Advanced Solutions in Object-Oriented Mechatronic Simulation

Ph.D. Thesis

Juhász, Tamás

Supervisor:

Dr. László Vajta, associate professor
Dept. of Control Engineering and Information Technology

Budapest, 2008
Declarations

I – Tamás Juhász – declare hereby that this thesis has been created only by me and all parts that are imported or rewritten with the same content from others’ work, are unambiguously marked and their sources are given.

I also declare that this thesis contains no material which has been accepted for the award of any other degree or diploma in any university.

Budapest, 29th August 2008
# Table of Contents

**CHAPTER 1**  
INTRODUCTION  

1.1 Motivation  
1.2 Extended contents  

**CHAPTER 2**  
FROM A CAD CONSTRUCTION TO MODELICA MODELS  

2.1 Background  
2.1.1 Object-Oriented Mathematical Modelling  

2.2 Related work  
2.2.1 Virtual prototyping solutions  
2.2.2 Existing CAD model translators  

2.3 Chosen methods and tools  
2.3.1 Source: Pro/Engineer CAD system  
2.3.2 Target: Modelica language and Dymola simulator  

2.4 The basic steps  

2.5 The Physical Modelling XML file  

2.6 Building an internal model using the XML description  
2.6.1 The initial parsing step  
2.6.2 The scene graph as the core data structure  
2.6.3 Adding objects to the scene graph  
2.6.4 Adding joints to the scene graph  

2.7 Importing VRML geometry  
2.7.1 Problem of VRML – PMXML assignment  
2.7.2 Matching the VRML hierarchy with the scene graph  

2.8 Emphasizing other domains  
2.8.1 A new multi-domain object library  
2.8.2 Actuated joints in RobotMax  

2.9 Creating a test motion  
2.9.1 Editing a keyframe  
2.9.2 Processing of the motion  
2.9.3 Introducing a complex example
2.10 Creating new Modelica components 27
   2.10.1 The standard MultiBody library in Modelica 27
   2.10.2 My Actors package 28
   2.10.3 My Joints package 30
   2.10.4 My Motors package 31

2.11 Exporting to Modelica models 32
   2.11.1 Generating the pure mechanical model 32
   2.11.2 Generating the mechatronic model separately 39

2.12 Conclusion 39

CHAPTER 3 CONTACT PROCESSING IN MODELICA MODELS 41

3.1 Introduction 41

3.2 Background 42

3.3 The main tasks of contact processing 42

3.4 My new models 43
   3.4.1 A self-explaining example 44

3.5 Collision detection 45
   3.5.1 Creating a Modelica-C wrapper for the task 46

3.6 Collision response in general 47

3.7 Impulse-based method 48

3.8 Force-based method 49
   3.8.1 First assumptions 50
   3.8.2 Determining contact points 51
   3.8.3 Determining normal direction 51
   3.8.4 Penalty forces: introducing a temporary spring 52
   3.8.5 Constraints on the contact force 52
   3.8.6 Contact force based on linear spring model 53
   3.8.7 Contact force depending on penetration depth 54
   3.8.8 Contact force based on spring and damper model 55
   3.8.9 Determining tangential direction 57
   3.8.10 My friction spring model 57

3.9 A case study 59
   3.9.1 Using the presented workflow 60
   3.9.2 The control algorithm for the robot 61
   3.9.3 An analysed real world motion 64
   3.9.4 Comparison with the simulation 64
List of Theses

Thesis 1.................................................................................................................................................. 4
Thesis 2.1................................................................................................................................................. 44
Thesis 2.2............................................................................................................................................... 58
Thesis 3.................................................................................................................................................. 67

List of Tables

Table 1: Basic steps of the translation process .................................................................................. 12
Table 2: RobotMax → Modelica mapping ......................................................................................... 35
Table 3: Labelling and adjacency matrix of the GX parameter graph ........................................ 83
Table 4: Design of experiments ........................................................................................................ 84
Table 5: Intrinsic depth perception levels ......................................................................................... 87

List of Figures

Figure 1: UML class-diagram of PMXML information classes ....................................................... 14
Figure 2: UML class-diagram of my scene-graph classes ............................................................... 15
Figure 3: A weld joint .......................................................................................................................... 17
Figure 4: A serial joint-set ................................................................................................................ 18
Figure 5: A construction with its scene graph and virtual edges .................................................. 18
Figure 6: A single entity with multiple shapes in RobotMax.......................... 19
Figure 7: XML schema of my Object Library........................................... 22
Figure 8: The schematic diagram of a DC motor with permanent magnet.... 23
Figure 9: The Motor Library browser of RobotMax................................. 24
Figure 10: Symbols of various joints in RobotMax .................................. 24
Figure 11: Motion editing inside RobotMax .............................................. 26
Figure 12: My Actors package ................................................................. 28
Figure 13: Shape model........................................................................ 28
Figure 14: Collider model...................................................................... 29
Figure 15: My Joints package................................................................. 30
Figure 16: The Revolute and Prismatic models ..................................... 30
Figure 17: The JointDrive model............................................................ 31
Figure 18: A kinematic loop and a cut-joint in RobotMax...................... 33
Figure 19: Mechanical model of the double cardan-axle in Dymola ...... 38
Figure 20: Inspecting forces and torques in Dymola.............................. 39
Figure 21: Scheme of contact processing during dynamic simulation .... 42
Figure 22: Side view of a collision scenario ......................................... 44
Figure 23: Schematic view of the previous example............................. 44
Figure 24: Contact modelling with penalty-forces............................... 49
Figure 25: Determining the normal direction..................................... 51
Figure 26: The restitution factor function............................................ 56
Figure 27: Spring model for static and dynamic friction......................... 57
Figure 28: The construction of Anton.................................................. 59
Figure 29: Virtual Anton in the editor of RobotMax............................... 60
Figure 30: Control structure of Anton................................................... 61
Figure 31: Simulated vs. real contact forces....................................... 65
Figure 32: Linear perspective.............................................................. 69
Figure 33: Size consistency.................................................................. 69
Figure 34: Overlapping shapes........................................................... 69
Figure 35: Distant moisture and aerial perspective............................... 70
Figure 36: Depth effect of defocusing.................................................. 70
Figure 37: Effect of shading................................................................. 70
Figure 38: Disparity in binocular vision............................................... 71
Figure 39: Horopter and the borders of binocular image fusion............ 71
Figure 40: Anaglyph spectacles............................................................ 73
Figure 41: LCS glasses.......................................................................... 74
Figure 42: The case of convergent cameras (viewed from above)........ 76
Figure 43: The test room for the experiment.......................................... 80
Figure 44: The measurement model..................................................... 81
Figure 45: The GX graph in constellation parameter space.................. 83
Figure 46: Distribution of absolute, subjective answers....................... 85
Figure 47: Answers for static and dynamic scenes............................... 87
Acknowledgement

Hereby I would like to express my sincere gratitude and appreciation to my director of studies and supervisor, Dr. László Vajta, for his expert guidance and mentorship, his corrective suggestions for this dissertation and his encouragement and continuous support at all levels since my undergraduate years and during my PhD studies.

I would like to thank my colleagues at the Department of Control Engineering and Informatics: Dr. István Loványi, for generously providing the advanced 3D visualization hardwares and Dr. Ferenc Vajda and Tamás Urbancsek for their kind assistance in organizing and conducting the psychophysical experiment and nonetheless for Miklós Vogel, who was my adviser for the master’s thesis in informatics for his motivations on the direction of early research on mobile robotic simulator systems.

Moreover I am very thankful for my Marie Curie programme-supervisor, Prof. Ulrich Schmucker that he offered me the opportunity to gather great experience on the research field of multi-domain modelling and simulation as a fellow in the Early-Stage Research Training mobility programme of the European Commission at the Virtual Engineering Expert Group of the Fraunhofer Institute for Factory Operation and -Automation in Magdeburg.

Finally, I would like to thank my family for their life-long love and support of my research career. Without them, this work could not have been completed.
Chapter 1

Introduction

1.1 Motivation

Virtual engineering offers a completely new aspect of product development, as thereby all sections of the product life cycle can be independently analyzed and in parallel continuously optimized in the virtual world. Customer's requests can be purposefully included into the requirement specification of the product by means of interactive virtual models. Component and system integration on the basis of the virtual reality (VR) and simulation make the practical verification of desired product characteristics possible in very early development stages, thus reducing development risk significantly. This is very important, because 70 to 80 percent of the final costs are already determined in the development phase of a new product.

Additionally the parallelism of different development domains (e.g. mechanical engineering, electrical engineering and control systems) leads to a synergic outcome: a substantial acceleration of the development procedure. On the basis of shared and/or compatible databases verifications of solutions can be accomplished in the development phases, without even manufacturing a physical prototype for early testing.

A substantial driver of the improvement of virtual techniques in the product development is the automobile industry. The obligation to even shorter development times, connected with drastically increased complexity of the components being used and their compound interaction lead to the fact that already in an early stage of the product development the modules have to be previously verified and optimized by various VR techniques [9]. These methods make the complex relations clearly perceivable, letting the engineers detect design errors earlier. Distant goal is thereby the "digital vehicle" as complete virtual representation of the appearance and physical behaviour of a real vehicle.

This directive leads inevitably to a challenge for the small and medium sized enterprises (SME) in the supply chain: they must develop their systems also in such a way, where they have both the geometry and the virtual functional model of each product at their disposal, much earlier than the real manufacturing would begin.

The parameters of a virtual product can be optimized based on a functional model and its dynamic simulation. Building a multibody system (MBS) out of the existing CAD information is a significant task of the virtual engineering workflow. Usually a designer or construction engineer has to ask help from a simulation expert if he might want to inspect the behaviour of the given virtual
product by utilizing a dynamic simulation of that. The designers and construction engineers of the SME aren’t fully aware of the know-how of modelling and simulation of complex mechatronic behaviour. However the lack of tools that are tailored to the relevant smaller company’s wishes (function range, operability, price) targeted for multi-domain modelling and simulation only on the basis of the established development environments (e.g. construction CAD), is even bigger.

In order to describe and simulate a functional model of a system, a big number of descriptive languages and simulation-environments exist already. Such a simulation model has to be both syntactically and semantically correct, which requires extreme attention if it is created manually. As the development process is usually iterative (e.g.: changes in the requirements, varying some parameters during optimization) the continuous tuning of the model multiplies the risk that the human modeller can fail. This explains that there is a large demand for a computerized solution for the translation from a construction model to a multibody simulation model. With this help all the relevant physical parameters of a construction can be embedded largely automated in the simulation.

In majority of the simulation cases the contacts between moving mechanical parts and / or the environment has to be detected and their physical effect must be taken into account. The simulation of industrial manipulators or mobile robots lays stress on the performance and accuracy of this contact processing task.

One practical application of mechatronic simulation is the planning of a teleoperation mission by means of virtual reality techniques (e.g.: for training purposes). In such cases the proper visual feedback is very important for the human operator. During the real-time visualization of a mechatronic simulation it can be taken into account that the relative motion and the changing occlusions in the sight have already a strong depth clue, which can assist the operator to navigate in the virtual reality quite well. The binocular techniques using auxiliary devices for stereoscopic visualization can also help, but these have usually more disadvantages: wearing extra glasses can be inconvenient and the simulator-sickness (because of the eyes’ lack of ability to focus on any virtual object) can occur already after 10-15 minutes of usage.
1.2 Extended contents

A complete virtual engineering workflow embraces multi-domain modelling and simulation including the detailed, adequate visualization of the results: during my work I focused on these research areas.

In Chapter 2 I specify a highly automated translation process that helps a designer to generate multi-domain simulation models from a product’s CAD data. Based upon fundamental graph-theory cogitation, I discuss the algorithm behind a structural conversion to and from a tree structure. In my work this intermediate tree structure plays the role of the main object hierarchy, which is also the key element of the visualization, the subject related to Chapter 4, too.

Using collision shapes extracted from the CAD geometry data I could manage to add collision processing support to a modern, object-oriented multibody simulation environment. I discuss my relevant research on collision response methods in Chapter 3. The contact forces and torques are determined using penalty formulation with interim spring and damper elements, and are parameterized by material properties of the contacting surfaces. One other novelty of my approach is the decoupling of multiple possible collision shapes from the rigid body model.

Chapter 4 guides the reader to the subject of visualization. Among the application areas of mechatronic simulation the virtual reality-based training of teleoperation missions is here emphasized. The simulation being interactively controlled by a human operator needs to present also adequate visual feedback in order to support navigation in the virtual environment. In this chapter of my thesis I present my work on researching on the importance of dynamic motion among the depth clues of visualization. A monocular visualization is relatively easy to handle, while it needs much less resource in comparison to binocular stereoscopic visual methods. By teleoperation the amount of information that can be transmitted between the remote system and the operator is usually limited, and in this manner “cheaper” solutions are demanded. I recognized that the psychophysical depth perception level caused by the presented motion-parallax effect is relatively big, though in the literature I did find any quantitative comparisons with the binocular techniques (using auxiliary spectacles) about the importance of the role of dynamic motion. Focusing on the stereoscopic techniques available at our department (polarizer-, liquid crystal shutter- and anaglyph glasses) I designed a psychophysical experiment that measured the quantitative relevance of motion in numerous dynamic 3D scenes.

