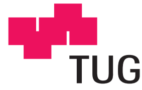


Graz University of Technology

Institute for Computer Graphics and Vision

Master's Thesis

Interactive Decomposition of Large Assemblies

Bernhard Kerbl

Graz, Austria, November 2013

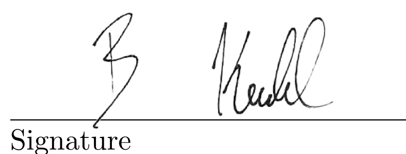
Thesis supervisors

Univ.-Prof. Dipl.-Ing. Dr. techn. Dieter Schmalstieg

Dipl.-Ing. Dr. techn. Denis Kalkofen

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.



Graz, Austria	**November 5, 2013**	
Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, Österreich	**5. November, 2013**	
Ort	Datum	Unterschrift

Abstract

This thesis presents a selective disassembly planning system that is capable of constructing animated step-by-step instructions for solving a user-defined disassembly problem. In contrast to complete disassembly, the system focuses on detecting sequences that lead to the efficient removal of a limited number of specified parts. The assembled products for disassembly planning are represented by a list of polygon meshes that correspond to the individual parts contained in the assembly. The employed algorithms have been optimized to allow handling of complex products with a large number of parts and high geometric detail. The system is divided into two separate modules. The first module performs detailed analysis of the product and extracts relational information that can then be assessed in an interactive planning application that constitutes the second module. By utilizing the parallel computing capabilities of modern graphics processing units, we achieve high performance during assembly analysis. A tolerance mechanism has been incorporated into the system to account for imprecisions and artefacts in the available mesh data. Computed disassembly sequences consist of translating motions that result in the exposure of all specified parts for removal. A graphical user interface provides means for visualization and extensive editing of disassembly sequences. By restricting the space of tested motions to a discrete set of translations, the system can provide immediate visual feedback to all changes made by the user. During animation of the step-by-step instructions, visual cues are employed to highlight important aspects of the disassembly procedure. The assemblies that have been examined in our system contain up to 512 parts and 700,000 triangle primitives.

Keywords. disassembly planning, parallel computing, animation, user interaction

Kurzfassung

Die vorliegende Arbeit präsentiert ein System zur Planung von selektiven Zerlegungsprozeduren. Die nötigen Schritte um ein benutzerdefiniertes Zerlegungsproblem zu lösen werden animiert dargestellt. Im Gegensatz zur vollständigen Zerlegung liegt das Hauptziel dieses Systems im Erkennen von Sequenzen die das effiziente Entfernen einer limitierten Anzahl von Bauteilen erlauben. Die analysierten Baugruppen bestehen jeweils aus einer Liste von polygonalen Oberflächengittern, welche die einzelnen Bauteile des fertigen Objekts darstellen. Die verwendeten Algorithmen wurden für die Anwendung auf komplexe Produkte mit einer hohen Anzahl von Bauteilen und detaillierter Geometrie optimiert. Das System besteht aus zwei getrennten Modulen. Das erste Modul führt eine detaillierte Analyse der Baugruppe durch und extrahiert relationale Informationen welche im zweiten Modul, der interaktiven Planungsapplikation, verarbeit werden. Durch die Verwendung von paralleler Berechnung mithilfe von modernen Grafikprozessoren wird die beanspruchte Zeit für die Analysephase erheblich verkürzt. Um Ungenauigkeiten und anderen Artefakten der verwendeten Flächengitter vorzubeugen, wurde eine Methode zur Berechnung von Bauteilen mit Toleranz eingeführt. Die errechneten Zerlegunssequenzen bestehen aus verschiebenden Bewegungen, welche alle gesuchten Bauteile freilegen. Eine Bedienungsoberfläche wird bereitgestellt um Sequenzen zu visualisieren und zu bearbeiten. Durch die Beschränkung auf eine diskrete Menge von möglichen Bewegungsrichtungen ist die Evaluierung aller vom Benutzer durchgeführten Veränderungen in Echtzeit möglich. Während der Animation der einzelnen Instruktionen werden visuelle Hilfestellungen verwendet um unscheinbare oder verdeckte Bauteile während der Entfernung herauszuheben. Die Baugruppen, welche in unserem System evaluiert wurden, bestehen aus bis zu 512 Bauteilen und 700,000 Dreiecken.

Schlagwörter. Zerlegungsplanung, Parallele Berechnung, Animation, Benutzerinteraktion

Contents

1 Introduction **1**
1.1 Disassembly Planning System 2
1.2 Program Workflow 5
1.3 Structure of Thesis 5

2 Related Work **7**
2.1 Terms and Concepts 8
2.2 Configuration Space 9
2.3 Assembly & Disassembly Planning Systems 12

3 Concept **27**
3.1 Challenges for Disassembly Planning 27
3.1.1 Geometrically Feasible Disassembly Paths 27
3.1.2 Complexity of Removal Actions 29
3.1.3 Tolerance 30
3.1.4 Partitioning 32
3.1.5 Real World Constraints 34
3.1.6 Editing Disassembly Paths 35
3.1.7 Disassembly Sequence Visualization 37
3.2 System Structure 37
3.3 Preprocessing Module 38
3.3.1 Setting Fixed Parts and Separators 38
3.3.2 Contact Information 39
3.3.3 Part Groups Generation 39
3.3.4 Mesh Shrinking 40
3.3.5 Blocking Relationships 42
3.3.5.1 C-Space Object Generation 42
3.3.5.2 Singular Translating Motions 45
3.3.5.3 Dual Translating Motions 46
3.3.5.4 Bounding Box C-Space Objects and Separator Occlusion . . 48
3.4 Disassembly Planning Application 49

3.4.1 Disassembly Customization 50
3.4.1.1 Partitioning 51
3.4.1.2 Removal Actions 53
3.4.1.3 Extended Motions 53
3.4.2 Disassembly Path Computation 54
3.4.2.1 Iterative Part Removal 55
3.4.2.2 Dependency Detection 57
3.4.3 Correcting Disassembly Paths 59
3.4.4 Animation and Illustration 60
3.4.4.1 Exploded Disassembly Preview 60
3.4.4.2 Step-by-Step Animation 61

4 Implementation **65**
4.1 Preprocessing Module 65
4.1.1 CAD Data Conversion 66
4.1.2 Part Groups Generation 66
4.1.3 Iterative Mesh Shrinking 68
4.1.3.1 Minimal Distance Calculation 69
4.1.3.2 Cell Grid Creation 70
4.1.4 Contact Information 72
4.1.5 Polyhedral C-Space Evaluation 73
4.1.5.1 C-Space Object Generation 73
4.1.5.2 C-Space Object Evaluation 75
4.1.5.3 Separator Occlusion and Bounding Box C-Space Evaluation 79
4.1.5.4 Detecting Unities 80
4.1.6 Storing Static Disassembly Information 81
4.2 Disassembly Planning Application 81
4.2.1 Loading and Initial Partitioning 81
4.2.2 The Planning View 82
4.2.3 Disassembly Path Computation 83
4.2.3.1 Blocking Relationships Storage and Access 83
4.2.3.2 Iterative Partition Removal 84
4.2.3.3 Dependency Hierarchy 87
4.2.4 Explosion Diagram Preview 88
4.2.4.1 Partition Placement 88
4.2.4.2 Static Motion Blur 90
4.2.5 The Animation View 91
4.2.5.1 Phase Snapshots 91
4.2.5.2 Animation Paths 92
4.2.5.3 Animation Phase Preview 92
4.2.5.4 Visual Cues 92

4.2.6 Rendering Styles . . . 94

5 Examples and Discussion **95**

5.1 Assembly Data Sets . . . 95

5.1.1 Preprocessing Runtimes . . . 95

5.1.2 Required Storage . . . 97

5.1.3 Influence of Optimization Methods . . . 98

5.1.4 Influence of Mesh Shrinking . . . 98

5.2 Disassembly Examples . . . 99

5.2.1 Press . . . 99

5.2.2 Pneumatic 6-Cylinder Engine . . . 103

5.2.3 Drill . . . 105

5.2.4 Radial Engine . . . 108

5.2.5 Mecanum Wheel . . . 110

5.2.6 Aviation Engine . . . 114

6 Conclusion **119**

A Acronyms and Symbols **123**

Bibliography **125**

List of Figures

1.1 Press device assembly . . . 3
1.2 Explosion diagram of press assembly . . . 3
1.3 Step-by-step instructions for disassembling press device . . . 4

2.1 Using C-Space for robotic path planning . . . 10
2.2 Minkowski sum as generated from sliding ([20]) . . . 11
2.3 Simple product for demonstrating the AND/OR graph ([6]) . . . 13
2.4 AND/OR graph generated for the simple product assembly ([6]) . . . 14
2.5 Separable parts with contact faces and normal vectors ([35]) . . . 15
2.6 Inseparable parts with contact faces and normal vectors ([35]) . . . 15
2.7 Simple 5-part assembly with two DBGs for specified directions ([32]) . . . 17
2.8 3-part assembly with contacts and corresponding NDBG ([32]) . . . 18
2.9 2-part assembly with original and optimized NDBG ([26]) . . . 19
2.10 Program flow of the assembly planning tool by Thomas et al. ([30]) . . . 21
2.11 Stereographic projection of C-Space obstacles ([30]) . . . 22
2.12 Automatically generated assembly instructions for a product ([3]) . . . 23
2.13 Insets used to highlight assembly of small parts ([3]) . . . 24
2.14 Explosion diagram of a turbine model ([19]) . . . 25
2.15 Removal influence graph for a simple assembly ([29]) . . . 26

3.1 Local & global freedom . . . 28
3.2 Disassembly with singular translating motions . . . 29
3.3 Disassembly with dual translating motions . . . 30
3.4 Erroneously self-intersecting nut-and-bolt assembly . . . 31
3.5 Disassembly with partitioning . . . 32
3.6 Editing disassembly paths by defining designated removal actions . . . 36
3.7 Assembly with separating object . . . 39
3.8 Generating part groups . . . 40
3.9 Iterative mesh shrinking of a bolt nut with different parameters . . . 42
3.10 C-Space object generation with two input polygon meshes . . . 43
3.11 C-Space object generation with translated input objects . . . 44

3.12 C-Space object generation with rotated input objects 45
3.13 Tolerancing singular translating motions through perspective projection . . 46
3.14 Evaluating dual translating motions through orthographic projection 48
3.15 Optimizing blocking relationships by considering separators 50
3.16 General peeling algorithm 54
3.17 Storing blocking information for dual translating motions 56
3.18 Press assembly with exemplary disassembly problem 57
3.19 Dependency hierarchy for a given disassembly problem 58
3.20 Radial aircraft engine with exemplary disassembly problem 61
3.21 Explosion diagram for a given disassembly problem 62
3.22 Explosion diagram with applied static motion blur 62
3.23 DPA displaying the press assembly with animation phases 63
3.24 DPA during animation of an instruction for part removal 64

4.1 Effects of part grouping for preprocessing the Cylinder Block assembly . . . 67
4.2 Transformation of vertices during iterative mesh shrinking 69
4.3 Iterative mesh shrinking applied to the mesh of a knot 70
4.4 Iterative mesh shrinking applied to the mesh of a grate 70
4.5 Iterative mesh shrinking applied to the Stanford dragon 71
4.6 Space partitioning of input objects to speed up iterative mesh shrinking . . 72
4.7 Minkowski sum of two tessellated spheres with naive and robust culling . . 75
4.8 Showcase of test cases for parallel Minkowski sum generation 76
4.9 Influence of target resolution on evaluation of singular translating motions . 78
4.10 Bounding box C-Space objects and separator occlusion 80
4.11 Top-down and bottom-up examples of hierarchy-based explosion diagrams . 90

5.1 Accumulated runtimes for preprocessing stage 96
5.2 Accumulated disk space required for static assembly information 97
5.3 Illustration of the press assembly 100
5.4 Examined disassembly problem for press assembly and involved parts . . . 100
5.5 Explosion diagram for the press assembly 101
5.6 Step-by-step instructions for disassembling the press 102
5.7 Pneumatic 6-Cylinder Engine with illustrated disassembly problem 103
5.8 Initial explosion diagram for the pneumatic engine 104
5.9 Improved explosion diagram for the pneumatic engine 104
5.10 Dual translating motions used for disassembling the pneumatic engine . . . 105
5.11 Manual drill with illustrated disassembly problem 106
5.12 Editing designated removal actions for the drill assembly 107
5.13 Initial and improved explosion diagrams for disassembling the drill 108
5.14 Radial Engine with illustrated disassembly problem 109
5.15 Explosion diagram for disassembling the radial engine 110

5.16 Animation phase preview for removing small bolts in the radial engine . . . 111
5.17 Animation with visual cues for removing bolts in the radial engine 111
5.18 Mecanum Wheel with illustrated disassembly problem 112
5.19 Animation phase preview for removing rollers from the mecanum wheel . . 113
5.20 Explosion diagram for disassembling the mecanum wheel 114
5.21 Aviation Engine with illustrated disassembly problem 115
5.22 Explosion diagram for disassembling the aviation engine 116
5.23 Animation with visual cues for removing fixtures in the aviation engine . . 117

List of Tables

3.1 Requirements for AND/OR graph generation 34

4.1 Geometric attributes of the sample test cases for Minkowski sum generation 75

5.1 Geometric attributes of products chosen for disassembly 96

List of Algorithms

4.1 Peeling algorithm for iteratively removing free partitions 85
4.2 Detection of removable partitions 86
4.3 Method for choosing suitable removal action 87
4.4 Dependency hierarchy creation 89

Chapter 1

Introduction

Most of the more complex mechanical objects and designs of modern technology are comprised of several parts that interact dynamically to provide a certain functionality or statically to build a certain structure. These parts are usually created separately and later combined to form the final product. There is often more than one way to assemble or disassemble such products, based on the order and the manner in which parts are removed. Methods for computer-aided assembly and disassembly in computing aim to reveal and suggest suitable procedures for constructing or deconstructing such objects. Generally, complete and selective methods are distinguished, where the first is concerned with the complete disassembly of the object, while the latter focuses on finding minimal instruction sequences that only result in the removal of a given set of parts.

In computer graphics, disassembly algorithms are most commonly found in systems for the automatic generation of explosion diagrams or step-by-step assembly manuals. Most of these approaches exploit findings and methods from the domain of robotics in order to detect possible disassembly procedures and use them as the basis for visualization. Existing algorithms for finding and validating these procedures are usually strongly dependent on the shape information of the individual parts. Thus, a a suitable 3D representation is required for evaluation, such as CAD data sets or polygon meshes. The involved computational effort is usually directly proportional to the level of detail provided by these objects.

Whereas explosion diagrams focus on providing visually appealing part constellations to convey information about the structure of an object, generation of instruction manuals requires additional concern regarding the feasibility of each instruction. For large assemblies, the effort for avoiding visual clutter and finding valid instructions increases

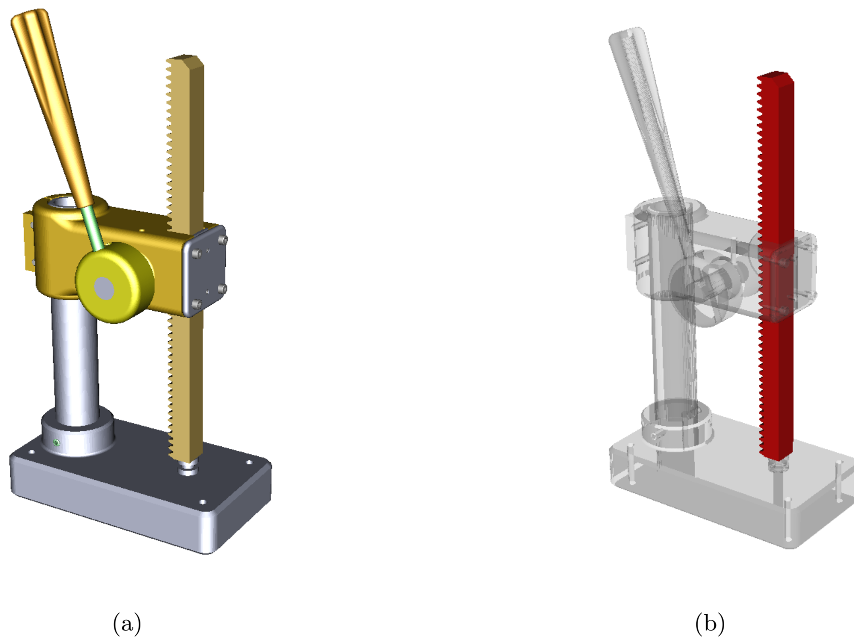
with the number of parts to the point where fully automatic methods become infeasible. Furthermore, a number of factors influence the perceived quality of a disassembly procedure, many of which are complex or may depend on the preferences of the user. Thus, user interaction is often key for extracting additional information about parts that can be used to reduce the search space for candidate disassembly procedures. However, even with support for user interaction, finding or verifying the most suitable path for disassembly can be a tedious task due to the high number of possible alternatives.

1.1 Disassembly Planning System

This thesis presents an interactive system for detecting, editing and visualizing possible disassembly sequences that result in the removal of one or more specified components from an assembled product. The underlying algorithms are designed to handle assemblies with several hundred parts and high geometric detail. The minimal user input to the system is a specific **disassembly problem**, which is defined by polygon meshes representing the parts in the assembly and the set of required components to be removed. An exemplary assembly is depicted in Figure 1.1(a), for which a disassembly problem is defined by the see-through triangle meshes and a part required for removal in Figure 1.1(b).

Any sequence of actions that leads to the removal of all required components is considered a viable solution to the disassembly problem. The system automatically calculates basic spatial constraints and conveys solutions to the user through suitable visualization and animation techniques. Suggested disassembly sequences that are not acceptable can be corrected or modified in real-time by setting parameters for each part via the user interface. To provide a quick preview of a possibly lengthy procedure, explosion diagrams that statically illustrate the decomposition can be generated (see Figure 1.2). For a detailed and lucid visualization of the necessary steps, animated instruction manuals are available (see Figure 1.3).

The presented system aims to support a high variety of assemblies by relaxing some of the most common requirements in previous approaches, such as numerical exactness of the input data or a low number of contained parts. By exploiting the graphics pipeline and parallel processing for computationally expensive calculations, we can reduce the runtime during critical steps of the assembly analysis. We use experimental evaluation methods to increase the probability of detecting feasible disassembly steps. The key method to the functional features provided lies with the disassembly path computation method, which evaluates the results of the preceding analyses and detects the most suitable disassembly



(a) (b)

Figure 1.1: (a) An assembly of a press device consisting of 23 parts. (b) Schematic illustration of a disassembly problem with the triangle meshes of the geometry as see-through models and a required part for removal (red).

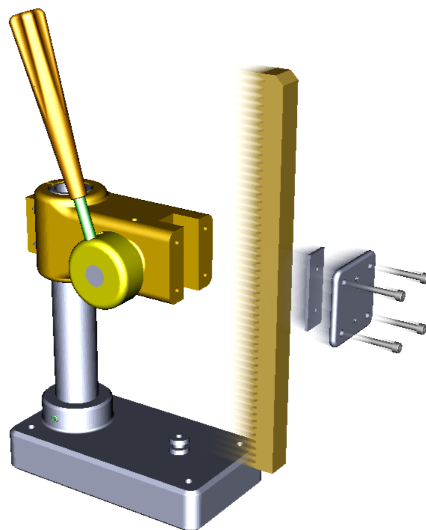


Figure 1.2: Explosion diagram representing a disassembly procedure for a disassembly problem. Static motion blur is applied to convey how each individual part is removed.

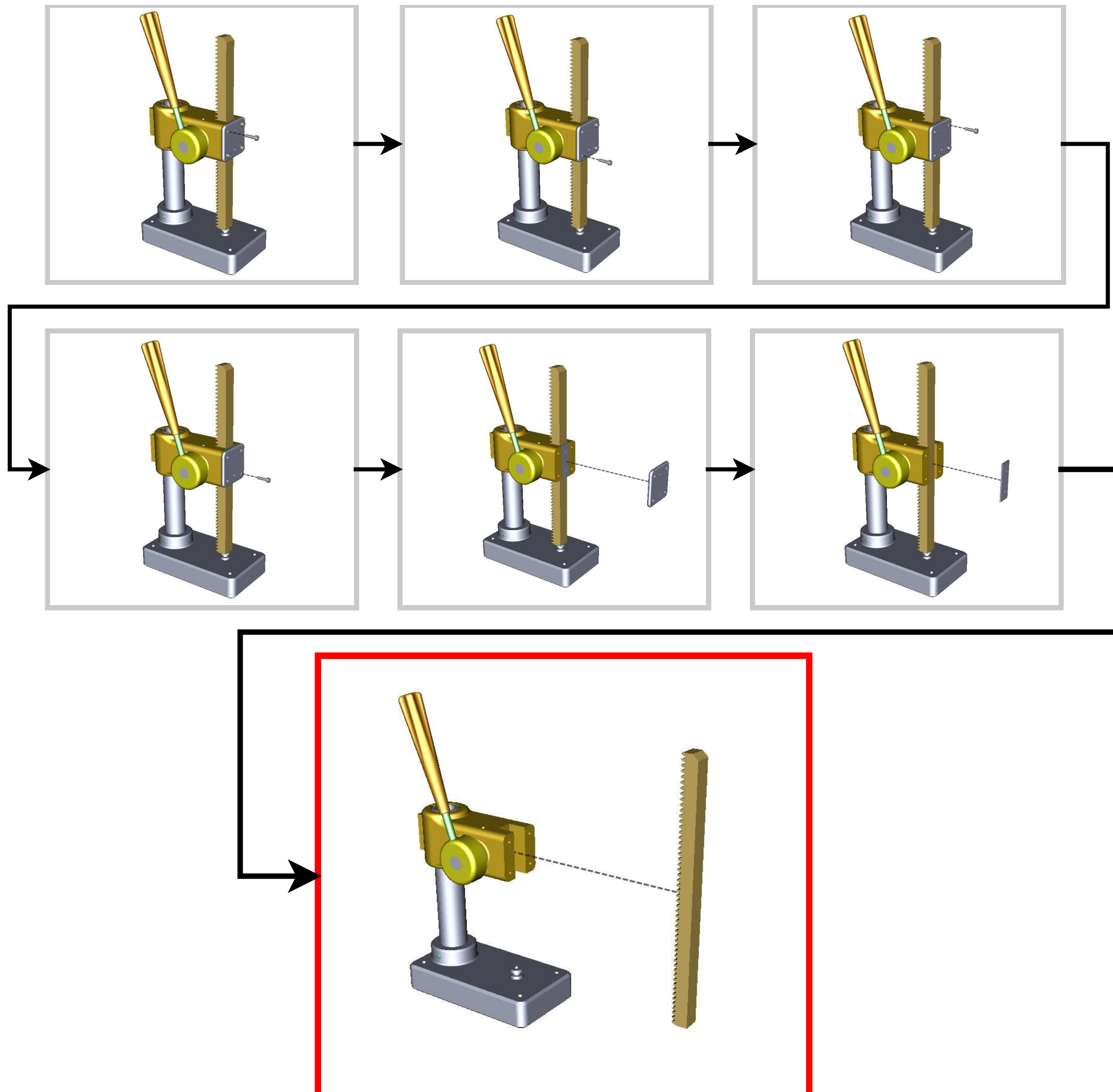


Figure 1.3: A series of instructions that were automatically created based on the disassembly problem illustrated in Figure 1.1(b). Disassembled parts are translated along the direction in which they were removed to separate them from the remaining assembly. Guidelines connect each translated part to its initial position to establish visual coherence.

procedure based on geometrical feasibility and user defined constraints. In contrast to most existing systems, we test part removal not only for singular translating motions, but also for dual translating motions. The provided disassembly planning interface allows for a high degree of customisability, thus enabling the user to recreate a wide range of possible sequences virtually in our system.

1.2 Program Workflow

The system consists of an automatic preprocessing module for handling computationally expensive analyses of assemblies and a planning application that processes the resulting information, as well as user input. By dividing the functionality of the system in this manner, the hardware requirements for running the planning application can be kept low, thus ensuring its compatibility with a variety of (possibly mobile) devices. An assembly that is input to the preprocessing module is analyzed regarding the potential interactions and relationships between parts. The analysis is based solely on the triangle meshes representing the individual parts in the assembly. We extract relational information about parts for a set of discrete removal actions and store them for further use in the planning application. Using standard rendering functionality, we evaluate blocking relationships, which represent the feasibility of performing translating motions on parts to move them past each other. In order to enable tolerance mechanisms when detecting these blocking relationships, we shrink the meshes and perform all ensuing evaluations using their reduced form. The preprocessing module explicitly exploits the features of the graphics pipeline to provide both high precision and acceptable runtimes, even when dealing with complex assemblies.

The disassembly planning application (DPA) combines the information extracted in the preprocessing phase with the knowledge supplied by the user to detect disassembly procedures based on a simple and fast peeling algorithm. The DPA provides an environment for changing properties of individual components in the assembly, thus affecting the outcome of the disassembly path calculation. Suitable methods for removing individual parts are either explicitly defined by the user or detected automatically during disassembly path computation. Once a solution has been found, the DPA can be used to illustrate it in several ways: we utilize explosion diagrams to convey the ramifications of changing made by the user by giving a single-frame preview of the complete process. Detailed depiction of the single instructions in chronological order is provided via step-by-step animation. In cases where no suitable solution to the disassembly problem can be found, the DPA helps with detecting and correcting the causes for failure.

1.3 Structure of Thesis

This thesis is structured into 6 main chapters. The following listing concludes the first introductory chapter. In Chapter 2, we summarize publications and documents from

related work in the field of disassembly planning and corresponding supplemental material. The characteristics and potential problems of different assemblies, as well as possible solutions and eventual approaches taken in our system concept, are being discussed in Chapter 3. The actual implementational details of our system are outlined in Chapter 4, along with examples and preliminary measurements demonstrating the functionality of algorithms that were specifically devised to handle large assemblies. A showcase of exemplary assemblies, as well as numerical and qualitative evaluation of the corresponding disassembly procedures generated by our system can be found in Chapter 5. Chapter 6 concludes this thesis by summarizing the insights made in the previous chapters and lists open problems that provide potential material for future research in the field of computer-aided disassembly planning.

Chapter 2

Related Work

This thesis describes an interactive disassembly planning system that is capable of handling highly complex assemblies and seeks to provide qualified features for editing and visualizing selected disassembly procedures. A suggested procedure is only valuable to a user if it can be performed in a corresponding real-world setup, i.e. if it is naturally feasible. However, the same is also true for the inverse problem of assembly planning. It is trivial to see that in the majority of cases, reversing the procedure for disassembling a product will yield a valid way for assembling it. In fact, finding a disassembly procedure and inverting it is the most common approach to obtaining step-by-step instructions for assembly [2, 3, 13, 14, 18, 35]. Thus, the challenges for disassembly planning are basically congruent with those of assembly sequencing. Consequently, most findings from the domain of computer-aided assembly can be directly incorporated in disassembly planning systems. For the remaining part of this chapter, the commutability of assembly and disassembly planning will not be pointed out explicitly when describing contributions from either domain.

Most assemblies cannot be decomposed in any arbitrary order, but rather exhibit a certain dependency of parts. For instance, the contents of a box that is covered by lid may only be accessed once said lid has been lifted. Similarly, a bolt cannot be removed from an assembly unless its corresponding nut has been loosened as well. Although these associations may seem trivial to an observing human individual, they require involved algorithms to automatically register in a software solution. Where such methods are not available or fail to identify possible solutions, user interaction can provide additional information to allow the program to proceed. However, relying on user input alone makes disassembly planning a tedious task and may render it practically infeasible for untrained individuals to handle large assemblies due to their complexity.

Several approaches have been examined and evaluated, ranging from fully automatic to mostly input-dependent programs. The results obtained by individual implementations usually differ in terms of required runtime for processing a given assembly and the overall versatility of the system. This chapter presents supplemental material to provide a basic understanding of a concept called the "configuration space" and the associated algorithmic challenges, as well as a selection of notable publications that describe the functionality, traits and performance of existing assembly/disassembly planning systems .

2.1 Terms and Concepts

In this thesis, the term **part** refers to the physical or logical representation of a single indivisible component in an assembly (e.g. a screw or a bolt). The terms **partition** and **subassembly** both denote a non-empty group of distinct parts, although a partition is usually the result of splitting an assembly into non-intersecting groups of parts, whereas subassembly may define any conceivable subset of the complete assembly. The smallest possible partition has a size of one, which equates to the logical representation of a single part. Partitions are labeled using $\boldsymbol{P_x}$, where $\boldsymbol{x}$ may be replaced with a letter or number, based on the context in which it is used. $\boldsymbol{P_0}$ always denotes the partition that contains all other partitions, and thus represents the entire assembly. Except for $\boldsymbol{P_0}$, each partition possesses a parent partition, from which it can be broken off during disassembly. Two partitions sharing the same parent partition may be referred to as **siblings**. A part or a partition may be described as **active**, if it has not been removed from its parent partition. **Inactive** partitions and parts denote subassemblies that have been disassembled and no longer have any influence on other subassemblies.

Removal action denotes an action that results in the removal of a subassembly from its parent partition. Note however, that the removed subassembly itself may very well be decomposed further to reveal yet smaller subassemblies it contains. Thus, the contents of a partition that has already been removed may still be considered in ensuing steps. In this thesis, we only consider removal actions that are comprised of translating motions, unless stated otherwise.

A **blocking relationship**, formally represented as $B_r(A, B, \omega)$, defines a boolean value that states whether a specified part A is blocked by another part B when using removal action ω. If the evaluation of a blocking relationship returns true, it is said to be positive, otherwise we refer to it as being negative. Since groups of boolean values can be efficiently defined via the bits of larger data structures, the blocking relationships for one part A and

a removal action ω with all other parts may be stored in a chunk of allocated memory. The blocking relationships of a part A can thus be envisioned for each possible removal action ω as a sequence of binary values, where each value represents the positive or negative evaluation for this action with another part B. A part B where $B_r(A, B, \omega)$ is true may be referred to as a **blocker** of ω for A. If there exists any removal action ω for A such that $B_r(A, B, \omega)$ is true, A can be said to be **influenced** by B.

Contact-coherence describes the condition for a group of parts that is fulfilled if each part is directly or indirectly connected with every other part in the group. A direct connection is present, if two parts are in direct spatial contact with each other. An indirect contact between two parts is present if by means of recursively following direct contacts a sequence of traversals exists which links the two parts.

The terms **disassembly sequence** and **disassembly path** are used interchangeably throughout this thesis, both describing a unique solution to a given disassembly problem. However, **disassembly sequence** bears a strong relation to the exact instructions given in chronological order (e.g. during disassembly animation), whereas **disassembly path** denotes one specific solution from a potentially large number of alternatives (e.g. during disassembly planning).

2.2 Configuration Space

The concept of the configuration space (C-Space) and its application to spatial planning is discussed by Lozano-Perez in [22]. Given a target object and a set of obstacles, the C-Space encodes at each location whether the corresponding configuration is feasible for the target without causing intersection or collision. This is achieved by expanding all obstacles using Minkowski differences such that the target itself can be coherently reduced to a point. The Minkowski difference of two parts $A \ominus B$ is equivalent to the Minkowski sum with one part inverted $\neg A \oplus B$. The feasibility of moving the target from one point to another in a straight line can be conveniently evaluated by connecting these two points and intersecting the resulting line segment with the expanded C-Space objects. If no intersections are found, the translation can be performed without interference by any obstacle. By concatenating a set of feasible translations, more complex paths can be computed. Figure 2.1 illustrates an example of using C-Space obstacles for efficient path planning in robotics.

Due to their similarity and the fact that the Minkowski sum is more commonly explored in contemporary research than the Minkowski difference, the following paragraphs will

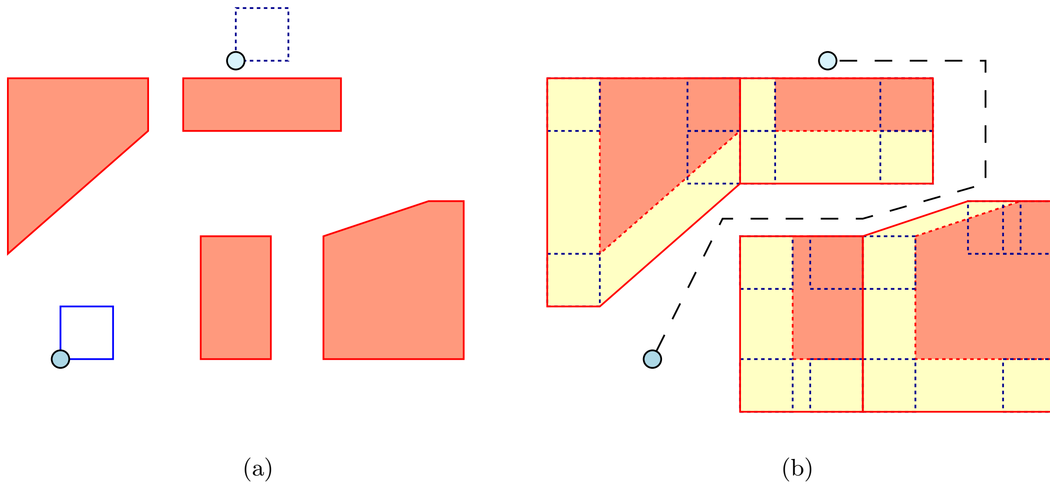


Figure 2.1: (a) A scene containing a robot (blue solid), conceptually represented by a square, that is to be moved from its initial position to a target location (blue dashed) without interference by the obstacles (red). A reference point on the robot (light blue) is chosen to establish the coordinate system for C-Space computation. (b) The generated C-Space obstacles (red+yellow) occupy more space. In order to detect a feasible path for the robot, it suffices to establish a sequence of line segments that connect the two reference points without intersecting the expanded obstacles.

focus on concepts and findings related to the generation of polyhedral Minkowski sums exclusively.

The Minkowski sum of two bodies A and B is formally defined by

$$A \oplus B = \{a + b \mid a \in A, b \in B\} \tag{2.1}$$

The visual interpretation can be understood as the result of ”sliding” one object along the perimeter of the other. Figure 2.2 illustrates the Minkowski sum as the result of sliding along the perimeter of a polygon.

The calculation of the Minkowski sum can be efficiently extended to 3D space for convex polyhedra [7]. A straight-forward implementation can be achieved by generating all possible vertices according to Equation 2.1 and constructing the convex hull. However, computing the Minkowski sum of two non-convex polyhedra is less trivial. Most existing approaches to this problem are based on one of two methods: convex decomposition and convolution [20]. Although the basic concepts of both methods are relatively simple, the primary challenges for implementing a solution for general Minkowski sums lies with ensuring the robustness of the underlying calculations [21].

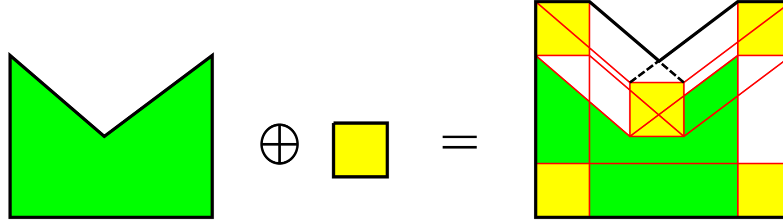


Figure 2.2: Generation of the Minkowski sum of two polygons by sliding one along the perimeter of the other. Edge candidates are created for each vertex of the yellow square as it moves along the edges of the green polygon. The new boundary is established by detecting the outermost edges and resolving intersections (adopted from [20]).

One example for Minkowski sums through convex decomposition is given in [9]. By decomposing polyhedra into several convex subsets such that the union of these sets yields the original, the Minkowski sum of two non-convex bodies A and B is calculated by repeatedly applying an algorithm for convex polyhedra to all pair-wise combinations (α, β) where $\alpha \in convex_subsets(A)$ and $\beta \in convex_subsets(B)$. The implementation is released as part of the CGAL framework and can employ 3D Nef polyhedra and precise predicates in order to obtain robust results [1]. Due to the fact that exactness is ensured at all stages, the algorithm requires several minutes to process polyhedra featuring ∼1,000 facets on contemporary hardware [9].

A faster, less precise convolution-based implementation is described by Lien [21]. A set of potential candidate facets for the Minkowski sum of two polyhedra is generated and pruned in a brute-force manner using a set of geometric criteria. A bounding volume hierarchy is created to efficiently detect all intersections in the remaining facets. Each facet is then split into subfacets along its intersections. The generated subfacets are iteratively stitched together to form simple regions that are either entirely located on the boundary of the Minkowski sum or completely contained inside. In a final step, all interior regions are identified using collision detection and consequently discarded, thus leaving only the regions that comprise the boundary of the Minkowski sum. Although this method produces nearly-exact results and is substantially faster than most previous approaches, processing may take several seconds or even minutes for input objects with ∼10,000 facets [21].

To reduce the runtime complexity of the task, Minkowski sums can be approximated to obtain fast results. Varadhan and Manocha present one approach that is based on convex decomposition in [31]. For all pairs of decomposed subsets, the Minkowski sum is calculated as the convex hull of all candidate vertices. A variation of the Marching Cubes algorithm is employed to approximate the boundary of their union. Several criteria are

tested to cull non-contributing primitives. The algorithm achieves accurate approximations of the exact results. However, the performance gain is rather small when compared to the approach by Lien.

Li and McMains propose a parallel algorithm for computing voxelized Minkowski sums [20]. Previous to the voxelization stage, their implementation generates a Minkowski sum that is suitable for rendering. For each edge and vertex, a number of conditions is tested based on their local topology. The necessity of performing these tests restricts the input to proper 2-manifolds without artifacts. All candidate primitives that may contribute to the appearance of the Minkowski sum are created and tested using a set of culling criteria. The authors provide four propositions with corresponding mathematical proof to reduce the number of candidates to less than 1%. Although not all redundant primitives may be culled, the appearance of the result is identical to the exact Minkowski sum when viewed from outside, since all superfluous primitives are contained exclusively on the inside of the boundary. A parallel implementation for CUDA is outlined which may be easily reproduced. The corresponding results state that generating Minkowski sums of polyhedra with ∼100,000 facets for rendering is feasible in a matter of seconds if parallel computation can be employed.

2.3 Assembly & Disassembly Planning Systems

Natarajan and Kavraki et al. provide early research into the feasibility of disassembly planning, as well as the limitations and complexity measures for assemblies [16, 24]. In particular, Kavraki et al. focus on the complexity of the assembly partitioning problem. Assembly partitioning in this respect describes the procedure of finding all valid ways to break up an assembly into smaller, detachable groups of parts which can be removed one after another to achieve complete product disassembly. It is shown that the problem is NP-complete if no additional constraints are added to reduce the search space. However, the algorithmic complexity may be considerably reduced by only allowing contact-coherent operations. The authors suggest that with regard to this constraint, significantly lower runtime complexity may be achieved.

