL code that can't really be further optimized. The other procedures also take a considerable amount of time, especially contact analysis and hierarchy creation, but the overall duration of nearly three minutes for one of the bigger shapes is still manageable, because the analysis needs to be only once and can be reused almost instantly if it is serialized.

	bunny	frog	ankylo	rosetta	boeing757
Mesh Faces	633	13216	10230	16901	108986
Mesh Area	0.424	0.423	0.608	0.141	0.323
Load & Preparation [ms]	90	163	273	570	1630
Mesh Repair [ms]	10	118	103	284	1521
Segmentation [s]	7.8	11.3	10.3	29.9	116.6
Contact Analysis [s]	14.6	12.4	21.1	14	45.8
Hierarchy Creation [s]	0.7	2.2	6	130.1	102.2
Overall duration [s]	23.2	26.2	37.9	175	267.7

Table 5.1. Performance of the shape analysis for shapes of varying complexity.

5.2 Synthesized Shapes

It is always possible to generate an interpolation between two shapes, but the results have a varying quality depending on the similarity between the two input models. The main reason is the fact that during the shape matching procedure part redistribution using eigen-transforms could not be implemented even with the greatest amount of effort expended. A different solution also mentioned in [Jain et al. 2012] is simple comparison of a part and node centers, which is used instead and it unfortunately doesn't always yield the best results for shapes that are not very similar. Also the contact enforcement

using mass-spring system or a similar method wasn't implemented due to lack of time, which would further improve the shapes created. The results of shape synthesis from the categories of animals, dinosaurs, humanoids and space aircraft can be seen in figures 5.8, 5.9, 5.10 and 5.11.

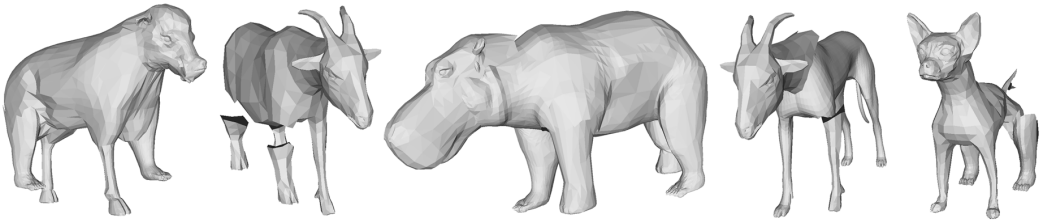


Figure 5.8. Shapes created from the models in the animal category.

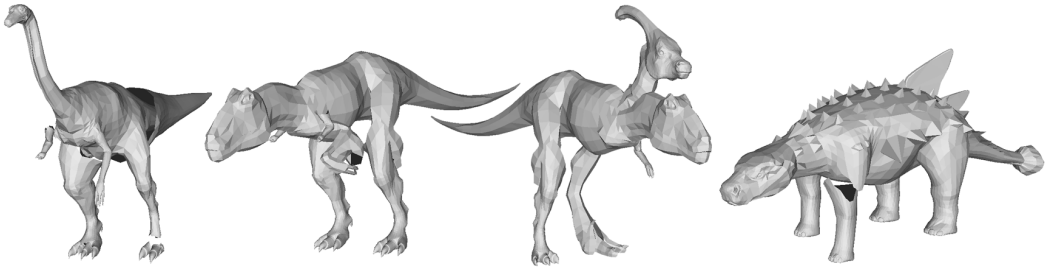


Figure 5.9. Shapes created from the models in the category of dinosaurs.

Figure 5.10. Shapes created from the models in the humanoid category.