All my presented results have been integrated in the development of my integrated mechatronic model authoring and visualization tool, which was already used in multiple projects to create, use and present mechatronic simulation models effectively.
Chapter 2

From a CAD Construction to Modelica Models

If you want to analyse the real-time behaviour of macroscopic multibody systems, simulating the dynamics of perfectly rigid bodies (by neglecting elasticity) is usually an allowable approximation. Macroscopic articulated multibody structures are nowadays designed with the help of interactive Computer Aided Design (CAD) applications. Currently the most, widely spread commercial CAD systems do not offer multi-domain simulation support, which would allow evaluating a new product’s dynamic behaviour based on a functional model at an early development stage.

The gap between designers and simulation experts issues in a cumbersome manual work during constructing a parametric simulation model of the complex product that is being designed in a CAD environment rather independently. In machine production a family of component parts with varying parameters has to be also designed repeatedly. Furthermore the product development has an iterative nature by itself: some internal parameters must be fine-tuned, according to model assessment or verification processes. These parameter changes would require tremendous manual recalculations if there was no computerized support for creating a simulation model.

In this chapter it is presented that using my mechatronic model authoring and visualization application, a smooth workflow can be achieved from the widely-spread Pro/Engineer construction CAD system to the multi-domain simulation world of Dymola.

Thesis 1 [1][2][3]

a) I have created a new, automated method that translates a CAD construction plan into a consistent mechanical model in Modelica language. During this method I build a special, hierarchical graph structure: on the one hand it eases handling kinematical loops, and on the other hand it can also be used later in the effective visualization of the hierarchical model.

b) In my extended method active components can be chosen from a multi-domain model library interactively, and these parametric models can be assigned to the nodes of my mechanical model’s hierarchical graph. This extended method can serve the complete functional mechatronic model of the CAD construction. □
2.1 Background

CAD models are hierarchical structures built from assemblies and parts. The outermost assembly is the construction itself (later denoted by \( K \)): it may contain multiple parts and other subordinate assemblies.

In order to create a multibody system model (MBS, later denoted by \( M \)), some information existing in the CAD world has to be extracted. For rigid bodies the minimal required physical parameters are the masses, inertia-tensors and locations of gravity centres. If collision processing is also a part of the simulation, the shapes of the CAD parts must be extracted, too. The constraints between parts in the CAD assembly are to be represented by joint types that are supported in the target simulator.

2.1.1 Object-Oriented Mathematical Modelling

Mathematical models used for analysis in scientific computing are inherently complex in the same way as other softwares are. One way to handle this complexity is to use object-oriented techniques. The basic terminology of object-oriented programming is defined in [49]:

- **Objects** are collections of operations that share a state. These operations are often called methods. The state is represented by instance variables, which are accessible only for the operations of the encompassing object.
- **Classes** are templates from which objects can be created.
- **Inheritance** allows us to reuse the operations of a class when defining new classes. A subclass inherits the operations of its parent class and can add new operations and instance variables.

Note that the strict requirement regarding data encapsulation is not fulfilled by object oriented languages like Java or C++, where non-local access to instance variables is allowed. It is more important that the previous definitions are suitable for describing the notions of object-oriented programming, but they are too restrictive for the case of object-oriented modelling: where a class description may consist of a set of equations that implicitly define the behaviour of some class of physical objects or the relationships between objects. Functions should be side-effect free and regarded as mathematical functions rather than operations. Explicit operations on state can be completely absent, but can be present. If a system of such equations is solved symbolically, the equations are transformed into a form where some (state) variables are explicitly defined in terms of other (state) variables. If the solution process is numeric, it will compute new state variables from old variable values, and thus operate on the state variables.
The basic terminology of *object-oriented modelling*:

- An *object* is a collection of variables, equations, functions and other definitions related to a common abstraction and may share a state. Such operations are often called methods. The state is represented by instance variables.
- *Classes* are templates from which objects or subclasses can be created.
- *Inheritance* allows us to reuse the equations, functions and definitions of a class when defining objects and new classes. A subclass inherits the definitions of its parent class and can add new equations, functions, instance variables and other definitions.

The concept of *declarative* programming is inspired by mathematics where it is common to state or declare *what holds*, rather than giving a detailed stepwise algorithm on *how to achieve* the desired goal as is required when using procedural languages (such as Fortran or C). This relieves the programmer from the burden of keeping track of such details. Furthermore, the code becomes more concise and easier to change without introducing errors.

However the causality – i.e. which variables are regarded as input, and which ones are regarded as output – is usually not defined by such an equation-based model. There are usually many possible choices of causality, but one must be selected before a system of equations is solved.

The declarative paradigm introduces *acausal* modelling, as the mathematical equations are always bidirectional. The main advantage is that the solution direction of equations will adapt to the data flow context in which the solution is computed. The concept of *object-oriented mathematical modelling* can be summarized as follows:

- Object-orientation is primarily used as a structuring concept, emphasizing the declarative structure and reuse of mathematical models.
- Dynamic model properties are expressed in a declarative way through mathematical equations.
2.2 Related work

2.2.1 Virtual prototyping solutions

There is a huge amount of modelling and simulation applications on the software market, tailored mostly towards general mechanical virtual prototyping. In this subsection I give only a very brief overview of some of these softwares, focusing on their simulation possibilities [14].

- **VisualNastran 4D** from MSC Working Knowledge provides an integrated environment for motion and FEA (Finite Element Analysis) simulation and complete suite of tools for the development and communication of physics-based mechanical virtual prototypes. Constraints and drivers can be defined by numeric or equation input in the formula editor, or with tabular data.

- **ADAMS** – developed by Mechanical Dynamics Inc. – provides a fully integrated virtual prototyping environment. In addition to the powerful modelling and visualization capabilities includes an analysis engine called ADAMS/Solver, which converts an ADAMS model to equations of motion, and then solves the equations, typically in the time domain. ADAMS/Solver can resolve redundant constraints, handle unlimited degrees of freedom, and perform static equilibrium, kinematic, and dynamic analyses.

- **Dynamic Designer/Motion** and **Simply Motion**, two other products from Mechanical Dynamics Inc., provide a full integration with AutoCAD Mechanical Desktop. Simply Motion written also in AutoDesk’s ARX development language extends the design automation capabilities of Mechanical Desktop to include realistic 3D dynamic motion simulation. Simply Motion anticipates the mechanical designer needs and automatically updates the motion data. Through a browser (called IntelliMotion Browser), the user can add joints, springs and input motion to the mechanical model.

- **DADS** (standing for Dynamic Analysis and Design System, available from LMS International Inc.) performs assembly, kinematic, dynamic, inverse dynamic and preload analysis. It incorporates advanced numerical methods to solve Differential Algebraic Equations (DAE) using both implicit and explicit solvers.

- The **Working Model 3D** tool permits construction of a mechanical model with joints (free and motor-controlled), springs, dampers and ropes. It also has built-in collision detection options. Working Model 3D can import assemblies from SolidWorks. However Working Model 3D is a *closed* mechanical simulation system and user defined control code can be used there in a very limited way.
The disadvantage of these virtual construction prototyping solutions is that they don’t support multi-domain simulation to this very day, and it is complicated – or even impossible – to integrate them with other simulator environments (e.g.: Matlab). The actual solutions are limited to certain types of mechanical systems. Though some packages offer Simulink interface (for co-simulation of control systems), they are not flexible enough for other domains or perform very poorly.

2.2.2 Existing CAD model translators

The SimMechanics™ software of MathWorks extends Simulink® with tools for modelling and simulating mechanical systems [60]. It is integrated with MathWorks’ control design and code generation products, enabling you to design controllers and test them in real time with the model of the mechanical system. MathWorks has also implemented a CAD→XML parameter exporting plug-in for both SolidWorks and Pro/Engineer environments. The exported XML data is used by SimMechanics to generate a physical model that represents the CAD construction. However, the result is still a single-domain, mechanical model, which is hard to extend with non-mechanical components.

At the Programming Environment Laboratory of the Linköping University, Sweden some articles have been published on translating a mechanical CAD construction to Modelica models, using the following source systems:

- AutoDesk’s Mechanical Desktop [14]
- Solidworks [27]

Both of these articles are using the obsolete Modelica v1 notation (it is understandable because the language improved continuously for these years), but their biggest disadvantage is that the authors disregard / don’t emphasize the original kinematical structure of the CAD construction. They don’t focus on detecting special loop structures, which can cause the result model to be erroneous and faulty when simulating.

Nevertheless the suggestion of article [14] to integrate the translation and visualization process inside the CAD environment can be very productive: this is going to be considered in my future work (see section 5.2.2), too.
2.3 Chosen methods and tools

During my research I followed the sequence of literature overviews, problem definition, program-, system- and experiment design, early implementation tests, approval tests, evaluation and case studies comparing real system behaviour.

In the next sections I present the key software tools I have chosen as the source and the target of my CAD model translation process.

2.3.1 Source: Pro/Engineer CAD system

Pro/ENGINEER – or shorter Pro/E – is a professional CAD/CAM/CAE modelling software developed by Parametric Technology Corporation (PTC). Its newer Wildfire versions are used worldwide in the car industry, for example by Audi, Volkswagen, Skoda, Seat, Maserati, Porsche or Toyota.

In Hungary the Pro/E system is popular with both local affiliates of the leading international vehicle manufacturers and Hungarian enterprises as well. Audi Hungaria Motor Ltd., the company with the largest export in Hungary, and the third largest engine manufacturer in the world is one of the largest Hungarian Pro/ENGINEER users [54]. They are planning further investment and will continue to increase their weight in Hungary, as a result of their recently announced joint research and development project with the BUTE.

The original Pro/E was created in 1988 by Dr. Samuel P. Geisberg, with the help of some colleagues: the program supported creating complex 3D models, assemblies, and 2D measured drawings. Later it caused a major change in CAD industry when first released by introducing the concept of Parametric Modelling.

Rather than models being constructed like a mound of clay with pieces being added or removed to make changes, the designer constructs the model as a list of parametric features, which are stored by the program and can be used to change the model by modifying, reordering, or removing them. Besides the geometry and material, each 3D model part has many physical parameters: e.g.: mass, inertia-tensor and location of centre of gravity. On demand the physical parameters of the respective parts – that are dependent on the actual material and geometry input parameters – are recalculated internally.

There is a large amount of commercial and non-commercial applications (e.g.: “3D_Evolution” or “TransMagic”) available on the market offering native conversion between common standard (STEP, IGES) and other well-known (AutoCAD, CATIA, Inventor, Pro/Engineer, SolidWorks, Unigraphics) CAD data formats. My case study about the six-legged mobile robot Anton (which I present in section 3.9, on page 59) also started with 3D_Evolution and the automated conversion of the original SolidWorks construction to Pro/Engineer’s format.
2.3.2 Target: Modelica language and Dymola simulator

Modelica and Dymola have been chosen as my target modelling language and multi-domain simulator. The development and promotion of the free Modelica language is organized by the non-profit Modelica Association [57]. Modelica is a modern language supporting non-causal modelling with bidirectional mathematical equations and object-oriented paradigm to facilitate reusing of knowledge in many modelling domains at the same time. Besides process-oriented and control system components it provides modelling abilities in various engineering domains (including mechanical-, electrical-, hydraulic-, pneumatic- and thermal subsystems).

Modelica offers a general type system that unifies object-orientation, multiple inheritance and templates within a single class (model) construct. The components are based on standardized interface definitions, and can contain formalisms such as ordinary differential equation systems (ODEs), differential algebraic equations (DAEs), state machines or Petri nets.

Modelica programs are built from classes. Like in other object-oriented languages, a class contains variables, i.e. class attributes representing data. The main difference compared with traditional object-oriented languages is that instead of functions (methods) equations are used to specify behaviour. Equations can be written explicitly, like \( a = b \), or be inherited from other classes. Equations can also be specified by the `connect` statement. The statement `connect(v1,v2)` expresses general Kirchhoff coupling between the `connector` variables v1 and v2: the potential values of those are meant to be equal, and the flow values (e.g.: electrical currents) sum to zero for each connected node. This gives a flexible way of specifying topology of physical systems described in an object-oriented way using Modelica.

The Modelica language supports general multi-domain modelling, where arbitrary control algorithms can be specified for the mechatronic models. For a brief overview about the language please refer to the article [16]. The complete reference of the 2.1 version of the language can be found in the book [18].

Dymola – Dynamic Modelling Laboratory – is a commercial simulation environment for Modelica with unique multi-engineering capabilities. Using Dymola it is possible to simulate the dynamic behaviour of mechanical, electrical, thermodynamic, hydraulic, pneumatic, thermal, power and control components described in Modelica language. The flexibility of Dymola depends on the powerful Modelica language and its aforementioned technologies.