In [6], De Mello and Sanderson analyze the drawbacks to storing assembly plans as lists of actions and outline the benefits of using the AND/OR graph structure for representing the possible ways to partition assemblies into all possible subassemblies. The AND/OR graph offers a compact method for encoding the relations between sets of parts, and how they can be split into smaller partitions. The AND/OR graph can be understood as a tree,

where the root represents the assembled product. It is comprised of a number of nodes and hyperarcs which connect one predecessor node with two or more successor nodes. Each hyperarc denotes a specific disassembly operation that is performed on the partition represented by the predecessor node. The successor nodes to which the hyperarc points are the respective separated child partitions that result from performing the corresponding disassembly operation. The computational effort for generating and storing the contents of the AND/OR graph can be significantly reduced by linking all references to a specific partition to the same node. Thus, multiple hyperarcs may point to the same successor node, which also reduces visual clutter if the AND/OR graph is visualized. The authors demonstrate the concept using a simple example assembly consisting of four parts (see Figure 2.3). The corresponding AND/OR graph is depicted in Figure 2.4.

Lambert focuses on optimizing the generation of the AND/OR graph using formal precedence relations to reduce the complexity of the problem [17]. The list of all partitions in the AND/OR graph is created by intersecting the set of connected subassemblies with the set of subassemblies that were found to be detachable according to the precedence relations. A compact representation of all possible disassembly procedures can then be extracted and evaluated. Furthermore, a simple algorithm for detecting optimal sequences is presented, based on a cost vector that defines a value for each part corresponding to the severity or the "cost" of its removal.

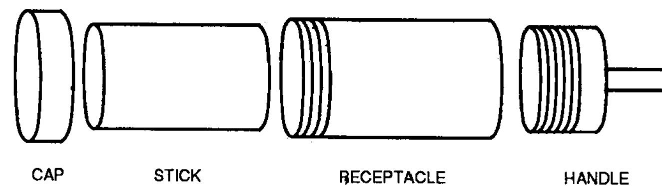


Figure 2.3: A simple product example used by De Mello and Sanderson to demonstrate the use of the AND/OR graph (adopted from [6]).

Targeting the problem of geometric reasoning for assemblies, Bourjault as well as De Fazio and Whitney have proposed methods for extracting blocking relationships and part dependencies based on a number of questions that have to be answered by the user [4, 5]. While the approach taken by Bourjault requires up to $2l^2$ yes-or-no questions where l denotes the number of parts in the assembly, De Fazio and Whitney have significantly reduced the number of posed questions to $2l$ by requiring the user to provide answers in precedence-logical form. The answers can be converted to create a number of relationship rules, which can be converted to generate all possible disassembly sequences. The

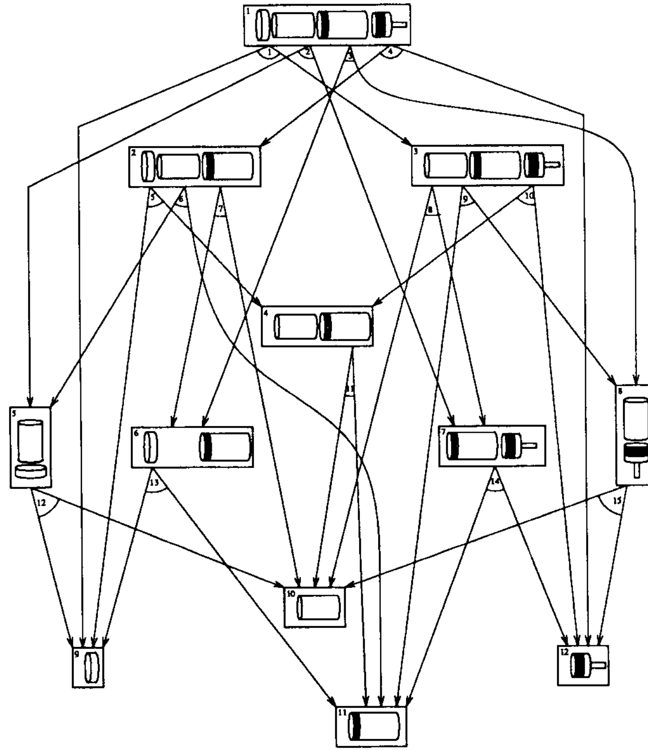


Figure 2.4: The AND/OR graph generated from analyzing the simple product assembly. Hyperarcs indicate the possible ways of splitting partitions into yet smaller partitions (adopted from [6]).

corresponding information can be stored in a liaison graph, which strongly resembles the AND/OR graph, although its focus lies more on the encoding of different assembly stages than providing information about detachable components.

Woo et al. proposed a complete algorithm that included automatic methods for efficiently establishing geometrical relationships and dependencies of parts in the assembly based on detailed analysis of their surface [35]. They introduce an undirected face adjacency graph (FAG), which lists all contact faces, i.e., surfaces where two distinct parts touch. Along with the two facets and the identifiers of their parts, the outward normal vectors of the planes supporting the facets are stored for each contact. If a part is removed from the assembly, the graph is updated and all entries referencing this part are discarded. A part is considered free, if all normal vectors of its contact faces in the graph lie in a common hemisphere. Examples of two possible part setups with corresponding contact faces and normal vector evaluation can be found in Figures 2.5 and 2.6 respectively.

All free parts are inserted into a disassembly tree, the traversal of which then yields

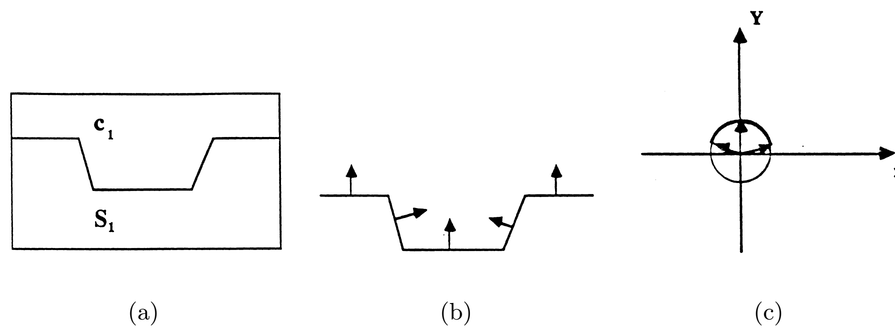


Figure 2.5: Two separable parts S_1 and c_1 where the normals of the contact faces lie in a hemisphere. Images adopted from [35].

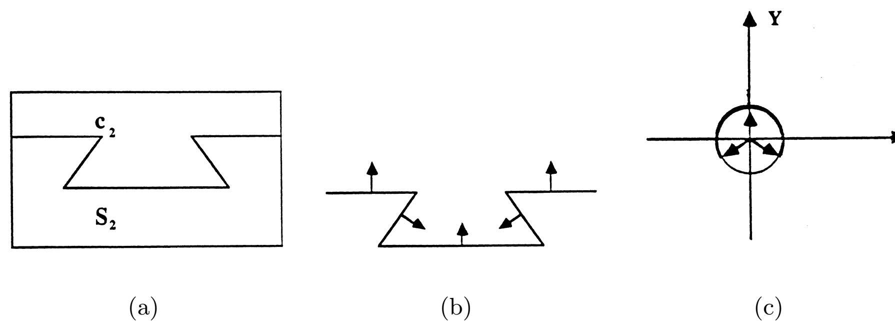


Figure 2.6: Two parts S_2 and c_2 that cannot be separated in 2D since there exists no hemisphere that contains all normal vectors of the contact faces. Images adopted from [35].

either a disassembly or an assembly sequence in linear time, depending on the direction in which it is being traversed. The implementation of such a sequencing system is easily reproducible and guarantees a low runtime complexity. However, the applicability of this approach is limited to straightforward assemblies that require only simple translating motions to remove individual parts. Furthermore, the algorithm is based on the assumptions that all parts can be removed one at a time, i.e. it is unable to detect solutions that require the removal of larger partitions in order to proceed.

Based on their research into AND/OR graph representation and the corresponding compact data structure for storing assembly paths, Homem de Mello and Sanderson have proposed a complete algorithm that incorporates mechanisms for geometric reasoning, while also allowing to evaluate paths that require complex partitioning [13]. By recursively dividing the assembly into all possible pairs of connected partitions, until only single parts

are left, all ways of partitioning the assembly are tested. For each of these generated partition pairs, the algorithm evaluates whether a valid translating motion exists to separate them. Testing the separability is based on the detection of contacts between the individual parts of both partitions. The authors define three classes of contact relationships for categorization, namely plane-plane, cylinder-bolt and cylinder-hole type contacts. These relationships are used to compute the space of feasible infinitesimal translating motions that do not cause the two partitions to collide. If such a motion exists, they are assumed to be separable. Thus, the partitions represent a valid partitioning of their union and the corresponding hyperarc is added to the AND/OR graph. The authors note, that this approach does not automatically consider obstacles that are not in direct contact with the part they block, which requires them to model these relationships using virtual contacts.

In [32], Wilson provides fundamental insight into the domain of computer-aided assembly planning. A detailed analysis of the challenges and previous solutions is given, as well as a formal context for describing assemblies. The main contribution is the highly involved assembly planning system GRASP, which incorporates algorithms for automatically finding geometrically feasible operations based on the shape of the input parts. The space of tested motions includes straight-line translations as well as rotations. The complete set of valid assembly paths is generated by building the AND/OR graph, which can subsequently be evaluated using a search heuristic or a best-candidate method to find a suitable assembly plan. In an analysis of basic assembly planning approaches, two implementations for generating the AND/OR graph under the constraint of contact coherence are provided as pseudo code. In order to improve on these generate-and-test methods, the author introduces the concept of two auxiliary data structures, namely the directional blocking graph (DBG) and the non-directional blocking graph (NDBG). Several classes of contact relationships are defined, which are used to create the contents of these data structures. A DBG encodes the blocking relationships between parts for a given direction. Parts are represented by nodes. A directed arc in the DBG indicates that the part from which it originates is blocked in the given direction by the part to which it points. An exemplary assembly with two DBGs for different directions is illustrated in Figure 2.7. Wilson further describes an algorithm for quickly detecting detachable and connected groups of parts in a DBG, thus demonstrating its suitability for generating the AND/OR graph.

A direction for which a DBG is built can be interpreted as a point on the unit sphere. The vector leading from the center of the sphere to the point lying on its surface represents

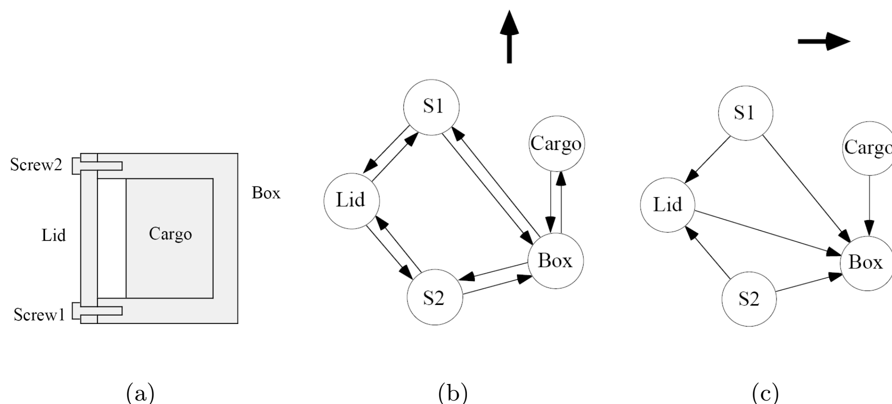


Figure 2.7: A simple assembly of a crate with cargo, consisting of 5 individual parts. Two DBGs represent the blocking relationships of the parts for the respectively specified direction. Images adopted from [32].

this direction. Since the number of points that can be placed on the unit sphere is infinite, so is the number of potentially dissimilar DBGs. The main purpose of the NDBG is thus to sensibly partition the unit sphere into separate regions for which a DBG should be created. By analyzing the contact information that was previously extracted, half spaces can be defined that split the unit sphere into separate sections, which are referred to as cells. All directions – represented by the points on the sphere – that are enclosed by the boundaries of one common cell share the same DBG. Thus, in order to build all unique DBGs, it suffices to select a representative vector for each cell on the surface of the sphere and calculate the corresponding graph. Figure 2.8 shows an assembly with several plane-plane contact relationships. The half spaces supporting the contact faces are represented by circles on the unit sphere, dividing it into four sections which correspond to different DBGs. One benefit of this design is the fact that the DBGs on separate sides of a boundary vary only slightly. Thus, updating the existing DBG when crossing a boundary into a new cell is much more efficient than building a new DBG from scratch.

One drawback of the basic NDBG approach results from it only storing the contact information of the parts. Thus, if no additional information is provided, blocking relationships are assumed to be a subset of the contact relationships. Consequently, the algorithm fails to detect obstacles of a part with which it is not in direct contact. Wilson proposes to either encode the configuration space of each part into the NDBG or to use a two-stage approach in which free partitions are first suggested using the basic NDBG and then verified globally via collision detection. However, polynomial runtimes cannot be guaranteed

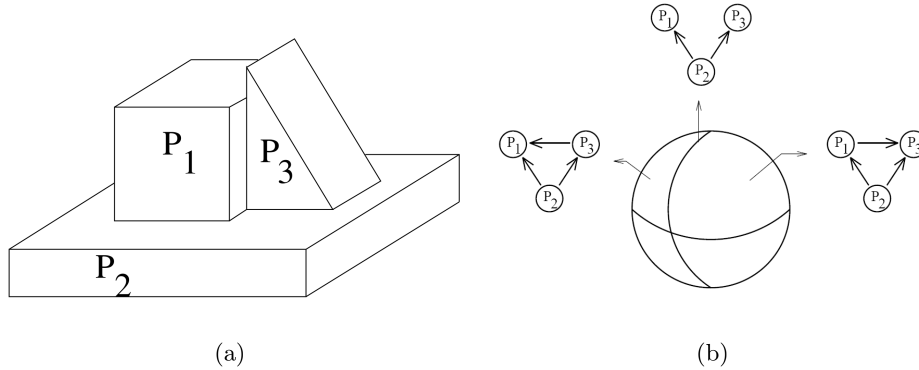


Figure 2.8: A 3-part assembly and the corresponding NDBG with contact faces represented by arcs on the unit sphere. Each separate section represents a cell and a DBG that can be created from it. Note that an arc separating two cells may itself represent an infinitely small cell. Images adopted from [32].

for either approach.

The overall performance, capabilities and possible extensions of the NDBG have been thoroughly explored [11, 12, 23, 34]. By combining the NDBG with the "interference diagram" of the assembly, Wilson and Latombe lay out a set of algorithms for examining possible assembly paths that consider complex insertion operations. In [11], the authors discuss the case of testing a bounded number of k consecutive translations and show that by combining these two data structures, complex operations for inserting or removing a partition can be identified. However, it is observed that the main problem of this approach and similar strategies that employ the NDBG is the high runtime complexity. Since both the number of parts examined and the vertex count in the associated polygon meshes directly influence the performance of the algorithm, its applicability is limited by the size and the geometric detail of the assessed assembly.

Building on the concept of the NDBG and the AND/OR graph, Romney et al. present the Stanford Assembly Analysis Tool (STAAT) [26]. The system implements the general approach by Wilson and detects feasible assembly paths that consist of straight-line translating motions. Contact relationships required for building the NDBG are automatically inferred from analysis of the input CAD data sets. The authors contribute several optimizations which directly affect the runtime required for detecting all feasible partitionings in the AND/OR graph. A compact version of the NDBG is proposed, which greatly reduces the number of DBGs that need to be evaluated (see Figure 2.9).

The system notably provides a proof of concept for considering also those obstacles of

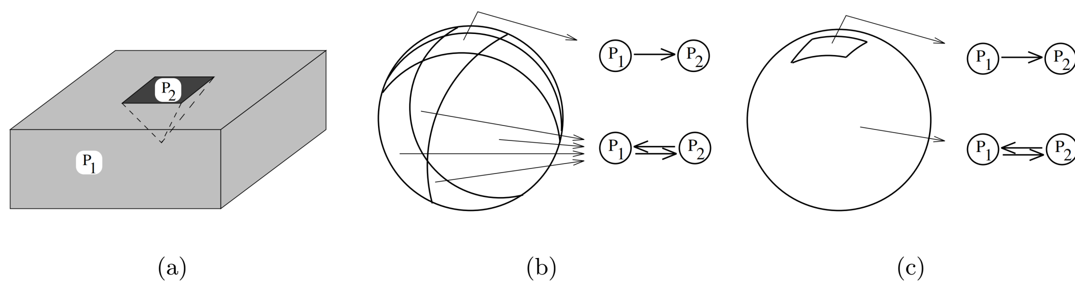


Figure 2.9: (a) A simple setup with two examined parts P_1 and P_2. (b) NDBG created from analyzing the contact relationships. A number of separate cells can be identified that lead to the same DBG. (c) The optimized NDBG uses arc segments instead of full circle arcs to describe contacts. The number of cells is significantly reduced. Images adopted from [26].

a part with which it is not in direct contact. As proposed in [32], the system first detects partitions that are not blocked by any part in their immediate vicinity and then verifies their global accessibility. This is done by "sweeping" the geometric primitives of the parts and intersecting the extruded shapes with all potential obstacles. In comparison with the runtimes for finding assembly sequences and complete AND/OR graphs using Wilson's GRASP tool, a significant speedup is achieved. The authors note their intention to improve the system by directly integrating C-Space obstacles in the NDBG and discarding the sweeping mechanism.

A significant step towards practical usability of computer-aided assembly planners was taken by Jones et al. [14]. They describe an interactive tool for planning assembly procedures called *Archimedes*, which is capable of efficiently creating and animating plans that consist of straight-line insertions and rotations. The authors highlight the fact that besides geometrical feasibility, human interaction can provide valuable information to significantly speed up the process of finding an assembly plan. They provide an extensive library of strategic and tactical constraints, which can be selected and parametrized by the user to convey preferences or unwanted actions to the system. Thus, in order to arrive at a satisfactory solution, the system starts with an initial plan and then cycles through a view-constrain-replan procedure, where the user inspects the current result, modifies it by providing additional information and eventually asks the system to recalculate a candidate solution with the added constraints. As mentioned in [33], the Archimedes tool can exploit graphical hardware to ensure that all actions can be performed without causing collision between parts. The largest assembly that has been successfully evaluated in the

Archimedes system consisted of 472 parts, for which the replan phase in each iteration required several minutes [14].

Agarwal et al. present an efficient algorithm that considers only a discrete set of k directions along which the parts can be moved [2]. Input is required in the format of polygon meshes representing the parts of the examined assembly. For each tested direction $\overrightarrow{d}$, the system maintains the set of parts that are free to move to infinity in $\overrightarrow{d}$ without causing a collision. These parts are referred to as *maximal* in $\overrightarrow{d}$. The system then iteratively removes one of the free parts and updates the maximal sets with those objects that in turn become free due to the removal of that part. The authors propose a data structure for maintaining the maximal features of polyhedra in order to allow efficient querying and updating of the sets. The algorithm generates k-directional assembly/disassembly plans that may involve translations and rotations around the translating axis. A valid sequence can be found in $\mathcal{O}(m^{4/3+\epsilon})$, where m corresponds to the number of vertices in the polyhedra. The authors note that, compared to [32], no sequences can be detected that require the removal of larger partitions. However, due to the rather low runtime complexity, they suggest it as a viable alternative suitable for most common assemblies.

Reasons for investigating tolerances in mechanical assemblies and possible solutions have been discussed in [18, 25]. The main motivation in [18] stems from the fact that industrial components often contain slight variational errors or are not built exactly as they have been specified. Furthermore, individual parts in an assembly may be replaced by similar variations of the same base model. In order to account for these possibilities, they describe assemblies that use a designated language for encoding tolerance in their parts. For all edges in the original part geometry, tolerance zones specify how much its relative distance to a chosen point of origin may vary. The authors note that it is a basic requirement for all variations of an original part to preserve the initial topology, although it is not stated how this can be ensured computationally. A method is presented for evaluating assembly plans with tolerance by introducing a "strong" and a "weak" NDBG. Corresponding algorithms can be employed to query whether a valid assembly path can be found for all possible variations of all parts (strong NDBG). If this is not the case, a secondary procedure can determine whether *any* variation of toleranced parts can be feasibly assembled (weak NDBG). However, the outlined procedures focus on 2D polygonal assemblies. The authors observe that especially calculating the weak NDBG for 3D polygon meshes poses a challenging topic for future research.

Thomas et al. identify feasible assembly sequences using stereographic projection to

evaluate the C-Space objects of the parts in an assembly for a discrete set of straight-line translations. The corresponding vectors are obtained by uniformly sampling the complex plane obtained from the Moebius transformation of the Riemann's sphere [30]. Figure 2.10 outlines the program flow of their system with intermediate steps and results.

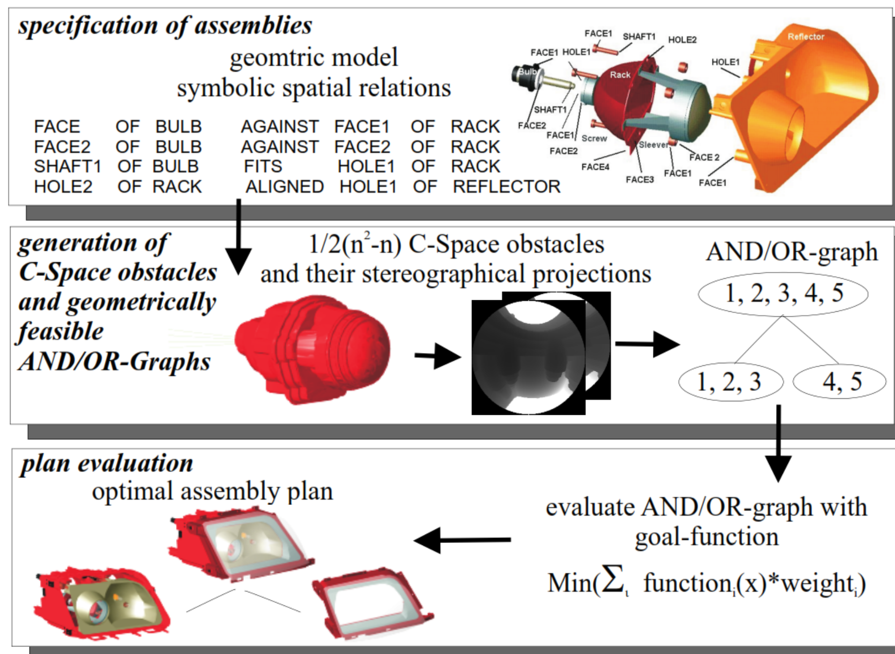


Figure 2.10: Outline of the program flow for the assembly planning system described by Thomas et al. in [30]. The system generates pair-wise C-Space objects and uses a $2\frac{1}{2}$D projection to discretely sample the space of possible separating motions. All feasible partitions are created and inserted into the AND/OR graph, from which an assembly plan is selected that is optimal with respect to a defined goal function (adopted from [30]).

The program accepts polygon meshes as input, along with specifications that provide hints to the precedence relations between the individual parts. The C-Space objects are created by applying convex decomposition on the input meshes and calculating the union of the Minkowski differences that are obtained using the convex polyhedra. A parametrizable tolerance mechanism is introduced by shrinking the decomposed subparts by a given value. A user-defined parameter defines the resolution of the texture targets for stereographic projection. By referring each texel to a location on the Riemann's sphere via the Moebius transformation, a set of discrete directions is implicitly obtained from this parameter. Ray casting is employed to check for intersections with the C-Space objects using the vectors that represent the sampled directions. The resulting information is then written to two separate 2D textures for each part pairing (see Figure 2.11).

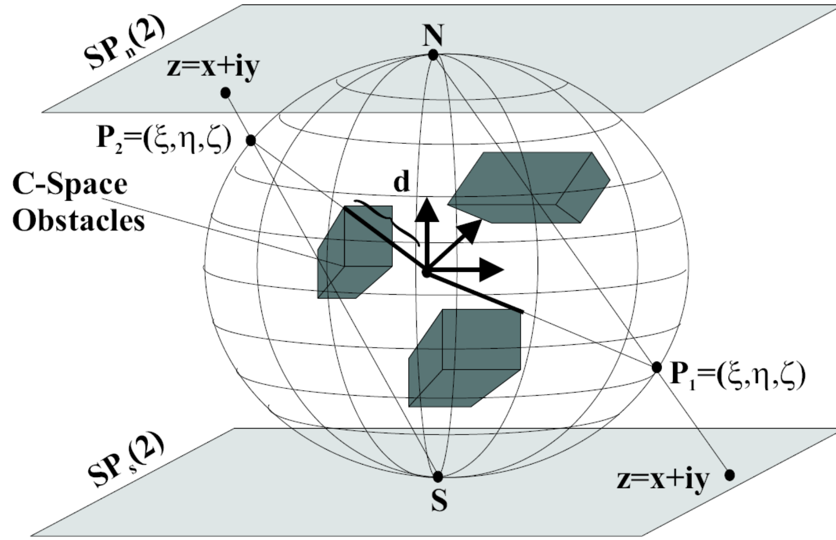


Figure 2.11: Stereographic projection of the C-Space obstacles for a given part pairing using two target textures. Each texel of either texture represents a point on the Riemann's sphere and thus a direction that can be sampled via ray casting (adopted from [30]).

Each texel stores a depth value encoding the distance one part can move in the corresponding direction before colliding with the other. A value of 1.0 indicates that the direction is unblocked. The complete AND/OR graph is created by calculating all connected partitions and determining whether they are detachable by evaluating the blocking relationships in the textures. The authors suggest that this is most efficiently performed by exploiting graphics hardware to quickly generate the union of all relevant textures that were stored for the parts in the group. For reasons of efficiency, binary images are used where each pixel represents whether the conditional ($distance < 1.0$) is true or not. If the final union contains a zero valued pixel, the corresponding direction is unblocked for the entire group of parts. The implied parallelism allows for quick evaluation of all disassembly paths that can be performed when considering the discrete set of tested translations. By employing the assembly-by-disassembly policy, the system allows the definition of a goal function that can be used to detect an optimal assembly plan.

Agrawala et al. applied the discoveries of automatic assembly planning to the field of computer graphics [3]. Their work focuses on automatically arranging effective step-by-step assembly instructions based on a computed assembly sequence. They reason that for most assemblies, it suffices to evaluate and store blocking relationships for the 6 principal directions only. For the corresponding 6 translating motions, DBGs are created and main-

tained by the system. Blocking relationships are inferred from contacts between parts, thus disconnected obstacles are not explicitly avoided during assembly. The system accepts a number of constraints considering partitioning, symmetry and other properties that are to be ensured in the resulting assembly sequence. A set of images depicting the stages of the procedure are generated, with possibly multiple parts being assembled in the same stage (see Figure 2.12).

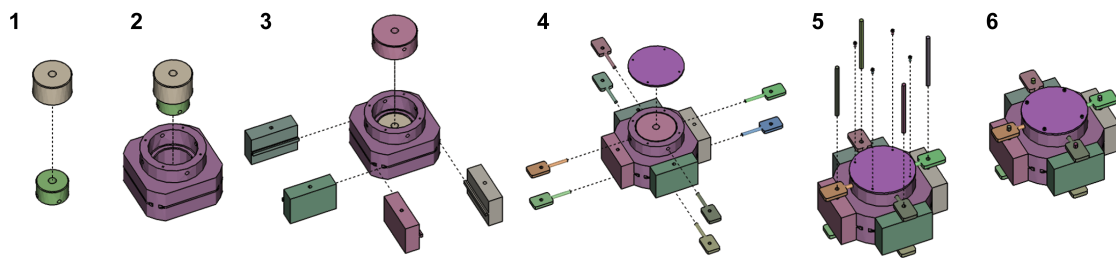


Figure 2.12: Assembly instructions for a product as generated by Agrawala et al., including the final assembled state. Guidelines indicate the relative motion applied to mate sets of parts. Images adopted from [3].

The system employs a scoring system based on visibility to ensure that parts that are added early on will not occlude parts that are added at a later time. Thus, each stage builds on the reference frame of the previous stage to ensure that the instructions may be easily understood. Repetitive steps may be omitted to draw focus to more significant assembly instructions and reduce the number of output images. To capture the smaller parts that are difficult to make out during assembly, insets were manually added to some of the pictures (see Figure 2.13).

The authors conclude with the insight that the problems of *planning* and *presenting* assembly sequences are strongly interrelated and should best be addressed cooperatively. Furthermore, they comment that by extending the space of possible part motions, the set of assemblies that can be processed in their system would increase significantly.

A similar approach is presented by Guo et al. [8], where feasible disassembly sequences are detected based on blocking relationships that are inferred from part contacts. However, they extend the space of possible separating motions by employing an analytical approach. Parts of examined assemblies are segmented to form patches that are fitted with algebraic surfaces. Circular loops in these surfaces are detected and clustered to define a group D of potential separating motions. The problem of partitioning is considered based on the results of a user study that revealed the preferred tendencies for grouping parts to be coaxiality, symmetry and contact coherence. In cases where the automatically suggested

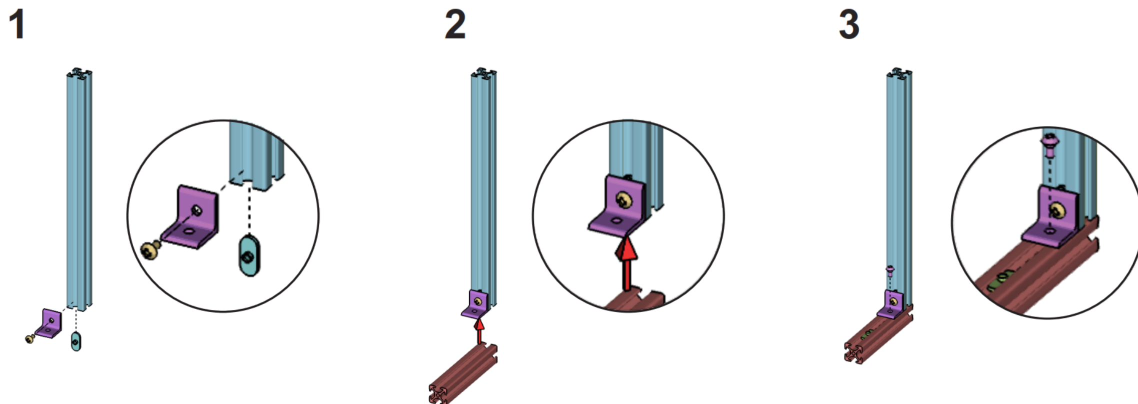


Figure 2.13: Due to their small size, the precise action to be applied to some parts is difficult to make out. Insets were manually added by Agrawala et al. for clarification. Images adopted from [3].

partitioning hierarchy fails to satisfy the requirements of the user, an interface provides the means for editing all aspects of the hierarchy. To avoid situations where loose parts may fall out of place due to gravity, the system allows the definition of a base part and an ***up*** vector. Based on this information, the product can be disassembled top to bottom, thus reducing the risk of collapsing. Computed sequences can be illustrated in a step-by-step manner using either part offsets with guidelines or animation paths. In their concluding statement, the authors comment on the limitations imposed on their system by requiring input to exhibit a high degree of exactness and its inability to consider complex motions such as extended translations.

Li et al. extend on the work by Agrawala et al. in order to automatically create explosion diagrams of assembled products based on a corresponding disassembly sequence [19]. The authors provide mechanisms for handling interlocking parts, part hierarchies and common cutting operations in explosion diagrams. They introduce the explosion graph, a relational data structure that is incrementally built by detecting and inserting removable parts over time. Removability is evaluated by testing the feasibility of straight-line translations for the main axes of the coordinate system. In addition, the user may define arbitrary directions to be tested as well. In-depth information about subhierarchies in the product can be provided to introduce localized structures in the resulting explosion diagram. For each subhierarchy, the system generates a separate explosion graph, enabling partitions at any level of the hierarchy to expand and collapse independently. If the calculation of a disassembly sequence is stalled due to interlocking parts in the assembly, the largest detachable partition is computed and removed. The system provides methods

for calculating explosion diagrams that specifically expose one or more user-defined target parts. Furthermore, an interactive user interface offers various tools for editing and assessing explosion diagrams. Selected features include riffling, label placement and smooth animations for establishing visual context between the initial and the exploded state of the parts. Figure 2.14 illustrates the explosion diagram that was created for the model of a turbine.

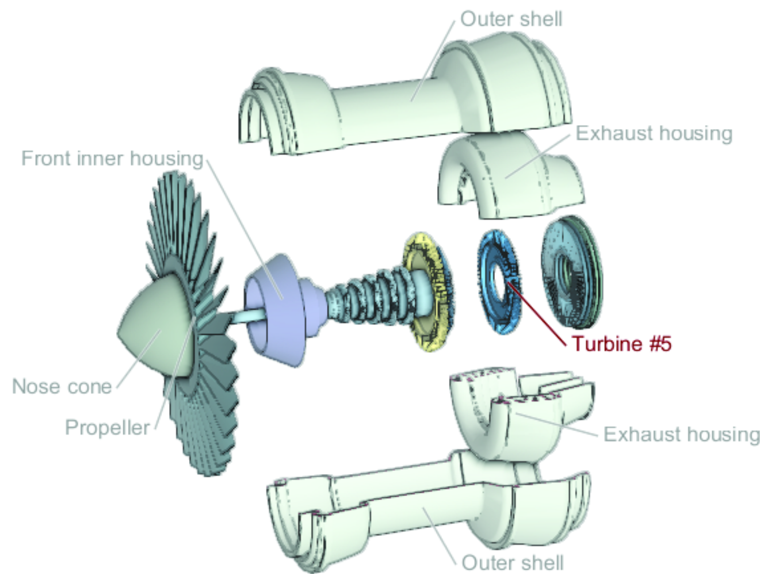


Figure 2.14: Explosion diagram illustrating the structure of a turbine model (adopted from [19]).

Srinivasan and Gadh have specifically focused on the concept of **selective** disassembly which aims at removing any given set of target parts as efficiently as possible [28, 29]. Instead of performing complete disassembly of a product, it suffices to find a sequence of instructions for extracting all parts that inhibit the removal of the targets. An algorithm for finding the disassembly sequence with the lowest number of simultaneous removals is outlined in [29]. The approach is based on the "onion peeling" algorithm, where all parts that can be independently removed at a given point in time are stored in a corresponding set. By removing the current layer and increasing the time variable, other parts may in turn become detachable and define the contents of the next inner layer. Starting with 0, each point in time thus represents a "layer" that can be peeled off the assembly.

The space of tested removal actions is comprised of a discrete, possibly user-defined set of straight-line translations. Blocking relationships are extracted from contacts and spatial

constraints, which allows the removed parts to avoid connected and disconnected obstacles equally. The sets representing the peeled layers as well as the blocking relationships are entered into a removal influence graph (RG). The RG stores each part as a node on a layer corresponding to the time when it was removed. Blocking relationships are represented as directed arcs emanating from nodes pointing to entries on higher layers. An example of an RG for a simple assembly is depicted in 2.15. Once the RG has been topologically sorted, a feasible disassembly sequence can be directly inferred.

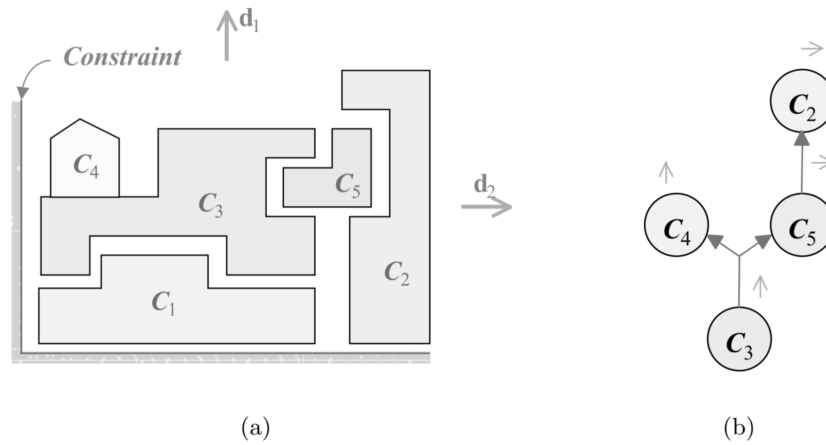


Figure 2.15: (a) Simple assembly constrained by an unmovable adjacent structure with two specified directions being tested for removal. (b) The removal influence graph that was generated for the selective disassembly of part C_3. Images adopted from [29].

Chapter 3

Concept

3.1 Challenges for Disassembly Planning

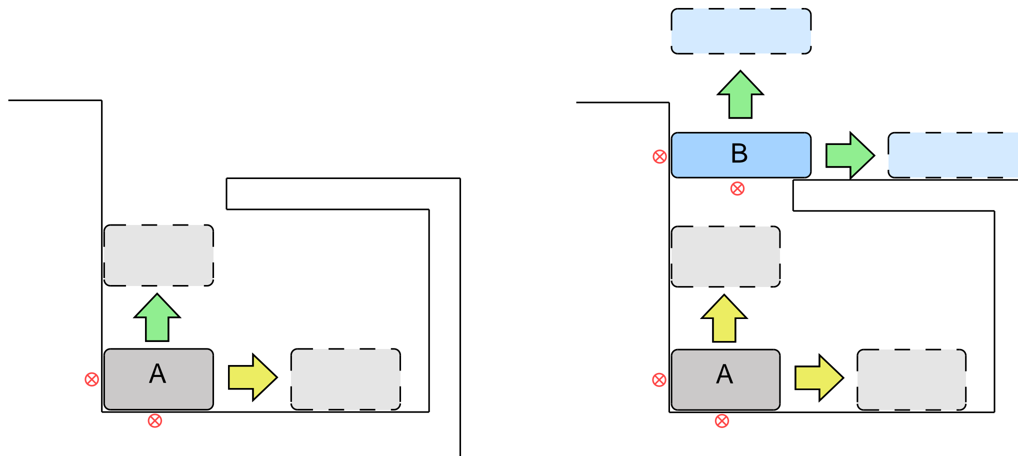
When looking for adequate solutions for a specified disassembly problem, there are numerous criteria that need to be taken into consideration. The perceived quality of a chosen disassembly path may depend greatly on the implied effort for removing individual partitions, overall feasibility or personal preferences. Apart from criteria for identifying suitable disassembly paths, there are several computational restrictions to conventional approaches for disassembly path generation that pose notable challenges for implementing an efficient disassembly planning system.

3.1.1 Geometrically Feasible Disassembly Paths

Geometrical feasibility defines the key prerequisite for all disassembly sequences with real world applications. Any interaction with the components of an assembly is said to be geometrically feasible, if it can be performed without causing collisions or mesh intersections of the considered parts. During disassembly planning, each performed removal action must be geometrically feasible with respect to the active partitions. Disassembly paths containing removal actions that do not fulfill these requirements are discarded. The term **subassembly removability** in this context denotes the existence of a geometrically feasible removal action for a subassembly at a given point in time. Most assemblies contain subassemblies that are initially blocked and thus cannot be removed directly. Their extraction may become geometrically feasible once some of the more accessible partitions have been removed and become inactive. Therefore, monitoring subassembly removability over time is key for detecting and suggesting geometrically feasible disassembly paths.