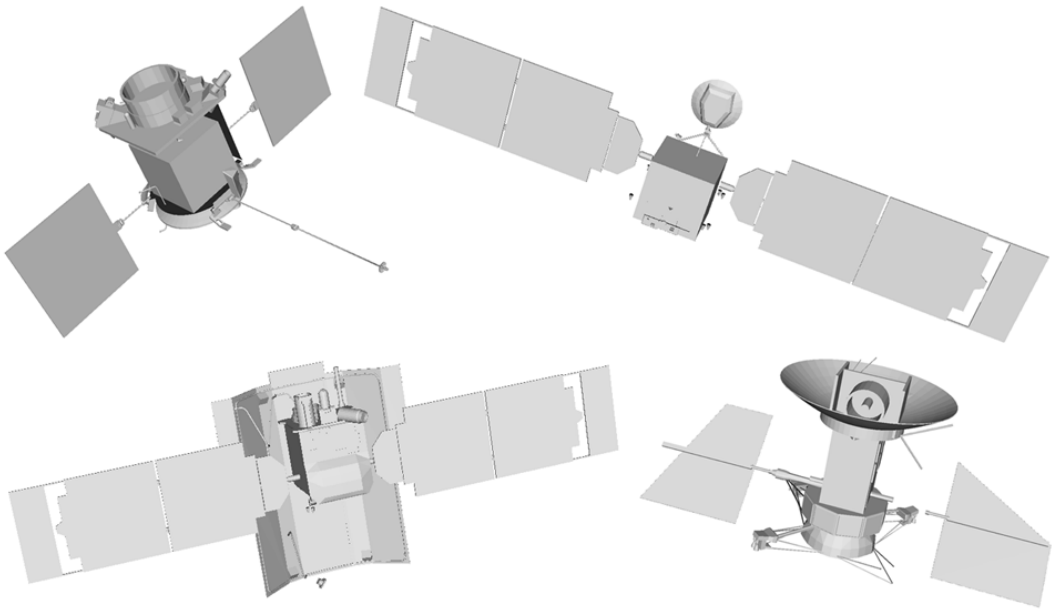


Figure 5.11. Shapes created from the models in the spacecraft category.

Chapter 6
Conclusion

Three methods of example-based 3D-model synthesis [Jain et al. 2012, Zheng et al. 2013, Kalogerakis et al. 2012] were reviewed and compared. The method by [Jain et al. 2012] was chosen for implementation for its ease of use, undemanding input shape requirements and flexibility and it was further studied.

Many useful open-source external libraries dealing with linear algebra, shape segmentation, point cloud sampling, mesh processing and user interface creation were found and employed, which greatly helped with the whole process of implementation.

Shape analysis consisting of mesh segmentation, point cloud sampling, symmetry and contact detection was implemented. For the shape segmentation an implementation of the SDF segmentation created by Shapira et al. was used at first. The integration wasn't simple and took approximately two weeks of time to do. The segmentations produced also suffered from many errors. It was then later found that a CGAL implementation was released at the beginning of April, which took about two days to integrate and produced much better results.

The application is able to produce interpolations for a pair of shapes with a varying quality for four categories (animals, dinosaurs, humanoids and spacecraft). Unfortunately the meshes used in [Jain et al. 2012] are not freely available [1]) so a direct comparison between the interpolated shapes can't be made. The redistribution of the parts using parent node eigen-transforms lead to strange results and had to be substituted with redistribution using part centers only, which is a method also mentioned in [Jain et al. 2012] but not as sophisticated. Also the contact enforcement wasn't implemented. Due to these two facts it is probably safe to assume that the shapes wouldn't be very similar for greatly dissimilar inputs. Similar inputs, on the other hand, yield acceptable results. Although the article [Jain et al. 2012] was pretty thorough there were some vague parts that most likely prevented an exact realization of the method. The authors were contacted to make the misunderstandings clear but no answer was received.

Future Work. The part redistribution using eigen-transforms could be fixed to really unleash the potential of the shape interpolation. Contact enforcement method such as a mass-spring system could also be implemented to improve the plausibility of the meshes created. To do this, the hierarchy of the created shape would need to be updated to use the contacts of the source and target shapes. A mass-spring system would then be applied to enforce the contacts of the newly created shape. Vega FEM library's[2]) implementation of a mass-spring system could be used.

Although the performance of the application isn't bad, there is always room for improvement. As mentioned in [Jain et al. 2012] contact analysis could be done using an AABB tree as explained in [van den Bergen et al. 1998].

[1]) `http://www.doschdesign.com/products/3d`
[2]) `http://run.usc.edu/vega/`

The input shapes could be refined using a Winding Numbers application [Jacobson et al. 2013] to ensure that any mesh could be used as the shape segmentation from the CGAL library requires polyhedral surfaces as its input.

The other reviewed methods [Zheng et al. 2013, Kalogerakis et al. 2012] could be implemented and integrated into the application to allow the user to synthesize new shapes even if the results of the other methods aren't plausible, because each of the methods is expected to work best for a limited range of shape categories. These two methods require segmented and annotated shapes so other shape segmentation methods [Huang et al. 2011, Sidi et al. 2011] could be implemented as well.

References

[Attene and Falcidieno 2006] Attene, M., and Falcidieno, B. 2006. Remesh: An interactive environment to edit and repair triangle meshes. In *Shape Modeling and Applications, 2006. SMI 2006. IEEE International Conference on*, IEEE, 41–41.

[Cheeseman and Stutz 1996] Cheeseman, P., and Stutz, J., 1996. Bayesian classification(autoclass):theory and results.