I emphasize here the non-causal modelling possibility, with bidirectional data flow between components, as a major advantage of Dymola / Modelica (for example comparing against Matlab / Simulink).
2.4 The basic steps

The translation from a given Pro/Engineer assembly to Modelica description needs several steps. The process itself has to be deterministic, given the same source CAD model it has to produce always the same output. The following sequence represents the workflow:

- I use a plug-in for Pro/Engineer that allows exporting the given CAD assembly to a single (so called "Physical Modelling XML") descriptor file, which represents the original mechanical structure in a tagged XML text [60]. This file includes the original hierarchy of the CAD structure. It represents each constraint between two parts with a set of joints. The global transformation (position and orientation information) and physical parameters (inertia-tensors, masses, etc.) are also stored there for each part, but there is no information about their shapes, at all.

- If you want to model collision between the bodies during the simulation (see Chapter 3 for further information), the shape information of the CAD parts is essential: under Pro/Engineer the geometries can be exported to standard VRML format. In a general case you get one (or more) hierarchy file(s) and the geometries of individual parts in separate .WRL files. Hierarchy files correlate to subordinate CAD assemblies, and can contain transformations and references to subordinate hierarchy- or geometry files, but cannot contain any shape information. The geometry files define the actual shapes always in local coordinate system.

- My own-developed mechatronic authoring and visualization application (called RobotMax) has a core translator logic that can import the aforementioned XML descriptor file, and generates an internal mechanical multibody system (MBS) out of that information first. The VRML geometry is optionally imported and converted to an internal representation for supporting collision detection and visualization, too.

- By default only the gravity force is applied to the mechanical structure: in order to test dynamic behaviour of it, there must be also additional external forces acting on the structure. In RobotMax the translated and pure mechanical MBS model can be extended interactively with various electromechanical elements (e.g.: parametrical motors can be chosen from a model library, and can be added to the passive joints to make them actuated) to form a more complex mechatronic (i.e. multi-domain) model that can interact with its environment in an active way during the simulation.

- A reference motion that is going to be used during the dynamic simulation can also be designed by the user of RobotMax for each active joint. Either forward or inverse kinematics stays at the user’s disposal for this step.
Finally the internal mechatronic model of RobotMax – including motion information – is exported to Modelica models using the built-in conversion module, and can be simulated in Dymola. Dymola supports basic 3D visualization using .DXF triangular mesh shapes: these are optionally generated during the export process, too.

The next table summarizes the main steps of the workflow in a clear way:

Table 1: Basic steps of the translation process

<table>
<thead>
<tr>
<th>Step</th>
<th>Tool</th>
<th>Commercial</th>
<th>Input</th>
<th>Output</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAD System</td>
<td>yes</td>
<td>User</td>
<td>CAD</td>
<td>construction</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Pro/Engineer</td>
<td>yes</td>
<td>CAD</td>
<td>VRML Files</td>
<td>geometry</td>
</tr>
<tr>
<td>2</td>
<td>Pro/E-to-SM</td>
<td>free</td>
<td>CAD</td>
<td>XML File</td>
<td>MBS</td>
</tr>
<tr>
<td>3</td>
<td>RobotMax</td>
<td>own</td>
<td>VRML</td>
<td>XML</td>
<td>DXF Files</td>
</tr>
<tr>
<td>4</td>
<td>RobotMax</td>
<td>own</td>
<td>User</td>
<td>Modelica</td>
<td>geometry + MBS</td>
</tr>
<tr>
<td>5</td>
<td>RobotMax</td>
<td>own</td>
<td>User</td>
<td>Modelica</td>
<td>electro-mechanic</td>
</tr>
<tr>
<td>6</td>
<td>Dymola</td>
<td>yes</td>
<td>Modelica</td>
<td>DXF</td>
<td>SIMULATION</td>
</tr>
</tbody>
</table>

2.5 The Physical Modelling XML file

A physical modelling (PMXML) descriptor file of a K construction gives the CAD information that is required to generate a multibody system. A PMXML file is composed from a tree-hierarchy of Subsystems. The Subsystem-hierarchy reflects the original hierarchy of the CAD assembly and its sub-assemblies, thus it has always a tree structure. Each subsystem contains the following sets and entities:

- a set of the rigid bodies (*denoted by T*)
- a set of coordinate systems (*or frames, denoted by FR*)
- a set of primitive joints (*denoted by J*)
- a RootPart object (*optional, also member of T*)
- a set of contained subsystems

The unique RootGround entity in the top-level subsystem is also a member of T: it is a virtual body that has a fixed pose in the World inertial system, and can be treated as a fixed body with infinite mass.

A RootPart is optional in each subsystem. It is a zero-dimensional body, without mass or any associated physical information. It can carry various coordinate-system frames, just like the other, normal bodies in T. It is usually “welded” together with other rigid bodies (*welding* represents a fixed, 0-dof transformation constraint between those parts).
Each Frame in the FR set is a coordinate system that is identified with an integer ID, and belongs to a single Body object in T. A unique one-way association exists in the direction FR→T. A Frame defines the initial pose of its origin in the Cartesian space, including a position vector and an orientation matrix.

Each subsystem enumerates the sequence of bodies that belong to it. The rigid body elements in T must have one local origin Frame (CS1), one centre of gravity Frame (CG) and can have multiple additional Frames associated (CS2, CS3, ...), optionally. The physical parameters (masses, inertias, surface areas, volumes, etc.) are also stored for all bodies in the given subsystem.

The constraint relations between pairs of mechanical parts are represented by SimpleJoint entries in the PMXML file. Such an entry has the following form:

\[
FR_B – J_1, J_2, ..., J_n – FR_F
\]

\[
FR_B, FR_F \in FR; B, F \in T; J_i \in \{W, P, R, S\}
\]

("B" stands for Base body and "F" for Follower body, \(J_i\) is a primitive joint – see later – from the \(J\) set). It is not absolutely necessary that the referred bodies \(B\) and \(F\) are in the same subsystem, as in Pro/E there can be constraints between parts in different sub-assemblies, as well.

Each SimpleJoint entry is representing a constraint between two frames (using their integer IDs) belonging to two bodies, having \(FR_F\) and \(FR_B\) frames, removing up to 6 degrees of freedom between those. Note that there can be multiple primitive joints between two bodies (in a serial connection).

A primitive joint can be either weld (yields zero degree-of-freedom), prismatic (1-dof, translational), revolute (also 1-dof, rotational) or spherical (3-dof, point-to-point): \(W, P, R\) or \(S\) type, respectively. All other possible joint types can be represented by connected joint-sets. For example, a 2-dof cylinder-joint is decomposed to the series of one revolute and one prismatic joint.

### 2.6 Building an internal model using the XML description

In this section I present that out of the physical modelling XML information (or shorter: PMXML) the F graph, an internal tree structure is built in RobotMax, which is a new, .NET-based mechatronic model authoring and visualization software tool I have developed in the recent years. It has been interposed into the design workflow to finally achieve the conversion to Modelica models from the output of the Pro/E CAD system highly automated, and is representing an internal multi-domain mechatronic model.

As it happens, a tree structure is being used also in standard VRML to represent the scene graph, because it allows a convenient way to describe relative transformations and relations, and it adds up to a great performing visualization, using the right graphics APIs.
2.6.1 The initial parsing step

The PMXML data structure can be parsed into a bunch of C# classes I have created. The class diagram can be seen on the next figure:

Figure 1: UML class-diagram of PMXML information classes

The original structure of the PMXML file is transformed first to the hierarchy of the following XMLNode objects:

- Each XMLSubsystem object can have an optional reference to its RootPart as an XmlBody. It can also have child XMLSubsystem entries, thus forming a tree hierarchy of the original PMXML Subsystems.

- An XMLFrame instance stores the pose of a frame entry in the PMXML FR set, and has a globally unique integer ID.

- The XMLJoint class is related to the PMXML SimpleJoint entries. It has the so called Base and Follower frames’ reference IDs and a collection of XMLPrimitive primitive joint objects.

- The XMLPrimitive class is encapsulating a primitive joint in the set J of the PMXML file, and stores axis information and the type of the primitive joint.

- The XMLGround entities represent either the global RootGround entity or a Body object of the PMXML file, and contain one XMLFrame reference, as the CS1 local origin frame. They have information which XMLSubsystem they belong to.

- The XMLBody is an XMLGround, and stores also the physical data of a rigid body (mass, inertia and the centre of gravity “CG” XMLFrame reference). It can store optionally a collection of XMLFrame references.
2.6.2 The scene graph as the core data structure

The following figure shows an extract of my C# classes related to the F scene-graph in RobotMax:

![UML class-diagram of my scene-graph classes](image)

Figure 2: UML class-diagram of my scene-graph classes

- My scene graph has a tree structure (connected and acyclic graph), built of `SceneNode` objects as **vertices**. Each `SceneNode` instance stores a relative transformation to its **single** parent `SceneNode`, and can have unlimited number of `SceneNodes` as children. The **edges** of the scene graph are always between a parent and a child `SceneNode`. Each `SceneNode` stores a collection of one or more `SceneObject` references: those are the objects being actually transformed from local into the Cartesian world space by the actual derived transformation of that `SceneNode`.

- A `SceneObject` is an abstract movable object in 3D space, whose transformation is stored in its associated `SceneNode`. Only their `Renderable` variant may contain visual information.

- A `CBody` instance (that is also a `Renderable`) represents a rigid body in the internal multibody system. It stores the physical information of the given body: mass, inertia, etc. It does not have a physical shape, and is represented optionally with a dummy geometry (a wireframe cube) for visualization.

- An `Entity` (which is also a `CBody`) object may have multiple associated `CShape` geometry objects (with VRML geometry converted to boundary polygonal surface representation and material information), thus it can be rendered by the 3D engine. If VRML geometry (see section 2.7) is also being imported (which is a must, when collision processing is taken into account – see Chapter 3), the dummy geometry is replaced immediately, after the `Entity` gets at least one `CShape` instance associated.

- The `CJoint` base class stores information of a joint in the scene. Only the basic prismatic, revolute and spherical joints are supported in RobotMax. Other, higher DOF joint types can be formulated using a serial composition of those (e.g.: a cardan joint = 2 revolute joints with perpendicular axes). Each `CJoint`
instance has an axis vector (defined in local space), optional lower and upper limits of allowed motion and an optional reference to a CMotor instance as the associated drive model (see section 0).

2.6.3 Adding objects to the scene graph

After the whole PMXML file has been parsed into intermediate C# objects in the memory (see section 2.6.1), the internal scene graph of RobotMax can receive the new objects (i.e.: new vertices will be added).

First the single RootGround object – as a new CBody instance – is added to a new child SceneNode, directly under the unyielding root SceneNode in the graph. The origin frame pose of the XmlGround instance is used to setup the new SceneNode’s transformation: the initial position and orientation are defined in this way. This new CBody instance has also a static indicator flag enabled, thus it becomes a fixed, static body in the scene.

A very similar task has to be done for all XmlBody entries. In case of a normal Body (or a RootPart) in the T set, a new Entity (or a new CBody) instance has to be added via a new child SceneNode (with initial pose already set) under the root SceneNode. At this point all bodies are already added to the scene graph as top-level nodes, but there isn’t any constraint (edge) between them, thus – excluding the RootGround – all of them have six degrees of freedom in the Cartesian space.

2.6.4 Adding joints to the scene graph

The (1) relations represented by XMLJoint instances are interpreted as the following sequence:

\[ \text{Follower} \rightarrow J_1 \rightarrow \ldots \rightarrow J_N \rightarrow \text{Base} \]  

(2)

where \( a \rightarrow b \) indicates an edge in the tree, showing that ‘\( a \)’ is the child of ‘\( b \)’ in the scene hierarchy.

Because it is not forbidden to have kinematic loops in the original CAD structure, the first, ‘\( \rightarrow \)’ edge is only conditional: seeing that creating a cycle in the scene tree graph must be prevented! In order not to lose any structural information, this edge will be kept virtual in \( F \), and along that the cycle will be cut to two branches in the tree. \( J_1 \) is a so called cut-joint in this case.

Such a sequence – with \( N \) primitive joints between Base and Follower – can represent new vertices and edges in the scene graph. The vertices for Base and Follower objects have already been added to the graph (in section 2.6.3): just the \( N \) new \( J_i \) joint vertices and the \((N+1)\) new \( a \rightarrow b \) edges are possible missing. Two branches of processing can follow, distinguishing between the types of the given \( J_i \) primitive joints.
Welding constraints

A “weld” joint means that the Follower and Base bodies are basically glued together, forming a single rigid body. Accordingly there can be only a single $J_1$ weld primitive ($N=1$) between two different bodies, as this one alone removes all the possible six relative degrees of freedom. Note that one of the bodies is usually a RootPart or the RootGround. In a welding case the Follower body’s SceneNode is simply relocated to be a child of the SceneNode of the Base body, resulting only in a new edge in the F tree:

![Diagram of a weld joint]

Real joint-sets

There are $N$ pieces of real joints involved in this other processing case. The new edges building the tree structure of the scene graph are added one at a time, in the following order: $J_N \rightarrow Base$, $J_{N-1} \rightarrow J_N$, $\ldots$, $J_1 \rightarrow J_2$ and finally (only optionally!) the Follower $\rightarrow J_1$.