It can generally be assumed that removing a subassembly involves - but is not necessarily restricted to - a translating motion. Based on this assumption, Wilson distinguishes two degrees of freedom for subassemblies, **local** and **global** [32]. Local freedom describes the existence of an infinitesimal translating motion that is geometrically feasible over a distance greater zero. It is commonly determined by detecting the contact faces of parts and using their supporting planes to reduce the space of possible motions. If the space of possible motions for a subassembly is non-empty in respect to its contacts with all other active partitions, it can be considered locally free.

A subassembly is considered globally free, if there exists a translating motion that can be extended to infinity without causing part collisions. In contrast to local freedom, global freedom thus evaluates the geometric feasibility of completely removing one subassembly from the remaining assembly. Methods for determining global freedom include mesh sweeping, rendering-based approaches and C-Space computation [26, 30, 33]. Figure 3.1 illustrates the concept of local and global freedom for two different setups.



(a) Although part ***A*** is locally free in one direction (yellow arrow), it cannot be removed from the assembly along the respective vector. The green arrow indicates that the part is also globally free and can thus be successfully removed with an upward translating motion.

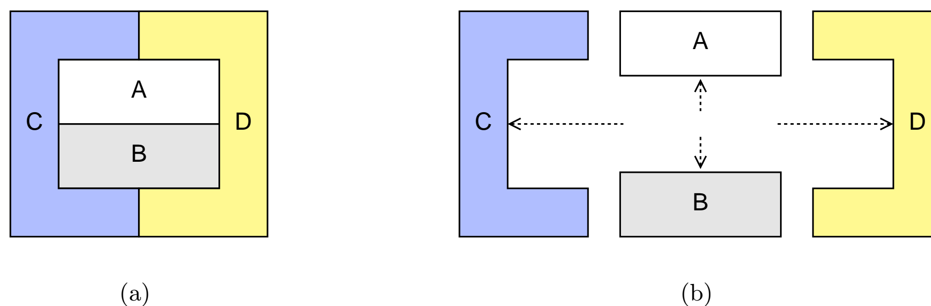
(b) Part ***A*** is now only locally free in both directions. However, removing the initially globally free part ***B*** from the scene results in the same setup as in (a), thus ***A*** becomes globally free through the removal of ***B*** and has its subassembly removability updated.

Figure 3.1: Two setups (a) and (b) for simple assemblies for which a number of removal actions exist. The space of possible motions is restricted by the contact faces of parts (red circle). Locally free and globally free directions are indicated by yellow and green arrows respectively.

Previous implementations for detecting geometrically feasible removal actions vary based on their underlying data structure, the extent of supported removal actions and the requirements for subassembly removability. Since global freedom is usually harder to verify, local freedom is often used to reduce computational demands. For instance, Agrawala et al. consider a subassembly to be removable if it is locally free [3], while Romney et al. use local freedom to reduce the search space when testing for globally free subassemblies [32]. However, the definition of local freedom implies that it can only be used robustly with numerically exact input. For more imprecise assemblies, the distance between two subassembly surfaces that are supposed to touch may not necessarily be zero. Therefore, the task of detecting and verifying local freedom becomes a matter of trial-and-error thresholding. For spuriously self-intersecting assemblies, no feasible disassembly sequence may be found due to the fact that some of the parts are constantly colliding and thus blocking each other. Geometrical feasibility thus becomes a highly complex problem when dealing with imprecise or lossy representations of the original assembly.

3.1.2 Complexity of Removal Actions

The most basic form of removing or inserting a part into an assembly is by using a singular or straight-line translating motion, e.g. removing a lid from a box or pulling a wheel from an axle. Some products can be completely disassembled by using singular translations only. Due to their simplicity and detailed research on suitable implementations, singular translations are the most common removal actions to be evaluated in previous disassembly algorithms [2, 3, 13, 14, 18, 35]. Figure 3.2 shows a simple assembly where each part can be removed using only singular translating motions.

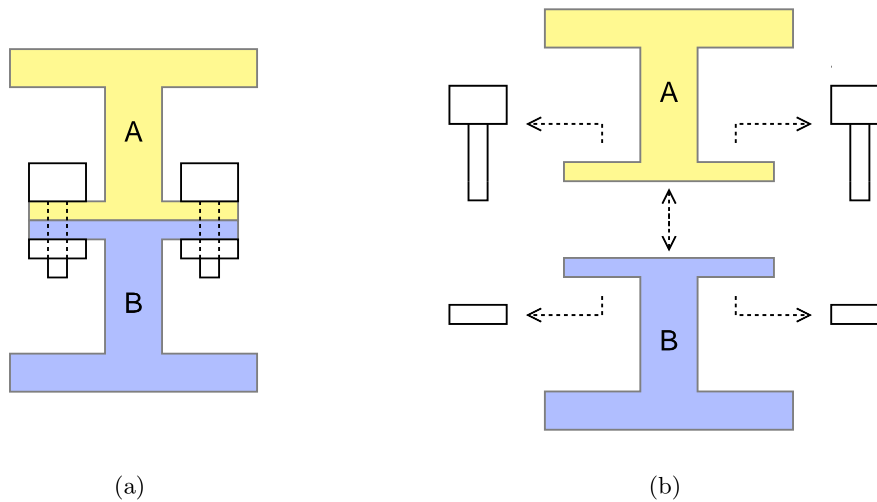


(a) (b)

Figure 3.2: (a) A simple assembly consisting of four parts. (b) Each part in the assembly can be accessed and removed over time using only straight-line translating motions.

However, many assemblies contain parts that can only be removed using more com-

plicated motions or intermediate stops. For instance, a screw may only be removed by performing a twisting motion. Fitting a bolt may require it to be moved into place in one direction before being fastened in another. Figure 3.3 represents an assembly that cannot be decomposed if only singular translating motions are being considered.



(a) (b)

Figure 3.3: (a) Two parts A and B being held together by a pair of nuts and bolts at their point of contact. Although the assembly is rather simple, there is no viable solution when only singular translations are being considered. (b) If testing removal actions involves the evaluation of two or more consecutive translations, removing the nuts and bolts is geometrically feasible and the assembly can be decomposed completely.

Since the complexity of considered removal actions is directly linked to the computational demands of automatic disassembly path generation, most implementations to date focus on singular translations only, with few exceptions. For instance, Wilson also includes the rotation of parts in the set of possible removal actions [32]. Efficiently testing the feasibility of removal actions containing more than one translating motion in 3D space is considered an open problem [10]. Where necessary, multi-translational motions have occasionally been dealt with using time-consuming path planning computations [32].

3.1.3 Tolerance

Calculating the blocking relationships and contact information for an assembly usually involves thorough analysis of the geometry information provided for its parts. If the input is guaranteed to be numerically exact, this information can be automatically extracted by qualified algorithms with adaptive precision. However, most available representations of

assemblies, such as CAD data sets, are usually constructed by human individuals. Therefore, exactness and quality of the resulting models may be influenced by the experience and skill of its creator. Even if the creation process is executed with maximum precision, artifacts may occur when converting CAD models to polygon meshes due to loss of detail during tessellation. Possible artifacts include disconnected or floating parts, spuriously intersecting polygon meshes or surface deformations. Such artifacts can cause false results for calculated contact information or blocking relationships. For instance, a part may be scaled to be slightly bigger than it is in relation to the parts with which it interacts in the assembly. This may cause overlapping part geometries that cannot be separated from each other with any tested removal action. Figure 3.4 illustrates an example of two intersecting CAD models that may easily be considered valid by the designer, but cannot be correctly processed by automatic disassembly systems.

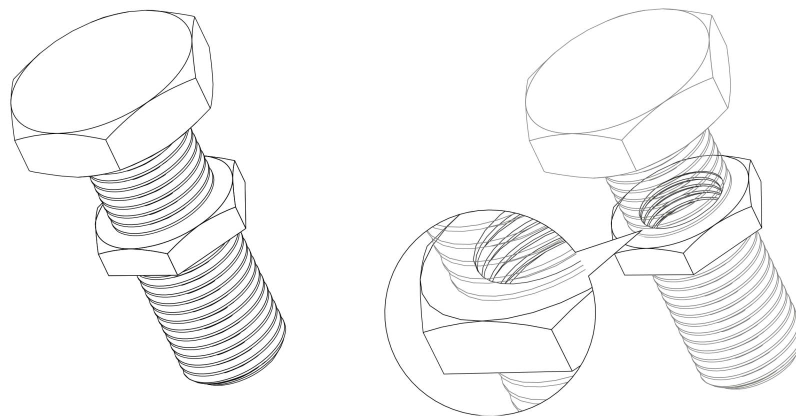


Figure 3.4: A nut and bolt assembly that appears valid when viewed as opaque objects. Applying transparency to the model reveals that the bolt actually does not fit the opening of the nut and the two parts are overlapping. While a human subject may still easily grasp the underlying principle and the necessary procedure for separating the two parts, an automated system without tolerance mechanisms cannot deduce a geometrically feasible disassembly sequence.

To account for the mentioned artifacts, tolerance values can be applied during the analysis of the assembly. Latombe and Wilson suggest a descriptive language encoding the tolerance zones for each part and provide methods for evaluation in 2D. Thomas et al. shrink the components of input assemblies prior to evaluation of blocking relationships in order to decrease the probability of spuriously prohibited removal actions. Enabling part tolerance in the field of automatic disassembly is still at an experimental stage, but is nonetheless a key factor when targeting general assemblies.

3.1.4 Partitioning

In disassembly planning, partitioning is the process of splitting an assembly into groups of parts that can be tested for removal, except for the top-most partition $\boldsymbol{P_0}$, which represents the entire assembly. A partition may itself contain a number of smaller partitions. The structure of an assembly can thus be represented by a hierarchy of partitions, where each partition is linked to its parent, which is defined as the next larger partition from which it was split off. The simplest form of partitioning is to equate each part to a partition and directly linking them to a common parent $\boldsymbol{P_0}$. Using this approach, an assembly consisting of n parts contains n partitions that are immediate children of $\boldsymbol{P_0}$ and can be tested for removal at each point in time. However, this form of naive partitioning may not suffice to find a fitting solution for more complex disassembly problems. Figure 3.5 depicts an exemplary assembly, that can only be resolved if two of its individual parts are temporarily fused to be removed as one larger partition.

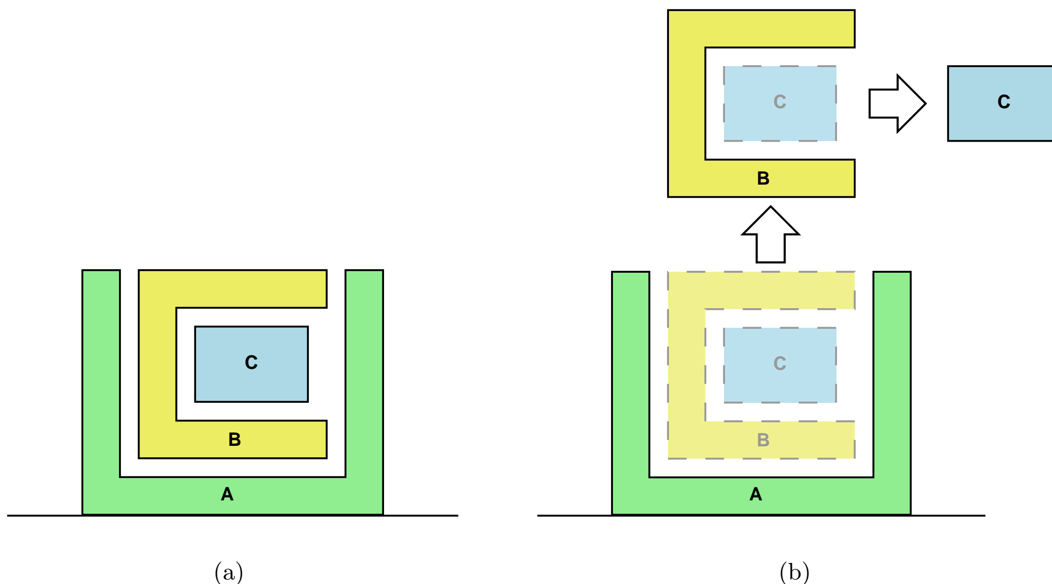


Figure 3.5: (a) A simple setup consisting of three single parts. With naive partitioning, we obtain three partitions $\boldsymbol{P_A}, \boldsymbol{P_B}, \boldsymbol{P_C}$, none of which can be removed from the assembly. (b) By fusing partitions $\boldsymbol{P_B}$ and $\boldsymbol{P_C}$, a new partition $\boldsymbol{P_{BC}}$ is formed. The partition $\boldsymbol{P_{BC}}$ can be removed using a singular translating motion. $\boldsymbol{P_{BC}}$ itself can then be further decomposed in order to reveal its contents $\boldsymbol{P_B}$ and $\boldsymbol{P_C}$.

Starting with the top-most partition $\boldsymbol{P_0}$, there may be a high number of ways to create valid partitionings. A partitioning of $\boldsymbol{P_A}$ into a group of smaller partitions $\{\boldsymbol{P_1}, \boldsymbol{P_2}, ..., \boldsymbol{P_N}\}$

is valid, if none of the newly created partitions intersect and for each partition $\boldsymbol{P_i}$ with $0 < i < N$ there exists a geometrically feasible removal action to separate $\boldsymbol{P_i}$ from $\boldsymbol{P_A} \setminus \boldsymbol{P_i}$. The entirety of valid partitionings for an assembly can be effectively calculated by building its AND/OR graph [6]. AND/OR graphs consist of nodes that represent partitions in the assembly and hyperarcs that indicate all valid ways in which they can be split into yet smaller partitions. The AND/OR graph is usually generated by recursively checking the feasibility of all potential partitionings for a target partition, starting with $\boldsymbol{P_0}$. If there are no additional constraints, the number of partitions that can be generated and tested in this way for an assembly containing n parts is as high as 2^n. It is obvious that for large assemblies with several hundred parts, testing all possible 2^n partitions quickly becomes infeasible. Common constraints to reduce the size of the AND/OR graph include only considering two-handed and contact-coherent partitionings. Two-handed in this context describes the constraint that a partition may only be split into two smaller partitions, thus exactly two hands are required to perform each operation. Contact-coherent partitionings also fulfill the requirement that there are no disconnected or floating parts in either child partition. In 1992, Wilson elaborated several approaches for AND/OR graph generation based on these constraints. However, it was estimated that without providing yet further constraints, the expected upper boundary for assemblies to which these methods could be applied lies at ∼50 parts [32]. Since the computational power in processing units has grown tremendously over the past few years, we implemented a naive AND/OR graph generating method to execute and evaluate its performance under contemporary circumstances. More efficient implementations have been proposed in [34], however, they cannot guarantee polynomial runtime when testing global freedom, nor do their results differ from those of the basic approach in terms of required storage and structural complexity. The results in Table 3.1 show that the computation of AND/OR graphs is still very demanding even when using today's hardware, and can hardly be applied to larger assemblies. Depending on how strongly connected an assembly is, its AND/OR graph may grow to consume several gigabytes for assemblies consisting of only 20 parts and can take rather long to compute.

For assemblies of moderate size, calculating the AND/OR graph is practically feasible. Once generated, the AND/OR graph itself requires evaluation in order to find a suitable disassembly path. For instance, it may be of interest to find the disassembly path that requires the smallest number of removal actions. However, the number of possible disassembly paths may be far greater than the number of valid partitionings. Therefore,

Testcase	#Parts	Connectivity*	#Partitionings	Time	Size
Cargo	7	weak	81	17ms	12 KB
Valve	14	weak	4,308	316ms	372 KB
Blocks 64x1x1	64	weak	43,680	1s 184ms	7.2 MB
Blocks 3x2x2	12	strong	176,422	4s 255ms	28.2 MB
Blocks 3x3x2	18	strong	106,286,977	2h 54m 25s	14.5 GB

* In absence of a better measure, the connectivity of the assemblies with regard to the number of contacts for each contained part was classified either as ”strong” or ”weak”.

Table 3.1: Result data gathered from AND/OR graph generation procedures using sample test cases. The structure of the assembly in each considered test case has a very noticeable effect on the computational demands to generate the final result. Rather than the number of parts in an assembly, the governing factor influencing the number of potential partitionings is the strength of its overall connectivity.

evaluating the AND/OR graph may take considerably longer than building it [30]. In cases where generating or searching the complete AND/OR graph is not feasible due to runtime or memory limitations, greedy search algorithms may be employed to pursue at each branch the most efficient local alternative in order to obtain an estimate of the ideal disassembly path. However, these approaches cannot preclude the existence of a superior solution unless the entire space of possible alternatives is examined. Furthermore, the AND/OR graph does not lend itself to disassembly path editing. Navigating between alternative sequences in order to find more suitable solutions in the AND/OR graph may be confusing due to the complexity and sheer size of the data structure.

Alternative approaches to the partitioning problem usually focus on the selection of one specific disassembly path based on predefined partitioning rules. For instance, Guo et al. automatically create partitions based on spatial criteria such as coaxiality or part symmetry [8]. Li et al. use subassembly splitting to determine the largest free partition on demand at each point in time if none of the smaller partitions can be removed [19]. Since these rules may not always produce the ideal disassembly path for specific requirements, disassembly planning systems are often equipped with a mechanism for editing or defining custom partitions. Partitioning is thus an essential factor in disassembly planning and may be highly dependent on user interaction for finding optimal disassembly paths.

3.1.5 Real World Constraints

Real world assemblies are often structurally complex. In order to achieve its designated purpose, the characteristics of its subassemblies may show a high degree of diversity.

These characteristics become important when considering the execution of disassembly instructions under real world conditions. For example, a machine may consist of numerous parts with different stability, buoyancy or resiliency. The removal of supporting structures from such an assembly may put an unplanned strain on the remaining parts, thus causing damage or causing the assembly to collapse. Including the influence of physical forces such as gravity or friction in the disassembly planning process would require a highly involved simulation system, the implementation of which is beyond the scope of this thesis. Other constraints may arise when dealing with potentially hazardous components, such as electricity conducting wires, which require a specific course of action to ensure safe removal. Future technological developments may also introduce new elements or forces that need to be taken into account. Ultimately, user-specific preferences for performing certain removal actions may also influence the perceived adequacy of a disassembly path (e.g. removing only parts that are below a certain weight). The list of potential criteria for finding an adequate assembly sequence can be extended ad libitum. Disassembly planning thus involves a high number of volatile influences that can change over time and may even differ from user to user.

3.1.6 Editing Disassembly Paths

As mentioned above, a disassembly problem is defined by the input meshes representing the assembly and a set of partitions that are required to be removed. Given a disassembly path as a chain of consecutive, possibly interdependent removal actions that provide a solution to this problem, modifying the disassembly path to fit an expected outcome can become quite complicated. If a specific removal action at an early point in time is modified, later instructions may become infeasible or require additional effort for removing partitions that were not considered in the original path. Thus, for a modified removal action representing instruction I_x in a disassembly path containing n steps, all instructions $I_{x+1}, I_{x+2}, ...I_n$ need to be either verified for feasibility or discarded such that a new disassembly path can be calculated automatically from this point onward. However, changing removal actions that appear early in the disassembly path may have little or no effect on how the required partitions themselves are removed. Assuming that the underlying disassembly computation method focuses on the shortest possible disassembly path, Figure 3.6(a) shows an assembly where the initial solution (see Figure 3.6(b)) is not ideal with regard to the user's preferences. Customizing the first instruction yields the same result, complemented by an unnecessary part removal (see Figure 3.6(c)). In order to adjust the

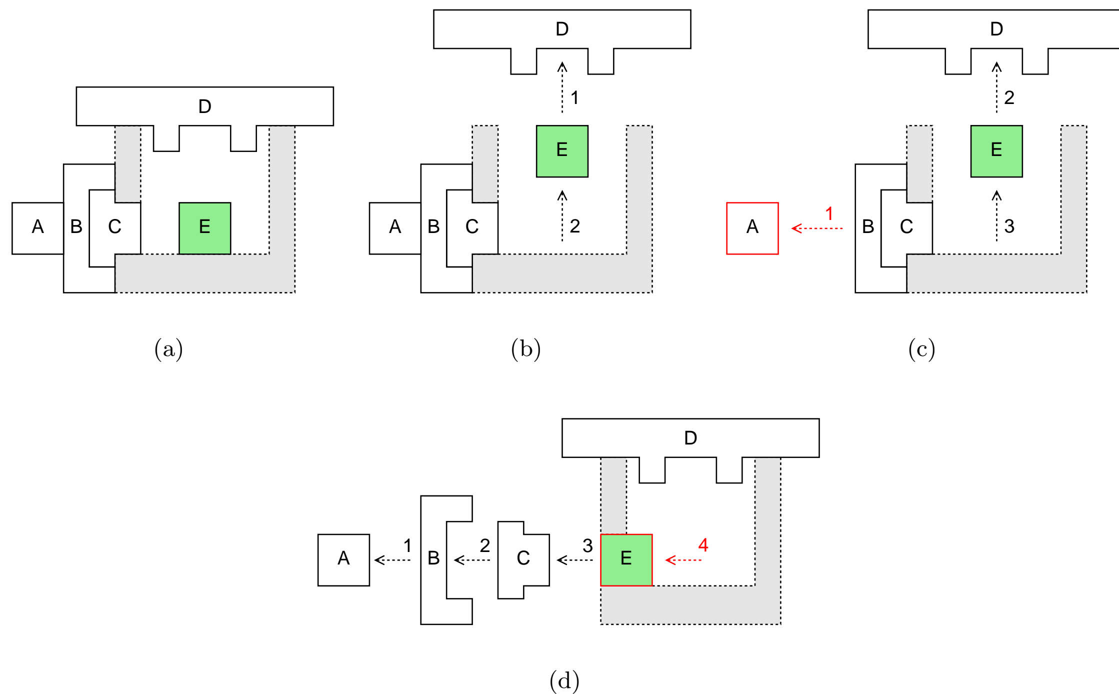


Figure 3.6: (a) An assembly consisting of five parts and a fixed base structure. Part E is the required part for removal. For this example, we assume that the user has instructions that part D should not be moved if it can be avoided due to its heavy weight. (b) The most straightforward approach for a solution as detected by a system that prioritizes short disassembly paths. (c) After changing the first removal action to a translation of A, all ensuing removal actions need to be recalculated. However, the shortest disassembly path is still obtained by removing part D after A, followed by E. (d) By directly defining the removal direction of the required part instead, the system is forced to calculate a new solution that ensures the feasibility of this action.

disassembly path, the user may have to change several removal actions until the suggested solution based on the priorities of the system conforms to the expected outcome. For editing a disassembly path in this manner, in-depth knowledge about all partitions that are involved in the procedure is needed. Since the intended purpose of the automatic disassembly path computation is to reveal ways to access the required partitions, modifying a suggested path top down until they are removed as expected defeats the purpose of the system. An alternative way for editing a disassembly path may be based on a set of rules that can easily be changed for partitions so that the disassembly path is explicitly modified bottom up. For instance, defining the direction that should be used for extracting the required part in the example in Figure 3.6(d) leads to an entirely new solution.

3.1.7 Disassembly Sequence Visualization

A disassembly sequence may consist of a high number of consecutive removal actions. Instruction manuals usually illustrate the required actions in a step-by-step manner, which is very helpful for understanding the details of the process. However, these methods only provide transient information from one disassembly stage to the next. Additionally, pinpointing unwanted actions in the disassembly procedure may become very time-consuming since the user has to locate the corresponding frame or image first. In order to allow the user to quickly evaluate the quality of different disassembly sequences, other techniques should be used to give an overview of the implied actions (e.g. explosion diagrams). Ideally, selecting a new disassembly sequence or changing its intermediate steps should be simple and produce instant visual feedback in order to make the impact of the modifications easily comprehensible. Relevant partitions that are involved in the disassembly sequence should be easily distinguishable from the remaining assembly. During animation, small partitions that are being removed may be occluded by larger partitions, which may cause the user to miss necessary removal actions. Therefore, instructions that are at risk of being overlooked should be highlighted by using visual clues.

3.2 System Structure

The system structure is based on two separate modules. The preprocessing module is concerned with the automatic conversion and optimization of geometric data, as well as calculating contact information and discrete blocking relationships for each part in the assembly. The assembly-specific data sets generated by this procedure are henceforth referred to as **static assembly information**. Extracting the static assembly information from geometric data alone is computationally expensive and usually takes up a considerable portion of the disassembly planning procedure [19, 30]. However, given the correct parametrization by the user, the static assembly information needs to be calculated only once for each assembly. Thus, the preprocessing module is logically decoupled from the remaining functionality of the system, which is provided by the Disassembly Planning Application (DPA). The DPA evaluates the output of the preprocessing module and additional user-defined constraints in order to detect, validate and visualize possible solutions. Furthermore, the user may define custom removal actions that were not tested automatically in the preprocessing module, thus extending the space of possible disassembly solutions.

3.3 Preprocessing Module

The preprocessing steps for generating static assembly information involve a number of analytical methods for converting and examining the input meshes that have been extracted from CAD data sets. Since the system is targeted towards handling general assemblies containing a high number of parts, we employ tolerance mechanisms that can be parametrized by the user, as well as optimization methods to accelerate the required calculations for testing the geometrical feasibility of a discrete set of singular and dual removal actions.

3.3.1 Setting Fixed Parts and Separators

An assembly may often contain parts that are either not supposed or simply impossible to move during the entire disassembly procedure. For many assemblies, there exists a basic structure, such as the floor plate in an automobile. Furthermore, rivets might be welded into the basic structure, or wooden parts may be attached to it using glue. Neither of these conditions can be derived from the geometric representation of the assembly alone, however, they clearly inhibit the movement of these parts. They are usually disassembled last, and can trivially be considered removable if all other parts have been successfully removed. These parts can be completely excluded from the disassembly path computation process, thus it is not necessary to calculate their static assembly information. Since the basic structure of an assembly is often comprised of larger parts that occupy a considerable amount of space, excluding them from the ensuing preprocessing steps can cause a noticeable reduction of the runtime. In addition, it provides an intuitive way for the user to convey information about the assembly to the program that can be used during disassembly path computation. Since all disassembly paths involving the removal of the given fixed parts are effectively discarded, finding and selecting a suitable part may become more convenient. For this reason, we allow the user to define a set of fixed, unmovable parts as the first preprocessing step.

The term separator in this context refers to a special instance of a fixed part that divides the assembly into isolated spatial sections. Since a separator is not removed during disassembly, defining them enables the system to infer logically independent groups of parts. Combinations of parts in separated groups are assumed incapable of forming removable partitions. This alleviates the impact of large assemblies on the computational demands for the remaining steps of the preprocessing phase. Figure 3.7 illustrates a case where the number of potential partitions is reduced with regard to a separator.

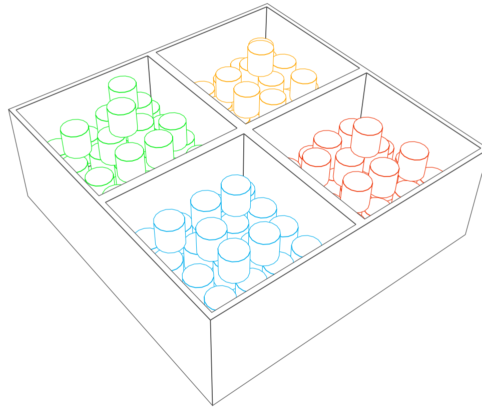


Figure 3.7: Four stacks of objects contained within a separating structure. Each individual color marks a set of parts that represent a logical spatial group. Partitions that involve members from separate groups are not considered sensible candidates for removal.

3.3.2 Contact Information

The contact information for an assembly describes all parts that are in direct contact with each other. This information is usually extracted by checking parts pairwise and determining whether there exist mutual contact faces, i.e. a facet in each part that can be considered to touch a facet of the other. The contact information for the entire assembly is stored in an undirected contact graph, where each nodes represents a part and arcs denote that two parts are in contact. We compute these arcs by testing each part $\boldsymbol{A}$ with every other part $\boldsymbol{B}$ in its vicinity and check for each facet of $\boldsymbol{A}$ if its minimal distance to any facet of $\boldsymbol{B}$ is smaller than a given threshold D. For each pair $(\boldsymbol{A}, \boldsymbol{B})$ where this is true, an arc is inserted into the contact graph.

3.3.3 Part Groups Generation

Many assemblies contain base components that are being used multiple times in different locations. One common example would be the use of standardized nuts and bolts as supporting elements in a model. Furthermore, the same base components are often reused in common assemblies with an inherently repetitive structure, such as the pistons and valves for cylinder heads in car engines. A large assembly that contains hundreds of parts may therefore be comprised of only a few base components. Based on these assumptions, we identify geometrically identical parts in an assembly and generate according part groups for ensuing preprocessing steps. We define two parts to be geometrically identical, if they

can be made to appear congruent by applying a translational motion to one of them. Thus, parts with different rotation or scaling applied to them may not be added to the same group, even if they are instances of the same base component. The benefit of using part groups is two-fold: first, analytical methods in the preprocessing stage may infer properties for multiple parts from a single mesh. Since parts in a group only differ in terms of translation, any translationally invariant method that is applied to a part generates one result for the entire group. Second, the same geometric information can be reused internally during rendering, which greatly reduces the time required for loading and converting input meshes for visualization. An example for applied part groups generation is depicted in Figure 3.8.

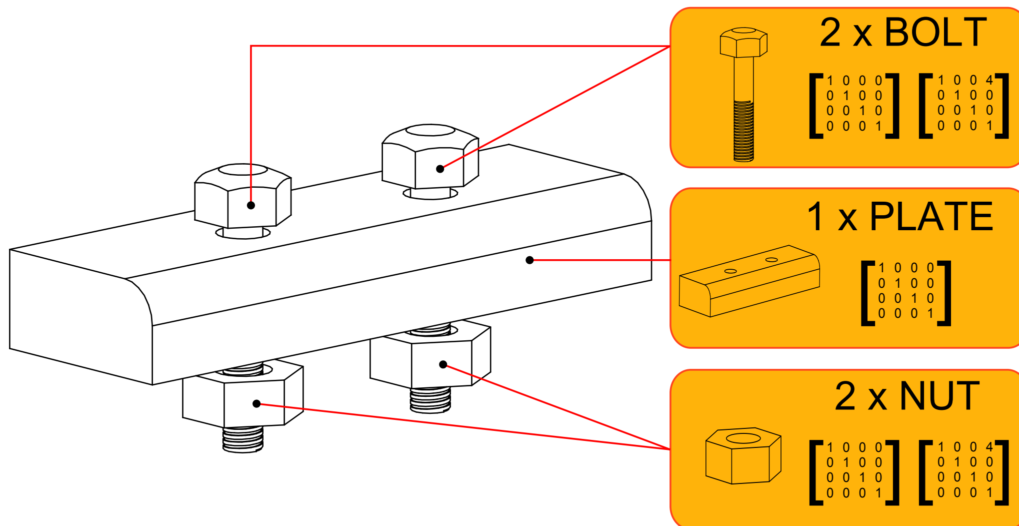


Figure 3.8: An assembly consisting of five parts is divided into groups by reusing the same mesh information. The resulting three groups are defined by one representative polygon mesh and a group of transformation matrices that are used to place all instances of the base component in their proper location.

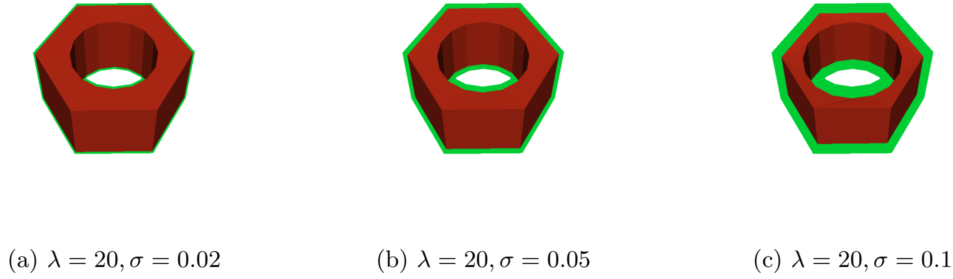
3.3.4 Mesh Shrinking

In order to account for artifacts caused by imprecisions in the geometric representation of parts, we use an iterative algorithm to shrink the representative polygon meshes of each part group by a fraction of their original extent. By doing so, we introduce a mechanism that enables the user to define a tolerance value that can help reduce the probability of false positive results during blocking relationship analysis. Though the name may suggest otherwise, mesh shrinking is not equivalent to down-scaling the object in question. Since

scaling moves each vertex closer to or farther from the reference point around which scaling is performed, the resulting object may occupy space that was formerly vacant. Consequently, a down-scaled model may cause positive results for blocking relationship evaluation where the original would not. Instead of scaling, mesh shrinking thus performs a reduction of the original mesh that ensures that the blocking relationships of the reduced object are a logical subset of the original.

The result of the mesh shrinking is similar to the principle of erosion of 2D images or 3D volumetric data sets in computer graphics. In contrast to conventional erosion, the employed method ensures that the skeleton of the mesh remains intact. Thus, the basic shape of each part is preserved when shrunk. During the shrinking process, the mesh is morphed by moving each vertex in the direction of its weighted inward vertex normal, away from the surface. We calculate the vertex normals as the average of all normal vectors of incident triangles and weight the influence of each normal vector by the opening angle of the triangle edges in contact with the vertex. We then determine a safe distance by which each vertex can be moved without causing the mesh to become degenerate or self-intersecting. This process is repeated for a number of iterations as specified by the user. The maximum distance by which a vertex should be moved can be defined by the user as well. Figure 3.9 displays different results of the mesh shrinking algorithm being applied to the model of a bolt nut with a given number of iterations λ and a percental value σ by which the mesh should be reduced. Holes and openings become noticeably larger while protruding structures and prominent features are scaled down.

Since bolts and nuts are very common elements in most kinds of assemblies, being unable to handle threaded models can become a severe constraint for disassembly planning. However, unfastening a threaded bolt involves a twisting motion, i.e. a combination of translation and rotation. Testing the feasibility of such removal actions is far from trivial since the ratio of translation and rotation speed is dependent on the structure of the threading, which may vary from model to model. As a positive side effect of the mesh shrinking process, bolts may be reduced such that the threading is no longer considered an obstacle when testing translating motions. Consequently, it becomes possible to process a wide range of highly detailed assemblies that contain threaded nuts and bolts without accounting for complex twisting motions in the set of supported removal actions.



(a) $\lambda = 20, \sigma = 0.02$ (b) $\lambda = 20, \sigma = 0.05$ (c) $\lambda = 20, \sigma = 0.1$

Figure 3.9: Iterative mesh shrinking method applied to a simple bolt nut model. λ defines the number of iterations performed, while σ denotes the threshold for shrinking the part based on its original bounding dimensions. The resulting mesh is rendered in red as an overlay to the original object in bright green. In contrast to simply scaling the part, the reduced object is completely contained within the mesh of the original and does not penetrate space that is not occupied by the original model as well. By increasing the opening in the bolt nut, the probability of finding valid removal actions for the associated bolt is increased accordingly.

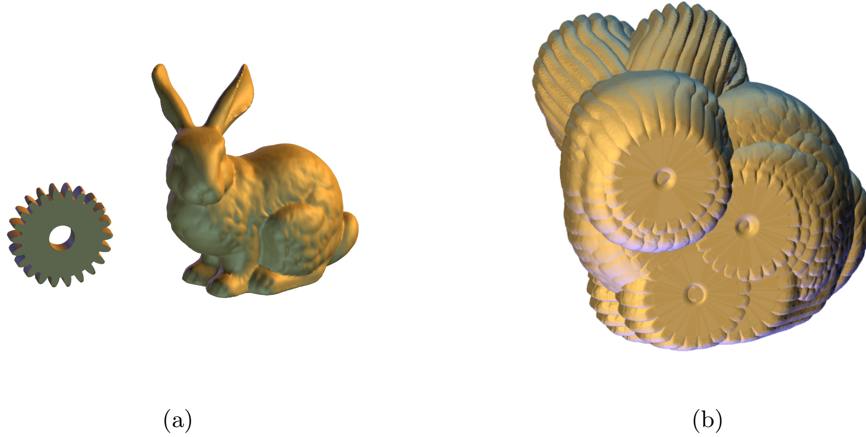
3.3.5 Blocking Relationships

Blocking relationships define a key factor to disassembly planning, since the removability of a subassembly can be directly inferred from them at each point in time; if for a given part A and a removal action ω all blocking relationships $B_r(A, B, \omega)$ with each active part B are negative, ω can be used to remove A from the assembly. We calculate the blocking relationships for each part in the assembly using a discrete set of removal actions consisting of singular and dual translating motions. We make sure that for each removal action, the final translating motion can be extended into infinity. This way, if a removal action ω with no positive blocking relationships exists for a part A, we can conclude that A is globally free. We use a parallelized method for generating detailed C-Space objects which can easily be evaluated with high precision for all removal actions by exploiting the rendering pipeline.

3.3.5.1 C-Space Object Generation

C-Space approaches are commonly found in robotics and path planning applications. They provide a precise method for evaluating the feasibility of moving one object past another. A C-Space object for testing translating motions for an object A with respect to another object B is generated by using the Minkowski sum to compute $\neg A \oplus B$. The Minkowski

sum of two polyhedra can be envisioned as the result of "sweeping" one object along the boundaries of the other. An exemplary setup with the resulting C-Space object is illustrated in Figure 3.10. The output of this computation is a collection of geometric primitives that allow us to efficiently determine translating motions that can be extended into infinity without collision: if a ray can be cast from the coordinate origin (i.e. null vector) in a given direction $\boldsymbol{d}$ without hitting any of the primitives, moving A past B in direction $\boldsymbol{d}$ is geometrically feasible. C-Space object generation and testing can be directly adapted to calculate the blocking relationships for pair-wise part combinations. For a tested moving part A and a translating removal action ω, $B_r(A, B, \omega)$ is positive for every part B where at least one of the line segments representing the translating motions of ω extending from the coordinate origin intersects $\neg A \oplus B$.



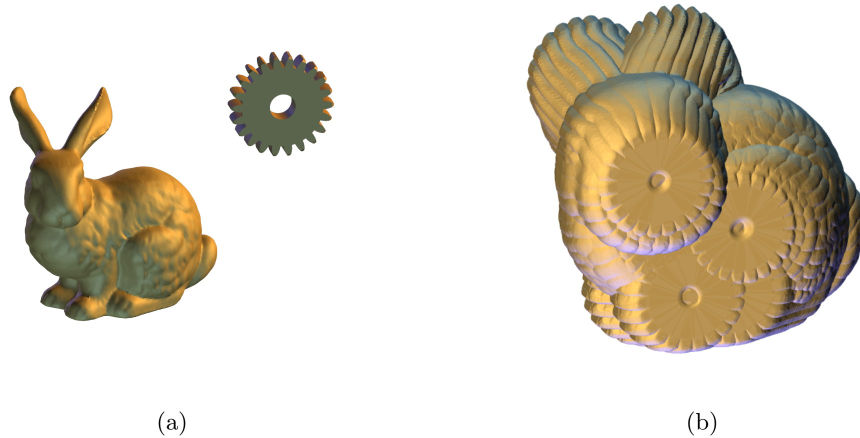
(a) (b)

Figure 3.10: (a) The initial setup for C-Space object generation, with two input polygon meshes, gear and bunny. The gear represents the moving part A, while the bunny is considered a potential blocking object B. (b) The final C-Space object of A and B that is obtained by calculating the Minkowski sum $\neg A \oplus B$. The generated primitives illustrate the shape that results from "sweeping" the inverted gear along the boundaries of the bunny model.