[Fu et al. 2008] Fu, H., Cohen-Or, D., Dror, G., and Sheffer, A. 2008. Upright orientation of man-made objects. *ACM Trans. Graph. 27*, 3.

[Huang et al. 2011] Huang, Q., Koltun, V., and Guibas, L. 2011. Joint shape segmentation with linear programming. *ACM Trans. Graph. 30*, 6 (Dec.), 125:1–125:12.

[Jacobson et al. 2013] Jacobson, A., Kavan, L., , and Sorkine-Hornung, O. 2013. Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH) 32*, 4, 33:1–33:12.

[Jain et al. 2012] Jain, A., Thormählen, T., Ritschel, T., and Seidel, H.-P. 2012. Exploring shape variations by 3d-model decomposition and part-based recombination. *Comp. Graph. Forum 31*, 2pt3 (May), 631–640.

[Kalogerakis et al. 2012] Kalogerakis, E., Chaudhuri, S., Koller, D., and Koltun, V. 2012. A probabilistic model for component-based shape synthesis. *ACM Trans. Graph. 31*, 4 (July), 55:1–55:11.

[Mitra et al. 2006] Mitra, N. J., Guibas, L., and Pauly, M. 2006. Partial and approximate symmetry detection for 3d geometry. *ACM Transactions on Graphics (SIGGRAPH) 25*, 3, 560–568.

[Shapira et al. 2008] Shapira, L., Shamir, A., and Cohen-Or, D. 2008. Consistent mesh partitioning and skeletonisation using the shape diameter function. *Vis. Comput. 24*, 4, 249–259.

[Sidi et al. 2011] Sidi, O., van Kaick, O., Kleiman, Y., Zhang, H., and Cohen-Or, D. 2011. Unsupervised co-segmentation of a set of shapes via descriptor-space spectral clustering. *ACM Trans. on Graphics (Proc. SIGGRAPH Asia) 30*, 6, 126:1–126:10.

[van den Bergen et al. 1998] van den Bergen, G., Van, G., and Bergen, D. 1998. Efficient collision detection of complex deformable models using aabb trees. *J. Graphics Tools 2*.

[Wang et al. 2010] Wang, Y., Xu, K., Li, J., Zhang, H., Shamir, A., Liu, L., Cheng, Z., and Xiong, Y., 2010. Symmetry hierarchy of man-made objects.

[Yaz and Loriot 2000] Yaz, I. O., and Loriot, S. 2000. Triangulated surface mesh segmentation. In *CGAL User and Reference Manual*, 4.4 ed. CGAL Editorial Board.

[Zheng et al. 2013] Zheng, Y., Cohen-Or, D., and Mitra, N. J. 2013. Smart variations: Functional substructures for part compatibility. *Computer Graphics Forum (Eurographics) 32*, 2pt2, 195–204.

Appendix A
List of Abbreviations Used

3D	Three-dimensional
AABB	Axis-aligned bounding box
C#	C# programming language
C++	C++ programming language
C++11	The most recent version of C++
CGAL	Computational Geometry Algorithms Library
GUI	Graphical User Interface
OBJ	Wavefront .obj File
OFF	Object File Format
OpenGL	Open Graphics Library
PCA	Principal Component Analysis
PCS	Precalculated Shape File
PDF	Portable Document Format
PLY	Polygon File Format
Qt	Qt User Interface Framework
SDF	Shape Diameter Function
sFarr	Symmetric Functional Arrangements
SVN	Apache Subversion
VCG	Visualization and Computer Graphics Library
VRML	Virtual Reality Modeling Language

Appendix B
Contents of the Attached DVD

```
DVD/
 bin/           Executable binaries with necessary DLL files
 lib/           Compiled libraries used
 models/        Input models with their precomputed versions
 src/           Visual Studio 2013 solution
 tex/           TeX source codes of the thesis
 thesis.pdf     The thesis in PDF
 README.txt     Information about the DVD and compilation
```

Appendix C
List of Important Parameters

Name	Symbol	Source Code Name	Value
Number of Clusters	–	number_of_clusters	7
Smoothing Lambda	–	smoothing_lambda	0.15
Sample Count Multiplier	μ	SAMPLE_COUNT_MULTIPLIER	100 000
Minimal Sample Count	ω	MINIMAL_SAMPLE_COUNT	200
Contact Threshold	–	CONTACT_THRESHOLD	0.001
Part Similarity Threshold	–	SIMILARTIY_THRESHOLD	0.025
Symmetry Quality Threshold	–	SYMMETRY_QUALITY_THRESHOLD	0.8