In step ($i=1..N$) the $J_{N+1}$ primitive joint have to be added to the scene graph. The respective $C\text{Joint}$- (either -Prismatic, -Revolute or -Spherical) class is instantiated and added to a new SceneNode vertex in the graph, which is inserted as child under the SceneNode-$i$. The very first parent SceneNode is the vertex holding the Base body. As the new SceneNode becomes the parent in the next step, it is sure that these new vertices form a straight path in the graph, without causing a cycle:

$$\text{SceneNode}_N \rightarrow \ldots \rightarrow \text{SceneNode}_1 \rightarrow \text{SceneNode}_0$$

After the previous $N$ steps there are $N$ new vertices and $N$ new edges in a chain in the scene graph. The joint $J_1$ sits on the leaf SceneNode vertex.
However the last step with the edge $Follower \xrightarrow{\cdot \cdot \cdot} J_1$ needs some attention, because it might cause a cycle in $F$. First I have to check whether it is violating the acyclic property of the $F$ scene graph: if the answer is no, then the $Follower$ body’s $SceneNode$ is relocated to be the child of the $SceneNode_N$ (which is holding $J_1$). Otherwise the $SceneNode_N$ remains the leaf node, and $J_1$ joint puts a marker on the $Follower$ body by storing the $SceneNode$ reference as its “second parent”: this is the virtual edge I mentioned before, which preserves the original structural information (the $F$ scene graph is a special tree).

This has a practical advantage: $J_1$ with the second “virtual parent” can simply be a cut-joint (see also section 2.11), which is found here as a side effect. The next figure shows an example construction that has multiple kinematic loops:

This example construction has two marked cut-joints (“$D—C$” and “$F—G$”) that both introduce a virtual edge for the $F$ scene graph.
2.7 Importing VRML geometry

After the information of the PMXML file has been converted into the scene graph, the shapes of the (so far point-mass, dummy-shaped) Entities still need to be loaded for two main reasons:

- they can serve as collision shapes in the multibody simulation
- they are used for visualization

2.7.1 Problem of VRML – PMXML assignment

It is important to know which geometries form together a single rigid body, when consistent collision handling is also a goal during the simulation (see Chapter 3). For the present the exporting of PMXML and VRML files under Pro/Engineer is done completely independently. Unfortunately in the former file there is no direct reference to any VRML information. Later I will override this by creating a new plug-in for Pro/E (using the Pro/Toolkit API [58]).

The partially auto-generated names inside these source files (e.g.: “Obj01”, “Obj02” vs. “Obj”, “Obj-1”) are neither globally unique nor match each other. In order to find the correspondence between VRML and XML domains, a sophisticated procedure has to be followed.

It can also happen that even an entire subordinate CAD assembly is represented by a single rigid body. This is usually the case, when there is no degree of freedom left inside that assembly (i.e.: the assembly is rigid). In such cases there are less rigid bodies in the scene than actual VRML geometries, as the VRML hierarchy is deeper than that one in the PMXML file. The next figure shows a simplified Mitsubishi robot assembly, imported into RobotMax:

![Figure 6: A single entity with multiple shapes in RobotMax](image)

This robot model has six segments and 5 joints. Every segment of the robot is rigid, thus it transforms to a single Entity in RobotMax. However in Pro/Engineer the second segment (marked with a thin red wireframe on the
middle) is a rigid assembly of 3 parts, which is exported to three VRML geometry files (on the right hand side I show that it is built from 3 parts)! In the PMXML description such a rigid assembly is handled as a single body. According to my process described below in the next section, the Entity of the second segment will receive 3 CShape (VRML) references.

2.7.2 Matching the VRML hierarchy with the scene graph

All VRML geometry nodes have a relative, homogenous transformation matrix (which can arise derived from their respective parents, recursively), from which one can retrieve their global pose (position and orientation) in the 3D world coordinate system. This derived 4x4 matrix is also used to transform the local vertices of a given VRML shape into the global (World) coordinate system during rendering, too. Fortunately pose information (arisen from the CS1 frames originated from the PMXML file) is included also in each new SceneNode vertex in the F scene graph of RobotMax, holding the transformation of its contained Entity object. I have a C collection of such new SceneNodes that have been added to the scene graph in section 2.6.3.

As I load the VRML scene node-by-node, I can track the actual homogenous transformation matrix in Cartesian space. There is always an actual E Entity held by a SceneNode in C, to which I assign the actual VRML geometry as a new CShape object: if this 4x4 matrix changes, I have to search for a possibly new E Entity.

Filtering the candidates first by position

First I check the elements of the C collection, whether there is a single SceneNode entry that has exactly the wanted position (within a fair tolerance, of course). If I find a single positive match, that one has the new E Entity. Without any matching in the C set, the E remains unchanged, of course. The fact of no matching SceneNode is to be found in C indicates that the VRML hierarchy is deeper than the XML hierarchy, thus the old E Entity will receive all the subsequent VRML geometry information.

Filtering the candidates by orientation

If there were more candidate SceneNodes having the actual derived position, I continue filtering those by their actual derived orientation. At this point I can get one or more positive matches. In the former case the single match has the new E Entity.
Filtering the candidates by names

Finally, in very extraordinary cases there can still be more than one entry in C having the same pose in the 3D space. As I mentioned before, the VRML export module of Pro/E includes also the names of CAD parts in “Model_Info” prototype VRML nodes. In the final search I compare the prefixes of these model names with the prefixes of the remaining candidate SceneNodes’ names (each SceneNode is named after the XmlBody entry that was used to create it). For example “Obj01” can match with “Obj” or “Obj-1”, thus giving possibly a new Entity to store the new VRML geometry into.

It is hardly imaginable that there are more parts in the CAD assembly with exactly the same pose and name. This would indicate that there is an error in the source CAD plan.

2.8 Emphasizing other domains

After the CAD-import steps there is only a pure mechanical, passive structure in RobotMax. As I mentioned before, the goal of the simulations is to verify the dynamic behaviour of the structure while external, varying forces also act on it (besides the uniform gravity, of course).

The passive mechanical structure must be equipped with energy-sources that allow carrying out an active, controlled motion. The source of such energy can come from other engineering domains, such as electrics, electronics, pneumatics or hydraulics. I emphasized the electrical machines as primary mechanical energy sources in my work.

2.8.1 A new multi-domain object library

I have designed a new XML-based library that can contain folders of multiple engineering domains. The XML schema in text can be found in the appendix. My library has a hierarchical structure, on the top level there are the folders.
**Figure 7: XML schema of my Object Library**

**Folders:**

In the library each *object_folder* is a container of *object_class* entries in a given engineering domain. The *Electromechanical* folder is emphasized here as the container of the most commonly used actuator elements in my mechatronic models so far.

**Classes:**

A class encapsulates the abstract information of a given functional model. The actual objects that are from a class are also named the *instances* of a class.

Inside an *object_folder* each *object_class* entry can contain a path for a Modelica implementation. A class also declares all *parameters* that the instances can use. The class parameters must have a unique name, a valid data-type and a Boolean flag, called *editable*: if this refers to true, the given parameter’s value remains changeable also in the actual instance that uses it. Optionally a class parameter can have a physical unit, an explanatory / commentary string and lower / upper bounds.

An example is the "*Permanent magnet DC machine*" class, which has the Modelica implementation under *Motors.DCMachines.DC_PermanentMagnet* (see section 2.10.4). Figure 8 shows the model’s schematic diagram, which is generated from its graphical annotation in Dymola:
Figure 8: The schematic diagram of a DC motor with permanent magnet

**Objects:**

The actual instances of a model class are the model objects. The object entries are enumerated inside their declaring object_class entry. Each object has a 32 digits-long global unique ID that is used in a client application (which uses the Object Library) for referencing purposes. Each object must define the actual values of the class parameters it belongs to.

An entry of a concrete permanent magnet DC motor (KRS-784 ICS digital servo motor, product of Kondo Kagaku Co. Ltd.) looks like:

```xml
<object name="KRS-784ICS" ID="628F99AAAA7444D5A2A36A6FA1175594">
    <parameter name="J_Rotor" value="1.2E-7" />
    <parameter name="VaNominal" value="8" />
    <parameter name="IaNominal" value="1.2" />
    <parameter name="rpmNominal" value="8200" />
    <parameter name="Ra" value="0.78" />
    <parameter name="La" value="2.1E-5" />
    <parameter name="Ratio" value="251" />
</object>
```

Note that all the parameters’s physical units are defined in the model class, not in a given instance.
2.8.2 Actuated joints in RobotMax

All joints imported from the CAD structure (over the PMXML description, see section 2.6.4 on page 16) are passive and have no bounds of the allowed motion range, by default. If a prismatic or a revolute joint is selected, you can use the Motor Library browser of RobotMax (see Figure 9) to manually assign a motor to the selected joints to make them actuated:

![Motor Library browser of RobotMax](image)

Figure 9: The Motor Library browser of RobotMax

Unless a motion is defined (see next section), each actuated joint has a fixed target value during the simulation. Unless otherwise initialized, the default target is 0 (in [m] or [°] units: for prismatic or revolute type, respectively).

The following figure shows the types of joints supported in RobotMax:

![Symbols of various joints in RobotMax](image)

Figure 10: Symbols of various joints in RobotMax

The upper row of Figure 10 shows the passive joint types of RobotMax (using brighter colours: a prismatic, revolute and spherical one, respectively). Only the prismatic and revolute joints can be actuated: their symbols are shown in the bottom row of the figure (note the difference of the colours and stronger contrast) and they are easily distinguishable from the passive ones.
2.9 Creating a test motion

In RobotMax there is a possibility to define a keyframe-based reference motion, whereby the joint reference signals are calculated as the function of time, depending on so called keyframes. The keyframe has a time stamp $T_i$, and it stores a snapshot of the posture of the actual structure at that time.

There are two basic kinds of motion planning support in my tool: you can rely either on forward or inverse kinematics. This allows you to create complex dynamic test scenarios and – using the final simulation – inspect the behaviour of the mechanism under the load caused by the dynamic effects 0.

2.9.1 Editing a keyframe

I designed two methods in RobotMax for setting up a posture of the actual assembly in a given keyframe:

- You can pick up one of more joints and adjust their angles directly, either interactively (with the mouse) or manually (using the keyboard).
- You can select a single part – as the end-effector – and using the interactive (mouse) or manual (keyboard) way you can set up the demanded pose of that in Cartesian space, while the dependent other parts (i.e. the joints’ values on the chain towards the root element) are updated automatically. This is the well-known inverse kinematics method.

2.9.2 Processing of the motion

The calculation of the actuated joints’ continuous reference signals is based on these keyframes. The continuous segments of the signals are generated in respect of the actual and the previous keyframes’ data. This process depends on the relation between the actual keyframe and its predecessor: it can be one of the following two types:

- Pure forward logic-based (F-type)
- Inverse-kinematics based (I-type)

In the former case a section of the reference signal is generated via interpolation between the actual (F-type) and the last keyframe’s stored joint values. The interpolation is done in joint space. Currently only first order (linear) interpolation is implemented in RobotMax.

An inverse kinematics-based keyframe (I-type) must be used if a part has to undergo a transformation in the Cartesian space during the $[T_{i+1}, T_i]$ interval. For example following a linear path with a manipulator tool requires this technique. Such I-type keyframes can contain multiple target objects which are processed parallel: this is useful for multiple independent, cooperating manipulators (e.g.:
two robot arms mounted on the same platform, like fingers of a hand). The joints that are not involved in any IK chain will automatically use the joint-space interpolation (like by F-type keyframes).

### 2.9.3 Introducing a complex example

Figure 11 shows the keyframe-based motion editing interface of *RobotMax*, while a CAD construction has already been imported from Pro/Engineer (using the workflow presented so far in previous sections). The CAD model is a courtesy of Fachhochschule Ulm [53].

![Figure 11: Motion editing inside RobotMax](image)

This double cardan-axle example (for rear suspension) has two prismatic and six revolute joints. The construction is power-driven by two motors:

- One motor drives the rightmost actuated revolute joint (dark red). There are two *F-type* keyframes for this joint, defining the start and end reference angles for the simulation. As in an F-type keyframe there is linear interpolation of the reference angles, the angular velocity will be constant.

- The leftmost prismatic joint (dark blue) is also actuated. The slider in the left cage has three reference positions set using 3 *I-type* keyframes: the initial (it is depicted), the bottom- and the uppermost position are stored.