Although evaluation of C-Space objects is a fast and precise way to determine blocking relationships, the creation of precise C-Space objects in an acceptable amount of time becomes challenging when considering highly detailed meshes with several thousand triangles [9, 21, 31]. We have implemented the approach proposed by Li and McMains in [20]. They describe a robust parallel algorithm for generating Minkowski sums that may include redundant primitives, but are well suited for rendering purposes. By exploiting the computation capabilities of GPGPUs in combination with the highly parallel work-

load of Minkowski sum calculation, a noticeable speedup is achieved when compared to conventional approaches. Using this method, we are able to quickly compute the C-Space objects for triangle meshes with 100,000 triangles and more.

Since the shape of an object generated via Minkowski sum calculation is independent of the relative position of the two input objects, the operation can be considered translationally invariant. The setup in Figure 3.11 produces a C-Space object whose appearance is identical to the result in Figure 3.10, even though both input objects have been moved from their original position. However, rotating or scaling an object will change the outcome, as can be seen in Figure 3.12.



(a) (b)

Figure 3.11: (a) A slightly modified setup, where each object has been displaced using translating motions. Again, the gear model represents the moving part A and is thus inverted during C-Space object computation. (b) The C-Space object that was generated for the modified setup. When compared to Figure 3.10, there is no noticeable change in appearance, since the result of the Minkowski sum calculation is translationally invariant.

In Section 3.3.3, we defined that the same rotation and scaling factors are applied to all members in a part group. Parts in the same group only differ regarding their spatial position, which is encoded in the fourth column of the associated transformation matrix. Hence, since the result of the Minkowski sum calculation is translationally invariant, we do not need to create the C-Space objects for all possible part pairings in the assembly. Instead, we only generate the C-Space objects for all pair-wise combinations of part groups and simply change the position of the coordinate origin to evaluate the blocking relationships for parts at different locations. For two part groups $\mathcal{A}$ and $\mathcal{B}$, the common C-Space object is built from their respective representative meshes. The new origin for evaluating the possible motions of a part $A \in \mathcal{A}$ with respect to a potentially blocking part $B \in \mathcal{B}$ is

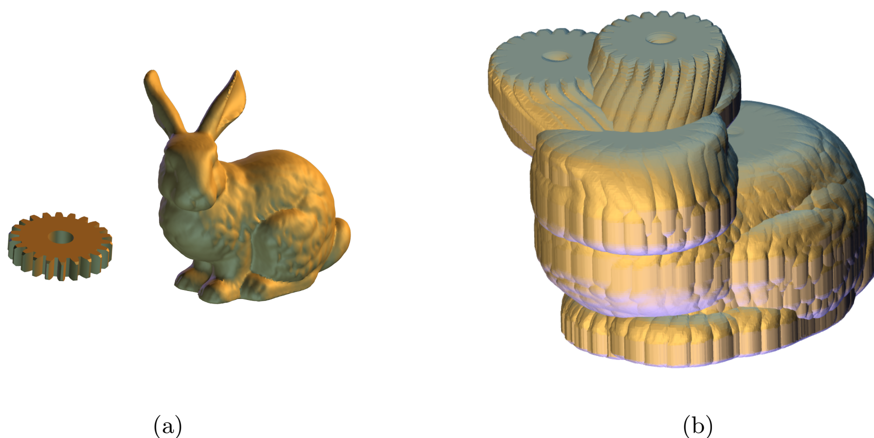


Figure 3.12: (a) Another setup for C-Space object computation, where the gear model representing the moving part A has been rotated by 90°. (b) The obtained C-Space object differs strongly from the results in Figures 3.10 and 3.11, since the Minkowski sum calculation is not invariant to rotation.

then simply obtained by subtracting their respective translations in world space, formally $Origin' = Trans_A - Trans_B$.

3.3.5.2 Singular Translating Motions

Singular translating motions are the most common removal actions to be tested during disassembly path computation. They can be applied to parts that can be removed in one straight motion, such as the lid of a crate or the top-most element of a stack. Whether a feasible motion is detected strongly depends on the directions that are being tested. Some existing implementations use analytical approaches to detect arbitrary free directions, while others refer to a standard set of directions such as the Euclidean unit vectors. We use a discrete set of singular motions to reduce memory and computational demands at this stage; blocking relationships for a part are calculated for the positive and negative instances of its three principal axes. Thus, altogether six singular translating motions are tested for each part. We evaluate the feasibility of moving one part past another for each of the singular motions by checking for intersections with the C-Space object. However, instead of casting a ray in the tested direction, we use perspective projection with a small, customizable field of view (FOV). The angle α that defines the extent of the FOV can be used to apply tolerance to the blocking relationship evaluation: if a part is blocked in the tested direction, the motion may still be considered geometrically feasible

if there exists an unblocked direction that diverges by only a few degrees. We test this by setting up a quadratic high resolution render target and counting the samples that have successfully passed the rasterization stage of the rendering pipeline. If the number of passed samples is lower than the resolution of the render target, at least one free direction exists whose directional vector from the origin lies within the viewing frustum. Perspective projection and sample counting can be performed efficiently using the standard features of the OpenGL rendering pipeline. Figure 3.13 shows a simple C-Space object that is evaluated with tolerance using a viewing frustum with FOV α and the rendered image in a 8x8 render target. By repeating this procedure until all blocking relationships have been evaluated, we receive a list of blockers that is written to an output file for use during the disassembly path computation.

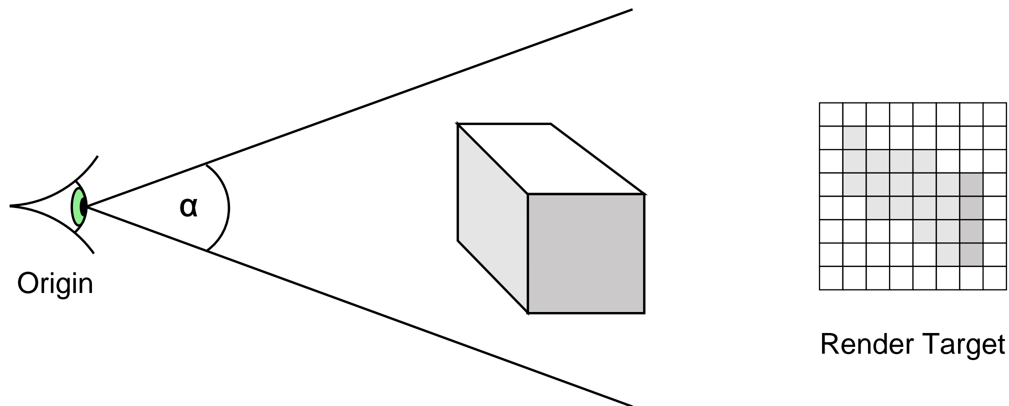


Figure 3.13: Perspective projection and field of view for testing singular translating motions with C-Space objects. The opening angle α of the viewing frustum defines the tolerance for evaluating a given direction. This image illustrates a case where a very large α has been chosen for demonstration purposes only. While a straight line extending from the origin would directly intersect with the C-Space object, the evaluation of the perspective projection leaves some pixels of the render target unset. Thus, a valid passing motion exists that only diverges from the tested direction by an angle smaller than $\sqrt{2}\alpha$.

3.3.5.3 Dual Translating Motions

A noteworthy feature of our disassembly planner is the ability to automatically calculate blocking relationships for a number of dual translating motions. We use a discrete set of vector pairs, which contain a primary and a secondary direction that define the first and second translating motion respectively. We only test right angular dual translating motions, which means that we require that the directional vectors of the first and second

translation form a right angle. The set of tested vector pairs for each part is created by forming all combinations that satisfy these requirements from the positive and negative Euclidean unit vectors as well as the principal axes of the object. Thus, up to 48 dual translating motions are evaluated for each part in the assembly by default. Additional vectors can be introduced by the user prior to calling the preprocessing module in order to further extend this set. For each of these motions, we define a number of evenly spaced positions along the primary direction where the first translation may stop and the second translation ensues. Each of these stopping points can be interpreted as the origin for testing a separate singular translation in the secondary direction. Thus, for each stopping point, a data set of potential blockers may be acquired as discussed in Section 3.3.5.2, although tolerance via FOV is omitted. Technically, choosing a number of stopping points n thus increases the number of tested dual translating motions and according data sets to $24n$. However, we prefer to relate the term dual translating motion to the concatenation of these data sets for the primary direction, which is also referred to as the **dual chain**. Additionally, we need to store for each dual translation the set of parts blocking the primary direction and the according distances. This information is necessary to determine which segments of the dual chain should be evaluated during disassembly path computation.

Given a set of C-Space objects, we use orthogonal projection to obtain the contents of the dual chain for a dual translating motion D. We use a camera setup where the $\boldsymbol{up}$ and $\boldsymbol{z}$ vectors are defined by the primary and secondary direction of D respectively. The vertical size of the view plane in world space defines the space of possible stopping positions in the primary direction. The projected image is written to a render target with a horizontal resolution of 1 pixel. The vertical dimension of the render target directly defines how many stopping points in the primary direction are considered. The result of rendering a C-Space object is evaluated by checking each pixel in the render target and, if it has been set, adding the ID of the tested blocker to the corresponding data set in the dual chain. Figure 3.14 illustrates the process of acquiring the contents of the dual chain using a camera setup for orthogonal rendering.

Since we use all valid combinations in the two given sets of right angular vectors, for each dual direction D there exists a tested dual direction D' such that the primary direction of D equals the secondary direction of D'. Also, the first segment in each dual chain is chosen to lie at the origin, hence there is no translation in the primary direction. Thus, the primary blockers of D can be easily extracted by directly copying the information from

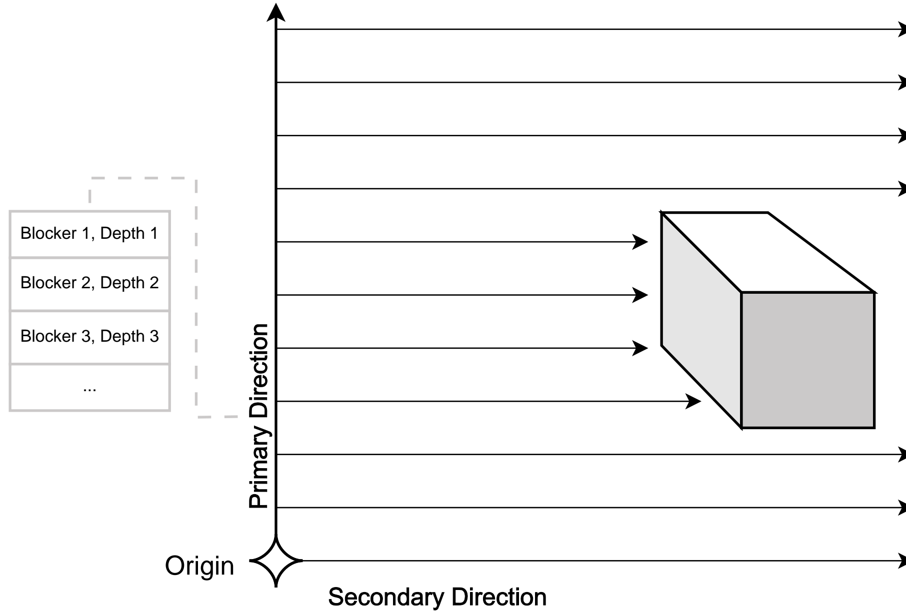


Figure 3.14: A C-Space object being rendered using orthogonal projection to evaluate dual translating motions. The primary translation is represented by the ***up*** vector of the camera that is used for rendering, the secondary translation is defined by its z-axis. In addition to the sets of blockers for each segment of the dual chain, a list of value pairs is maintained for each dual motion that contains the IDs of all blocking parts in the primary direction and the according distances.

the first segment in the dual chain of D', where the corresponding depth values are stored as well.

3.3.5.4 Bounding Box C-Space Objects and Separator Occlusion

In order to reduce the computational demands for blocking relationship detection, we use approximation and occlusion culling techniques to skip unnecessary calculations. The evaluation of bounding box C-Space objects can be considered as a filtering stage to the actual procedure; although the parallel Minkowski sum method is quite efficient, it may take some time to complete if both input meshes feature a high number of triangles. However, creating a C-Space object from the axis-aligned bounding boxes (AABB) of two parts can be achieved in constant time and provides a robust approximation of the real C-Space object. Therefore, prior to creating and evaluating the C-Space objects of their triangle meshes, we evaluate the blocking relationships for parts using only their bounding boxes. However, since a bounding box is just an approximation of its contained polygon mesh, we simply check for each removal action whether any samples are written

to the render target at all. If so, the part against which the removal action is tested is a potential blocker. For each part and each removal action, a list of potential blockers is maintained which are then tested again using the representative meshes of their respective part groups. If for two part groups $\mathcal{A}$ and $\mathcal{B}$ no part $A \in \mathcal{A}$ exists that is influenced by any part $B \in \mathcal{B}$ considering the bounding box C-Space objects, we can trivially discard the C-Space computation of their representative triangle meshes.

If however two members of $\mathcal{A}$ and $\mathcal{B}$ respectively are influenced by each other, we may still skip all polyhedral C-Space evaluations of A and B for removal actions ω where $B_r(A, B, \omega)$ as obtained from bounding box C-Space evaluation is negative. Consequently, assemblies with a high number of simple parts and a low number of part groups can be processed much faster, since for those setups the highest computational effort is required when processing the rendering stages during polyhedral C-Space evaluation.

We can further enhance the described method by making use of the information provided by the user concerning separating parts as mentioned in Section 3.3.1. We discard the computation of all blocking relationships that become irrelevant in the presence of parts that define a separating, fixed structure for the assembly: if for a removal action ω part A is blocked by a separating part, blocking relationships for ω with parts that lie beyond the separator can be discarded. Figure 3.15 depicts an assembly where 50% of all blocking relationships can be trivially discarded prior to detailed evaluation.

Prior to the evaluation of the bounding box C-Space objects for a part A, we generate the polyhedral C-Space objects of A against all separating parts and render them to the depth buffers of the render targets for C-Space evaluation. Thus, if depth testing is enabled, no samples will be drawn to the render target for a bounding box C-Space object of A and B that is occluded by a polyhedral C-Space object of A and a separating part. Consequently, the number of C-Space object evaluations is reduced even further. Based on the assumption that for many assemblies a low number of separators occupy a considerable amount of space, this can have noticeable influence on the program's runtime.

3.4 Disassembly Planning Application

The DPA provides the user interface for computing, editing and visualizing disassembly sequences. In contrast to the preprocessing module, the DPA was designed for real-time use on a wide range of computing systems. Thus, the requirements for performance are deliberately kept low to ensure compatibility with less powerful equipment. This is strongly reflected in the employed algorithms for disassembly path computation and

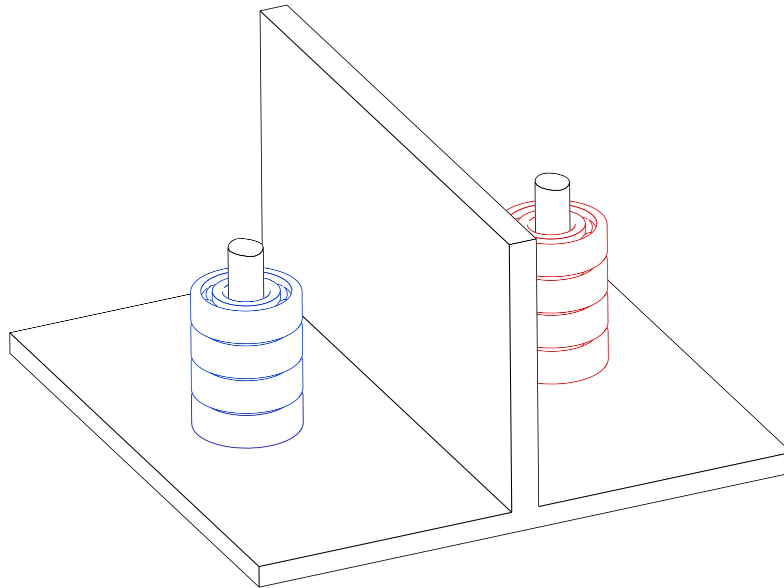


Figure 3.15: An assembly consisting of a base plate with an elevated panel that separates two stacks of ball bearings. Due to the shape of the base plate, all possible blocking relationships between blue and red parts become irrelevant for any singular or right angular dual translating motion. The number of required blocking relationship evaluations is thus effectively halved.

customization. Based on the user input, a specific disassembly problem may be defined by selecting required parts. The planning application first automatically evaluates the static assembly information from the preprocessing steps in order to suggest solutions using a simple peeling algorithm, without making any assumptions about partitioning or order of removal. Disassembly sequences can be altered by changing the properties of individual parts, thus changing their behavior and implied constraints during the disassembly path computation. Removal actions that were not considered in the preprocessing phase can be entered by hand and verified to complement the available set of blocking relationships. For each suggested disassembly sequence, a preview is generated using hierarchy based explosion diagrams, as well as a step-by-step series of instructions for animation.

3.4.1 Disassembly Customization

Due to the factors considered in Sections 3.1.2 and 3.1.5, an automatically computed disassembly path may not provide an acceptable solution to a given disassembly problem. Since a disassembly path consists of a chain of consecutive removal actions whose geo-

metrical feasibility is dependent on the chronological order in which they are performed, editing the path itself can be a rather complex task (see Section 3.1.6). Instead of editing individual steps, we use modifiable properties that can be set for each partition and define their behavior during disassembly path computation.

3.4.1.1 Partitioning

The process of partitioning is based on the concept that the entire assembly can be represented by a hierarchical collection of partitions. The **code** of a partition describes a set of values that identify the parts of which the partition is comprised. Each partition is defined by its code, immediate children and parent partition. The union of the codes from all immediate children of a parent partition must be equal to the code of the parent. The immediate children of a parent partition are themselves sub-partitions of the parent. A sub-partition of a parent partition can only contain a subset of the parts that define the parent. No parts can be shared between partitions that are not connected via such parent-child relationships. Since we do not consider partition creation and testing during the disassembly path computation, all partitions and their relationships need to be defined beforehand. When a new partition is built from a set of smaller partitions, the removal actions against which it can be tested are obtained by finding all removal actions that have been evaluated for all contained parts. The minimal set of potential removal actions for a partition in our system is thus defined by the dual translating motions from Euclidean unit vectors, since these are evaluated for each movable part in the assembly. The corresponding blocking relationships of the new partition are defined by the union of blocking relationships of the children. Positive blocking relationships with parts inside the created partition are discarded, since the partition should not be blocked by its own contents. Furthermore, all blocking relationships that involve parts outside of the new parent partition are removed from all of its sub-partitions.

Automatic Partitioning Initially, the program uses naive partitioning to create the basic setup of partitions. Thus, for every part X that is loaded to the assembly, a separate partition $\boldsymbol{P_X}$ is created and attached as an immediate child to $\boldsymbol{P_0}$. Since we calculate the blocking relationships during C-Space evaluation on a per-part basis, in this initial step we only need to copy the available blocking relationships for a part to the according partition. Next, we fuse all pairs of partitions that are "stuck together" according to the information from the preprocessing stage: due to imprecisions in the mesh representation

of an assembly or the requirement for complex removal actions that cannot be resolved using the employed mechanisms, the final assembly may contain pairs of parts that cannot be separated from each other. For instance, there may be no removal action that can be used to separate part A from part B and vice versa due to intersections of their polygon meshes. Before initializing the DPA for disassembly path computation, we automatically fuse these groups of inseparable parts into larger partitions. By combining the pair-wise sets of inseparable parts, bigger clusters may be formed. For instance, the part pair (A, B) may be inseparable, as well as (B, C), thus defining a cluster $\{A, B, C\}$. For each of the eventual clusters, a new partition is created whose immediate children are the contained parts. The benefits of this method are two-fold: first, there is a chance that two parts cannot be separated because they were never intended to, thus the program automatically adopts this concept. Second, even if there exists a valid solution for separation in the original assembly, the user may never require to access one of the inseparable parts. Also, since there is no possibility for automatically detecting a way to remove the stuck parts one by one, fusing them into one partition can only increase the chances for removal. Thus, removing the automatically generated partition may be feasible and become part of any disassembly sequence.

Custom Partitioning In addition to the automatically created partitions, the user is free to define any number of partitions that abide to the basic rules of partitioning. We support three operations for partitions, namely fusing, collapsing and flattening. Fusing requires the selection of a number of partitions $\{\boldsymbol{P_A}, \boldsymbol{P_B}, \boldsymbol{...}\}$ that share the same parent partition $\boldsymbol{P_P}$, which are then combined to form the children of a new partition $\boldsymbol{P_x}$. $\{\boldsymbol{P_A}, \boldsymbol{P_B}, \boldsymbol{...}\}$ are removed from the list of children of $\boldsymbol{P_P}$, while $\boldsymbol{P_x}$ is added as a new child. Collapsing works on a single partition $\boldsymbol{P_x}$ and attaches all of its immediate children to its parent partition. Subsequently, $\boldsymbol{P_x}$ is deleted and all references to it removed.

Consecutive utilization of fusing and collapsing may create complex subhierarchies inside a partition. In order to quickly remove all hierarchical structures, we provide a flattening operation that detects all indivisible partitions contained in a partition $\boldsymbol{P_x}$ and eliminates all intermediate sub-partitions. The result of flattening $\boldsymbol{P_x}$ is a partition $\boldsymbol{P'_x}$ whose immediate children are without exception partitions of size one (i.e. singular parts). For instance, applying the flattening method to $\boldsymbol{P_0}$ after loading an assembly will factually nullify the effects of automatic partitioning.

3.4.1.2 Removal Actions

For each partition in the assembly, one of the evaluated removal actions can be assigned to be used during disassembly path computation. Defining the designated removal action for a partition can be helpful if the user has specific ideas or in-depth information about the way the partitions should be removed. Changing the designated removal action of a required object can also have considerable impact on the solution that is generated by the system (see Section 3.1.6). Thus, the ability to freely select which removal action should be used for a specified partition provides an intuitive tool for customizing the disassembly path.

In order to help the user identify which removal action should be selected, we place 3D arrows in the scene to visualize the corresponding motion relative to the edited partition. For singular translating motions, a single arrow extends from the center of the partition and points in the direction that represents the vector of translation. For dual translating motions, two arrows are used, their orientation being defined by the primary and secondary directions of the dual motion respectively. The first arrow originates at the center of the edited partition, while the second arrow extends from the tip of the first. The length of the first arrow depends on which primary stopping position should be used. The user is free to choose any evaluated primary stopping position, thereby modifying the extent of the first and the position of the second arrow.

3.4.1.3 Extended Motions

In addition to the removal actions that were evaluated as part of the static assembly information, we allow the user to define custom removal directions that consist of an arbitrary number of consecutive translations. When selecting this option, the user is queried to define a series of vectors that are then concatenated to represent the desired motions for the removal action. Following this definition, the DPA calculates the blocking relationships of the new removal action for disassembly path computation. Since the DPA is not necessarily run on a system that is capable of programmable parallel computing, we cannot employ the C-Space method described in Section 3.3.5.1. Instead, we use sweeping to generate a set of primitives that can then be tested for intersection with potential blockers. For each translating motion t_i in the custom removal action, the triangle meshes of the partition are first moved according to the preceding translation t_{i-1}. For each edge e in the repositioned triangle meshes, we create a duplicate of e, denoted as e', which is translated by t_i. e and e' can then be combined to define a quadrilateral

with parallel opposite sides. Each created quadrilateral is tested for intersection with the triangle meshes of the remaining partitions in the assembly. This way, accurate blocking relationships can be calculated for arbitrary sequences of translations.

3.4.2 Disassembly Path Computation

The core functionality of the disassembly planning system is embedded in the disassembly path computation method. The main task of this method is to suggest suitable disassembly paths to the user, while ensuring that all rules and properties that were set for individual partitions are upheld. Since our concept targets potentially large assemblies with high geometrical detail, it is key that we use an approach that scales well and is not dependent on the complexity of the input objects. We have decided to employ a simple peeling algorithm that iteratively detects and removes free partitions at each point in time which is also referred to as a peeling **phase**. Figure 3.16 displays an example of an assembly where each partition is labeled with a number that indicates the according phase in which it is free to be removed. The iterative partition removal is followed by the detection of dependencies between moved parts in order to discard unnecessary removal actions from the suggested disassembly procedure.

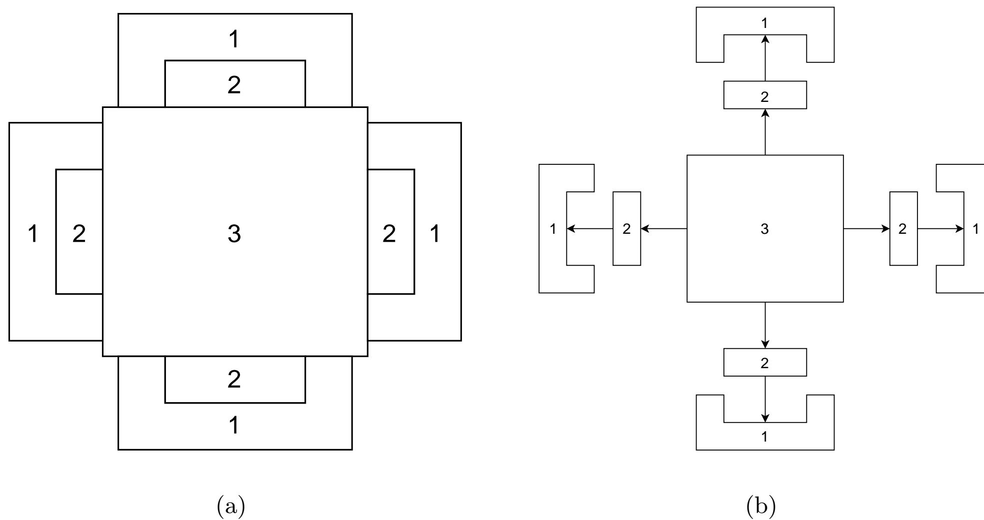


Figure 3.16: (a) An assembly where each partition is labeled with a number indicating the phase during which it can be removed using a peeling approach. (b) The exploded assembly where each partition is moved according to the free removal action that was detected in the corresponding phase.

3.4.2.1 Iterative Part Removal

During the peeling procedure, we iteratively check and remove all partitions for which a geometrically feasible removal action exists in each phase. We start by adding all immediate children of P_0 to the list of observed partitions. If a partition P_A is free, it is removed from the list of observed partitions and marked with the number corresponding to the phase during which its subassembly freedom was detected. All blocking relationships that involve exactly one of the parts contained in P_A are set to **false**, thus all partitions that are influenced by the contents of P_A except for its sub-partitions have their blocking relationships updated. Furthermore, all immediate children of P_A are appended to the list of observed partitions. Since the immediate children of a parent partition P_P have no positive blocking relationships with parts outside of P_P, the system is able to compute accurate disassembly steps of multiple separate partitions in a single phase. The procedure stops as soon as all required partitions have been successfully removed. We distinguish two separate conditions for determining subassembly removability. For partitions where the user did not select a designated removal action, the system iteratively checks all blocking relationships corresponding to the available removal actions for each phase. A partition $\boldsymbol{P_A}$ is marked free if at least one removal action of $\boldsymbol{P_A}$ is geometrically feasible with respect to all other active partitions. In this case, ω is stored for $\boldsymbol{P_A}$ to be used during dependency detection. If, however, a designated removal action has been declared for $\boldsymbol{P_A}$, the system will only test in each phase the geometrical feasibility of this one action.

Detecting Free Singular Translating Motions Singular translating motions are the simplest type of removal actions, thus we choose to test them first in each phase of the disassembly path computation and thus prefer them over more complex removal actions. Since in each phase all positive blocking relationships related to the removed parts are invalidated, we simply check for each partition $\boldsymbol{P_x}$ whether the set of positive blocking relationships for any singular translating motion ω is empty. If so, all partitions that contain blockers of ω have been disconnected from the parent partition of $\boldsymbol{P_x}$. Thus, $\boldsymbol{P_x}$ can be successfully removed from the assembly using the corresponding singular translating motion.

Detecting Free Dual Translating Motions Dual translating motions are more complex to handle, since we are considering movement in two separate directions. As mentioned in Section 3.3.5.3, for each dual translating motion we store its primary and sec-

ondary direction, as well as its dual chain and set of primary blockers. Which sets of the blocking relationships in the dual chain should be tested depends on the available segments at each point in time. The available segments of a dual chain can be inferred from its primary blockers. The primary blockers define a list of parts that inhibit the movement along the primary direction as well as their proximity. The distance of the closest primary blocker dictates which segments of the dual chain can be accessed for testing their corresponding blocking sets. The entries in this list can themselves be considered blocking relationships, and are updated in each phase along with the blocking sets of each segment. Thus, by removing a part that is a primary blocker of ω, more segments may become available for testing. Removing a secondary blocker may cause a blocking set of a segment to become empty, though the segment itself may be inaccessible. Hence, finding a free dual translating motion depends on two factors, namely identifying reachable segments of the dual chain and checking whether one of the corresponding blocking sets is empty. Figure 3.17 illustrates an example where certain parts of the dual chain are excluded from testing in the presence of primary blockers.

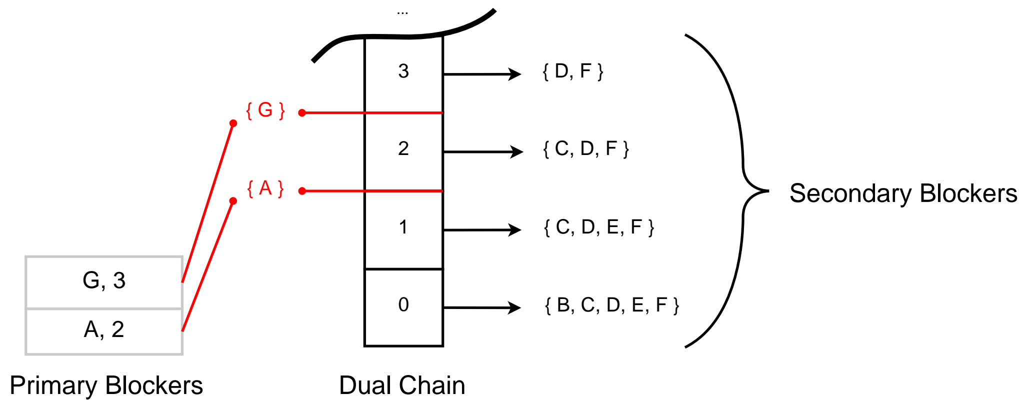
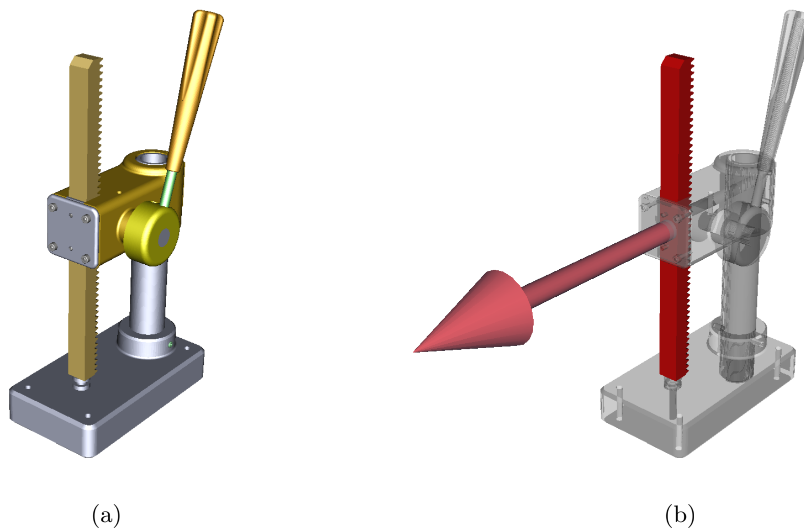


Figure 3.17: A schematic visualization of the blocking information that is stored for a dual translating motion of partition $\boldsymbol{P_x}$. The primary blockers represent those parts that inhibit movement along the primary direction. The corresponding distance defines the index of the first inaccessible segment in the dual chain if this blocker is present. Since moving in the primary direction past a blocker would be geometrically infeasible, all segments with an index higher than the distance of the closest primary blocker can be ignored. In each phase of disassembly path computation, all accessible segments are checked whether their corresponding blocking set is empty. If so, the dual direction can be used to remove $\boldsymbol{P_x}$ from the assembly. For instance, this would be the case if either $\{C, D, E, F\}$ or $\{A, C, D, F\}$ were to be removed. However, removing $\{D, F, G\}$ would not suffice since A is blocking off all segments with an index above 1.

Validating Designated Removal Actions For partitions that have designated removal actions, we only consider the set of corresponding blocking relationships. Extended removal actions as defined in Section 3.4.1.3 are automatically used as designated removal actions. However, singular and dual translating motions from the static assembly information can also be assigned to a partition, which effectively overrides the automatic detection of removal actions. The system is forced to keep the corresponding partition in the set of observed partitions until the designated removal action becomes geometrically feasible.

3.4.2.2 Dependency Detection

As mentioned above, the peeling algorithm involves removal of all partitions that can be extracted in each phase. Thus, once all required partitions have been disassembled, a high number of redundant partitions may have been extracted that have no influence on their removal. In order to detect and exclude pointless removal actions from the final set of instructions, the calculated disassembly paths are compacted to include only those actions that are necessary for removing required partitions. Figure 5.3 shows a disassembly problem involving the **Press** assembly and a selected required partition, along with a designated removal action, for which a compact disassembly path should be computed.



(a) (b)

Figure 3.18: (a) The complete Press assembly. (b) Display of a disassembly problem with a required partition $\boldsymbol{P_6}$ (red) and a designated removal action for singular translation indicated by the arrow.

In order to eliminate unnecessary steps from the final disassembly path, we build

a dependency hierarchy to recursively find all direct and indirect blockers of required partitions. An indirect blocker of partition $\boldsymbol{P_X}$ in this context is a partition that does not directly block $\boldsymbol{P_X}$ for its removal action, but is still relevant to its removal. For instance, if $\boldsymbol{P_A}$ is blocked by $\boldsymbol{P_B}$ for its stored removal action and $\boldsymbol{P_B}$ is itself blocked by $\boldsymbol{P_C}$, $\boldsymbol{P_C}$ becomes an indirect blocker of $\boldsymbol{P_A}$ because $\boldsymbol{P_A}$ cannot be removed before $\boldsymbol{P_C}$. We detect direct and indirect blockers by examining the results of the peeling algorithm starting with the required partitions and determining all dependencies bottom up. For each partition, we store its dependencies as a set of nodes. Figure 3.19 represents the resulting dependency hierarchy for the disassembly problem illustrated in Figure 5.3 with dependencies indicated by arrows.

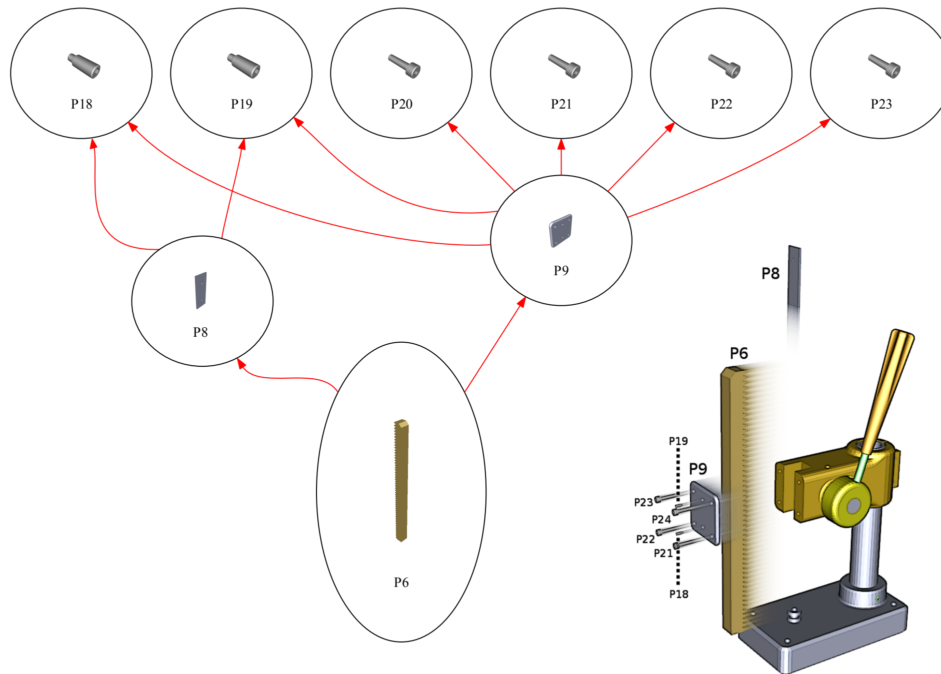


Figure 3.19: Dependency hierarchy for the disassembly problem imposed on the Press assembly. The dependencies of each partition are represented by red arrows connecting the nodes. Although $\boldsymbol{P_{18-23}}$ are not blockers of the removal action assigned to $\boldsymbol{P_6}$, they inhibit the movement of $\boldsymbol{P_9}$ and $\boldsymbol{P_8}$ and are thus considered indirect blockers of $\boldsymbol{P_6}$.

The set of dependencies for a partition is defined by the union of its direct blockers minus all of its indirect blockers, e.g. if a partition $\boldsymbol{P_A}$ is already blocked indirectly by a partition $\boldsymbol{P_B}$, no entry for $\boldsymbol{P_B}$ is made in the dependency set of $\boldsymbol{P_A}$ if $\boldsymbol{P_B}$ is also a direct blocker of $\boldsymbol{P_A}$. This allows us to compactly restructure the resulting hierarchy bottom up:

we define a new set of hierarchy layers, starting with the removal of required partitions as the bottom layer (with respect to dependencies between required partitions). We then check the set of dependencies for these partitions and add them to the hierarchy as the next higher layer. This procedure is repeated for each new layer until there are no more dependencies left to process.

The benefits of this approach are two-fold: first, each partition is removed as late as possible without stalling the disassembly procedure. Thus, it becomes easier for the user to establish a connection between partitions removed in one phase and the next, since each removed partition on layer i is directly dependent on the removal of partitions in layer $(i-1)$. Second, since removed partitions will be placed on the lowest possible hierarchy layer and our previewing method of disassembly paths involves explosion diagram generation based on the layers of the dependency hierarchy, removed partitions will be positioned closer to their initial position, ensuring a more compact representation in the resulting diagram (see Section 3.4.4.1 for details).

3.4.3 Correcting Disassembly Paths

If the disassembly path computation does not arrive at a state where all required partitions have been removed using the described peeling method, algorithm execution is considered to have failed. This can be due to the following reasons:

- **Infeasible user constraints.** The user may choose a designated removal action that would cause a partition to be translated through a fixed part or separator. Another possibility is the creation of a circular dependency, where in a group of partitions $\boldsymbol{P} = \{\boldsymbol{P_0}, \boldsymbol{P_1}, ...\}$, each partition $\boldsymbol{P_x} \in \boldsymbol{P}$ is blocked for its designated removal action by at least one other partition $\boldsymbol{P_y} \in \boldsymbol{P}$.

- **Complex removal actions.** The tested set of singular and dual translating motions does not suffice to find a feasible removal action for a partition which needs to be removed.

- **Incorrect partitioning.** A partition $\boldsymbol{P_A}$ cannot be removed, because it is blocked by another partition $\boldsymbol{P_B}$. However, if removing $\boldsymbol{P_A}$ and $\boldsymbol{P_B}$ together is possible and also elemental for disassembling the original assembly, the partitioning hierarchy needs to be updated.

- **Erroneous blocking relationships.** A part A is erroneously blocked by part B for a given removal action ω, even though performing ω of A against B is geometrically

feasible in the original assembly. This may be due to insufficient tolerance values or extensively overlapping meshes.

The DPA helps the user iteratively resolve these issues. If a disassembly path computation fails, we recursively check the blockers of designated removal actions if they have been set, starting with the required parts, until we arrive at a group of irremovable partitions that have no designated removal action. These are then marked as problem partitions for the user to edit. During this process, we also check for futile removal actions and circular dependencies to reveal all infeasible user constraints. If the problem is caused due to two or more partitions being blocked when they should be removed simultaneously as one larger partition, the user can apply the methods described in Section 3.4.1.1 to edit the partitioning hierarchy. As shown in Section 3.4.1.3, extended removal actions can be defined to take care of complex, initially untested removal actions. Finally, we provide an interface to erase specified positive blocking relationships for any removal action.

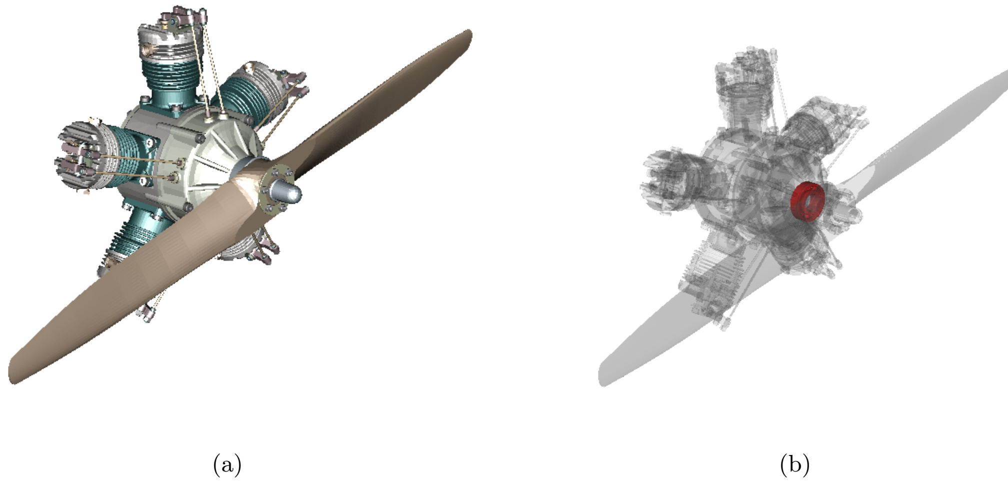
3.4.4 Animation and Illustration

For complex or lengthy disassembly sequences, visual cues play an important role; they enable the user to quickly grasp the individual instructions and required removal actions. While textual instructions are usually more compact and more easily transferred, illustrations and animations establish a visual context between the virtual model and the real assembly. We thus implement a variety of visualization techniques to explicitly convey the steps of a selected disassembly sequence using explosion diagrams and step-by-step animations.

3.4.4.1 Exploded Disassembly Preview

In order to visualize the ramifications of a selected disassembly path, we use explosion diagrams to give an overview of the involved instructions. Each part is translated according to its associated removal action. We make sure that the resulting diagram contains no intersecting objects and conveys the coherence of consecutive removal actions using the hierarchical dependency information extracted during disassembly path computation. An example of a disassembly problem and the explosion diagram representing the calculated disassembly path are shown in Figures 3.20 and 3.21 respectively.

We start by defining a set of parts S_0 representing the lowest layer of the dependency hierarchy and an axis-aligned bounding box $\mathcal{BB}$ that encloses the entire assembly. We then calculate a new position for each part in S_0 such that there is no intersection with



(a) (b)

Figure 3.20: (a) An assembly of a radial aircraft engine as displayed by the DPA. (b) A disassembly problem is defined by selecting the partition highlighted in bright red (ball bearing) for removal. Transparency is applied to visualize the structure of the assembly and its contained components.

$\mathcal{BB}$ or any other part in S_0. $\mathcal{BB}$ is then extended to contain the AABBs of the recently moved parts. We repeat this procedure with $S_1, S_2, ..., S_N$ until all layers of the hierarchy have been processed. Furthermore, we provide an option to activate static motion blur to allow for a better understanding of the removal action that is applied to each removed partition. Figure 3.22 shows a screenshot of an exploded disassembly preview in the DPA with static motion blur enabled.

3.4.4.2 Step-by-Step Animation

After the disassembly path computation routine has successfully ended, we generate a set of phases for disassembly sequence animation. Each phase includes at least one removal action. Similar objects that can be removed at the same time are automatically joined to form groups that participate in the same animation phase. For each phase, we take a screenshot of the assembly before the corresponding extractions are applied. Partitions that are being removed as part of an animation phase are highlighted in red. The screenshots are displayed at the bottom of the screen and can be used to navigate between the different stages of the disassembly procedure, as shown in Figure 3.23.

Selecting one of the screenshots causes the program to show a preview of the animation phase, where all moved partitions are placed in their final positions, connected to their

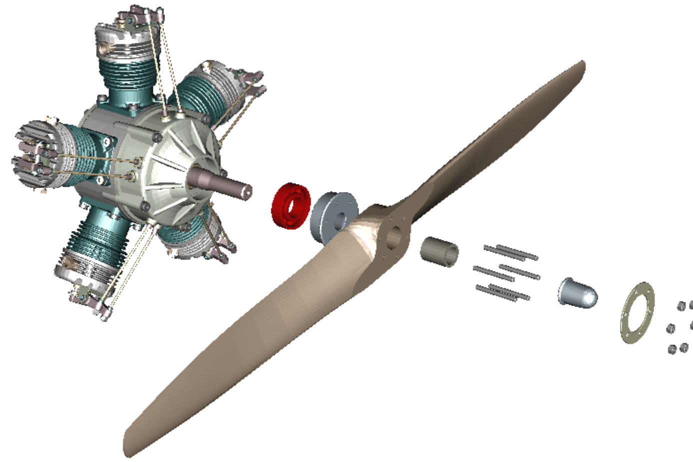


Figure 3.21: Explosion of the partitions involved in a computed disassembly sequence provides an overview of the required removal actions. Each partition is positioned relative to its original location, translated according to its stored removal action and displaced to avoid intersections.

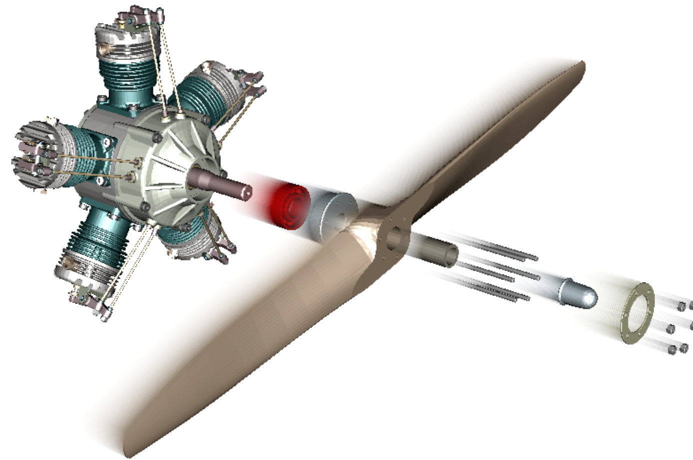


Figure 3.22: Static motion blur is applied to the explosion diagram for a disassembly sequence. The actual motion of each partition is indicated by the blurred images of the partition at intermediate positions during removal.

initial configuration in the assembly by guidelines. In order to provide a clear overview, the camera position is automatically adjusted so that all removal actions of the current phase are captured inside the viewing frustum. Once the camera has been repositioned in this manner, the user is free to change all extrinsic parameters at will to allow focus

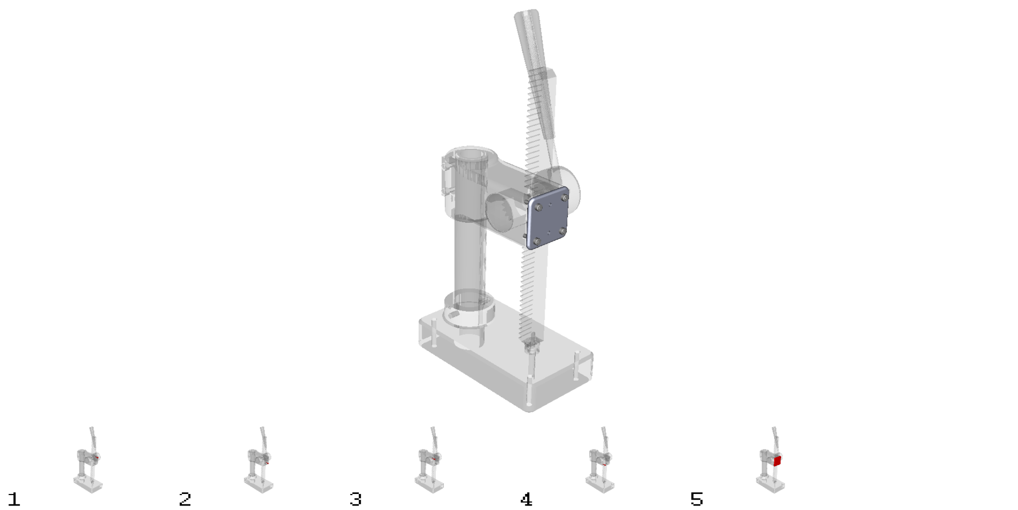


Figure 3.23: The DPA displaying the Press assembly and the screenshots of animation phases for a disassembly sequence. The rendering mode for creating the screenshots can be altered to suit specific requirements; we applied transparency to increase the visibility of the highlighted partitions (red). The images serve both as a preview to the disassembly procedure and as a visual aid to navigate between animation phases.

on arbitrary details. For each extracted partition, the program plays an animation of it being removed along the corresponding directions. The animation is looped until the user either confirms the displayed instruction or aborts the animation process.

Animated partitions that fail to occupy a certain portion of the screen are distinctly highlighted by a billboard of a red circle enclosing them. Furthermore, a second camera is activated in the upper left corner of the screen that provides a close-up view of the partition being moved. Thus, removal of small or occluded partitions that could be easily missed during animation is particularly stressed by the DPA. Figure 3.24 shows a screenshot of the animation for removal of a small partition where billboard highlighting and close-up camera have been activated.

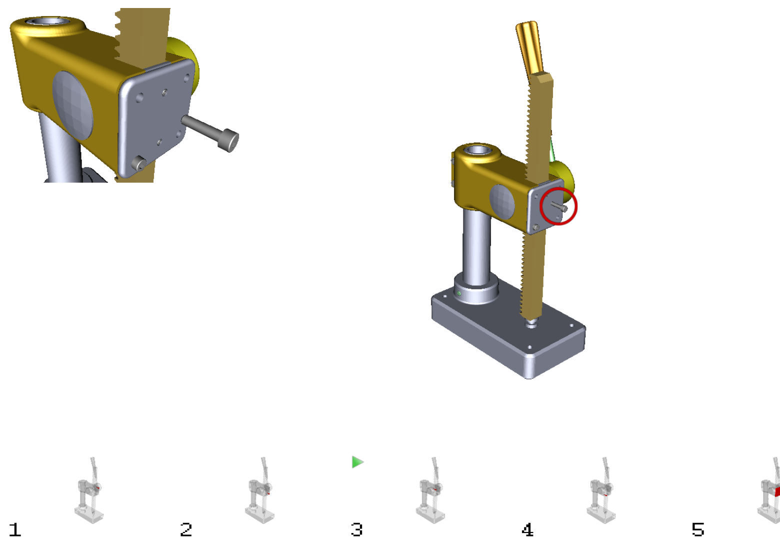


Figure 3.24: Screenshot of the DPA animating the removal of a part. The small green "play button" at the bottom of the screen indicates which animation phase is currently being viewed. The camera has been automatically positioned to capture the entire assembly and all removal actions of the phase in the viewing frustum. A relatively small bolt, which fails to occupy the minimal number of pixels in the view port, is being extracted in this phase. A billboard of a red circle is placed above it during animation to attract the attention of the user. Furthermore, a second view port of a close-up camera is activated in the top left corner to give a detailed view of the partition being removed.

Chapter 4

Implementation

The implementation of the disassembly planning system is based on the C++ programming language and several frameworks for enabling features from computer graphics and parallel computing. We use OpenGL and its scene graph wrapping extension OpenSceneGraph (OSG) for creating rendering environments and the user interface. The extended functionality of OSG includes cyclical animations and transformations, which are used for displaying disassembly procedures. For quick and easy usability of the DPA, we employ the lean AntTweakBar solution for editing disassembly paths and sequences. Parallel programming is enabled by the CUDA 5.0 framework, which also provides OpenGL interoperability for shared resources (e.g. textures) that can be used in both contexts with little overhead. We provide separate CMake build files for the individual modules to allow compilation of the more light-weight DPA on systems that have no support for CUDA 5.0.

4.1 Preprocessing Module

The input to the preprocessing module is provided by files encoded either in the Open Inventor or the Virtual Reality Modeling Language (VRML) format. The preprocessing module is strongly dependent on the parallel processing capabilities of the CUDA 5.0 framework in combination with a CUDA-ready graphics card. The runtime for calculating the static assembly information is thus directly influenced by the performance of the employed graphics card. Several methods of the preprocessing module require a device with a minimum compute capability of 1.2 to allow the utilization of atomic functions. CUDA/OpenGL interoperability is used at several points to allow rendering of primitives that are created by CUDA methods and systematic evaluation of textures that are used

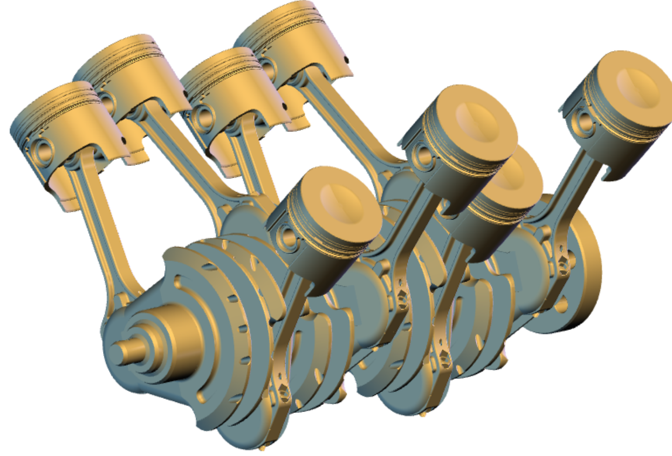
as rendering targets in the OpenGL pipeline.

4.1.1 CAD Data Conversion

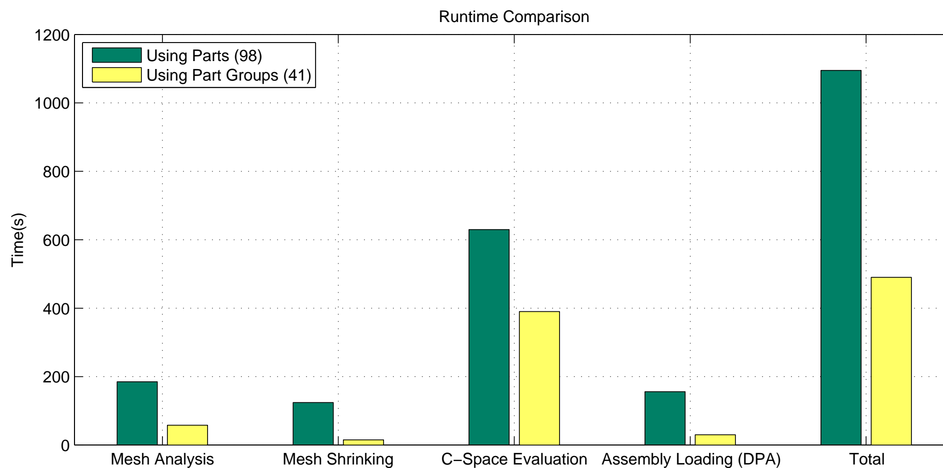
We use the SAP Visual Enterprise Author software to load and convert CAD input data sets to triangle meshes. The geometric detail of the converted parts can be influenced by altering the settings for the automated surface tessellation. The scene is then exported either in Open Inventor or VRML file format. No cameras or similar auxiliary nodes for describing the scene are exported. We make sure that the names describing components in the CAD file are kept and exported along with the geometric data encoding the triangle mesh. Instead of directly writing the absolute position of each vertex in a mesh, we choose to keep the original node hierarchy, including transformations. Thus, all available information about the transformations that were applied to parts during the creation of the CAD model is preserved as part of the output file.

4.1.2 Part Groups Generation

The reuse of components in CAD models is quite common; a node representing a specific component can be simply duplicated and placed in a different location. Usually, the duplicated object will carry the same base name or description to discern its origin. Since part names, node hierarchies and relative transformations are preserved in the exported Open Inventor and VRML files, the transformation matrix for each part can be recreated from the scene by starting with the outermost node and applying all inner transformations to create matrix M. For each object that is then read, M can be stored as the transformation matrix used to position this specific part. By assuming that the nomenclature in the provided CAD models is coherent, part groups can be generated simply by evaluating the extracted transformation matrix for all parts that share the same name. In order to detect differences in rotation or scaling, we compare the first 3x3 entries in the 4x4 transformation matrix. A part A is thus a member of a group $\mathcal{A}$, if the 9 upper left entries of M_A are equal to those in the transformation matrix of the first member in $\mathcal{A}$. Since all parts in a group share the same geometric information, the basic triangle mesh of the group representative is stored only once. For each such part group, a text file is created which contains the number of members in the group and the transformation matrices of all members. This information suffices to recreate a group of parts as defined in Section 3.3.3 for evaluation or rendering in all further processing steps. The positive effect of part grouping on the runtime is illustrated using the **Cylinder Block** assembly in Figure 4.1.



(a) Cylinder Block assembly consisting of 98 parts and 758K triangle primitives.



(b) Runtimes for different stages of assembly processing prior to usage in the DPA. All times were recorded using an i7 CPU @ 2.3 GHz and a GeForce GTX 675M GPU.

Figure 4.1: (a) The Cylinder Block assembly which contains numerous parts that are being reused in different locations. By detecting these duplicate components and extracting their transformation matrices, the 98 parts that the assembly contains can be represented by only 41 distinct part groups. (b) Comparing the run times of the preprocessing and loading stages of our disassembly planning system reveals the benefits of using part groups for assemblies with repeating structural patterns and large numbers of duplicate parts.

4.1.3 Iterative Mesh Shrinking

The mechanism for enabling toleranced parts in the evaluation of blocking relationships is implemented using a parallel iterative algorithm that shrinks the triangle meshes inward, i.e., it simulates the effect that erosion would have on a volumetric model of the same shape, with the constraint that the structural skeleton of the original must be preserved. Given the original mesh M of a part, the resulting reduced mesh M' is such that the number of intersections with other parts can only be less than or equal to those of M. Furthermore, it leads to the dilation of minuscule holes and openings in the resulting C-Space object for two triangle meshes, thus increasing the probability of detecting geometrically feasible motions and finding ways to separate even tightly fitting parts.

We assume that the input triangle meshes are free of self-intersections and that the basic shape of an object can be kept by ensuring that no intersections occur during shrinking. For each vertex $\boldsymbol{v}$ in a given triangle mesh, we detect the maximal distance by which it can be moved without causing intersections with other primitives of the object. This is done by determining for each incident face F of $\boldsymbol{v}$ the minimal distance between F and all other faces that are not direct neighbors of F. The resulting value defines the radius of a spherical area around $\boldsymbol{v}$ in which it can be safely placed without creating artifacts in the appearance of the model. We refer to this value as the safety radius of $\boldsymbol{v}$. However, moving a vertex $\boldsymbol{v}$ by its entire safety radius could cause other vertices to be locked in place, since their own safety radius evaluates to zero due to the movement of $\boldsymbol{v}$. This would result in a rather irregular mesh reduction with only a few vertices being moved by great distances. Furthermore, since the method is run in parallel for all vertices, we cannot guarantee that the accessed vertex locations have not been modified in the mean time. Moving two vertices by their respective safety radius could thus still cause intersections if the radius for the second vertex was calculated before the first vertex location had been updated. Therefore, we calculate the safety radii using the vertex locations of the original model and divide the resulting value by two to avoid conflicts. Using this distance to translate $\boldsymbol{v}$ in the direction of the inward normal vector yields the new position for $\boldsymbol{v}$. The normal vector is calculated for $\boldsymbol{v}$ by taking the average of the normal vectors of each incident face F of $\boldsymbol{v}$, weighted by the angle between the two edges of F emanating from $\boldsymbol{v}$. Figure 4.2 illustrates a vertex with the calculated normal vector and the safety radius defined by its incident faces, as well as the updated mesh after moving $\boldsymbol{v}$ accordingly.

The described method can only guarantee a crude approximation of an ideal reduced polygon mesh, since the safety radius is detected rather pessimistically by adopting the

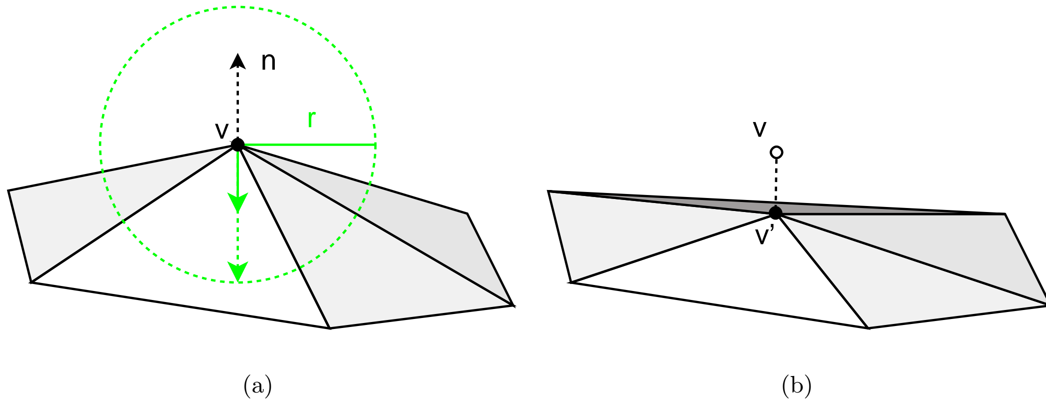


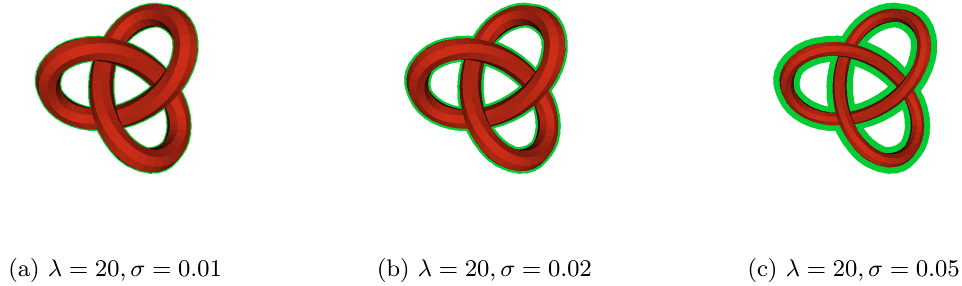
Figure 4.2: (a) A vertex $\boldsymbol{v}$ along with its incident faces. The radius r of the sphere in which $\boldsymbol{v}$ can be placed without causing intersections is defined by the minimal distance between the faces and all disconnected primitives. The calculated normal vector $\boldsymbol{n}$ is inverted and used to calculate the new position of $\boldsymbol{v}$ by translating it in the according direction by $\frac{r}{2}$. (b) The updated mesh after $\boldsymbol{v}$ has been moved accordingly, yielding the new vertex $\boldsymbol{v'}$ in the reduced mesh.

minimal distance of incident faces. Once each vertex has been moved in this fashion, there is a high probability that we can still find a safety radius greater zero. The method is thus applied repeatedly for a given number of iterations, which can be defined by the user by passing the according value as command line parameter. Furthermore, the maximum amount by which the object should be reduced can be defined to adjust the extent of tolerancing. Figures 4.3, 4.4 and 4.5 show the results of applying the shrinking method to different triangle meshes with a given number of iterations and a percental value denoting how much the object should be reduced.

4.1.3.1 Minimal Distance Calculation

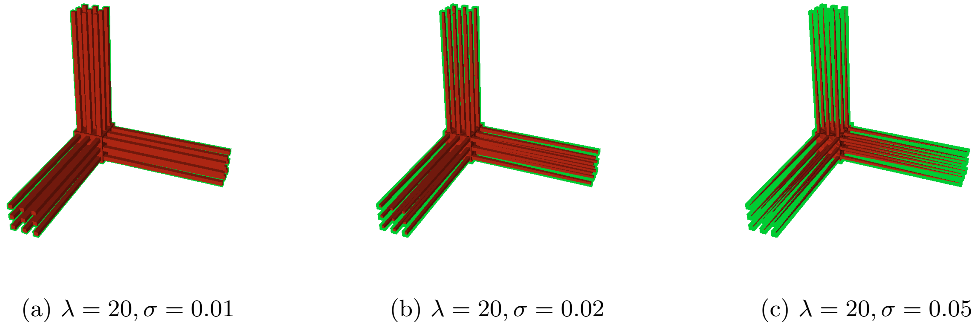
In order to find the safety radius of each vertex, for each triangle T in the mesh we determine the minimal distance between its surface and all other triangles that are not direct neighbors of T. The distance between two triangles T_A and T_B is trivially calculated by choosing the minimal value from the following results:

- Minimal distance between each vertex of T_A and the surface of T_B and vice versa
- Minimal distance between each edge of T_A and each edge of T_B
- 0, if any edge of T_A intersects the surface of T_B and vice versa



(a) $\lambda = 20, \sigma = 0.01$ (b) $\lambda = 20, \sigma = 0.02$ (c) $\lambda = 20, \sigma = 0.05$

Figure 4.3: The **Knot** model being shrunk by three different percental values σ using a fixed number of iterations λ. The space occupied by the resulting mesh (red) is a subset of the original mesh (green), thus reducing the probability for collisions or intersections.

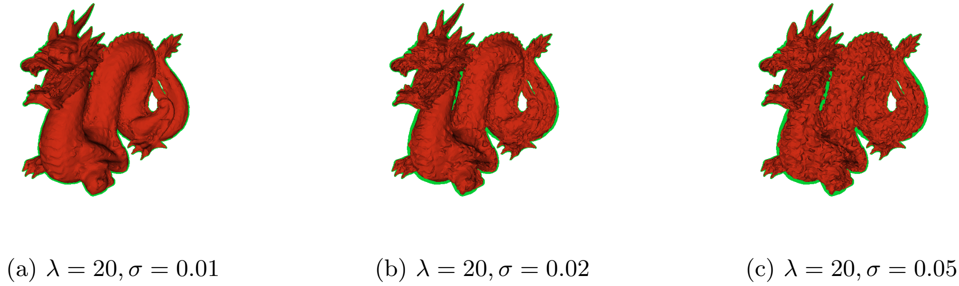


(a) $\lambda = 20, \sigma = 0.01$ (b) $\lambda = 20, \sigma = 0.02$ (c) $\lambda = 20, \sigma = 0.05$

Figure 4.4: The **Grate** model with results for different shrinking limit parameters σ. Compared to the original (green), it can be seen that the reduced mesh (red) still retains the basic shape of the original, but has a lower chance of causing collisions with other objects. Protruding elements are slimmer and may now easily fit corresponding openings.

4.1.3.2 Cell Grid Creation

The described algorithm requires us to determine the minimal distance between all possible disconnected pair-wise combinations of faces in a mesh. This information is then used during the evaluation of the safety radius for the vertices of the respective faces. This would imply calling the routine for calculating the minimal distance between two triangles $\mathcal{O}(n^2)$ times in each iteration. Even though we can exploit the CUDA functionality to determine the minimal distance for each triangle in a separate thread (thus reducing the experienced runtime to only $\mathcal{O}(n)$), the runtime still becomes an issue for highly detailed objects if we naively test all possible face pairings in this manner. Therefore, we use a regular grid space partitioning structure and only test the sets of candidate triangles

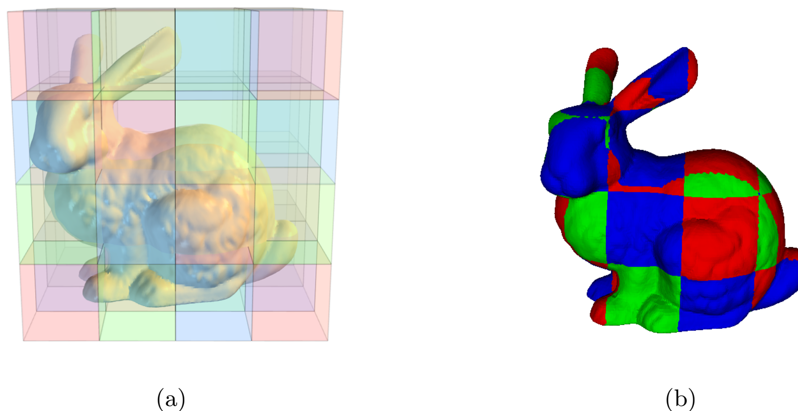


(a) $\lambda = 20, \sigma = 0.01$ (b) $\lambda = 20, \sigma = 0.02$ (c) $\lambda = 20, \sigma = 0.05$

Figure 4.5: Applying the parallel shrinking algorithm to the **Dragon** model. Since the original triangle mesh (green) features many regions of varying detail, as is common in organic shapes, the resulting meshes (red) are not reduced as effectively for a higher σ. However, although the model contains 100,000 triangles, a basic shrinking effect can be achieved with the given number of iterations λ in a matter of seconds.

that fall within a certain distance of each other. We have chosen to use a grid with dimensions 4x4x4, thus each triangle can be assigned to one or more of the 64 cells. In order to reduce the number of cells to which each triangles can be assigned, we must find an initial threshold distance d such that a triangle T is only assigned to a cell C if the distance between the AABB of T and the inside of C is less than d. Choosing a large d could result in all triangles being assigned to a high number of cells, thus defeating the purpose of this optimization. We have decided to calculate the value for d by checking the distance between each triangle T and the vertices of its neighbors that lie opposite of the connecting edges. The resulting value is commonly a sensibly small number and gives a valid threshold; assuming the object to be a non-degenerate, two-manifold mesh, the corresponding vertices would be tested in the naive approach as well since they define the corner of a triangle that is not connected to T. Figure 4.6 shows the different cells superimposed on the bunny model and the resulting assignment of triangles to cells.

This approach greatly reduces the number of calls to the minimum distance method, since each triangle in a cell can be processed by checking only the remaining triangles in the same cell. We further test the distance between the AABBs of the triangles to perform early-out rejection. All relevant cells are sequentially processed this way by launching consecutive CUDA kernel calls for each non-empty cell C, where each thread detects the minimal distance between one assigned triangle T and all other triangles in C that are not neighbors of T.



(a) (b)

Figure 4.6: (a) The cells of the 3D grid as a transparent overlay to the bunny model (b) The bunny model with each triangle primitive drawn in the color corresponding to the cell to which it was assigned.

4.1.4 Contact Information

We calculate the contact information for each part in the assembly, i.e., we detect for each part whether it is directly in contact with any other part in the assembly based on the geometry of the original, unreduced input object. The methods described in Section 4.1.3 can be reused for this purpose, since they are based on finding the minimum distance between triangles. The algorithm can be trivially extended to finding the minimum distance between two triangle meshes. One of the user-defined parameters to the preprocessing module is the threshold D which defines how far apart two parts can be and still be considered touching or in contact with each other. The parameter D can be directly used as the distance value d for creating cell grids and as an early rejection threshold, for instance by discarding all pairs of parts whose AABBs are separated by a distance greater D. There are, however, two crucial differences; first, instead of creating a cell grid for only a single object, we need to adjust the size of the cell grid such that it encloses two tested parts A and B. For each cell, two vectors need to be created to separate the triangles of A and B respectively. As before, all non-empty cells (i.e., cells in which triangles of both A and B are present) can then be tested using the CUDA kernel calls for each cell in the grid. The result of each initiated thread is then the minimal distance between one triangle of A and all triangles of B in the corresponding cell. Second, once two triangles have been found to be closer than d, the method can be terminated with a positive result. Since we only store the contact information on a per-part basis, parts can already be defined as in

contact if any two of their respective triangle primitives are found to be closer than D.

4.1.5 Polyhedral C-Space Evaluation

The fundamental constraints for detecting qualified disassembly paths are provided by the blocking relationships of the parts in the assembly. In order to enable the evaluation of global freedom, we employ C-Space object generation to automatically test geometrical feasibility for a discrete set of singular and dual translating motions. Both the generation and the evaluation of the created C-Space objects are implemented to exploit the rendering pipeline and parallel computing capabilities of modern GPGPUs for a fast extraction of all relevant blocking relationships, even when considering large assemblies.

4.1.5.1 C-Space Object Generation

We use a parallel method for the creation of C-Space obstacles which is largely based on the methods proposed by Li and McMains in [20]. Although their original implementation focuses on the creation of voxelized Minkowski sums, the early steps of this procedure involve the robust and fast creation of a polyhedral Minkowski sum which consists of primitives suitable for rendering.

The process is based on the findings by Kaul and Rossignac, which state that each primitive of the Minkowski sum of two polyhedra A and B can be generated by either translating a face of A by a vertex in B and vice versa, yielding a triangle, or by sliding an edge of A along another edge of B, thus creating a quadrilateral. Although these basic rules are simple to implement and appear to be well-suited for parallel execution using CUDA instructions as such, care must be taken of the sheer number of output primitives: for instance, if we consider two parts which both consist of only a few thousand faces, millions of output primitives will be created, which quickly inhibits the application of these rules due to the limitations of available memory on the GPU. Furthermore, although this definition yields a correct Minkowski sum when viewed from outside, the predominant amount of created primitives is redundant or invisible and does not contribute visibly to the final result. In order to avoid quickly running out of memory and generating obsolete primitives, Li et al. introduce four propositions for culling ∼99% or more of said primitives in most cases:

1. Given a face f_A of A and a vertex v_B of B, with n_A the outward facing normal of f_A, and e^i the ith incident edge pointing away from v_B. If $f_A \oplus v_B$ is a contributing triangle primitive, then $n_A \cdot e^i \leq 0, \forall e^i$.

2. Suppose e_A is an edge of A and e_B is an edge of B, f^0 and f^1 are the two incident triangles of e_A, and e^0 (or e^1) is one of the two edges of f^0 (or f^1) pointing away from e_A. Let f^2, f^3, e^2 and e^3 be defined similarly for e_B. If $e_A \oplus e_B$ is a contributing quadrilateral primitive, then either $(e_A \times e_B) \cdot e^i \leq 0, \forall e^i$ or $(e_A \times e_B) \cdot e^i \geq 0, \forall e^i, i \in \{0, 1, 2, 3\}$.

3. Suppose e_A is an edge of A and e_B is an edge of B. If either e_A or e_B is a non-convex edge, then $e_A \oplus e_B$ cannot be a contributing quadrilateral primitive.

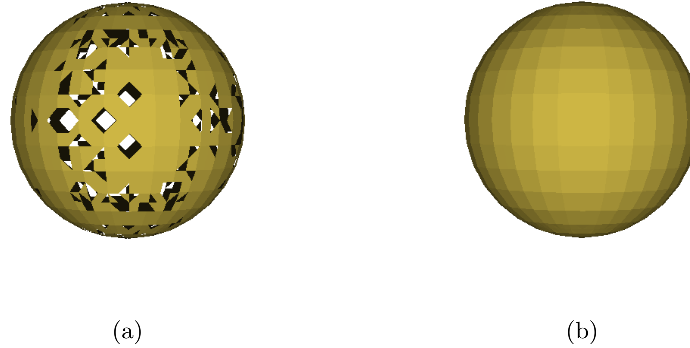
4. Suppose f_A is a face of A and v_B is a vertex of B. If v_B is a non-convex vertex, then $f_A \oplus v_B$ cannot be a contributing triangle primitive.

Since we use groups of parts, rules 3 and 4 need only be applied once for each representative in the group, as the required information can be extracted during the loading phase of the assembly from each input mesh, independent of the parts against which it will be tested during C-Space evaluation. For each part group representative, we thus store a compact set of indices which only references the convex vertices and edges in the triangle mesh. Thus, only rules 1 and 2 need to be handled in the CUDA kernels for C-Space object generation.

We do not use arbitrary precision since it is less applicable for parallel programming and also computationally expensive. However, naively implementing the geometrical tests using floating point precision can cause holes in the final result by erroneously culling contributing primitives. In order to avoid these artifacts, we use a variation of the $Orient3D$ method as proposed by Shewchuk and the according values for calculating dynamic error thresholds [27]. Figure 4.7 illustrates the improvement over the naive approach when using the $Orient3D$ method.

Although the proposed culling criteria cannot guarantee that all non-contributing primitives are discarded, the resulting number of primitives is greatly reduced. Li et al. suggest that the number of remaining primitives may be as low as 1% of the number of original candidates, which we can confirm. Table 4.1 shows the attributes of our test cases and the percentage of output primitives that were successfully culled.

The generated primitives are stored by writing to a shared resource vertex buffer object that is created via OpenGL/CUDA interoperability functionality. The created Minkowski sum is thus available for rendering immediately after the CUDA kernel has terminated. Figure 4.8 shows the test case setups for the Minkowski sum creation method and rendered images of the resulting sets of primitives.



(a) (b)

Figure 4.7: (a) Minkowski sum of two tessellated spheres, naively implemented using floating point precision for vector calculations. Due to internal imprecisions, some of the primitives are erroneously culled and create holes. (b) Using the robust *Orient3D* method and the according error margin as defined by Shewchuk, no holes are present in the resulting Minkowski sum.