All other joints are passive in the system (rotational ones: two pieces per each universal joint + one inside the slider for bearing; prismatic: between the inner and outer clawed parts).

*Later on in this chapter I will use this model to demonstrate the remaining steps of the model translation workflow.*
2.10 Creating new Modelica components

In this section I present the new Modelica package I created in order to support the new elements of the “CAD to SIM” workflow in a domain-independent way. This new package is based on the original standard multibody library shipped with Dymola. My Modelica package contains three sub-packages *Actors, Joints* and *Motors*, the details of which I will discuss in this section.

2.10.1 The standard MultiBody library in Modelica

The standard Modelica.Mechanics.MultiBody Library [37] is a free and extensible Modelica package providing 3-dimensional mechanical components to model mechanical systems – such as robots, mechanisms and vehicles – in a convenient way. It is included in the Dymola distribution and can be downloaded also from the Modelica Association’s website [57].

I emphasize the following features of the library:

- **About 60 main components**, i.e., joint, force, part, body, sensor and visualizer models are ready to use. One-dimensional force laws can be defined with components of the *Modelica.Mechanics.Rotational* and of the *Modelica.Mechanics.Translational* library and can be connected via available flange connectors to MultiBody components.

- **About 75 functions** exist to operate on orientation objects in a convenient way, e.g., to transform vector quantities between frames, or compute the orientation object of a planar rotation. The basic idea is to hide the actual definition of an orientation by providing essentially an *Orientation* type together with functions operating on instances of this type. *Orientation* objects based on a 3x3 transformation matrix and on quaternions are provided. As a side effect, the equations in all other components are simpler and easier to understand.

- A **World model** has to be present in every model on top level. Here the gravity field (currently: no gravity, uniform gravity, point gravity), the visualization of the world coordinate system and default settings for animation are defined.

- **Built-in animation** properties of all components, such as joints, forces, bodies, sensors. This allows an easy visual check of the constructed model. Animation of every component can be switched off via a parameter. The animation of a complete system can be switched off via one parameter in the *World* model.
• **Automatic state selection from joints and bodies.** Most joints and all bodies have potential states. A Modelica translator, such as Dymola, uses the generalized coordinates of joints as states if possible. If this is not possible, states are selected from body coordinates. As a consequence, strange joints with 6 degrees of freedom are not necessary to define a body moving freely in space.

### 2.10.2 My Actors package

The new models in my *Actors* package (see Figure 12) are extending the original Modelica rigid body and shape models with support for collision response (see Chapter 3 for the details) in a unique way. I discuss each element of the package in the followings.

**Body:**

This is the original implementation of the standard `Modelica.Mechanics.Multibody.Part.Body` rigid body model with parametrical mass, inertia tensor and one frame connector (it can have 12 potential states).

This model is a point-mass body with no geometry information.

**Junction:**

This model extends the `Modelica.Mechanics.MultiBody.Parts.FixedRotation` model that allows a fixed relative pose between its two coordinate systems, i.e. frame connectors. I have overridden the rotation part in order to use two orthogonal base vectors to represent the relative orientation of the frames.

This model represents the `SceneNode` transformations in *RobotMax*.

**Shape:**

This class extends the `Modelica.Mechanics.MultiBody.Visualizers.Advanced.Shape` model with an internal `Junction` declaration that allows a geometry having `frame_Shape` as local origin (CS1 frame) that can differ from the `frame_Actor`, which is used to connect to the associating `Actor` instance (to its centre of gravity CG frame).

The *Shape* model has the same meaning like the *CShape* class of *RobotMax*.
Collider:

This class extends my Shape model with the support for external collision force and torque, which can act on the Actor, associating this actual geometry. The generalized 6-dimensional force input is propagated to the Actor, which connects directly to the frame_Actor port.

The Collider instances have a unique integer index that allows them to communicate with the Collision Manager to refresh their external contactForce input. It has stiffness and restitution material parameters, an optional non-negative margin value (that allows having a “shield” with rounded edges around the polygonal surface approximation). With the contactHandling Boolean flag you can disable the collision response for this actual geometry.

For a self-explaining example including Collider models please refer later to section 3.4.1 on page 44.

Actor:

My new rigid body model encapsulates a Body instance, with parametrical mass, inertia, centre of gravity and initial pose elements. The central frame_a connector of an Actor can connect to multiple Shape or Collider instances (using their frame_Actor connectors). Therefore the Actor model relates to the Entity class of RobotMax.

Dummy:

This is also a rigid body model, but it has a fixed sized box shape, and cannot connect to any Shapes or Colliders. Consequently, the Dummy actors can never collide with others. They are conveniently representing the RootGround and RootPart entities (which are CBody instances in RobotMax).
2.10.3 My Joints package

In the followings I discuss the new models of my Joints package (see Figure 15).

**PrismaticAbstract, RevoluteAbstract and Spherical:**

These three models are representing the passive, pure mechanical parts of the translational- (1-dof), rotational- (1-dof) and ball-and-socket (3-dof) joints, all having two frame connectors that are hereby constrained. Except for the Spherical joint, these models have an abstract, 1-dimensional Drive flange that allows external forces to control the relative motion between their frame_a and frame_b connectors. Unless this flange is connected, these models behave as passive joints.

**Prismatic and Revolute:**

These are the non-mechanical parts of the abovementioned abstract and pure mechanical joints. They encapsulate a replaceable control system and drive model. On Figure 16 it can be seen that feedback information comes out of the jointDrive module and serves as an input for the jointControl module:

The Prismatic model uses an ideal gearbox (idealGearR2T) transforming the rotational motion of the driving subsystem into translational motion. Both models have a Target input signal for the reference angle / position that is a target to be achieved on the 1-dimensional Drive flange output. The non-linear limiter models set lower and upper limits of target values. An output Drive flange must connect directly to a respective passive joint’s Drive flange.
JointDrive:

Even in case of translational motion, rotational drive is being used internally, yet (later this has to be also replaceable):

![Diagram of JointDrive model]

Figure 17: The JointDrive model

The model contains replaceable, parametric Motor and Gear parts. For example the default permanent magnet DC motor and ideal gear combination can be easily replaced by an asynchronous induction machine with slipring rotor and a more realistic gearbox model with friction.

In Dymola there is a very useful feature: unless the connector interfaces of the new components are different, the “replaceable” elements can be transparently exchanged with new implementations, without influencing the existing structure and connections.

JointControl:

This model is the control subsystem of the connected joint. Its reference angle signal comes from the external, higher level motion control. It controls a rotational flange’s angle. The output of this module connects directly to the input voltage signal of the JointDrive model. Using internal sensors on the flange input connector it can utilize the actual angle, velocity and acceleration of the flange. My actual default implementation uses a simple proportional control with a constant gain parameter.

2.10.4 My Motors package

I created a collection of interface classes in this Modelica package to allow replacing the Motor part of the JointDrive model in a transparent way. I created a basic abstract machine model that contains a rotor part with a parametric inertia and the couplings to the electrical domain.

All other motor models have to inherit a basic interface, in order to keep a consistent coupling to higher level models that might use them as replaceable parts. I created three motor-families and put a few common models inside them:
• **Asynchronous induction machines**
  o with squirrel cage rotor
  o with slipring rotor
• **Synchronous induction machines**
  o with permanent magnet
  o electrical excited, with damper cage
  o with reluctance rotor and damper cage
• **DC machines**
  o with permanent magnet
  o with electrical shunt

The original implementation of the aforementioned motor models can be found in the standard *Modelica.Electrical* library. I have only generalized the interfaces to force compatibility.

### 2.11 Exporting to Modelica models

In this section I present how the internal mechatronic model of *RobotMax* is exported to Modelica models. The export process has two main steps:

- Exporting the pure mechanical structure ⇒ *RobotMAX_CAD.mo*
- Exporting other elements ⇒ *RobotMAX.mo*
  o the components of other domains (see section 2.8)
  o the reference motion signal (see section 2.9)

The two main steps generate two separate Modelica models. I also include some useful graphical annotations for Dymola: this eases the graphical editing on the 2D diagram at a later time (for example you can still add there hydraulic components, etc.).

#### 2.11.1 Generating the pure mechanical model

Before I present the first part of the model export procedure, I have to discuss the important question of possible kinematic loops in the mechanical structure.

**Kinematic loops**

When there are multiple connected bodies and joints forming a closed cycle, the structure contains a kinematic loop. Whenever a kinematic loop occurs in a
multibody system, non-linear algebraic equations are present on "position level". It is then usually not possible to select states by structural analysis during translation (which is possible though for non-loop structures).

Dymola – the interpreter and simulator of Modelica models – can detect a non-linear algebraic loop of equations and tries to reduce that to a system of coupled algebraic equations by just appropriate symbolic equation manipulation: this is done unseen from the user. In a kinematic loop no explicit "cut-joints" are asked for – compared to other multi-body programs (e.g.: in SimMechanics [60]) where the modeller has to make such a suggestion. Via the dynamic dummy derivative method the generalized coordinates on position and velocity level from one of the joints are dynamically selected as states during simulation. Whenever these two states are no longer appropriate, states from one of the other joints are selected. Due to the new handling of such over-determined DAEs in Dymola, the modeller does not have to take special actions if a general loop occurs in the model structure.

The cardan model I presented in section 2.9.3 on page 26 has one kinematic loop:

![Figure 18: A kinematic loop and a cut-joint in RobotMax](image)

The black arrows on the left hand side represent the increasing level of the scene graph hierarchy: the base plate (down) is the root element, the two side cages are its children, etc.

The exploded view on the right hand side has a red arrow, showing that inside the right universal-joint one revolute joint is automatically detected as cut-joint. The given cut-joint is at the leaf node of the right branch. This joint has a reference to the (red) cardan-cross part (the virtual edge is indicated with a blue arrow: it is the "second parent" of the joint, see section 2.6.4 on page 16) at the leaf node of the other branch, thus closing the loop.

This example kinematic loop has multiple degrees of freedom, so it is not a planar loop (which can have maximum 1-dof, see below).
Planar kinematic loops

In case of a planar loop the involved bodies have only a single degree of freedom altogether. However all planar loops result in a DAE that does not have a unique solution. This is a structural property that is determined by the symbolic algorithms. Since they detect that the DAE is structurally singular, a further processing is not possible. Without additional information it is also impossible that the symbolic algorithms could be enhanced because if the axes of rotations of the revolute joints are only slightly changed such that they are no longer parallel to each other, the planar loop can no longer move and has 0 degrees of freedom. Algorithms based on pure structural information cannot distinguish these two cases. The solver of Dymola needs help.

The usual remedy is to remove superfluous constraints, e.g., along the axis of rotation of one revolute joint. A Boolean flag must be set to true for exactly one cut-joint in each planar loop:

\[
\text{MultiBody.RevoluteJoint } jR(\text{planarCut} = \text{true});
\]

The other joints in each loop must use the default “false” value for the planarCut parameter. I focused also on the problem of detecting and handling planar loops (see later in this section) during the automated exporting of the mechanical structure of RobotMax.

All the class-objects of RobotMax presented in section 2.6 have the ability to create a Modelica text based on their actual state. The internal scene graph is exported completely automated, node-by-node, starting from the root SceneNode vertex, using the following recursive pseudo-code:

\[
\text{ExportNode(SceneNode } n) \{ \\
\quad \text{foreach SceneObject } o \text{ in } n.\text{Objects} \\
\quad \quad \text{ExportObject}(o); \\
\quad \text{foreach SceneNode } c \text{ in } n.\text{ChildNodes} \\
\quad \quad \text{ExportNode}(c); \\
\}
\]

The ExportObject method is overridden in the sub-classes of SceneObject, i.e. in CBody and CJoint (refer to the class hierarchy on Figure 2, page 15). Note that a user-friendly text file, including graphical annotations is generated during the export process.

Table 2 summarizes the mapping of RobotMax classes to my Modelica models (you can refer to section 2.10 for models of the Modelica column here):
Table 2: RobotMax → Modelica mapping

<table>
<thead>
<tr>
<th></th>
<th>Modelica</th>
</tr>
</thead>
<tbody>
<tr>
<td>SceneNode</td>
<td>Actors.Junction</td>
</tr>
<tr>
<td>CBody</td>
<td>Actors.Junction or Actors.Dummy</td>
</tr>
<tr>
<td>Entity</td>
<td>Actors.Actor</td>
</tr>
<tr>
<td>CShape</td>
<td>Actors.Collider or Actors.Shape</td>
</tr>
<tr>
<td>CJointPrism</td>
<td>Joints.PrismaticAbstract</td>
</tr>
<tr>
<td>CJointRevol</td>
<td>Joints.RevoluteAbstract</td>
</tr>
<tr>
<td>CJointSphere</td>
<td>Joints.Spherical</td>
</tr>
</tbody>
</table>

Exporting a CBody object:

A single rigid body (named “body”) is exported the following way:

1. Via traversing the scene graph towards the root I locate a SceneObject (called parObj): this is a valid parent object that belongs to the closest, non-empty ancestor SceneNode of body. If such parObj cannot be found, this indicates that body has the root SceneNode as parent, i.e. body is a top-level object (e.g.: the RootGround).