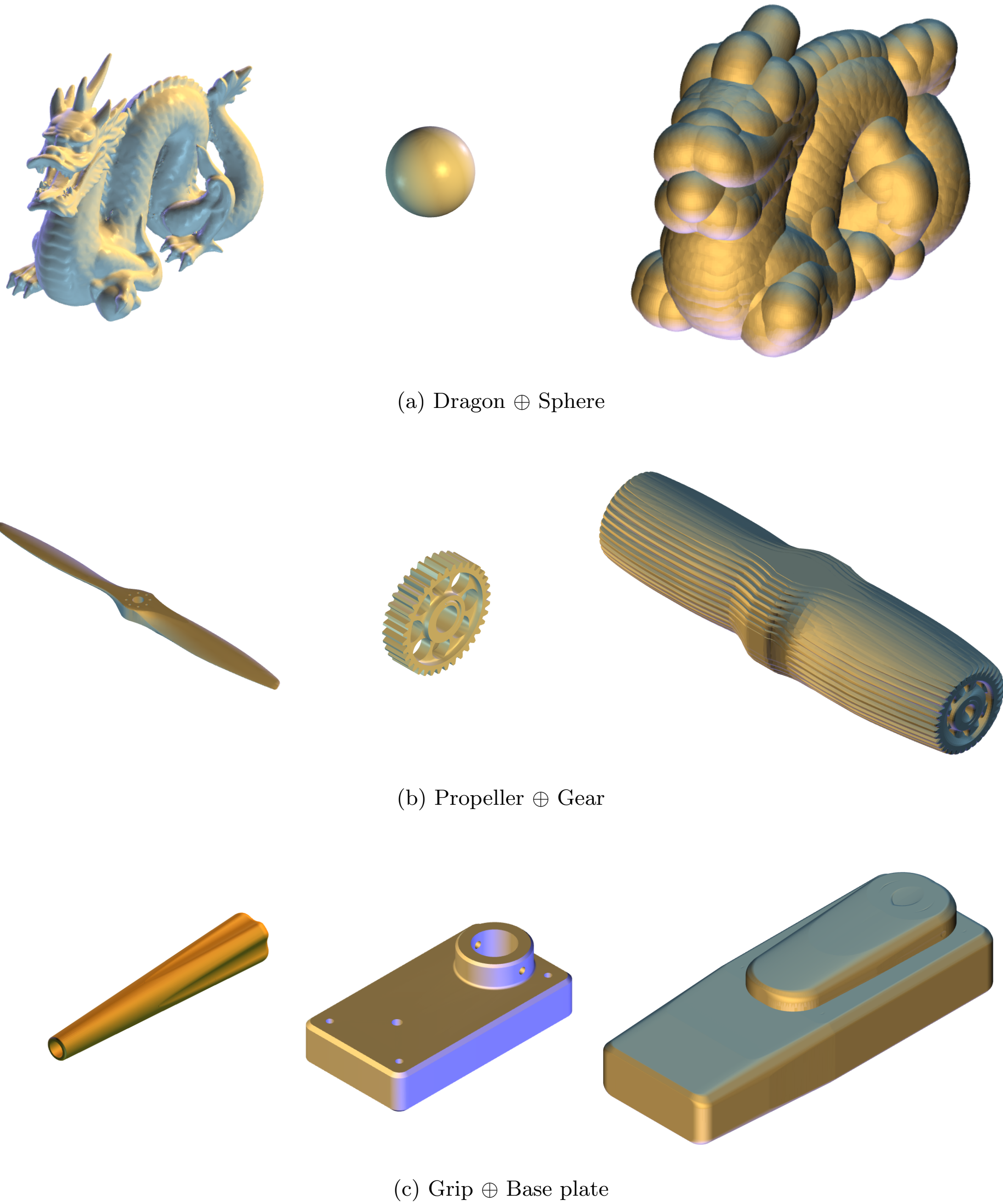
Testcase	#Vertices	#Edges	#Triangles	$Cull_T$	$Cull_Q$
Dragon ⊕ Sphere	50,000/1,922	150,000/5,760	100,000/3,840	99.8%	99.9%
Propeller ⊕ Gear	5,208/2,092	15,672/6,324	10,444/4,216	99.6%	99.7%
Grip ⊕ Base plate	62,681/4,116	179,423/12,390	116,738/8,260	99.9%	99.8%

Table 4.1: Geometric attributes of the sample test cases for Minkowski sum generation and percentage of triangle ($Cull_T$) and quadrilateral ($Cull_Q$) primitives that were culled before rendering. According to the definitions in [15], the number of potential output triangles without culling is trivially calculated using $(\#Vertices_A \times \#Triangles_B)+(\#Vertices_B \times \#Triangles_A)$, the number of potential output quadrilaterals is calculated as $\#Edges_A \times \#Edges_B$. Thus, without culling, each of the Minkowski sums resulting from our input test cases would require several GBs of memory.

4.1.5.2 C-Space Object Evaluation

Singular and dual translating motions representing a removal action ω of a part A against another part B are evaluated by using perspective and orthographic projection respectively to render the C-Space objects generated from part group representatives via $\neg A \oplus B$. In order to account for the different positions of parts in a part group, we calculate the camera position for rendering using $Origin' = Trans_A - Trans_B$.

Singular Translating Motions For singular translating motions, we use perspective projection to enable tolerance for C-Space evaluation based on the FOV α, which can be



(a) Dragon ⊕ Sphere

(b) Propeller ⊕ Gear

(c) Grip ⊕ Base plate

Figure 4.8: Showcase of our test cases used for evaluating the parallel Minkowski sum generation algorithm. Each example displays the two input objects and the resulting Minkowski sum created from rendering the generated triangle and quadrilateral primitives.

chosen by the user. The z-axis of the camera corresponds to the vector of the translation being tested. The FOV can be set by the user to allow for tolerance in the evaluation as described in Section 3.3.5.2. The parametrized call $gluPerspective(alpha, 1.0, near, far)$ is used for setting up the projection matrix in the OpenGL environment, where $alpha$ denotes the FOV in degrees, far is defined by the diameter of $\boldsymbol{P_0}$ and $near$ is chosen as $10^{-5} \times far$. The primitives of the Minkowski sum, consisting of one set of triangles and a second set of quadrilaterals, are stored in the shared VBOs and can be drawn to the render targets using the $glDrawArrays$ method with parameters $GL_TRIANGLES$ and GL_QUADS respectively. We use the standard occlusion query functionality of OpenGL to detect how many samples have actually reached the render target. By enabling depth testing and stencil testing, with the stencil functions $glStencilFunc(GL_GREATER, 1, 1)$ and operation $glStencilOp(GL_KEEP, GL_REPLACE, GL_REPLACE)$, we make sure that samples resulting from overdraw are not counted. According to OpenGL standard, a sample should be drawn only if the rasterized primitive encloses the center of the corresponding pixel. If we consider rasterization as the inversion of ray casting, it becomes obvious that each pixel in the eventual image corresponds to a tested motion that is defined by the vector from the origin to the center of the pixel on the view plane in world space. Thus, a pixel in the final image that is left blank equates to a ray through that pixel on the view plane that did not hit the target object. If the number of passed samples returned by the query is lower than the resolution of the render target, there is at least one pixel that was left blank and the associated translation direction diverges no more from the exact vector than $\sqrt{2}\alpha$. Otherwise, the blocking relationship is positive. Note that the resolution in each dimension of the render target should be set to an odd number, to ensure that one pixel always lies at the absolute center of the view plane and the exact original direction will be tested. Figure 4.9 illustrates an example where using an even number for the render target resolution causes the system to miss the original, unblocked direction.

Dual Translating Motions For dual translating motions, we use orthographic projection to simulate the process of testing multiple singular translating motions without tolerance from different positions that are equally distributed in the primary direction. The vector corresponding to the primary direction is used as the $\boldsymbol{up}$ vector for the camera setup, while the $\boldsymbol{z}$ vector is defined by the secondary direction. The resolution of the rendering target is such that on the vertical axis we find the number of intended primary stopping points, which can be defined by the user, with a horizontal resolution of only

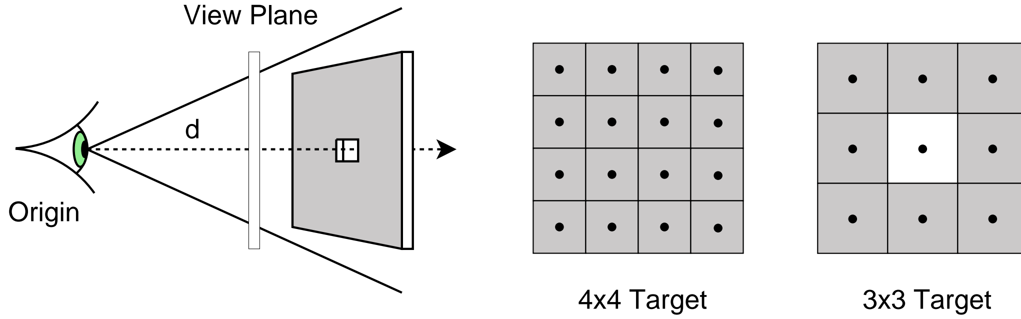


Figure 4.9: The setup for testing singular translating motions for direction d against a C-Space obstacle with holes shows the disadvantage of using even numbers for render target resolutions. Although the 4x4 target contains more samples of tested translations, it fails to consider the original translating direction itself. Using a 3x3 target, one pixel is located exactly at the center of the view plane. The vector extending from the origin through the center of that pixel is thus identical to the original direction.

one pixel. The dimensions of the view plane in world space are chosen such that the vertical extent is defined by the diameter of the bounding sphere of $\boldsymbol{P_0}$. There are no special requirements for setting the horizontal extent, but care should be taken that the dimensions are not of vastly different magnitudes to avoid imprecisions during rendering. The call to the method for generating an orthographic projection setup is made using $glOrtho(-s, s, -s, far - s, near, far)$, where s denotes half the distance between two consecutive primary stopping points in world space. Thus, each pixel in the render target equates to one primary stopping point and encodes the feasibility of moving from this location in the secondary direction. Rendering with the new camera setup is performed the same way as with singular directions, by binding the shared VBOs. The filled render target is then evaluated and the contents of its depth buffer are used to extract the secondary blockers for each primary stopping point. The concatenation of the resulting groups of blocking relationships yields the dual chain, which lists the secondary blockers for each primary position. The secondary blockers of a dual translating motion D' at the primary stopping position with index 0 are the primary blockers for any dual translating motion D whose primary direction corresponds to the secondary direction of D'. Thus, the primary blockers of any D are conveniently computed during the evaluation of D'.

4.1.5.3 Separator Occlusion and Bounding Box C-Space Evaluation

Separator occlusion and bounding box C-Space evaluation are executed prior to the polyhedral C-Space evaluation. Their main purpose the reduction of preprocessing runtime by culling all part pairings that do not require detailed C-Space evaluation. We first create for each part A the C-Space objects of its own AABB and the AABBs of all other parts. The resulting vertices of one such C-Space obstacle form again an AABB whose geometry can be defined using 8 vertices and 6 quads. We create the set of corresponding primitives that can then be directly used for rendering. The identifier of the part whose AABB was used for creating the according bounding box C-Space object is stored in the red vertex color value. Thus, for each part in an assembly of n parts, $n \times 8$ vertices with color information and $n \times 24$ indices are stored.

For each removal action ω_i that is being evaluated, we create a separate 3D texture render target τ_i. To enable C-Space object occlusion of separators for a part A from part group $\mathcal{A}$, we generate the polyhedral C-Space objects of the group representative of $\mathcal{A}$ with all separators and render it to the depth buffer of each τ_i with a camera setup corresponding to ω_i. We then bind the colored primitives that were generated for the bounding box C-Space objects of A and draw them in a way similar to that described in Section 4.1.5.2, with some optimizations; to exploit the parallel instructions of the graphics pipeline, we render all bounding box C-Space objects to a 3D texture τ_i at once and evaluate the color information to extract the complete set of potential blockers for ω_i. The process is illustrated in Figure 4.10 for an evaluated exemplary singular translating motion.

Back face culling is enabled to avoid overdraw of the boxes. We use a GLSL shader to select a 3D texture layer and convert the red color value such that for each part identifier a unique combination of T and γ exists, where T is a 2D texture target of τ_i and γ represents an RGBA color with one color value set to a power of 2 smaller 2^{24}. The final image is composed using the additive OpenGL blending function $glBlendFunc(GL_ONE, GL_ONE)$. Since the separators have already been drawn to the depth buffer, updating the depth values can be disabled, while depth testing remains enabled. As a result, the bits in the mantissa of the RGBA float values in the final image represent the identifiers of all bounding box C-Space objects that are closer to the origin than any separator. Thus, the combined pixels at position (x, y) of the different layers in τ_i encode the parts that may inhibit the corresponding translating motion. This set of potential blockers is extracted using a CUDA kernel which performs a parallel scan of the contents in the 3D render

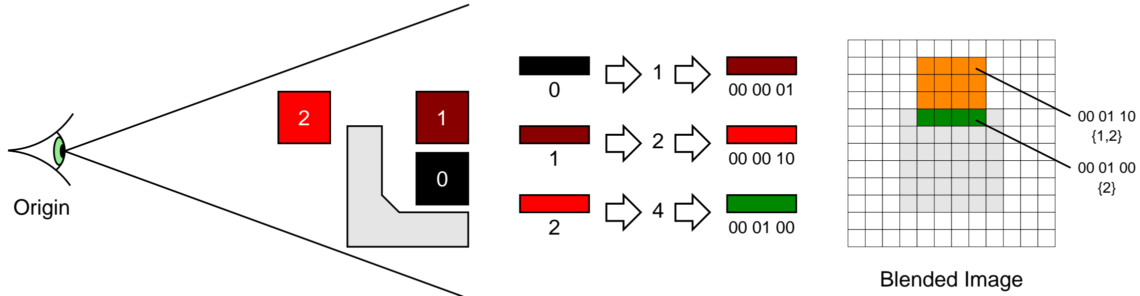


Figure 4.10: Illustration of the bounding box C-Space evaluation and separator occlusion for a straight-line removal action ω. For the sake of simplicity, this example uses a single 2D render target and a 6-bit RGB color palette with 2 bits for each color value. The gray polygon represents a polyhedral separator C-Space object that is rendered to the depth buffer of the according render target. The vertices of the bounding box C-Space objects encode the identifier of the part against which the test is performed in their red color value. Each incoming red value is transformed by the GLSL shader to a unique color with exactly one bit set, such that during additive blending no information is lost. The list of potential blockers that are not occluded by separators can then be extracted from the final image using a parallel scan.

target textures and returns the list of all identifiers that were detected. With contemporary hardware, the number of simultaneously active render targets is usually limited to 8. Since we use the mantissa bits of the RGBA float values for encoding part identifiers, the maximum number of bounding box C-Space objects that can be evaluated in one rendering pass thus becomes $8 \times 4 \times 24 = 768$.

4.1.5.4 Detecting Unities

The term unity in our system describes a pair of parts (A, B) for which no geometrically feasible removal action can be found to separate them from each other. This may be due to the system only evaluating a discrete set of translating motions. However, it is also possible that the parts were designed such that their conjunction, once established, is difficult or simply infeasible to undo. Thus, the existence of a unity does not necessarily indicate a failure of the preprocessing module. Information about these unities can be used during partitioning of the assembly to increase the probability of finding a feasible disassembly path. Thus, the system identifies and stores the set of inseparable parts by listing all pairs A, B for which no ω exists such that $B_r(A, B, \omega)$ is negative.

4.1.6 Storing Static Disassembly Information

The output generated by the preprocessing module is stored in a set of plain text and binary encoded files that can be easily transferred to be used by the DPA on another system or workstation. Each assembly file set defines a main directory with files containing structural information and two sub-directories where triangle meshes and blocking relationships are stored. The moving parts and separators are listed in two distinct files in the main directory. Basic information about the scene such as its dimensions, total number of parts and the set of unity pairs is stored in a designated scene file. Furthermore, contact information is stored in an upper triangular matrix in .csv format. The reduced triangle meshes are stored as .obj compliant files, while the original geometry data that is used for displaying the assembly is kept in its original format (either .wrl or .iv). All blocking relationships are sorted by their respective removal action and stored accordingly in a separate file for each moving part A. Blocking relationships of a removal action ω are encoded as a series of bits, where a 1 at position i represents a positive blocking relationship $B_r(A, \boldsymbol{P_{i+1}}, \omega)$. For dual translating motions, primary and secondary blockers of each segment in the dual chain are stored separately.

4.2 Disassembly Planning Application

The DPA allows the user to interact with the assembly and apply or modify the information that was obtained in the preprocessing phase to find or verify disassembly paths for a given disassembly problem. The application is intentionally light-weight regarding hardware requirements to allow running it on a wider range of systems. The user interface is based on the OSG framework and the AntTweakBar library. A simple interface is provided for defining required partitions and parameters for their removal using keyboard and cursor input. Potential solutions to disassembly problems are calculated almost instantly on contemporary systems, even when considering complex assemblies. The application employs several techniques to convey calculated disassembly procedures to the user and visualizes the individual instructions in a disassembly sequence with a strong focus on clarity and coherence.

4.2.1 Loading and Initial Partitioning

After starting up the DPA and selecting an assembly file set, the entire static assembly information is loaded to main memory. Each part group in the assembly is loaded and pro-

cessed separately. The representative triangle mesh is read and converted to a displayable OSG node for rendering. The reduced mesh in .obj format is loaded as well, since it may be required for the on-line generation of blocking relationships for user-specified removal actions. Once the part representative has been acquired, all individual instance of the base component are created by duplicating the original node and transforming it using the corresponding transformation matrix. For each part, blocking relationships for all evaluated removal actions are read and stored as the bits of designated integer groups. In addition to the integer groups representing the dual chain for dual translating motions, a set of value pairs is created for where each pair contains the ID of a primary blocker and according distance in the primary direction.

The initial partitioning is naively performed by creating one partition for each loaded part and adding it as a child to the top-most parent partition $\boldsymbol{P_0}$. We then proceed to parsing the file containing the list of unities and creating the according part clusters. For each cluster, the built-in functionality for manually handling partitions in the DPA is employed to fuse the individual parts to form larger partitions.

4.2.2 The Planning View

The planning view is the default screen that is displayed when the application is launched. Once the static assembly information has been loaded to memory, the assembly is displayed as a collection of selectable nodes, representing the partitions that correspond to the immediate children of $\boldsymbol{P_0}$. The standard OSG setup for viewing a scene allows the user to intuitively navigate through an assembly by controlling the camera position and view setup via keyboard and cursor. At each point in time, the system registers a designated focus partition, whose immediate children are being displayed, allowing the user to interact with them. This enables viewing and specifically editing the contents of selected partitions in detail. The focus partition can be changed by either directly selecting a partition and explicitly declaring it as the new focus or by switching to the parent of the current focus partition. Partitions can be selected by clicking on their 3D representation in the planning view with the *Ctrl* key pressed. The always-on-top status window is updated upon selection of a partition to display its properties. We use a menu bar based on the AntTweakBar library to allow easy manipulation of customizable parameters, such as designated removal action or setting the **required** flag for a partition. Partitioning operations can be called via the menu bar as well; while fusing can only be performed when two or more partitions have been selected, collapsing and flattening can only be applied if

exactly one partition is selected. Furthermore, the disassembly path computation can be launched from the menu bar to calculate a new disassembly path based on modified user constraints.

4.2.3 Disassembly Path Computation

Disassembly path computation is implemented using an iterative algorithm that checks the removability of partitions at each point in time based on the blocking relationships that were calculated for the preprocessed removal actions or extended motions. By evaluating the dependencies of required partitions, we discard redundant instructions and provide a lean solution to the posed disassembly problem, if one exists. If the system fails to find a geometrically feasible disassembly path, feedback is provided for the user to determine the cause of failure and possibly correct it. However, even if a geometrically feasible disassembly path can be found, it may not suit all preferences or requirements of the user (see Section 3.1.5). Since the parameters of partitions may require numerous modifications until a wholesome solution is found, the disassembly path computation needs to provide feedback with little latency to convey the ramifications of changes made by the user. Our implementation of the implied functionality targets instant computation and verification of disassembly paths for large assemblies.

4.2.3.1 Blocking Relationships Storage and Access

Monitoring and modifying blocking relationships pose the main tasks for disassembly path computation. In order to ensure the real-time constraint of the feedback for the procedure, optimizing these operations is essential. The blocking relationships that are used for detecting partition removability, as mentioned before, are not stored per partition, but per part, since the eventual partitioning of the assembly is unknown during the preprocessing stage. We initially use naive partitioning, thus each part has a corresponding base partition of size one in our system and any larger partition that may be automatically or manually created is a product of these base partitions. Since the list of nested children can easily be extracted from any larger partition, it suffices to monitor blocking relationships between these base partitions only. In order to do so, the blocking relationships in the static assembly information can be directly transferred; each base partition for an assembly of n parts is created with the same identifier i where $1 \leq i \leq n$ that was used for the corresponding part during the preprocessing stage.

We store the blocking relationships for a removal action ω of a partition in an assembly

of n parts as a sequence of n bits in an array of integers, where the bit at position $i \leq n$ corresponds to the blocking relationship with base partition $\boldsymbol{P_i}$. This allows us to update the blocking relationships of ω with any part in $\mathcal{O}(1)$. The corresponding memory for storing the blocking relationships for partitions must only be allocated once when they are created, since the number of parts in the assembly cannot change. In contrast, if we were to dynamically store blocking relationships with all existing partitions, the system would need to update all removal actions if a partition is manually created or destroyed.

4.2.3.2 Iterative Partition Removal

The algorithm for iterative partition removal is based on a general peeling approach for a given partitioning hierarchy. The decomposition of the partitions in an assembly is considered as a top-down process in our system, e.g. if $\boldsymbol{P_C}$ is a child a of $\boldsymbol{P_P}$, $\boldsymbol{P_P}$ needs to be removed before we can consider removal of $\boldsymbol{P_C}$. The benefits of this approach are two-fold; first, a partition can be considered free if it is not blocked by any other active partition with the same parent. Thus, in order to reduce computational overhead, all partitions are updated before iterative partition removal such that positive blocking relationships of removal actions of a partition $\boldsymbol{P_x}$ are discarded if the corresponding parts are not present in the partitions represented by the siblings of $\boldsymbol{P_x}$. Second, we can easily restrict the set of partitions to test for removal at each point in time by maintaining a set of observed partitions. The set may only be expanded by the children of parent partitions that are being removed. We start by monitoring only the immediate children of $\boldsymbol{P_0}$ in the set of observed partitions. If a partition $\boldsymbol{P_x}$ from the observed set is removed from the assembly, its immediate children become candidates for being added to the set while $\boldsymbol{P_x}$ itself is erased.

At each point in time t, we detect all partitions in the observed set for which an allowed and unblocked removal action exists. The set of partitions that are removable at time t define the **actors** of the peeling phase Φ_t. By removing the actors of Φ_t from the assembly, other partitions that were blocked by one or more of them may become removable in the following phase Φ_{t+1}. The procedure is repeated until either all required partitions have been successfully removed or no more actors can be found. The implementation of the peeling method is outlined in Algorithm 4.1.

The removal of free partitions in a phase Φ_t is performed in two stages, according to the implementation given in Algorithm 4.2. First, all partitions are detected for which a geometrically feasible removal action exists. This can either be the designated removal

Algorithm 4.1 Peeling algorithm for iteratively removing free partitions

```
1:  procedure Peel(partitions, required)
2:      observed ← ∅
3:      for all P ∈ children(partitions[0]) do          ▷ Start with children of P_0
4:          insert(observed, P)
5:      end for
6:
7:      t ← 0
8:      phases ← [ ]
9:      observed_old ← ∅
10:     while observed ≠ observed_old do
11:         if required ≠ ∅ then
12:             observed_old ← observed
13:             observed ← removeAllFree(required, observed)
14:             actors ← observed_old \ observed
15:             required ← required \ actors
16:             phases[t] ← actors
17:             t ← t + 1
18:         else
19:             break
20:         end if
21:     end while
22: end procedure
```

action of a partition or one of the removal actions that were evaluated during the preprocessing phase. If more than one unblocked removal action exists, the system selects the one with the highest score based on Algorithm 4.3, preferring singular translating motions over dual translating motions. The score of each removal action is inverse proportional to the distance which the partition has to cover to escape the AABB of the assembly using the corresponding motion.

Second, all remaining active partitions have the blocking relationships of their removal actions updated; all blocking relationships of active partitions involving an actor of Φ_t are nullified. Furthermore, the set of observed partitions is updated accordingly. The children of removed partitions become candidates for removal in the ensuing phase. Since removed partitions that do not contain any required partitions need not be decomposed any further, they will not be added to the set of observed partitions.

Algorithm 4.2 Detection of removable partitions

```
 1: procedure removeAllFree(required, observed)
 2:     removed ← ∅
 3:     for all P ∈ observed do
 4:         if hasRemovalAction(P) then        ▷ Partition has a designated removal action
 5:             R ← getRemovalAction(P)
 6:             if isFree(R) then                  ▷ No positive blocking relationships exist
 7:                 insert(removed, P)
 8:             end if
 9:         else
10:             if R ∈ isFree(P) then               ▷ Has unblocked tested removal actions
11:                 R ← chooseRemovalAction(P)
12:                 setRemovalAction(P, R)
13:                 insert(removed, P)
14:             end if
15:         end if
16:     end for
17:
18:     for all P ∈ removed do
19:         erase(observed, P)
20:         for all C ∈ children(P) do
21:             if containsRequired(C) then         ▷ Recursively check all nested children
22:                 insert(observed, C)                ▷ Add if contains required partitions
23:             end if
24:         end for
25:         for all S ∈ siblings(P) do                ▷ Check partitions with same parent
26:             if S ∉ removed then
27:                 for all V ∈ getParts(P) do          ▷ Part-based blocking relationships
28:                     if hasRemovalAction(S) then
29:                         R ← getRemovalAction(S)
30:                         removeBlocker(R, V)
31:                     else
32:                         for all R ∈ testedRemovalActions(S) do
33:                             removeBlocker(R, V)
34:                         end for
35:                     end if
36:                 end for
37:             end if
38:         end for
39:     end for
40:
41:     return observed                   ▷ Return set of observed partitions for next phase
42: end procedure
```

Algorithm 4.3 Method for choosing suitable removal action

```
1:  procedure chooseRemovalAction(partition)
2:      best_score ← 0
3:      best_action ← ∅
4:      for all S ∈ getSingularTranslatingMotions(partition) do
5:          if isFree(S) then
6:              score ← getScore(S)          ▷ Distance moved inside assembly AABB
7:              if score > best_score then
8:                  best_score ← score
9:                  best_action ← S
10:             end if
11:         end if
12:     end for
13:
14:     if best_action ≠ ∅ then
15:         return best_action
16:     else
17:         for all D ∈ getDualTranslatingMotions(partition) do
18:             if isFree(D) then
19:                 score ← getScore(D)          ▷ Distance moved inside assembly AABB
20:                 if score > best_score then
21:                     best_score ← score
22:                     best_action ← S
23:                 end if
24:             end if
25:         end for
26:
27:         return best_action
28:     end if
29: end procedure
```

4.2.3.3 Dependency Hierarchy

Each peeling phase Φ that was generated by the peeling algorithm is defined by a set of actors, which defines the partitions that are removed during Φ. However, many of these partition removals may not actually contribute to the exposure of required parts. Removing them would thus be unnecessary and therefore imply redundant disassembly actions. In order to prune these steps from the eventual sequence, we determine the direct dependencies of each partition and recursively add all partitions that have direct or indirect influence on the removal action of the required partitions. For each partition that was removed during a peeling phase, we calculate two sets which contain required and dependency partitions respectively. The set of required partitions simply lists all partitions

that need to be removed before the removal action of a partition $\boldsymbol{P_x}$ becomes geometrically feasible. Dependency partitions of $\boldsymbol{P_x}$ are defined by the set of partitions that block the removal action of $\boldsymbol{P_x}$, but have not been removed prior to the peeling phase containing $\boldsymbol{P_x}$. Thus, the dependency partitions of $\boldsymbol{P_x}$ describe a subset of its required partitions. From these sets, we can extract a collection of nodes that possess hierarchical relationships. Each node is identified via an ID that corresponds to the ID of the partition from which it is created. The parents of a node N are defined by the nodes corresponding to the IDs in the dependency set of the partition represented by N. The children of N are inversely all those nodes that have N as a parent. Due to the chronological arrangement of the different peeling phases, it is not possible to create any circular relationships, which simplifies the implementation of dependency hierarchy creation. Algorithm 4.4 demonstrates the method used for calculating the required and dependency sets to create all relevant hierarchy nodes with according parent-child relationships.

4.2.4 Explosion Diagram Preview

Explosion diagrams are often used to convey the structure of complex objects. The basic principle for creating explosion diagrams of a product dictates that each individual part is translated relative to its initial position, in a suitable direction to allow for its inspection. Ideally, the resulting constellation does not contain any overlapping or intersecting objects. This principle can be adapted for disassembly visualization, by substituting the explosion directions with the directions corresponding to the detected removal actions of the partitions. By recursively resolving part intersections, a basic explosion diagram can be generated, which serves as an instant preview to a potentially lengthy animation sequence. It provides quick, single-frame feedback to the user and can thus speed up the disassembly path editing process, since fundamental flaws or unwanted effects of the current settings are easily detected.

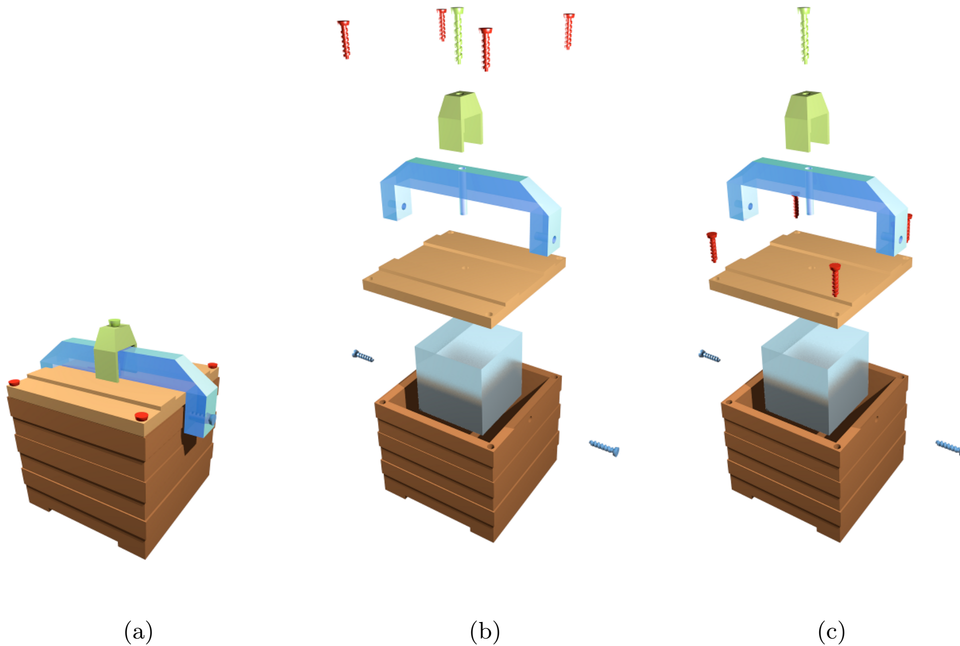
4.2.4.1 Partition Placement

The parent-child relationships already stored in the hierarchy structure can be used to create a bottom-up ordering for generating explosion diagrams. This way, we obtain a modified set of phases such that each partition is not removed as early as possible, but rather at the latest possible point without stalling the disassembly procedure (i.e. increasing the number of necessary removal phases). An array of phases with modified sets of actors is generated accordingly to replace the actors of the originally detected peeling

Algorithm 4.4 Dependency hierarchy creation

```
1:  procedure CreateHierarchy(phases, required)
2:      for all t from 0 to (size(phases) − 1) do
3:          actors ← phases[t]
4:          for all P ∈ actors do
5:              R ← getRemovalAction(P)
6:              blockers ← getBlockers(R)
7:              req_set ← blockers
8:              for all B ∈ blockers do
9:                  req_set ← req_set ∪ getReqSet(B)
10:             end for
11:             dep_set ← blockers \ req_set
12:             setReqSet(P, req_set)               ▷ Store required partitions set
13:             setDepSet(P, dep_set)               ▷ Store dependency partitions set
14:         end for
15:     end for
16:
17:     H ← ∅                                       ▷ Initialize empty hierarchy
18:     for all P ∈ required do
19:         recBuild(H, P)
20:     end for
21:
22:     return H
23: end procedure
24:
25: procedure recBuild(hierarchy, partition)
26:     if containsNode(hierarchy, partition) then  ▷ Check for node with same ID
27:         return getNode(hierarchy, partition)
28:     else
29:         node ← makeHierarchyNode(partition)
30:         dep_set ← getDepSet(partition)
31:         for all P ∈ dep_set do
32:             parent ← recBuild(hierarchy, P)
33:             addChild(parent, node)              ▷ Establish hierarchical relations
34:             addParent(node, parent)
35:         end for
36:         addNode(hierarchy, node)
37:         return node
38:     end if
39: end procedure
```

phases. Based on these new actor sets, the position of each partition $\boldsymbol{P_A}$ is calculated by iteratively detecting AABB intersections with partitions of earlier phases and moving $\boldsymbol{P_A}$ according to its removal action until no more intersections exist. As seen in Figure 4.11, using the bottom-up approach can result in a compact explosion diagram, where dependent partitions are closer to each other in comparison to using the original peeling phases.



(a) (b) (c)

Figure 4.11: (a) A modified version of the **Cargo** assembly depicted in Figure 2.7, with additional locking obstacles keeping the lid in place. (b) Explosion diagram created using the top-down hierarchy. Note that the red bolts fixating the lid have been moved by a considerable distance since the peeling phases are directly used as hierarchy layers. (c) Using the bottom-up approach, each element in the hierarchy is placed on the lowest possible layer. Therefore, the red bolts are removed just before the lid, which leads to them being located closer to it. This makes it easier for the user to understand the reason for removing the red bolts in the first place.

4.2.4.2 Static Motion Blur

We use the accumulation buffer to generate the static blur effect over a given number of frames N using multi-pass rendering. In each frame, all moved partitions are rendered in a position that is calculated as follows: for partition A with its initial and final exploded

location $\boldsymbol{L_I}$ and $\boldsymbol{L_F}$ respectively, the placement of A in frame i is calculated as $\boldsymbol{L_i} = \frac{i}{N} \times \boldsymbol{L_F} + (1 - \frac{i}{N}) \times \boldsymbol{L_I}$. The current content of the accumulation buffer is then updated by rendering and combining the contents of the render buffer with the previous data. We use $glAccum(GL_MULT, PERSISTENCE)$ to reduce the influence of the previous frames, where $PERSISTENCE$ is a parameter that can be set by the user to any value between 0 and 1. New color information is added via $glAccum(GL_ACCUM, 1{-}PERSISTENCE)$. The final image for display can be directly copied from the accumulation buffer to the main frame buffer using $glAccum(GL_RETURN, 1.0)$ once all N frames have been successfully processed.

4.2.5 The Animation View

The animation view is used for displaying disassembly phases and step-by-step partition removal instructions. Each animation phase Λ that was generated from the dependency hierarchy can be previewed in a static scene. The user can easily navigate between phases by selecting a corresponding snapshot from the menu at the bottom of the screen. This way, he may either step from preview to preview or choose to animate a selected phase. Camera positioning is achieved semi-automatically to ensure that initially, all visual events of Λ are visible, while giving the user the possibility to adjust rendering styles and viewing setups at will.

4.2.5.1 Phase Snapshots

For each animation phase Λ, we generate a snapshot that can be used for evaluating each phase and navigating between different stages of the disassembly. The images are rendered to textures, which are in turn used on quadrilateral geometry nodes that can be directly used for interaction in the OSG environment. We identify the actors and the set of partitions that remain after Λ has finished. We create a new, empty scene and clone the already loaded scene graph nodes of the corresponding partitions. The current rendering style for the planning view is used to draw the remainders of Λ. In order to highlight the actors of Λ, we modify their current appearance by using a bright-red opaque material. We then bind the texture rendering target for the geometry node corresponding to Λ and draw one frame of the new scene. The nodes containing the textured quads are then displayed at the bottom of the screen and can be used to preview the individual phases or select them for display as a step-by-step animation.

4.2.5.2 Animation Paths

We use the default implementation of the OSG Animation Path class to change the position of partitions during their animated removal. The animation path is built by converting the removal actions of each part to an array of structs representing singular motions. Each struct contains three variables, namely the type of the singular action, a vector and a floating point value. The animation interface was designed to enable support of translations and also rotations in future versions. The type variable thus distinguishes between $TRANSLATION$ and $ROTATION$. The vector defines the direction for translations and the rotation axis otherwise. The floating point value stores either the translating distance or the rotating angle. Any such struct can be easily converted and added as a segment to the animation path. Dual translating motions in our system thus generate an animation path with one intermediate stopping position, while extended motions may define an arbitrary number of stops. During step-by-step animation, each path is preset to loop infinitely until the user either quits the animation or confirms the apprehension of the displayed action using the *Right-arrow*/*Space* key.

4.2.5.3 Animation Phase Preview

Animation phases can be previewed in static 3D scenes, where each actor is placed in its final animated position, with guidelines connecting each offset partition to its initial location. In contrast to explosion diagram generation, no intersection tests are performed. Extrinsic camera settings are adjusted to ensure that each animation path of the phase is fully visible with the current setup. The user may step from preview to preview using the *Space* key. Alternatively, if a preview reveals that the details of the phase are of interest, he may choose to initiate the execution of the step-by-step animation by pressing the *Right-arrow* key. Once all paths of a phase have been animated and confirmed, the system automatically proceeds to the next preview.

4.2.5.4 Visual Cues

In order to clearly convey each instruction of the assembly sequence, we make sure that the animation path of removed partitions can be easily followed. At the beginning of each animation phase, the camera is placed such that all actors and remainders are at all times contained within the viewing frustum during animation. Thus, a partition that is being removed cannot vanish due to it leaving the screen, except for when the has user intentionally altered the camera position. However, since the zoom of the camera

is adjusted to cover the entire space occupied by static and animated partitions, smaller components such as screws or bolts may become difficult to locate. In order to improve their visibility, we provide mechanisms for highlighting actions that may be easily missed otherwise.

Visibility Testing Visibility testing is performed in every frame of the animation to activate visual cues dynamically if the camera position changes or the visibility of a partition varies over the course of animation. The test and corresponding reaction is based on two criteria. The first criterion is the size of the animated node when projected onto the screen via the main camera for the animation view. If the projection of its bounding sphere fails to consume at least $\frac{1}{10}$ of the screen, the first criterion for visibility is not fulfilled. This will cause the system to activate both the highlighting billboard and the close-up camera view. If, however, the node passes this first test, we check its visibility using occlusion testing, which is implemented using the standard OSG Occlusion Query Node. We set the threshold for samples passed to 0, thus the test fails only if the partition is completely occluded by other partitions. In this case, only the billboard is activated to convey to the user the current position of the occluded node.

Highlighting Billboard The billboard is drawn as a red circle with a transparent center so that the animated node remains visible. We use a quad geometry node that is scaled such that the circle encloses at least the bounding sphere of the corresponding partition when projected onto the screen. If the radius of the projected bounding sphere is less than $\frac{1}{10}$ of the larger view port dimension for the animation view, the billboard is rescaled to this value instead to ensure that the billboard itself occupies at least $\frac{1}{10}$ of the screen and cannot be missed. The billboard receives the same animation path as the node representing the highlighted partition, thus their movement is synchronized. A secondary camera is used for rendering the billboard on top of the contents in the color buffer post-frame with depth testing disabled. Thus, the billboard is always completely visible, even when the animated node itself vanishes due to its small size or occlusion.

Close-Up Camera We use a picture-in-picture approach to show the details of an animated node in close-up as part of the animation view. The upper left corner displays a separate a view port, that is activated if the visibility test fails due to the projected size of the node. The FOV for the close-up camera is set to 40°. The extrinsic settings are defined such that the viewing axis equates to that of the main camera, but always

focuses on the initial position of the node. The angle for viewing with the close-up camera can thus be changed by modifying the main camera of the animation view. The distance of the close-up camera from the partition is statically defined as $10 \times r$, where r is the radius of the bounding sphere of the node. This ensures, that the motion direction can be clearly discerned and the partition in its initial position is completely contained within the viewing frustum.

4.2.6 Rendering Styles

We use three different rendering styles for displaying the assembly; the scene graph nodes can either be displayed with variable transparency or as opaque objects. Based on the disassembly path computation, we also enable the user to set all partitions transparent except for those that are are removed during the disassembly sequence (*involved* rendering style), thus making it easy to distinguish partitions that participate in the procedure. We use the methods provided by OSG to customize the alpha value of node materials accordingly. The selected rendering style is global, i.e., it is used for the planning view as well as for all other visuals. Changing the rendering style of the DPA thus influences the generation of preview images and appearance of step-by-step animations. However, the preview images are only recalculated by request. It is thus possible to mix different rendering styles, e.g. by having the preview images display transparent objects while the animation uses opaque rendering style. The user can easily switch between the different rendering styles using the corresponding keyboard hot keys.

Chapter 5

Examples and Discussion

In this chapter, we present numeric and graphic results for disassembly planning procedures in our system, using a variety of input data sets and specified disassembly problems. We further discuss strengths and weaknesses of our system and the application of its individual features based on the suggested disassembly sequences and the resulting visuals. All images of the assembled products and illustrations of their disassembly procedures in this chapter were created by directly capturing the screen from the active DPA.

5.1 Assembly Data Sets

We consider 6 exemplary assembly data sets with varying numbers of parts and geometrical primitives. All data sets were created from openly available CAD models by exporting the input files to Open Inventor format using the SAP Visual Enterprise Author application. The list of test cases, along with the number of triangles, part count and input file size can be found in Table 5.1 as well as the number of part groups and base component meshes detected during the preprocessing stage. The largest assembly examined by us contains more than 500 parts and over 500K triangle primitives.

5.1.1 Preprocessing Runtimes

In previous systems, the runtime required for the analysis of assemblies was shown to grow rapidly with the number of contained parts, thus defining the limiting factor for considering complex assemblies [19, 30, 32]. Specifically, contact information and blocking relationship evaluation are usually the most expensive operations. Our system handles these steps in the preprocessing stage, which needs to be executed only once to create

Assembly	#Parts	#Triangles	File Size (MB)	#Groups	#Meshes
Press	23	23,646	2.4	18	17
Pneumatic Engine	121	100,380	9.6	84	26
Drill	141	292,360	27.4	119	94
Radial Engine	239	265,080	26.9	218	52
Mecanum Wheel	254	135,848	13.6	246	17
Aviation Engine	512	537,385	53.6	399	94

Table 5.1: The list of assemblies examined in this chapter. Each entry lists the number of contained parts, triangle primitives and the original input size in the Open Inventor file format. The columns on the right list the number of part groups and base component meshes that were detected by the preprocessing module of our system.

the static information data. Furthermore, the employed algorithms have been optimized to reduce the runtime through parallelization, part grouping and blocking relationship filtering. Consequently, our system can evaluate even complex assemblies such as the Aviation Engine model in a matter of minutes. Figure 5.1 lists the time required for the individual stages of preprocessing for each example assembly. All times were recorded using an i7 CPU and a GeForce GTX 680 GPU.

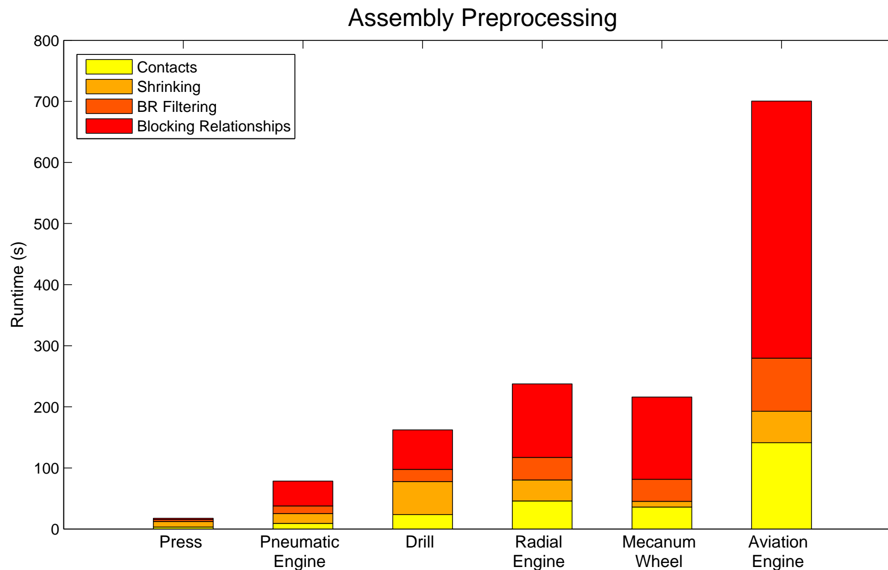


Figure 5.1: The runtimes for the preprocessing stages in our system for the 6 examined assemblies. Although we employ several methods for optimizing the corresponding steps, evaluation of the blocking relationships clearly consumes the majority of the required runtime.

5.1.2 Required Storage

Since we use part grouping to reduce the amount of meshes we need to store to the number of detected base components, the disk space required for storing the geometry information of a processed assembly is usually less than the size of the original input file. The storage required for blocking relationships is far greater, and also grows in a non-linear fashion. We consider for each part 6 singular translating motions corresponding to its principal axes and 48 dual translating motions which result from combining the Euclidean unit vectors and the principal axes to form all possible right-angular combinations. For each dual translating motion, we need to store a blocking relationship set for each of its primary stopping points. For our evaluation, the number of primary positions has been set to the default value of 255. The maximum number of evaluated motions and corresponding blocking relationship sets stored for each part in the assembly is thus calculated as $48 \times 255 + 6 = 12246$. For the Aviation Engine, almost 500MB of disk space are required to store the static assembly information. Although we can easily afford to store this amount of data on contemporary hardware, the time required for loading and writing these data sets may eventually become an issue if the number of parts is further increased beyond 1000 parts or more.

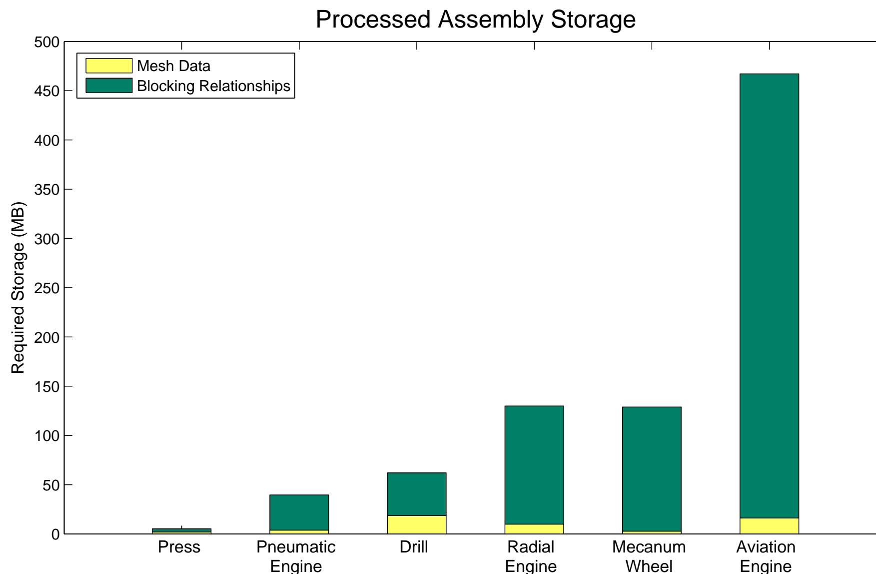


Figure 5.2: Disk space required for storing the static assembly information extracted in the preprocessing module for each input assembly. Considering that the Pneumatic Engine and the Aviation Engine contain 121 and 512 parts respectively, the growth in required storage is clearly non-linear.

5.1.3 Influence of Optimization Methods

The concepts of blocking relationship filtering and part group generation for the purpose of runtime optimization have been explained in Chapters 3 and 4. We have found these methods to have a considerable impact on the performance of the preprocessing module. The most apparent improvements were recorded using the Aviation Engine assembly. Without optimization, the total runtime for preprocessing the assembly took just over 48 minutes. Thus, compared to our optimized approach, naive processing takes almost 5 times longer (see Figure 5.1). Furthermore, if no part grouping is used, the DPA needs to load 512 mesh files instead of 94 each time the Aviation Engine is opened for disassembly planning. Thus, the optimization mechanisms not only affect the runtime for generating the static assembly information, but also the loading time in the DPA.

5.1.4 Influence of Mesh Shrinking

The effect of our mesh shrinking approach is difficult to evaluate objectively, since we are not aware of any suitable base line to which it can be compared. One theoretical possibility to allow classification of tolerance mechanisms for disassembly planning would be to provide a general test suite of part setups with an expected binary value for geometrical feasibility in the corresponding real-world assembly (i.e. "not separable" and "separable"). However, since the error in the provided models is often unintentional (either caused by imprecision or negligence), the criteria for such a test suite would also be ill defined.

One striking argument for the effectiveness of mesh shrinking in our system is the fact that, except for the Press assembly, none of the disassembly problems presented in this chapter could be solved with mesh shrinking disabled. For all examined exemplary assemblies, we employ the presented shrinking algorithm with $\sigma = 0.02$ and $\lambda = 20$.

One potential weakness of the employed algorithm is the fact that the shrinking distance defined by the provided parameter σ cannot be guaranteed, but is rather used as an upper bound. The algorithm is not well suited for organic shapes or triangle meshes that feature both very large and very small triangles, since under such circumstances the small distances separating the smaller triangles might affect the safety distance of the vertices in large triangles, resulting in little to none noticeable shrinking. Furthermore, the iterative nature and the corresponding requirement for a threshold value may lead to a lengthy trial-and-error process to find appropriate parameters. Since the mesh shrinking is executed at the very beginning of the preprocessing stage, it is not possible to verify the suitability of the selected values until the entire pipeline has been executed. If the result-

ing blocking relationships are found to be erroneous in the DPA, the entire preprocessing phase needs to be repeated.

5.2 Disassembly Examples

All renderings of disassembly explosion diagrams or animation phase previews were generated by taking screenshots from the active DPA system. For each examined assembly, we provide an image of the complete assembly, along with an illustration of the assessed disassembly problem. The selected set of assemblies contains diversified models which differ strongly in their overall design and complexity. Specific properties of the assembly or the disassembly problem – such as structural repetitiveness, strictly linear dependencies (e.g. in stacks) or a high variation of part sizes in the assembly – may elicit a certain behavior in the DPA or influence the perceived quality of the resulting disassembly path. These properties and corresponding effects on the results presented in the DPA are considered and discussed for each example. Potential weaknesses and possible means for improvement are pointed out for apparent artifacts that may affect the quality or usability of the program.

5.2.1 Press

The Press assembly (see Figure 5.3) has been used several times throughout this thesis. This is due to its comparably simple structure, which makes it a viable candidate for creating easy-to-understand visuals. Also, all possible disassembly paths are rather short, and the corresponding explosion diagrams hardly produce visual clutter. Therefore, it is the only assembly for which we have included a complete preview of the disassembly sequence to illustrate the details of disassembly visualization in the DPA.

To demonstrate the removal of core components, we have selected a part of the basic stabilizing structure for removal. Figure 5.4 illustrates the corresponding disassembly problem and displays the *involved* rendering of the assembly using the resulting disassembly path information.

The removal procedure can be visualized in two ways, namely explosion diagrams for quick user evaluations of detected solutions and step-by-step animated instructions. In the following, both techniques are demonstrated in Figures 5.5 and 5.6. Due to the low number of parts, the explosion diagram is rather compact and easy to understand. Furthermore, motion blur is activated to establish a visual correlation between the final

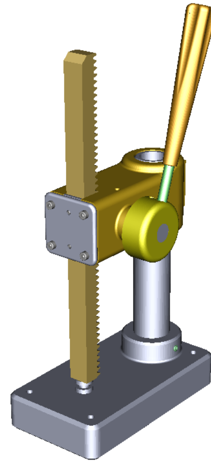
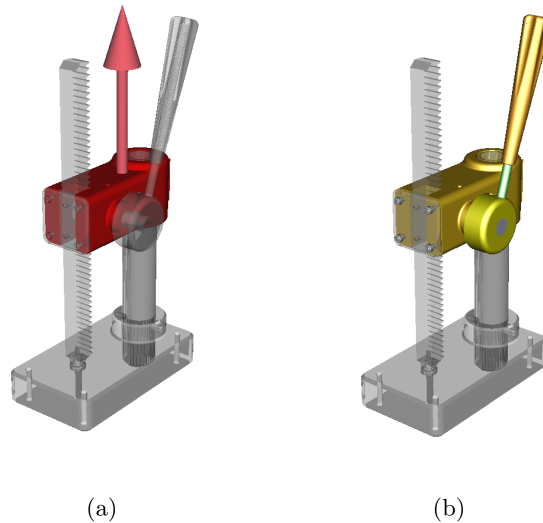


Figure 5.3: The Press assembly consisting of 23 parts, all of which can be removed, except for the base plate. This assembly serves as a control model and can be used to illustrate the behavior of the application for small assemblies.



(a) (b)

Figure 5.4: (a) The disassembly problem considered in this example. The part marked in red defines a basic structural component whose removal involves most parts in the assembly. (b) The partitions involved in the disassembly process are rendered as opaque objects, while the remainders are shown with transparency.

and initial position of all removed partitions. The disassembly instructions require removal of parts on both sides of the press, which is why the camera position was manually altered to provide a better view of the details.

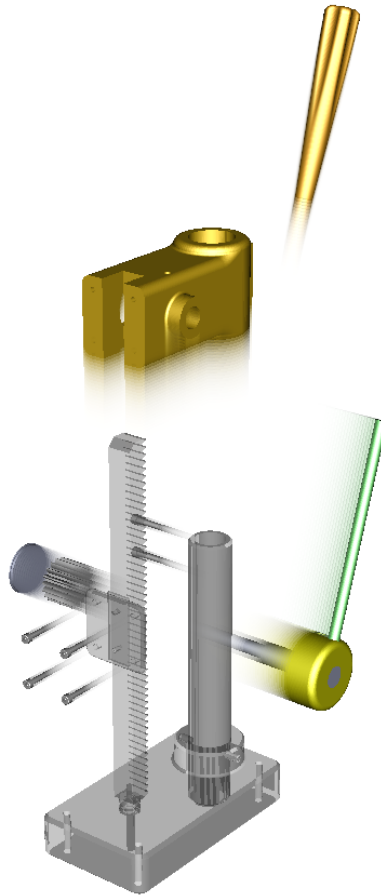


Figure 5.5: The explosion diagram illustrating the removal procedure for the disassembly problem. While this technique is mainly intended as a general overview for quickly evaluating the quality of disassembly paths, this particular example unambiguously conveys all removal actions due to the low number of parts involved and could possibly be used to directly perform the intended disassembly without further instructions.

Note that this disassembly sequence leaves no floating parts in the sense that all remaining partitions in the assembly are connected once the procedure is finished. However, since the connection of partitions is only based on contact information, we still experience some "quasi-floating" parts. Although the contact graph for the remaining partitions is connected, it seems unlikely that in a real-world example the components (especially the press bar itself) would remain in their current position without any additional support. On the other hand, such a disassembly procedure might very well be performed under different physical circumstances (e.g. low-gravity environments). Disallowing disassembly paths based on constraints other than geometrical feasibility would thus restrict the applicability of the system. Instead, we rely on user input to modify the disassembly path

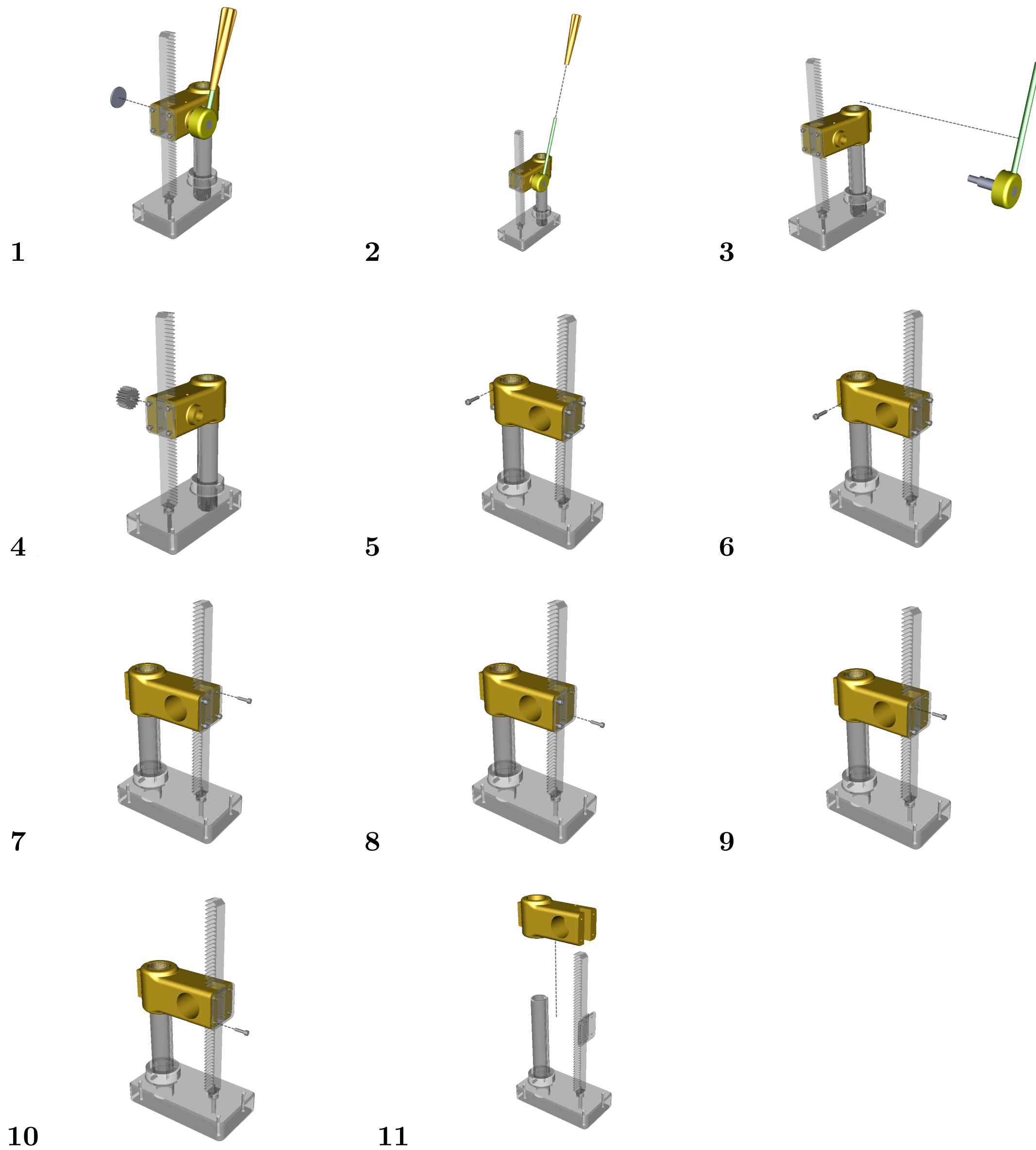
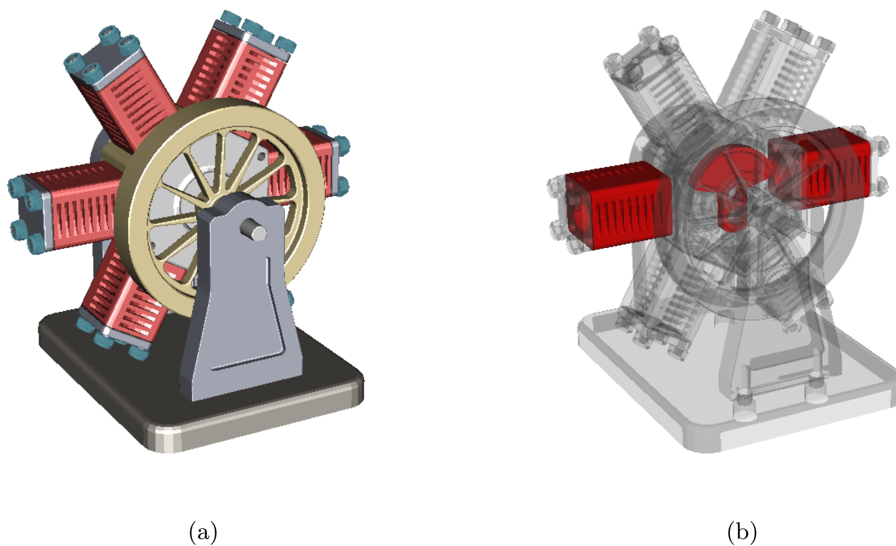


Figure 5.6: Disassembly instructions for the given disassembly problem in the Press assembly. Note that in the second image, the press appears to be significantly smaller. This is due to the program zooming out to make sure that the animation paths of the current phase are captured inside the viewing frustum. From image 5 onward, the assembly has been rotated manually to provide better visibility of the illustrated actions.

to avoid situations that inhibit their execution under the respective conditions.

5.2.2 Pneumatic 6-Cylinder Engine

We have chosen the model of a simple pneumatic engine with six cylinders as representative for instances of moderately complex assemblies. Figure 5.7 shows the complete assembly, as well as the selected disassembly problem using transparent rendering for all parts except the required components (red). This problem demonstrates the capabilities of the system to account for the removal of multiple required partitions in the resulting disassembly path.



(a) (b)

Figure 5.7: (a) The Pneumatic 6-Cylinder Engine assembly with six cylinder heads and simple internal mechanics. (b) The selected disassembly problem with three required partitions highlighted in red.

The base plate and the supporting pillars extending from it have been marked by us as fixed parts. As a result, most components that are involved in internal mechanics are removed via dual motions to avoid collision with the supporting structure. The explosion diagram for the initial solution can be seen in Figure 5.8.

Although it represents a feasible disassembly path, the partition placement and movement (indicated by static motion blur) seems not very intuitive. This is due to the program always selecting the first removal action for disassembly that is geometrically feasible. Furthermore, the diagram is unnecessarily expansive. These issues can be addressed in the DPA by modifying the designated removal actions of the individual partitions. Changing the according translational motions to a more straightforward choice (e.g. combination

of Euclidean unit vectors) results in a more compact explosion diagram that may also be understood more easily (see Figure 5.9). The eventual disassembly path involves a number of dual translating motions, the evaluation of which represents an important feature of the

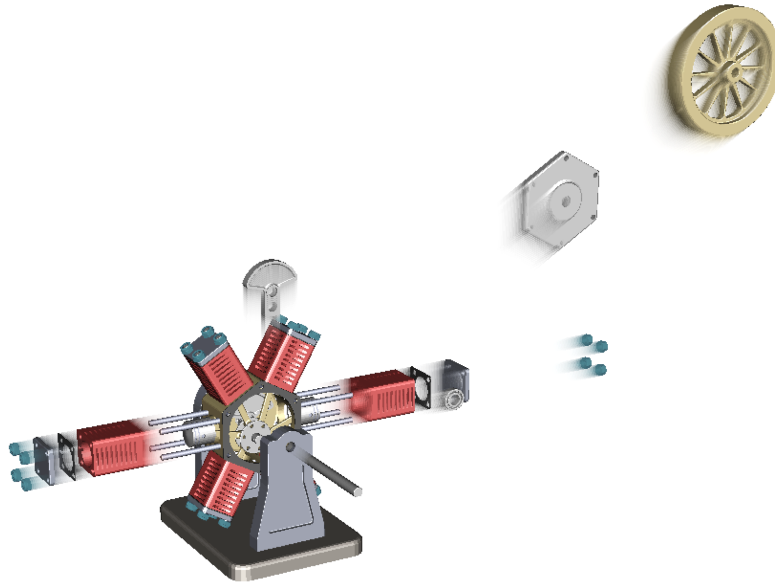


Figure 5.8: The initially calculated explosion diagram for the disassembly problem with multiple required components. Although geometrically feasible, some of the partition placements resulting from the disassembly seem unsuitable.

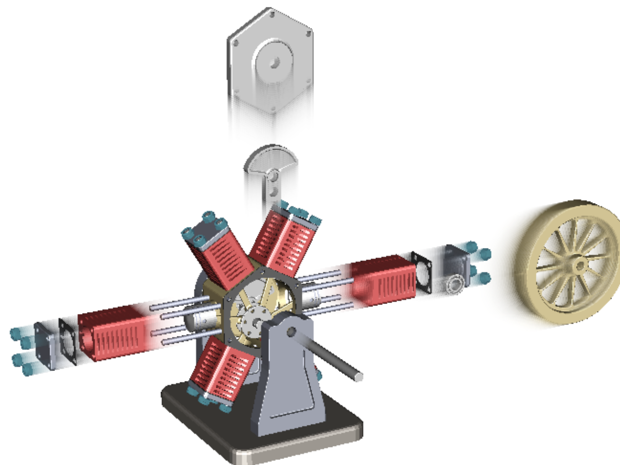


Figure 5.9: By changing the designated removal actions of individual partitions, the diagram becomes more intuitive and also more compact.

DPA. The preview of such a dual translating motion using the step-by-step instructions can be seen in Figure 5.10.

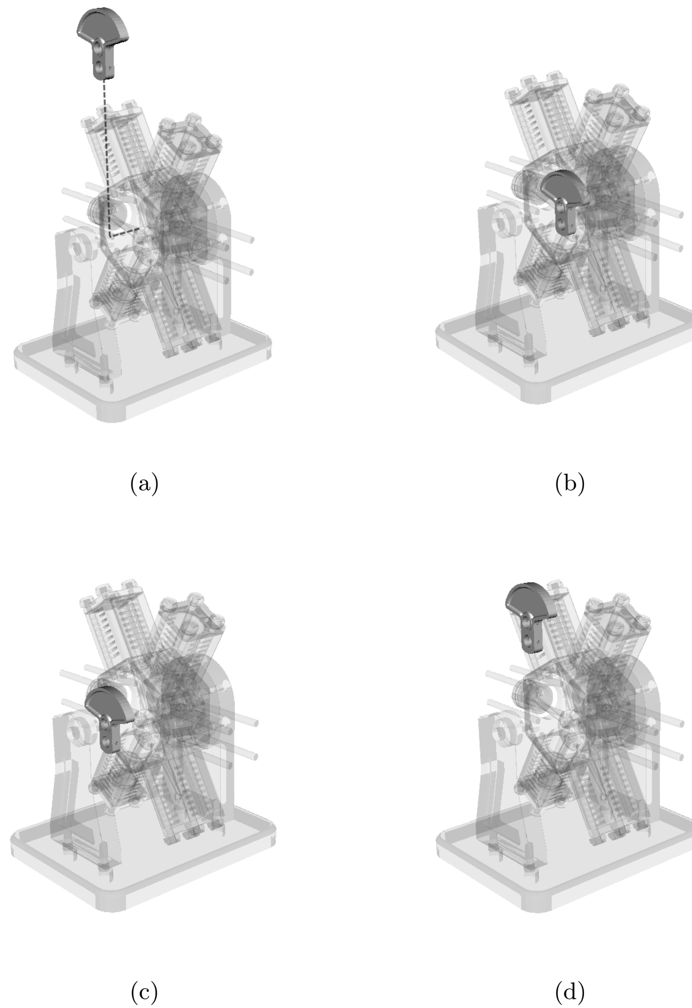
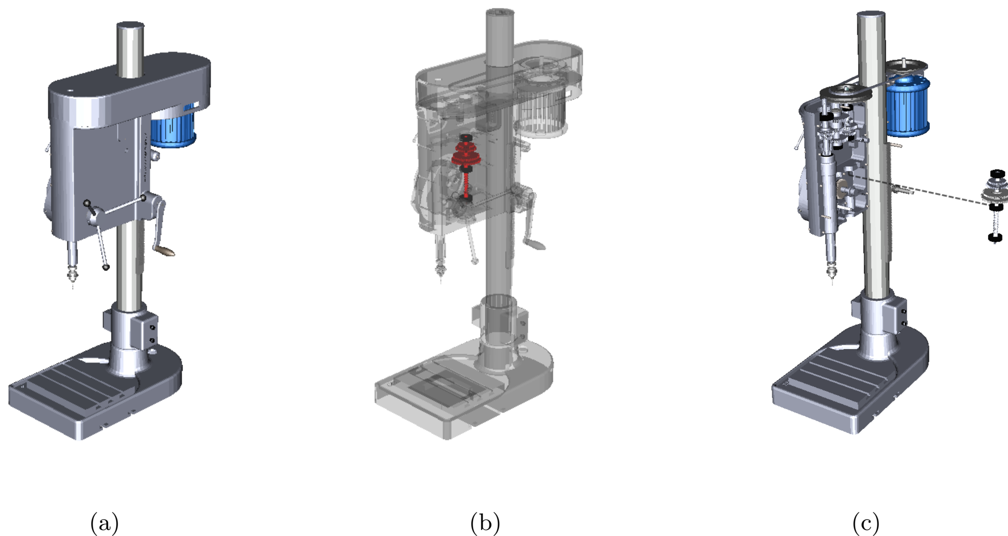


Figure 5.10: A dual motion in the disassembly path for the pneumatic engine is shown in (a). The translational path is indicated using guidelines connecting the intermediate positions. Images (b) through (d) depict screenshots of the animated step-by-step instructions with the final required partition being removed.

5.2.3 Drill

The Drill assembly represents a rather complex model of a manual drill stand, complete with fixation and drill plate. The disassembly problem for this assembly involves two of the key features of the DPA. The required component is actually a group of gears beaded

on a rod, which were manually fused to form a custom partition. Furthermore, efficient removal of this new partition requires the evaluation of dual translating motions. Figure 5.11 shows the complete drill assembly, a transparent view of the model with the required custom partition highlighted in red, as well as the final removal action being applied to the required partition. The base plate and main supporting pillar of the model were marked as fixed parts for the disassembly path computation.

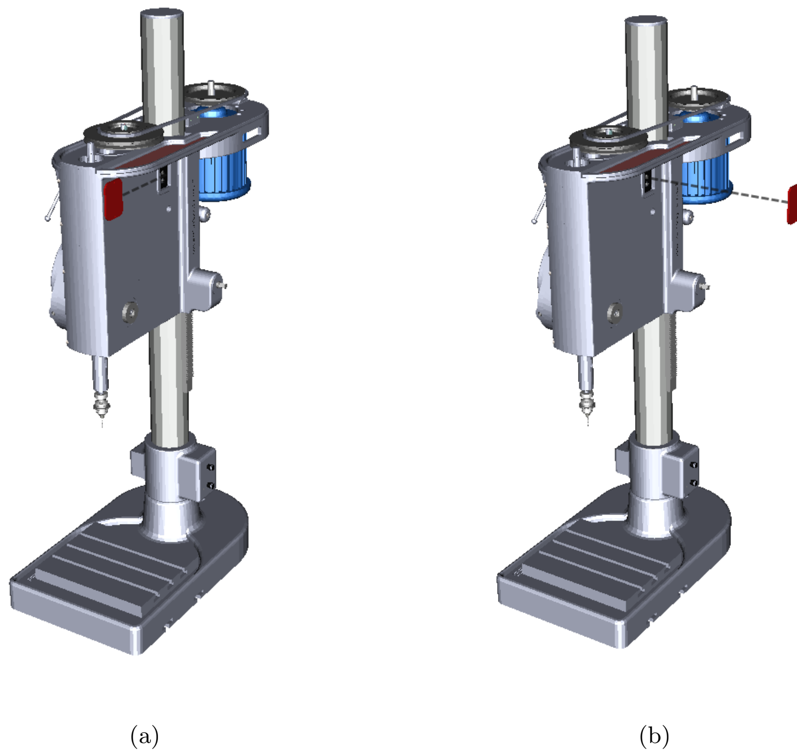


(a) (b) (c)

Figure 5.11: (a) The Drill assembly. (b) A custom partition consisting of a rod and multiple functional gears is selected as the required partition for removal. (c) The eventual removal action of the required partition. To reduce the number of partitions involved in the disassembly procedure, the system automatically chooses a dual translating motion.

Similar to the example of the radial engine, the initially detected disassembly path suffers from partitions being removed in ways that are feasible, but not necessarily ideal for visualizing the procedure. For instance, Figure 5.12 shows two possible options for removing a small cover lid from the casing of the drill. While both are geometrically feasible, the second one is clearly superior in terms of clarity and reproducibility.

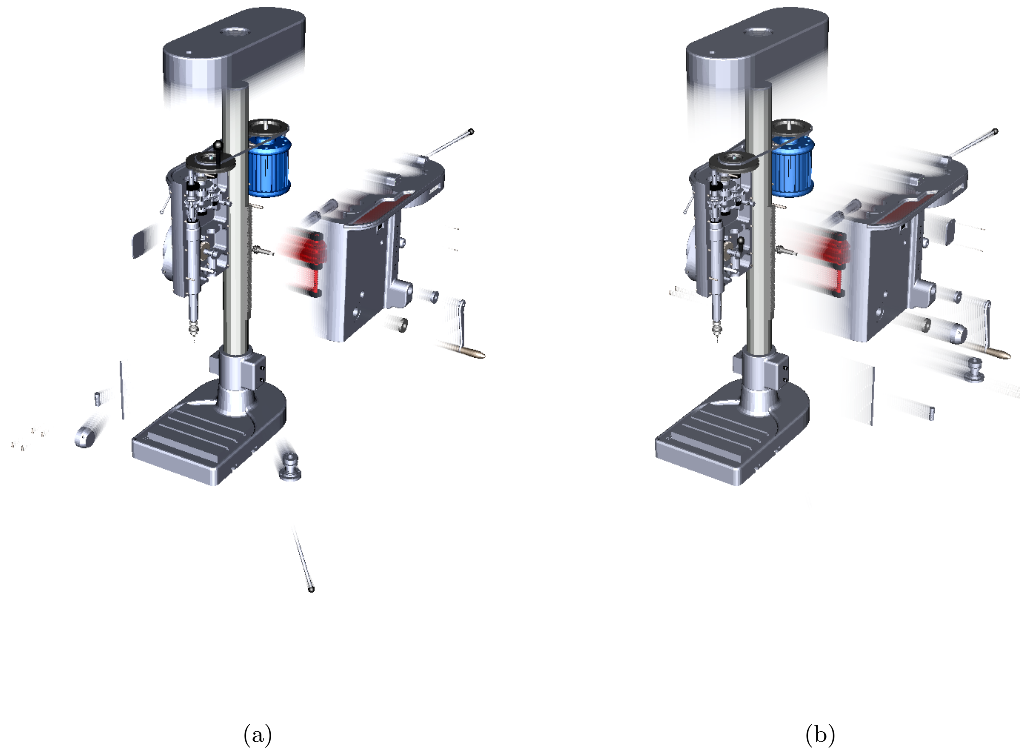
The drill assembly features a moderately high number of parts, as well as a strong sequential dependency for part removability, i.e. there are only few groups of partitions that can be removed in parallel in the same phase. Thus, the list of instructions for removing the required core partition contains more than 40 animation phases and even more individual removal steps. Furthermore, numerous dual translations are used to keep the number of partitions involved in the disassembly procedure at bay. The corresponding disassembly



(a) (b)

Figure 5.12: (a) The cover lid (red) is removed in a sliding motion, by translating it along one of its principle axes. Depending on the rendering style being used, this removal action be easily missed. (b) By removing the lid in a direction perpendicular to the surface of the case, the implied action becomes clearer and more noticeable.

procedure is therefore rather complex and cannot be easily understood from the explosion diagram alone. However, the explosion diagram can still provide a general overview and give hints as to which properties of the involved partitions should be modified to improve the quality of the disassembly path. Figure 5.13(a) shows the diagram corresponding to the initial solution, with some of the parts being placed in awkward positions. Although the DPA does not make any automatic assumptions about the preferences of the user for disassembly paths, it provides the necessary functionality to modify suggested solutions at will. These features were used by us to improve the perceived visual quality of the disassembly path from a general perspective, by reducing the set of directions used in the removal actions throughout the procedure to achieve an overall alignment along two perpendicular coordinate axes. Figure 5.13(b) shows the improved diagram after correcting the oddities indicated in the original by specifying designated removal actions for the according partitions.



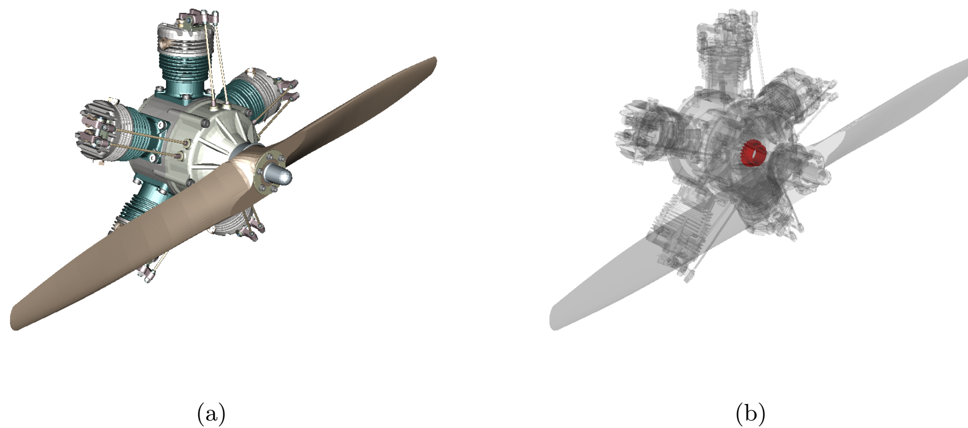
(a) (b)

Figure 5.13: (a) The initial explosion diagram created by the DPA. The disassembled partitions expose a certain coherence concerning the directions used for removal, which is broken by some parts that are placed in odd locations. (b) The explosion diagram representing the edited disassembly path. The overall coherence is increased by restricting the directions used for removal to two of the coordinate axes.