2. The parentNode is the SceneNode to which the parObj belongs (it was found in the previous step).

3. The isChild Boolean flag indicates that parentNode is not the root SceneNode. According to Table 2, a new Modelica model (either an Actor or a Dummy) is instantiated, and the centre of gravity vector, the inertia-tensor and the mass parameters are written to the text file output.

4. The initPose variable indicates whether the equations for the initial pose of the Actor / Dummy instance must be included (the isStatic parameter is true, if the body must be fixed in 3D space: the RootGround is also a good example here):

   \[
   \text{initPose} = \overline{\text{isStatic}} \lor \text{isChild}
   \]  

5. The graphical annotation is written directly after this, defining the location of the new icon block that represents this Actor / Dummy instance. This is necessary to make the block visible in the Dymola editor, too.

6. If isChild is true, a new Junction instance must be added between the body and parObj blocks, defining the fixed relative translation between those two parts. If parObj is a joint, the frame_b connector of it must be used to connect to the actual body’s frame_a connector. Adding a Junction element involves also a new graphical annotation (a line will be drawn between the respective frame connectors that are connected).
7. If neither isChild, nor initPose is true, the body part must be connected to the world inertial system, just like it would be a static body (child bodies can never connect to the world, because they already connect to their parent body). This connection requires both a function element, as well as the graphical annotations.

8. If the body is an Entity, it can store references to CShape instances that all must be exported right after the body instance.

Exporting a CShape object:

Each CShape represents a geometry that can be used for both visualization and collision-shape purposes, as well. Depending on the internal Collide flag of the parent CBody object, a CShape will be exported either as an Actors.Shape or as an Actors.Collider. Note that the user of RobotMax can set this Collide flag for any Entity instances separately and – for performance reasons – it is false, by default.

The following precautions must be followed to export a CShape instance to Modelica:

1. There are two independent and globally unique integer indices: “idx” and “DXF”, both starting from 1.

2. The idx is incremented after an Actors.Collider (originally a CShape that can collide) has been written to the output. This identifies this collision-geometry for the Collision Manager (that is responsible for calculating collision forces, see Chapter 3). A simpler non-colliding shape is translated to an Actors.Collider instance that does not use this unique index.

3. Dymola supports .dxf geometries for internal visualization: for this reason both the Actors.Collider and Actors.Collider models can refer to DXF format files. The DXF index increments for each CShape object: it is used to auto-generate a filename for every single geometry in the entire scene (1.dxf, 2.dxf, ...). If the internal visualization of RobotMax (see section 4.8) has higher priority over Dymola’s built-in, basic functionality, the storage of DXF geometries can be entirely disabled in order to increase the performance of the translation process. Otherwise the triangles of the shape are stored one-by-one in the respective .dxf file.

4. The graphical annotation is added to the new Modelica block, thus making it visible in Dymola on the model-editor screen.

5. Finally, the frame_a connector of the shape’s parent (Actors.Actor) and the shape’s frame_Actor connector are connected with a connect() equation in the output model.
Exporting a CJoint object:

The procedure of exporting a joint starts very similar to the aforementioned CBody case:
1. The parObj parent object has to be found.
2. The parentNode is the SceneNode to which the parObj belongs.
3. isChild is always true for a joint (because parentNode can never be the root SceneNode).

In case joint is a cutJoint, it hangs on a leaf- SceneNode and has also a second virtual parent (refer to sections 2.6.4 and 2.11.1). For each cutJoint it has to be determined first, whether it is a planarCut type or not: in order to assist the solver of Dymola.

For sample constructions with kinematic loops you might refer to Figure 5 (planar case) and Figure 18 (non-planar) in this chapter.

At the very beginning of the export procedure the entire scene-graph is analyzed by my SceneAnalyser class. The following tasks are carried out during an analysis:

- Generating the adjacency matrix of the scene-graph, including the virtual edges between a cutJoint candidate and its second virtual parent SceneNode, too.
- For each joint in the scene the kinematic loops that go through the given joint are detected and stored. Cycles (represented by SceneCircle instances) are found using a Breadth-First-Search algorithm.
- A global list is maintained that includes all possible unique cycles in the scene-graph.

Using the actual revolute joint, those loops that go through that joint, can be queried from the SceneAnalyser singleton. For each involved cycles a test must be done to make sure a cutJoint candidate joint is planarCut or not. One of the following cases can lead to a planar kinematic loop:

- There are at least three revolute joints in the loop having parallel axes.
- There are two revolute joints with parallel axes + one prismatic joint with an orthogonal axis or one spherical joint in the loop.

If one of the previous cases holds, the joint is a planarCut joint, thus the equation (3) must be present in the final Modelica code.

From now on I continue the general steps required to export a joint:
According to Table 2, a respective new Modelica joint model (e.g.: Joints.RevoluteAbstract) is instantiated, and the axis of the joint is written to the text file output. A graphical annotation is written after the declaration, too.

4. If the joint is not inside a kinematic loop, its initial angle and velocity must also be specified inside the new Modelica model instance.

5. A new Junction instance must be added between the joint and the parObj blocks. If the parObj is also a joint, the new Junction must connect to the frame\_b of it, instead of the frame\_a connector. The graphical annotation is also added for the new Junction instance.

6. If joint is a cut-joint – held by a leaf SceneNode and having a second virtual parent (see section 2.6.4, page 16) – a given kinematic loop can be recreated: the steps 1, 2 and 6 must be done again, but traversing in the direction of the other parent node.

The next figure shows the completely automatically generated Modelica model of the double cardan-axle example (you might refer to section 2.9.3, on page 26 for a rendered screenshot in RobotMax):

![Figure 19: Mechanical model of the double cardan-axle in Dymola](image)

The automatically added graphical annotations in the model allow displaying all components in the 2D editor of Dymola. I emphasized also the automated creating of a bright layout, thus the model remains human-friendly:

- The first row enumerates dummies, actors and their shapes, respectively.
- The second row contains the junctions (you can see that the Floor actor and the RootGround dummy part are both connected to the World, because they are fixed objects).
- In the last row it can be seen that there are two abstract prismatic joints (I marked them here with red arrows) and six abstract revolute joints in this mechanical model.
2.11.2 Generating the mechatronic model separately

The active model elements of other domains (section 2.8) and the reference motion (section 2.9) are to be exported to complete the mechanical model to an M’ mechatronic model. This step can be done independently from the exporting of M.

The domain independence is emphasized here: I could achieve that by exploiting the object-oriented model inheritance under Modelica. The pure mechanical model is used as the base class of the other one:

```
model RobotMAX extends RobotMAX_CAD;
...
end RobotMAX;
```

Unless the mechanical structure or the internal names of the actual model changes, you can make modifications in the non-mechanical domains (e.g.: fine-tune a motor parameter or create a new motion) and re-export only the M’ model (RobotMAX.mo) again: this won’t affect the M mechanical model (the RobotMAX_CAD.mo).

The other way: you can change a material parameter in the CAD system and re-export the M model only, and use the old M’ model with it. You won’t loose the work done on the mechatronic model (e.g.: you might have added there a pneumatic cylinder, as a new linear actuator).

2.12 Conclusion

The example double cardan-axle model – that I have presented in this chapter on Figure 11, page 26 – has been successfully translated and extended with motors and reference motion to form a mechatronic model.

The next figure shows a screenshot of its dynamic simulation under Dymola. The forces and torques acting on two selected parts (the bearing inside the slider and the left cardan-cross) are represented by single and double arrows, respectively (the left cage has been hidden for visibility reasons):

```
Figure 20: Inspecting forces and torques in Dymola
```
Commercially available virtual prototyping packages cannot be used directly for modelling the active motion of mechanical systems with attached components from other application domains.

I have presented a smooth, highly automated virtual engineering workflow to translate a CAD construction to a mechanical multibody system model described in Modelica, the standard model representation language. As a part of the process there is a possibility to extend the mechanical model with components from other engineering domains, inside the same integrated model authoring environment, RobotMax that I have introduced here.

The construction designer can work with arbitrary Pro/Engineer assemblies, extend them to mechatronic models, and analyze the simulation results under Dymola without much effort on manual functional modelling.

My new method allows building a mechanical model out of the scene-graph of RobotMax. The whole structure is analysed - including the virtual edges introduced by cut-joints - and the kinematic loops are processed in a consistent way during the translation process.
Chapter 3

Contact Processing in Modelica Models

Contact processing – including collision detection and collision response – is a key factor and a challenging problem in simulation of multi-body systems (MBS), where a balance between performance and accuracy has to be found, especially for real-time applications.

Matlab (with SimMechanics and Simulink) and Dymola (exploiting the Modelica language) are two widely-spread multi-domain simulators, and they are also often used with MBS models, though they do not offer contact processing, so far. Other multi-body simulators, like Vortex or ODE, which do support that, are limited to the single, mechanical domain in the rest of their functionality.

After giving an overview on the contact processing problem this chapter focuses on my research that was targeted for adding contact processing support to Dymola for extending the MBS simulation. At the end of this chapter I present a complex case study: using my presented workflow in Thesis 1, I created a virtual model of a six-legged walking robot “Anton” [59], and validated the extended mechatronic simulation under Dymola by comparing to the behaviour of the physical prototype of this experimental mobile robot platform.

3.1 Introduction

In mechanical systems certain machine elements might interact with each other. These interactions can be divided into two main categories:

- *Mechanical joints* are representing permanent constraints on relative motions between the connected parts.

- *Mechanical contacts* are observed while surfaces of bodies are touching each other. Two major phenomena – differentiated by the duration of the contact – can occur during such interaction:
  - *Resting or sliding contact* (where static or dynamic friction forces can arise).
  - *Impulsive collision forces* (almost instantaneous, typically short-time interactions caused by compression and decompression of contacting bodies).
3.2 Background

The Modelica.Multibody library [37] has the standard mechanical components that are used in a general multibody simulation in Modelica. Currently only rigid bodies and joints (mechanical constraints) are supported in the standard. The new models of my model translation process (see section 2.10) are built on these standard components.

On the other hand the handling of mechanical contacts between penetrating bodies is not supported in the standard library. According to this, the standard mechanical parts fall through each other uninterruptedly during the simulation. This shortcoming of the library is waiting to be solved for years.

There are numerous articles dealing with contact modelling in special cases under Modelica (e.g.: tyre + road contact), but these methods cannot handle general cases and they don’t rely on the standard mechanical elements of the library.

However there has been a suggestion [38] a few years ago, how contact processing could be included in the standard library, it is still in the research phase and left some questions open. My proposed solution differs from that in significant points.

3.3 The main tasks of contact processing

The important task of processing mechanical contacts during the simulation of multi-body systems (MBS) can be separated to two consecutive steps: collision detection and response calculation. The detection algorithm determines whether the simulated objects would interpenetrate each other if there weren’t any collision handling, which could prevent that. The response algorithm corrects the movement of bodies that would otherwise penetrate.

The next figure explains integration of collision processing inside a dynamic simulation environment [5]:

![Figure 21: Scheme of contact processing during dynamic simulation](image)

The following requirements have to be fulfilled:
• The contact processing support should be easily integrated to the already existing mechanical models, which use the standard elements.
• A key task is to track the shapes of the (moving) bodies for spatial relationship (including time coherence), and to extract information about the actual properties of each contact (contact points, penetration depth and – velocity, crossover surfaces or volumes, etc.). For this detection phase, the usage of external libraries is recommended.
• Based on the actual contact information and the materials of the contacting bodies the response (reaction forces and torques) has to be calculated.

3.4 My new models

The global World component of the standard Modelica multibody library defines the globally unique (singleton), fixed inertia-system, that the standard components are referring to. As the Modelica language is object-oriented, it allows model inheritance in a convenient way.

Each inherited model can extend its parent with new equations, and can transparently replace a parent component, too. In other words the inherited models are backwards compatible.

„Manager“:

The central element of my contact processing implemention is the (collision-) Manager model that inherits and transparently replaces the standard Modelica.MultiBody.World component.

The Manager moderates a bidirectional communication with the Collider shapes, negotiating collision detection and response calculation modules.

„Collider“:

I have already introduced this model briefly in section 2.10.2, as the one that extends my Shape model with collision support. It connects to an Actor and can influence the motion of that via propagating external collision forces.

Each Collider serves its frame_shape coordinate system’s pose (position + orientation) and velocity (both linear and angular) to the Manager, which forwards the information to the external collision detection module that can update its internal shapes’ states.

Based on the actual contact properties, collision responses are calculated, and the Manager tells each Collider the actual local collision forces that are acting in the contact points. As a Collider can have an offset local centre (see Figure 14 and the Pivot Junction element on page 29), the local forces and torques must be transformed to the frame.Actor connector, thus they act actually on the connected body’s centre of gravity.
In Modelica environment I separated the description of the standard rigid body model from multiple optional collision shapes. Regarding Thesis 1, this new method allows the decomposition of complex shapes – being extracted from a CAD system – into multiple convex ones, creating a better basis for a more stable – geometrically more robust – collision processing.

3.4.1 A self-explaining example

The following example has only two rigid bodies (actors): it has a free falling desk actor, which is falling down and colliding with the static floor actor:

With the help of the previously discussed Manager, the off-centre colliding shape (the dotted left leg) can act on the desk actor. The collision force $F_{\text{local}}$ (both normal and tangential components) is calculated at vector $p$ (collision point) in the local coordinate system of the shape, thus it causes also a $\tau_{\text{actor}}$ torque acting at the centre of gravity of the desk actor.

It is also shown here how a single concave object (a desk) is decomposed into multiple convex shapes: 5 box-shaped Colliders (1 top plate + 4 legs) are connected to the desk actor, while the floor has only a single Collider.

The next figure shows the schematic view of the floor + desk example scenario in the Dymola editor:

![Figure 22: Side view of a collision scenario](image)

![Figure 23: Schematic view of the previous example](image)
3.5 Collision detection

The collision detection plays an essential role in simulation of mechanical constructions, and is also used for trajectory planning (known as collision-avoidance) and further, computer graphics-related tasks. Besides indicating the fact that some object features are close enough (or already intersecting each other), the known algorithms can usually serve more details about the actual spatial circumstances: such as distances of closest features (where using actual movements the time of impact can be estimated), penetration depths, separation directions, collision surface areas or volumes, etc.

Irrespective to the Modelica environment, there are a lot of articles on inspecting spatial relations between arbitrary geometries, balancing between accuracy and performance. Some authors try to describe the problem analytically [39], while others – mostly emphasizing real-time performance over accuracy – use approximations [31][35][42].

On the market there is a bunch of non-commercial software packages that support collision detection e.g. SWIFT, Bullet, ODE and others [19][32], offering usually C or C++ interfaces for external applications.

Out of this set some tools (e.g.: V-Collide) – which can just indicate the existence of penetrating geometries and do not offer any useful contact parameters to be retrieved – must be excluded.

The one reason of collision processing is known as a hard problem that the bodies involved in a multi-body system might have very complex geometry. Furthermore, the physical accuracy of the simulation rests on the overall accuracy of collision processing algorithms. In order to be able to run a simulation in real-time, this phase of collision processing has to be fast enough.

Assuming there are N bodies in the scene, there can be \( \binom{N}{2} \) object pairs, which could interact with each other. In order to decrease the \( O(N^2) \) time complexity of the objects’ pair-wise intersection testing, the most known algorithms are divided into the following steps:

- **Collision culling** can assure groups of objects are spatially separated into cells / domains: proving they are independent, as the workspace is subdivided using spatial data structures like Sweep-and-Prune, Quadtree/Octree or BSP-Tree. The same techniques are also used for object culling to accelerate 3D rendering [10]. In a dynamic scene time coherence is utilized to refresh the data structures between consecutive calls for collision culling.
• In the Broad phase the pairs of surely non-colliding objects are found and excluded from each spatial domain. Various bounding volume types (each encompassing and fitting a single body, more or less) are forming a hierarchy – such as AABB-tree, sphere-tree [23], OBB-tree [20] or k-DOPs [30] – offering the advantage of very fast overlapping tests, whereby the potentially non-colliding object-pairs are excluded, again.

• The Narrow phase is the exact collision test, which uses the original detailed geometries of the remaining object pairs. In this part the closest features can be found, a contact manifold can be generated, contact points can be enumerated and contact parameters – such as contact plane, contact volume, contact normal and penetration depth – can be calculated.

3.5.1 Creating a Modelica-C wrapper for the task

Calling external functions is a preferable method to integrate collision detection algorithms with practical physical models, since body geometry is stored externally, anyway.

As Modelica supports external function calls in C language, I have created a general interface for the collision detection and contact parameter extraction under Modelica. The package contains Modelica functions which wrap to a common C interface. The functions are grouped to the following sections:

• Initializing / destroying: (create, destroy) these functions will be called right at the beginning / end of the simulation notifying the external module to allocate / release the required resources.

• Adding collision geometry: (addBox, addCylinder, addMesh, etc.) at the beginning of the simulation each Collider instance (see section 2.10.2) calls a respective function to announce its existence for the external module (unique ID and shape information are transmitted). The external module should decide whether the geometries are tessellated internally to triangle mesh or remain parametric for some cases.

• Setting other parameters: (setStiffness, setMargin, setRestitution) the stiffness and restitution material properties are required for the response calculation. Only the margin value influences the contact detection, as it defines the thickness of an outer shell around the surface of the shape, in order to smooth the surface or decrease visible penetration during the contact. All of these functions set constant parameters, so they are used only once at the initializing (t=0).

• Setting actual poses and velocities: (setPositions, setOrientations, setLinearVelocities, setAngularVelocities) as the Manager knows the position and orientation of each Collider, it can forward the whole information in the arguments of these functions each simulation step.
Invoke collision processing: (doCollision) this function will be called every simulation step by the Manager to update the external collision forces on all Colliders, regarding their actual contact geometry information.

I have integrated two collision detection libraries – SOLID and Bullet – using this common interface. My response calculation method – which is discussed in section 3.8 – uses also partially external implementation connecting to this interface.

For the integration of external, specific software modules into the contact processing problem I have created a general Modelica-C++ interface that unifies the integration of replaceable collision detection and response calculation implementation.

3.6 Collision response in general

Computation of contact response is a difficult task. The bodies might move in a complicated way, though their penetration should always be prevented. Here again there is a trade-off between efficiency and accuracy. One of the goals of Modelica simulations can be interactivity; therefore the computation should have at least the same speed as the processes in the real mechanisms.

There are analytical methods that are used in mechanical simulations where accuracy has higher priority than simulation speed. Tribology (from the Greek word ‘τριβο’, meaning ‘to rub’) is the science and technology of interacting surfaces in relative motions [36], and is commonly applied in bearing design where complex equations are taking the hydrodynamic properties of the lubricant – which compresses and decompresses during the contact – also into account. The performance of analytical methods is far away from real-time simulation.

Extreme accurate methods are known for computation of contact forces based on finite element methods, where bodies are subdivided into very small fragments [50]. It is required to cover the surface of colliding bodies with a mesh and elementary contact forces are to be computed for each point over the mesh. The collision response can be defined by integrating such elementary forces over the contact surface. These methods can be implemented in software packages for FEM-analysis (e.g.: ANSYS [51]), but cannot run fast enough on today’s computers for a real time simulation, either.

One of the general reasons of computing contact forces is the great variety of surface geometries. The contact model of Hertz was published already in 1882 [21], and discusses a linear-elastic model of contact. This article had an influence on further research, leading to the analysis of smooth, second order surfaces in [29] and in [40] where also Coulomb-friction model has been considered. Such systems in general use higher order polynomials to compute forces from geometrical relations.
However, many simulation applications do not require extreme accuracy. In many cases additional assumptions are taken into account providing high simulation speed, but decreasing the accuracy. As a matter of fact, different assumptions can lead to different computation methods but with the same (or nearly the same) computation results. In such cases, it is not important for the application what assumptions and methods were used.

Mathematically the impulse-based- and the force-based methods (penalty-force or Lagrange-multiplier) are known to formulate the collision response.

### 3.7 Impulse-based method

This method is based on the impulsive collision of pairs of objects [34]. It considers the impulse conservation law and operates with the impulses of the colliding bodies before and after the collision as well as with the restitution coefficient of materials. It requires instantaneous modification of the velocity variables, while assuming that

- Collision duration is negligible.
- Only one point of collision exists.
- The colliding bodies don’t move during the collision.
- No other forces than collision force act on the bodies.
- The impulse gives instantaneous change to the linear and angular velocities of the colliding objects.

The impulse-based approach can be easily used in MBS-based models if the collision impact on the other bodies in the system is negligible (i.e. in the system of free-flying bodies).

In other words this method is not suitable for models that contain constraints between bodies that might collide: the joints defined in the structure disallow free movement after the collision, so the velocities cannot be changed freely.

Nevertheless, since the velocity is not continuous in the impulse-based model, the traditional ODE solvers can’t be used. The continuous integration process in the solver should be stopped at the instant of collision and should be resumed with the new initial velocity.

An alternative approach is based on writing a system of non-differentiable equations and applying a Newton method [41] specially devised for such equations. This method has successfully been applied for body impact with friction [26].

Furthermore such idealized impact laws are only useful for stiff collisions. These properties restrict the applicability of the impulse-based method of the dynamical analysis of a multibody system.
3.8 Force-based method

An alternative approach of contact processing in multi-body mechanical systems is based on the force and torque model of collision. It is assumed that the contacting bodies penetrate each other and separation forces are caused by this penetration. These forces try to prevent further penetration and to separate the contacting bodies.

It is well known that during the collision relatively large forces occur between the colliding bodies for a very short period of time. The value and the direction of these forces can be approximately computed for each simulation time step. Though the force-based method leads to stiff ODE, this can be handled by solvers used with Modelica.

The many existing methods for the calculation of the force in the mechanical contacts and joints are divided into two following groups:

- **Force based methods with Lagrange multipliers formulation** models the mechanical constraints (contacts and joints) with the reactive forces, which are presented as Lagrange multipliers $\lambda$. The constraint forces perform no work on the environment and the physical meaning of the mechanical contact is lost. The mechanical interaction of the bodies caused by the contact is represented by these $\lambda$ reactive forces, which should be determined between two simulation steps in an additional optimization loop (see Figure 21) under consideration of the energy or/and impulse preservation laws.

- **Force based methods with penalty formulation** models the mechanical contact with a temporary stiff (possible nonlinear) spring. The active contact/friction forces (Figure 24) perform work on the environment and therefore the physical background of the mechanical contact is kept here. The mechanical interaction of the bodies caused by the contact is represented by the active forces $F_{\text{CONTACT}}$ and $F_{\text{FRICTION}}$, without any additional optimization between the simulation steps.

![Figure 24: Contact modelling with penalty-forces](image)

The force-based method, however, requires certain contact parameters, which can be time-consuming to calculate continuously. The following properties of the collision force should be taken into account:
• The collision force and collision torque acting on an object is zero if the object
does not collide.

• Between the start of collision and the end of collision a force is activated that
prevents further penetration.

• If an ideal collision is modelled (collision of points masses), the resulting
velocities after the collision are given by the law of preservation of linear
momentum.

• A contact force acting on a body resting on a horizontal platform
compensates, balances the gravitational force. Therefore such an object does
not move (its vertical acceleration and velocity is zero).

3.8.1 First assumptions

In order to balance between the required accuracy and available computational
power of Modelica simulations, the following rules must apply for collision
force computation:

• The collision force acting on a body is zero if it does not penetrate with any
other body.

• If body A penetrates body B, collision forces are created to act on the objects
A and B and are applied at the point of contact on each body. The force is
directed so that A and B are pushed away from each other due to this force.
The direction of this force corresponds to the shortest displacement that can
separate the bodies.

• The magnitude of the force is proportional to the depth of penetration of A
and B. This depth is the length of the shortest displacement that can separate
the bodies. This corresponds to the model of spring and damper, inserted
between the bodies. The bodies are rigid, but the spring and the damper are
not. This also corresponds to the physics of collisions between elastic and
homogeneous (isotropic) bodies.

Two bodies penetrate if the volume of their intersection is greater than zero, i.e.
there is at least one point of one body inside the other one. Determining the
volume of intersection is a very time-consuming problem: in most cases the
penetration depth is easier to be found and it accurate enough for real-time
applications [5] (refer to the case study in section 3.9).

The main advantages of such a force-based approach are the simplicity and the
possibility of using it for stiff and soft contacts. This approach works reasonably well if
several contact points are also present at the same instant time. The disadvantage of this
approach is that the variable length integrator’s step size should be reduced during the
contact time in order to catch the rapidly changing contact forces and torques.
3.8.2 Determining contact points

The point of contact can be defined in many different ways. The naive definition states that this is the point where two bodies touch each other the very first time (at the first instant of collision). Such a definition is only good for very short-duration contacts. If the bodies have longer contacts (i.e. the separation force should be computed during several time steps), then the point of the very first contact can differ from the point of contact a few steps later. An example of such behaviour is two bodies colliding and then keeping sliding contact. In this case the point of the first contact might influence the tangential forces (see later) but usually it cannot be used for evaluation of normal contact force for the next steps.

Collision detection software packages can determine a point which belongs to the intersection of the objects \( A \) and \( B \) if they collide. If the objects do not collide, the closest pair of their points can be tracked to assist the contact detection at a later time.

3.8.3 Determining normal direction

The normal direction of the collision force can be determined in several different ways. For a short contact it can be natural to define the direction of the collision force as opposite to the relative velocity of the contact points, i.e. the points of the first touch between the bodies.