5.2.4 Radial Engine

The Radial Engine assembly has a fairly complex structure and a high level of detail. Since it represents the complete model of a functional 5-cylinder engine, it contains a variety of different classes of components, combining large core parts such as the propeller or the casing, with rather small auxiliary elements for fixture. The duality concerning the size of these parts makes it an interesting test case to evaluate the performance of the DPA when dealing with heterogeneous assemblies. Figure 5.14 depicts the assembly using fully opaque rendering and the examined disassembly problem, which requires the program to remove a ball bearing that is located at the very center of the engine.

The diversity in shape and size among the parts in the assembly directly affects the



(a) (b)

Figure 5.14: (a) The Radial Engine Engine assembly as seen from the front. The internal mechanics which account for the majority of the parts are not visible. (b) A transparent rendering of the engine with the required target partition highlighted in red. Evidently, the assembly is not only comprised of its outer hull, but also contains complex functional components such as pistons, valves and gears.

quality of the explosion diagram representing the suggested solution. Figure 5.15 shows the generated diagram with enabled static motion blur. While the movement and spacing between larger components is lucid and a connection to their initial position is easily established, smaller parts are considerably spaced out in comparison, which makes it difficult to infer even the reason for their removal. This is due to the fact that the function for calculating explosion diagrams only considers the combined AABBs when creating a partition placement that is free of intersections. Although the employed algorithm provides fast performance and enables us to quickly process large assemblies, the perceived visual quality of the resulting diagrams may suffer. While the generated diagram may still be considered sufficient for the purpose of presenting an overview of the disassembly procedure, it is unnecessarily expansive. In this particular case, the quality can only be marginally improved through user interaction, since the DPA does not include tools specifically for editing the behavior of partitions when exploded.

It should be noted that the partition placement in the explosion diagram does not correspond to the distance over which the partition is moved during the step-by-step instructions. Thus, the animation of the procedure in the DPA may still be easy to understand where the explosion diagram fails to capture all involved steps in a single frame. Figure 5.16 illustrates the preview images of an animation phase where a group of

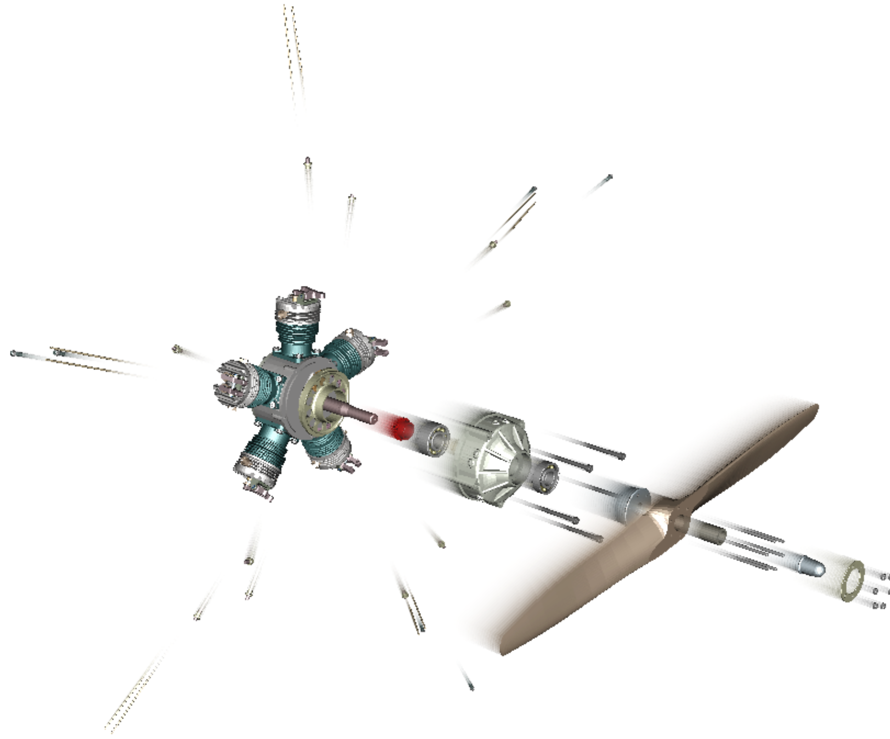


Figure 5.15: Explosion diagram generated for removal of the ball bearing (red) from the Radial Engine with static motion blur enabled. Some of the smaller components which are exploded in directions perpendicular to the main axis are hardly visible and separated from each other by unnecessarily long distances.

small bolts is removed. Note that in the explosion diagram, these bolts are both difficult to locate and also placed much further from their original position to avoid intersections with the partitions in the other phases.

To further reduce the risk of missing the removal of particularly small parts, the animation for removing the bolts during disassembly automatically activates the supportive visual cues included in the system. Figure 5.17 shows two screenshots of the DPA during animation with active close-up camera and circular billboard focusing on one of the bolts being removed.

5.2.5 Mecanum Wheel

Mecanum wheels enable motorized vehicles to perform omni-directional maneuvers without the need for conventional steering mechanisms. To achieve this, the mecanum wheel is

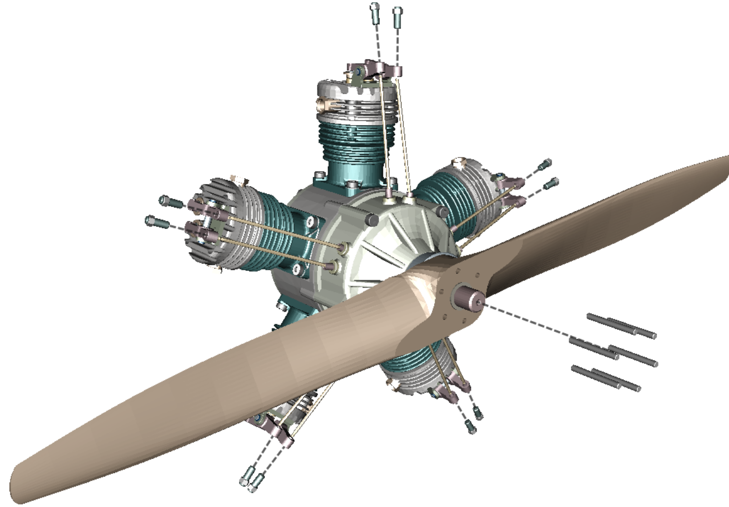
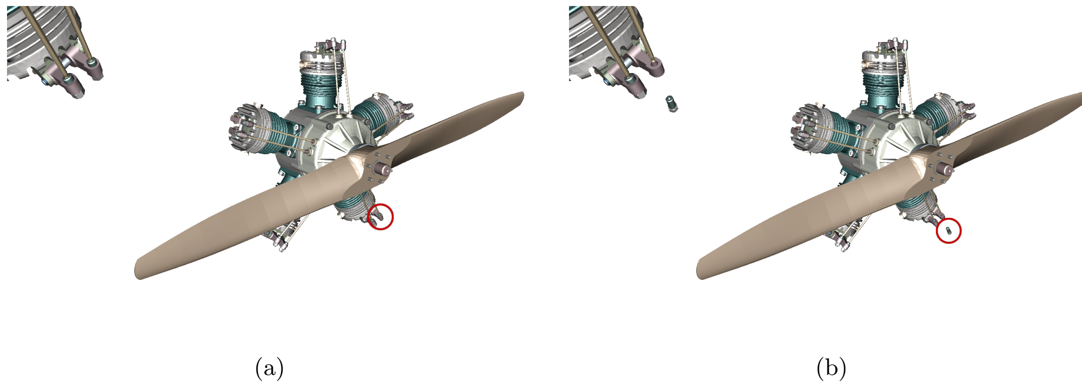


Figure 5.16: Preview of an animation phase covering the removal of fixture bolts. In contrast to the explosion diagram, the corresponding instructions are easily understood.



(a) (b)

Figure 5.17: Screenshots of the DPA animating the removal of a small fixture bolt. Since the component in question is comparably small in respect to the diameter of the full assembly, visual cues are activated (close-up camera and red circle billboard) to facilitate the comprehension of the illustrated instruction. (a) has the bolt in its initial position, while (b) shows it at the end of its animation cycle.

fitted with a set of rollers protruding from its perimeter at evenly spaced intervals. The structure of a mecanum wheel is thus highly repetitive, as the same components are being used multiple times in different locations. As such, it provides a good example for testing the effects of part grouping and aggregate animation phases. The Mecanum Wheel assembly examined by us is depicted in Figure 5.18 along with the corresponding

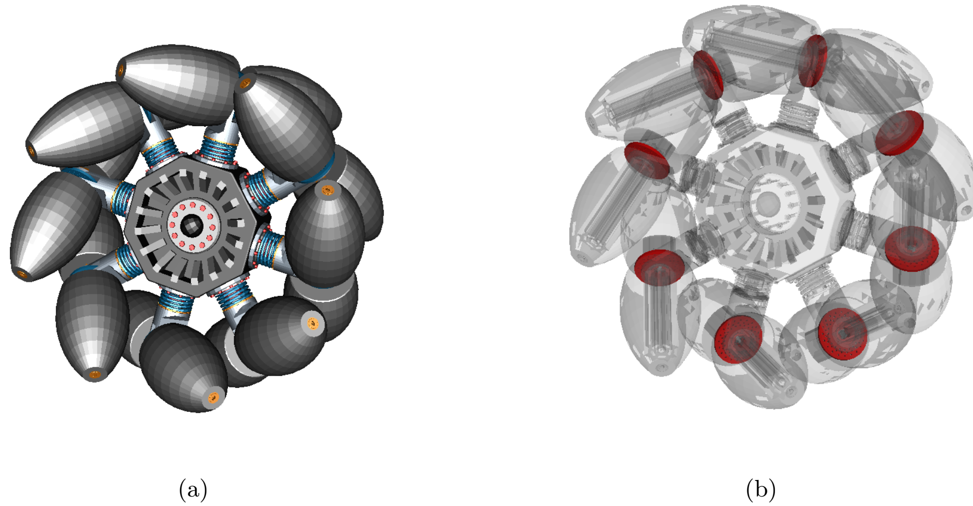


Figure 5.18: (a) The Mecanum Wheel assembly fitted with a set of rollers protruding from its base. The apparent repetitive structure suggests a strong potential for optimizing disassembly calculation and illustration. (b) The disassembly problem assessed for this model. 8 identical disc components are selected to be removed from their respective location.

disassembly problem. We have chosen to remove all 8 instances of a specific disc component from the assembly.

Surprisingly, the mecanum wheel does not lend itself to part group generation in our system during preprocessing. This is due to the fact that, although the same triangle meshes are being used multiple times, most of them differ in terms of the absolute rotation that is applied to them. The impact of part grouping on the runtime for the blocking relationship evaluation is therefore almost non-existent: the original number of 254 parts can only be reduced to 246 part groups. Although the complete runtime for preprocessing the model is below 4 minutes using contemporary hardware, its obviously repetitive structure should provide ample potential for optimizing C-Space object generation, which represents the most expensive processing step in terms of runtime (over 2 minutes). Theoretically, part grouping for assemblies such as the Mecanum Wheel could become more effective by monitoring and grouping parts based on their **relative** rotation. For example, two part groups $\{A, B\}$ and $\{C, D\}$ could be defined where the rotation applied to A and B may differ, but A is identical to C while B is identical to D when applying the same rotation matrix to A and B respectively. In this case, the C-Space object created for $A \oplus C$ could be reused for $B \oplus D$ by adjusting the camera to match the relative rotation of A

and B.

However, loading the assembly files to the DPA and the disassembly planning procedure itself both benefit from the apparent reuse of identical triangle meshes. Although they have different rotations applied to them, the corresponding geometry file needs to be read only once. The respective instancing and transformation is achieved efficiently via OSG by wrapping the same geometry node with distinct transformation nodes in the scene graph. Furthermore, the automatic partition grouping for animation phases effectively captures the total number of 32 instructions required for removing the discs in only 4 animation phases and the corresponding preview images. Figure 5.19 exhibits the phase preview of phases 2 and 4 respectively.

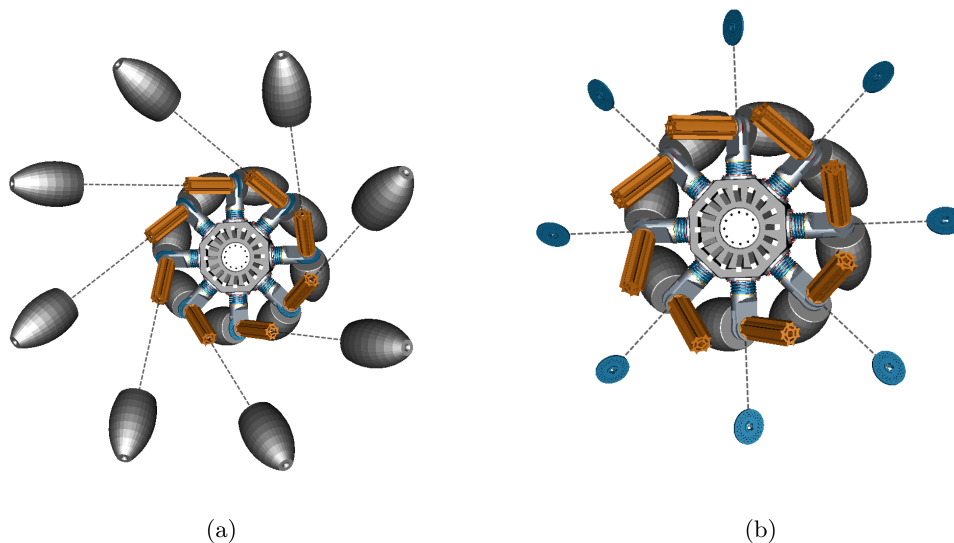


Figure 5.19: (a) Preview of the second animation phase, in which all 8 rollers are removed consecutively. All corresponding removal actions are coherently captured in one single image. (b) The final animation phase, during which the required disc components are successfully disassembled.

Since the solution to the disassembly problem posed for the mecanum wheel involves only singular translating motions, the resulting explosion diagram is easy to understand and comparably compact, although the number of removed partitions is rather high. This is also due to the relatively parallel structure of the dependency hierarchy created from the peeling phases. Each required partition is indirectly dependent on only three isolated objects that no other required partition depends upon. From this, we can conclude that the complexity of an explosion diagram does not depend only on the number of removed

partitions, but also on the degree of inter-dependencies and blocking relationships between the involved parts. Note that in this particular example, the set of peeling phases is congruent to the set of animation phases. As can be seen in Figure 5.20, there are 4 layers of removed partitions generated by peeling (from the outside inward: pins, rollers, rods and discs), which also directly represent the layers of the dependency hierarchy.

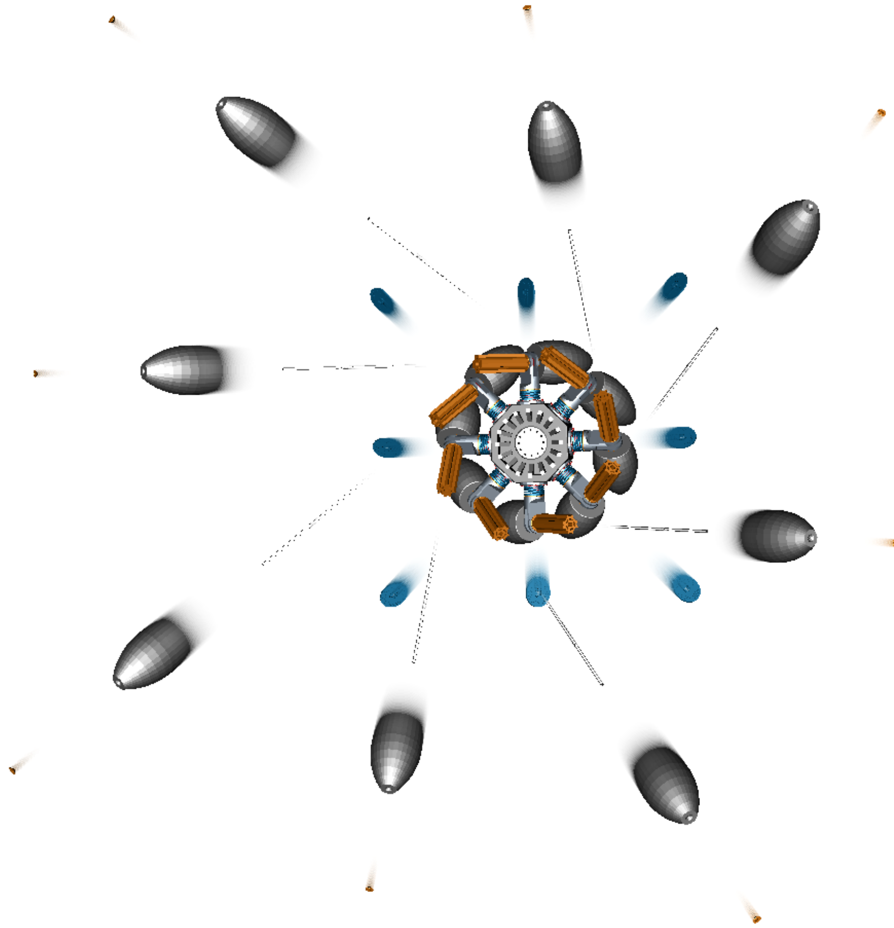
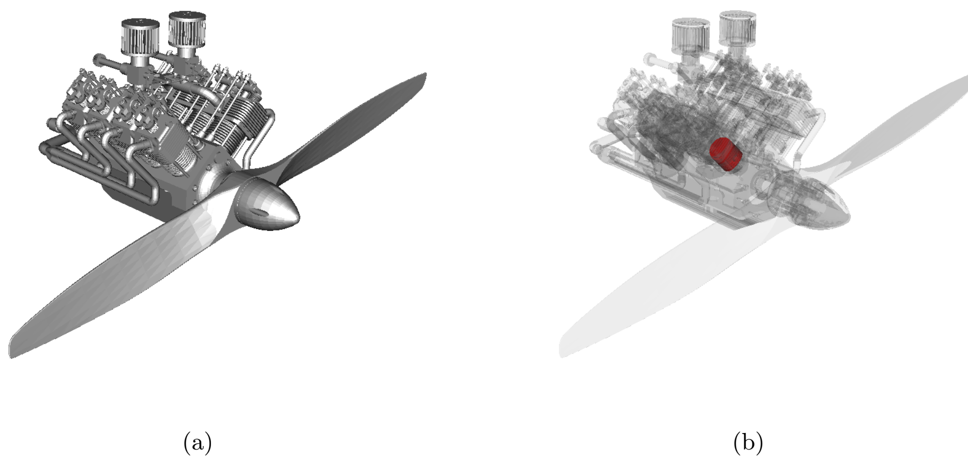


Figure 5.20: Explosion diagram for the mecanum wheel. Although the number of required steps involved in the disassembly is rather high, the diagram is both easy to understand and compact, due to the strictly isolated sets of dependent partitions.

5.2.6 Aviation Engine

The Aviation Engine assembly is the most complex model that has been evaluated in our disassembly planning system. The assembly represents a detailed model of a complete 8-cylinder propelling engine. It contains a high number of parts which strongly vary in

shape and size, many of which require dual translating motions to remove. Figure 5.21 depicts the assembled engine, along with a rather challenging disassembly problem, where one of the piston heads is selected for removal. The disassembly of this specific partition requires the removal of more than 50 indirect blockers. However, a feasible disassembly procedure can be computed or reevaluated with modified partition properties in less than 2 seconds on contemporary hardware, thus ensuring our constraint for immediate feedback to all user interaction.



(a) (b)

Figure 5.21: (a) The Aviation Engine assembly of a fully functional propelling engine. With a grand total of 512 individual parts, the model can be considered highly complex in terms of disassembly planning. (b) The corresponding disassembly problem. We have chosen one of the piston heads (red) for removal. Note that the piston is located beneath the cylinder cover, which represents one of the main structural elements in the assembly.

Figure 5.22 shows the explosion diagram illustrating the procedure in the DPA. It should be noted, that this image does not depict the initial diagram, but rather an edited version that has been adapted via the user interface to improve visual quality. The partition arrangement in the original is significantly messier and more confusing. Due to the lack of novelty in this respect when compared to the assessment of previous examples, the initial explosion diagram is omitted at this point. However, even the improved, compact explosion diagram is rather expansive, due to its numerous hierarchy layers, resulting from a high number of sequentially dependent partitions. Viewing it in its entirety requires the user to zoom out considerably in order to capture the complete scene in a single frame. For assemblies with a significantly higher number of parts and similarly strong sequential

dependency, arranging the partitions and the camera setup to provide a compact, yet expressive overview of the procedure may become infeasible altogether. We consider the disassembly problem of the Aviation Engine assembly to represent a borderline case in this respect.

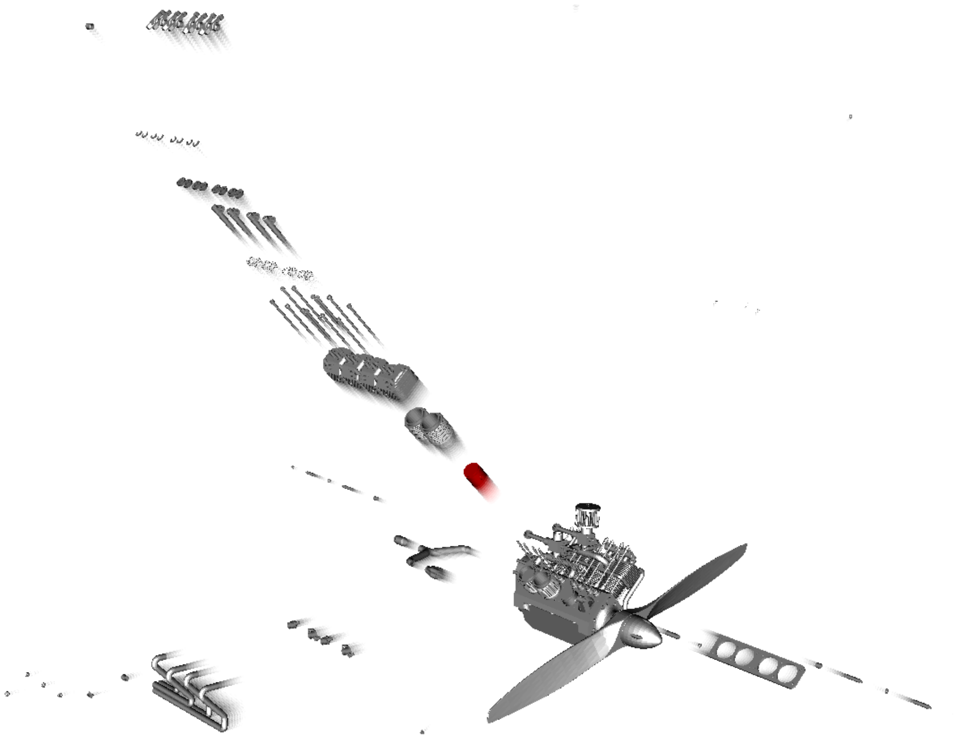
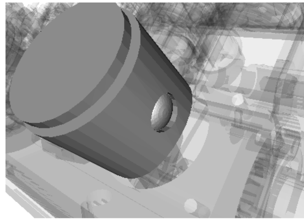
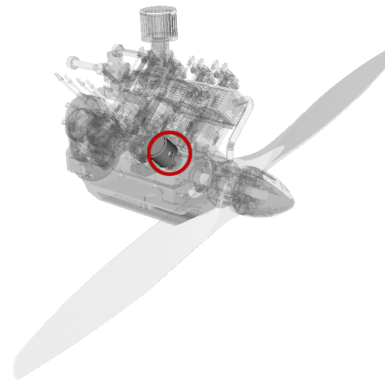
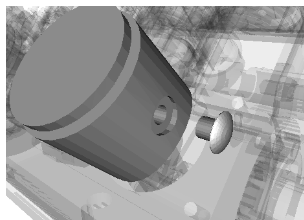
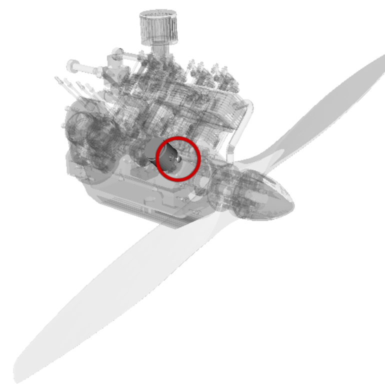


Figure 5.22: Explosion diagram of the Aviation Engine for the given disassembly problem. Although we have exploited part grouping and designated removal actions to improve upon its compactness and visual quality, the arrangement requires the user to zoom out considerably to capture all partitions. With a growing number of hierarchy layers, the explosion may eventually grow to a point where its extent no longer allows the user to discern individual partitions based on their appearance when looking at the full scene.

The step-by-step instructions of the disassembly procedure are represented by 21 separate animation phases. In terms of clarity, they are superior to the explosion diagram, which is mainly due to the fact that during animation, there is no need to resolve intersections with partitions contained in different animation phases. Illustrating the removal of smaller components is supported using the introduced highlighting mechanisms (billboard and close-up camera). Figure 5.23 displays the removal of the piston fixture with enabled visual cues.

(a)

(b)

Figure 5.23: Animation of the piston fixture being removed just before the required piston head can be extracted. Due to the small projected size of the fixture on the screen, the visual cues are triggered to enhance its visibility. (a) shows a screenshot of the DPA at the beginning of the animation cycle, while (b) marks its end, with the partition being placed in its final extracted position. Note that the rendering style that is used in the main animation window (*involved*) also affects the generated picture of the close-up camera.

Chapter 6

Conclusion

In this thesis, a system has been presented for interactively planning, editing and verifying disassembly procedures for products based only on the geometrical information of their components, with a special focus on enabling processing of complex assemblies (i.e. highly detailed models and a high number of parts). The most apparent problems in such an endeavor have been identified and discussed. By dividing the targeted functionality into two core modules, the more complex preprocessing stage is effectively separated from the light-weight disassembly planning application which provides a user interface and has much lower requirements in terms of available hardware. Therefore, once the static assembly information has been calculated on a designated machine, disassembly planning itself may be performed on a much weaker (potentially mobile) device. Throughout this thesis, the diversity of varying criteria for perceived disassembly path quality has been mentioned multiple times. For this reason, a solution was chosen that constrains the calculation of disassembly paths by geometrical feasibility only. By employing a C-Space approach to extract the blocking relationships that are used during disassembly path computation, it is ensured that all disassembled partitions for a suggested solution are both locally and globally free at their time of removal.

In order to allow a wide variety of assemblies to be processed in our system, common issues with available CAD data sets and the implications for the results of disassembly planning have been analyzed. A parallel algorithm for mesh shrinking is proposed to provide a tolerance mechanism for testing separability of individual parts. Thus, the system is not dependent on the input to be numerically exact. Using the GPU for creating and rendering the polyhedral C-Space objects provides a fast method for testing the geometrical feasibility of multiple discrete removal actions in parallel with high precision. In contrast

to most previous disassembly systems, the presented system considers not only singular, but also basic dual translating motions during the analysis of the assembly.

The input for polyhedral C-Space evaluation of each part is carefully filtered by using bounding box C-Space objects and separating part occlusion to trivially cull objects that have no blocking influence. Furthermore, the system detects and maintains groups of identical parts, which can be used to increase the performance during the preprocessing stage. The effect of these optimizations is most noticeable for large assemblies and provides a key element for enabling the system to process hundreds of parts in a matter of minutes.

Instead of using rules for partitioning or making assumptions about part interaction, these tasks are delegated to the user by providing a complete user interface for editing the properties of parts and partitions in the assembly. In doing so, our system allows the user to efficiently choose alternative disassembly paths and can provide instant visual feedback due to the linear nature of the employed disassembly computation algorithm. In addition to the set of precomputed removal actions, the DPA allows the user to test custom removal actions as concatenations of translating motions. Disassembly paths are available for display using two separate techniques, namely explosion diagrams and step-by-step instruction animation. Rendering techniques are employed to provide visual cues which increase the clarity of animated instructions and facilitate the overall interaction of the user with the system.

Although the system is capable of handling a wide range of assemblies, there are several limitations, most of which are directly related to the inability of detecting specific actions that might be required for disassembling a product. The system is currently restricted to testing translational movement only. Thus, removal actions that explicitly require rotations may not be detected*. Furthermore, all parts are assumed to be solid, since no information about deformability can be extracted from the part geometry. Although the space of removal actions that are automatically tested can be extended by the user to some degree, it is per definition discrete, thus any custom operation not included in this limited set needs to be defined manually in the DPA. While testing principal directions and Euclidean unit vectors may suffice for a large portion of assemblies, the system can be easily stalled by seemingly simple setups where one part needs to be removed in a direction that is not included in this set. Nevertheless, the discrete approach was chosen for the presented system due to its guaranteed high performance and low runtime, especially during assessment in the DPA where we achieve real-time feedback. The number of tested

*Note however, that for instance threaded screws and bolts may be indeed found to be separable in our system since the mesh shrinking algorithm can effectively cancel out the obstacles posed by the threading.

actions can be set by the user, which provides a mechanism for limiting required resources for large assemblies in cases where only a few simple removal actions need to be considered. Furthermore, analytical approaches such as the NDBG usually assume or require a certain numerical exactness of the input data set. This property can usually not be guaranteed for general CAD assemblies that were designed by amateurs or inexperienced users. However, with the constantly increasing power of modern CPUs and new advancements in this field of research, analytical approaches using NDBGs on inexact data sets may become feasible in real-time as well, especially when combined with strong user-defined constraints.

As has been shown in this thesis, evaluating all possible partitionings for complex, strongly connected assemblies is hardly feasible due to runtime and disk storage requirements. Furthermore, it can be argued that in order to arrive at a subjectively ideal disassembly path, editing an automatically generated partitioning hierarchy may require just as much or even more effort than defining all larger partitions by hand. Consequently, the system uses a comparably naive initial partitioning method and a simple peeling algorithm to detect possible solutions. While the algorithm can quickly detect a suitable solution for assemblies that only require removal of single parts, it completely relies on user interaction when considering advanced partitioning. Thus, the DPA may be unable to find a valid disassembly path that involves two or more individual parts being removed as one, unless the user provides the corresponding information. This may become increasingly troublesome with complex assemblies of large size, more notably so if the same partitioning patterns are used multiple times throughout the assembly. The problem of partitioning, though it has been thoroughly researched, is possibly one of the most difficult problems that currently stifle the versatility of applying fully automatic approaches to larger assemblies.

One of the techniques for visualizing the disassembly procedures is the utilization of explosion diagrams. Although they provide good visual feedback for simple setups, it has been shown in this thesis that even compact explosion diagrams may become too large to be displayed in their entirety and still convey necessary details in the scene. This may eventually become a limiting factor for their suitability when considering large assemblies.

While these limitations are quite severe, the methods described in this thesis aim to provide a worthwhile approach to a number of common problems in disassembly planning. The completed system exploits recent developments in parallel computing considering both hardware and algorithms to achieve high performance for large data sets and integrates the user as a key participant to provide additional information where no solutions can be

found. All mentioned shortcomings provide challenging areas for future research. In order to further improve the applicability of fully or partially automatic disassembly planning systems, sophisticated algorithms and elaborate visualization techniques are required and may contribute equally towards improving the quality of future implementations.

Appendix A

Acronyms and Symbols

List of Acronyms

GHz	gigahertz
KB	kilobyte
MB	megabyte
GB	gigabyte
FAG	Face Adjacency Graph
DBG	Directional Blocking Graph
NDBG	Non-Directional Blocking Graph
RG	Removal Influence Graph
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
AABB	axis-aligned bounding box
OSG	Open Scene Graph
GLSL	OpenGL Shading Language
VBO	Vertex Buffer Object
RGB	red-green-blue color space
RGBA	red-green-blue-alpha color space
FOV	field-of-view
DPA	Disassembly Planning Application
C-Space	Configuration Space

List of Symbols

K	thousand
$\neq$	inequality
$\neg$	negation
$\leftarrow$	assignment
$\mathcal{O}$	Big-O notation
$\cup$	union
$\cap$	intersection
$\emptyset$	empty set
$\setminus$	complement
$\overrightarrow{d}$	direction
$\in$	element of
$\notin$	no element of
$\oplus$	Minkowski sum
$\ominus$	Minkowski difference
ω	removal action
Φ	peeling phase
Λ	animation phase
B_r	blocking relationship
$\boldsymbol{P_x}$	partition
$\boldsymbol{P_0}$	top-most partition
σ	mesh shrinking percentage
λ	mesh shrinking iterations
α	FOV angle

Bibliography

[1] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[2] Agarwal, P. K., de Berg, M., Halperin, D., and Sharir, M. (1996). Efficient generation of k-directional assembly sequences. In *Proceedings of the seventh annual ACM-SIAM symposium on discrete algorithms*, SODA '96, pages 122–131, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[3] Agrawala, M., Phan, D., Heiser, J., Haymaker, J., Klingner, J., Hanrahan, P., and Tversky, B. (2003). Designing effective step-by-step assembly instructions. *ACM Trans. Graph.*, 22(3):828–837.

[4] Bourjault, A. (1984). *Contribution à une approche méthodologique de l'assemblage automatisé: élaboration automatique des séquences opératoires*. Université de Franche-Comté.

[5] De Fazio, T. and Whitney, D. (1987). Simplified generation of all mechanical assembly sequences. *Robotics and Automation, IEEE Journal of*, 3(6):640–658.

[6] De Mello, L. H. and Sanderson, A. C. (1986). And/or graph representation of assembly plans. In Kehler, T. and Rosenschein, S. J., editors, *AAAI*, pages 1113–1121. Morgan Kaufmann.

[7] Fogel, E. and Halperin, D. (2007). Exact and efficient construction of minkowski sums of convex polyhedra with applications. *Computer-Aided Design*, 39(11):929–940.

[8] Guo, J., Yan, D.-M., Li, E., Dong, W., Wonka, P., and Zhang, X. (2013). Illustrating the disassembly of 3d models. *Computers & Graphics*, 37(6):574–581.

[9] Hachenberger, P. (2007). Exact minkowksi sums of polyhedra and exact and efficient decomposition of polyhedra in convex pieces. In *Algorithms–ESA 2007*, pages 669–680. Springer.

[10] Halperin, D., Latombe, J.-C., and Wilson, R. H. (2000). A general framework for assembly planning: The motion space approach. *Algorithmica*, 26(3-4):577–601.

[11] Halperin, D. and Wilson, R. H. (1994). Assembly partitioning with a constant number of translations. Technical report, SAND94-1819, Sandia National Labs.

[12] Halperin, D. and Wilson, R. H. (1995). Assembly partitioning along simple paths: the case of multiple translations. In *ICRA*, pages 1585–1592. IEEE Computer Society.

[13] Homem de Mello, L. and Sanderson, A. C. (1991). A correct and complete algorithm for the generation of mechanical assembly sequences. *Robotics and Automation, IEEE Transactions on*, 7(2):228–240.

[14] Jones, R. E., Wilson, R. H., and Calton, T. L. (1997). Constraint-based interactive assembly planning. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 2, pages 913–920. IEEE.

[15] Kaul, A. and Rossignac, J. (1992). Solid-interpolating deformations: Construction and animation of pips. *Computers & Graphics*, 16(1):107–115.

[16] Kavraki, L., Latombe, J.-C., and Wilson, R. H. (1993). On the complexity of assembly partitioning. *Information Processing Letters*, 48(5):229 – 235.

[17] Lambert, A. J. D. (2002). Determining optimum disassembly sequences in electronic equipment. *Comput. Ind. Eng.*, 43(3):553–575.

[18] Latombe, J.-C. and Wilson, R. H. (1995). Assembly sequencing with toleranced parts. In *Proceedings of the third ACM symposium on Solid modeling and applications*, SMA '95, pages 83–94, New York, NY, USA. ACM.

[19] Li, W., Agrawala, M., Curless, B., and Salesin, D. (2008). Automated generation of interactive 3d exploded view diagrams. *ACM Trans. Graph.*, 27(3):101:1–101:7.

[20] Li, W. and McMains, S. (2011). Voxelized minkowski sum computation on the gpu with robust culling. *Comput. Aided Des.*, 43(10):1270–1283.

[21] Lien, J.-M. (2009). A simple method for computing minkowski sum boundary in 3d using collision detection. In Chirikjian, G., Choset, H., Morales, M., and Murphey, T., editors, *Algorithmic Foundation of Robotics VIII*, volume 57 of *Springer Tracts in Advanced Robotics*, pages 401–415. Springer Berlin Heidelberg.

[22] Lozano-Perez, T. (1983). Spatial planning: A configuration space approach. *IEEE Trans. Comput.*, 32(2):108–120.

[23] Lozano-Perez, T. and Wilson, R. H. (1993). Assembly sequencing for arbitrary motions. In *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, pages 527–532. IEEE.

[24] Natarajan, B. K. (1988). On planning assemblies. In *Proceedings of the fourth annual symposium on Computational geometry*, SCG '88, pages 299–308, New York, NY, USA. ACM.

[25] Requicha, A. A. G. (1984). Representation of Tolerances in Solid Modeling: Issues and Alternative Approaches. In Pickett, M. S. and Boyse, J. W., editors, *Solid Modeling By Computers*, pages 3–22. Plenum Publishing Corporation.

[26] Romney, B., Godard, C., Goldwasser, M., and Ramkumar, G. (1995). An efficient system for geometric assembly sequence generation and evaluation. In *Proc. ASME Int. Computers in Engineering Conference*, pages 699–712.

[27] Shewchuk, J. R. (1997). Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363.

[28] Srinivasan, H. and Gadh, R. (2000). Efficient geometric disassembly of multiple components from an assembly using wave propagation. *Journal of Mechanical Design*, 122:179.

[29] Srinivasan, H. and Gadh, R. (2002). A non-interfering selective disassembly sequence for components with geometric constraints. *IIE Transactions*, 34(4):349–361.

[30] Thomas, U., Barrenscheen, M., and Wahl, F. (2003). Efficient assembly sequence planning using stereographical projections of c-space obstacles. In *Proceedings of the 2003 IEEE International Symposium on Assembly and Task Planning*, pages 96–102, Besancon, France.

[31] Varadhan, G. and Manocha, D. (2006). Accurate minkowski sum approximation of polyhedral models. *Graph. Models*, 68(4):343–355.

[32] Wilson, R. H. (1992). *On geometric assembly planning*. PhD thesis, Stanford University, Stanford, CA, USA. UMI Order No. GAX92-21686.

[33] Wilson, R. H. (1998). Geometric reasoning about assembly tools. *Artificial Intelligence*, 98(1):237–279.

[34] Wilson, R. H. and Latombe, J.-C. (1995). Geometric reasoning about mechanical assembly. In *Proceedings of the workshop on Algorithmic foundations of robotics*, WAFR, pages 203–220, Natick, MA, USA. A. K. Peters, Ltd.

[35] Woo, A., Woo, T., Dutta, D., and of Michigan. Dept. of Mechanical Engineering & Applied Mechanics, U. (1990). *Automatic Disassembly and Total Ordering in Three Dimensions*. University of Michigan, Department of Mechanical Engineering and Applied Mechanics.