![Diagram of contact points and normal direction](image)

**Figure 25:** Determining the normal direction

There is a \( p \) penetration vector between these contact points the magnitude of which relates to the penetration depth: the shortest distance \( A \) and \( B \) can be separated along that direction.

For more accurate determination of force direction, a mesh based on colliding surfaces has to be constructed, and a normal vector to the surface in each mesh point is used as local force direction. The resulting force direction is then found by integrating the vectors of all the forces acting on the contact surface.
3.8.4 Penalty forces: introducing a temporary spring

The source of collision forces is in the following physical phenomenon. Initially, the bodies are compressed with each other and therefore deformed. This deformation causes reaction force and restitution (restoration) of the shapes. The bodies therefore separate from each other. This phenomenon can be modelled in different ways. The traditional approach is to model it as a temporary, virtual spring (with an optional damper).

At each instant during the collision the force values should be computed. The collision duration is very small, and the velocity of objects changes rapidly during the collision, thus the forces needed to change the velocity can be very large (i.e.: the stiffness of the spring is large).

3.8.5 Constraints on the contact force

- Conditions at the start of the collision (constraints $C_1$):
  - $F(0) = 0$: the force should be zero at the start of the collision.
  - $x(0) = 0$: the penetration depth should be zero at the start of the collision.
  - $v(0) = v_0 > 0$: the actual speed is known at the start of the collision

- During the collision ($C_2$):
  - $F(t) < 0$: the force should always push the object away from the obstacle.
  - $x(t) > 0$: the penetration depth is positive (the object penetrates the obstacle).
  - $v(t)$: the penetration speed is reduced to zero and – probably – to negative values

- If no external force act on the colliding body (or this force is negligible), it behaves exactly according to the impulse law ($C_3$):
  - The time when collision ends is $\tau$ (it is an unknown duration parameter).
  - $F(\tau) = 0$: the force should be zero at the end of the collision, and after that.
  - $x(\tau) = 0$: the penetration depth should be zero at the end of the collision.
  - $v(\tau) = v' = v_1 = -\varepsilon v_0 < 0$: the speed at the end of the collision can be predicted using the restitution coefficient $0 < \varepsilon \leq 1$.

- If a constant external force $F_{ext}$ acts on the colliding body, it either behaves as above or – in case $F_{ext}$ is large enough – the body rests on the obstacle, and the collision never ends ($\tau \to \infty$; $C_4$):
  - $\lim_{t \to \infty} F(t) = -F_{ext}$: the collision force should compensate the external force.
  - $\lim_{t \to \infty} x(t) = x_{rest} > 0$: the penetration depth should stabilize at some value.
  - $\lim_{t \to \infty} v(t) = 0$: the body rests, i.e. it does not move anymore.
There can be many definitions for $F$ satisfying these equations and relations. However, in most cases, the difference between these definitions (i.e. difference between the overall effects they cause) is negligible in comparison with the effect caused by the constraints. In practice, the definition for $F$ is sometimes not motivated by physics, but rather by numeric analysis. It is chosen such way that overall result of collision matches physical laws (i.e. preservation of impulse, and preservation of energy). In addition it should be chosen so smooth that numerical methods used in simulation are able to handle that.

### 3.8.6 Contact force based on linear spring model

In this section I will prove that the linear spring model alone is not enough to model contacting materials with different restitution values. $F$ depending on penetration position $x$ in a linear way has the form:

$$F(t) = \begin{cases} -K x(t) & \text{if } x(t) > 0 \\ 0 & \text{if } x(t) \leq 0 \end{cases}$$

where $K$ is a positive constant, called penalty coefficient. According to $C_1$ there is no additive constant factor in the equation. Physically this corresponds to a stiff spring (with stiffness $K$), temporarily placed between the objects during the collision. The expression $-K x(t)$ corresponds to an ideal spring. The expression contains $x(t)$ in the first power only. Therefore this method is a first order linear model.

For brevity of the solution, $K$ might be replaced by another positive constant, $K=q^2 m$. In this case – taking the constraints $C_1$ and $C_3$ into account – the system equations (assuming that external forces acting on the colliding bodies are negligible) appears as follows:

$$F(t) = -q^2 m x(t)$$
$$F(t) = m \ddot{x}(t)$$
$$\dot{x}(0) = v_0$$
$$x(0) = 0$$

This results in the equation:

$$\ddot{x}(t) + q^2 x(t) = 0$$

which has a solution:

$$x(t) = \frac{v_0 \sin qt}{q}$$

and so the velocity is:
\[ \dot{x}(t) = v_0 \cos qt \] (9)

When the collision ends (i.e. \( x(\tau) = 0 \) giving \( t = \tau = \pi/\theta \)) the velocity \( \dot{x}(\tau) = -v_0 \). This means that the resulting velocity does not depend on the mass, and does not depend on the penalty coefficient in front of \( x(t) \).

However \( \dot{x}(\tau) = v = -\varepsilon v_0, \ 0 < \varepsilon < 1 \): so the simple penalty-based linear model above is not good in the general case, and another model should be chosen.

### 3.8.7 Contact force depending on penetration depth

If the contact force if a function of only the penetration depth \( x(t) \), it can be proved that this is still not enough to model different material restitutions:

\[
F(t) = -p(x(t)) \\
F(t) = m \ddot{x}(t)
\] (10)

Thus

\[
\ddot{x} + p(x) = 0 \\
\] (11)

The \( p(x) \) **penalty** function is **positive** during the collision. Introducing \( \dot{x} = y(x) \):

\[
\ddot{x} = \frac{dy}{dt} = \frac{dy}{dx} \frac{dx}{dt} = \frac{dy}{dx} \dot{x} = y \frac{dy}{dx}
\] (12)

The equation (11) can be rewritten as:

\[
y \frac{dy}{dx} + p(x) = 0
\] (13)

By integrating the equation (13) we can get:

\[
\int ydy + \int_0^{x(t)} p(u)du = \frac{y^2}{2} + \int_0^{x(t)} p(u)du = \frac{\dot{x}^2}{2} + \int_0^{x(t)} p(u)du = C
\] (14)

where \( C \) is a constant. Because of (6), \( x(0) = 0 \), so at \( t = 0 \):

\[
C = \frac{\dot{x}(0)^2}{2} + \int_0^0 p(u)du = \frac{v_0^2}{2}
\] (15)

Now (15) is rewritten as:

\[
\frac{\dot{x}^2}{2} + \int_0^{x(t)} p(u)du = \frac{v_0^2}{2}
\] (16)

At the end of the collision \( x(\tau) = 0 \), thus
\[
\dot{x}(\tau)^2 = v_0^2 
\]

According to (11) and the fact that \( p(x(t)) > 0 \) is a positive penalty function, \( \dot{x}(t) < 0 \), so \( \dot{x}(t) \) is a monotonous decreasing function of time: (17) can have only one solution:

\[
v' = v(\tau) = \dot{x}(\tau) = -v_0
\]

This means that the collision will be always ideal elastic (\( \varepsilon = 1 \)) when the collision force depends only on the penetration depth, thus this model is not adequate for different restitution values, either.

### 3.8.8 Contact force based on spring and damper model

It has been showed in the previous sections that besides the penetration depths a new factor is essential to calculate the collision forces, as the magnitude of the velocity at the end of the collision must be reduced by the actual factor \( \varepsilon \). Because the penetration velocity changes gradually during the collision (actually it is decreasing monotonously), it was an obvious idea to use it as a new factor in the old (5) equation of \( F \). However this factor cannot be an additive one, because the \( F \) collision force must be zero right after the \( \tau \) collision time and also at \( t = 0 \) (right at the first moment of impact), while the penetration velocities are usually nonzero in these times.

According to these, the new formula – having a multiplicative \( H \) function depending on the normal penetration velocity – looks like:

\[
F(t) = \begin{cases} 
-H(\dot{x}(t))Km x(t) & \text{if } x(t) > 0 \\
0 & \text{if } x(t) \leq 0 
\end{cases} 
\]

(19)

If \( \varepsilon = 1 \), then \( H \) should equal to constant 1, thus for convenience:

\[
H(\dot{x}(t)) = 1 + G(\dot{x}(t))
\]

(20)

The simplest function for \( G \) is the linear one: \( G(v) = k \cdot v \), which is called the damping factor. The new formula of (6) is the following:

\[
F(t) = \begin{cases} 
-Km x(t)(1 + k\dot{x}(t)) & \text{if } x(t) > 0 \\
0 & \text{if } x(t) \leq 0 
\end{cases}
\]

\[
F(t) = m \ddot{x}(t)
\]

(21)

The \( \varepsilon \) restitution factor shows a dependency between the first moment collision velocity \( v_0 \), and the \( k \) constant: \( \varepsilon(v_0,k) \). The \( k \) damping constant is unknown: it has to be determined from the material parameters of the colliding bodies.
It is obvious that by multiplying \( v_0 \) with a \( p \) constant and dividing \( k \) with the same \( p \), the quotient \( \varepsilon = \frac{p \cdot v(\tau)}{p \cdot v_0} \) remains the same.

With Mathematica the equation (21) can be solved symbolically. Introducing a new independent variable \( u = \frac{v_0 k}{v_0} \), according to plots for \( 10^{-2} < u < 10^{2} \) (see Figure 26), the new \( \varepsilon(u,1) \) unknown function can be well approximated by a reciprocal formula:

\[
\varepsilon(u,1) = \frac{1}{1 + \frac{1}{v_0 k}}
\]

(22)

The next figure shows the relation of the restitution factor, the damping factor \( k \) and the impact velocity \( v_0 \):

![Figure 26: The restitution factor function](image)

Therefore the \( k \) factor has the following form:

\[
k = \frac{1 - \varepsilon}{v_0 \varepsilon} \quad 0 < \varepsilon \leq 1
\]

(23)

It can be seen that ideally \( \varepsilon = 1 \) leads to \( k = 0 \), which is the non-dampened case (when only a linear spring is involved, see section 3.8.6). Substituting \( k \) in (21), the final collision force magnitude in the normal direction can be calculated.
### 3.8.9 Determining tangential direction

The contact points can have a $v_{rel}$ relative velocity, the tangential component of which is:

$$v_{rel}^t = v_{rel} - e_n \cdot (e_n \cdot v_{rel})$$

(24)

where $e_n$ is the (unit length) normal direction vector of the penetrating contact; it is defined at $t=0$ to be parallel to the vector between the contact points. The tangential contact speed $v_{rel}^t$ is magnitude of the velocity vector in (24):

$$v_{rel}^t = |v_{rel}^t|$$

(25)

If this speed is greater than zero, a frictional force arises in the contact’s tangential direction: along an $e_t$ unit length vector in Cartesian space:

$$e_t = \frac{v_{rel}^t}{|v_{rel}^t|}$$

(26)

### 3.8.10 My friction spring model

I extended my spring and damper based material model also with the contact response in tangential direction. In my model the static and dynamic friction forces are switched at a relative tangential speed limit: $v_{rel}^{t,lim}$ that is dependent on the contacting materials. Thus the force acting along the $e_t$ tangential direction is either static or dynamic friction. Until the tangential relative speed limit $v_{rel}^{t,lim}$, a virtual spring from a fixed anchor point causes an increasing static friction arising between the colliding bodies. At the first moment of impact a contact point is anchored in space and a virtual spring is connected to it (see left side of Figure 27). During the collision this spring is extending between the anchored point and the actual contact point.

![Figure 27: Spring model for static and dynamic friction](image)

If the colliding bodies have already large enough relative velocity in tangential direction, a **dynamic friction** force will replace the static one. In this case in my model the “anchor” point is sliding together with the actual contact point in the tangential plane in a way that the virtual spring has a constant extension (see right side of Figure 27), thus causing a constant dynamic friction force in the opposite direction of the movement.
Thesis 2.2

For calculating the normal- and tangential reaction forces in simulated mechanical contacts in Modelica environment, I introduced a coupled spring + damper material model based on new material properties:

**Normal contact force:**

\[
F_{\text{NORMAL}}(t) = \begin{cases} 
1 + \frac{1 - e}{v_0 \varepsilon} \dot{x}(t) S_{\text{CONTACT}} x(t) & \text{if } x(t) > 0 \\
0 & \text{if } x(t) \leq 0
\end{cases}
\]  

(27)

where \(v_0\) is the normal component of the relative collision impact speed at the start of a contact, \(S_{\text{CONTACT}}\) is a spring stiffness factor [N/m], \(\varepsilon\) is the restitution factor (both depend on the materials of \(A\) and \(B\)), \(x\) is the penetration depth and \(\dot{x}\) is the normal component of the relative velocity of the contact points.

**Tangential contact forces:**

A static friction force is caused by a spring having stiffness \(S_{\text{FRIC}}\) that is anchored in the contact point and extends to the following length:

\[
l(t) = \int_0^t v_{\text{rel}}(t) \, dt, \quad 0 \leq t \leq \tau
\]

(28)

\[
F_{\text{static}}(t) = \begin{cases} 
\min\left(S_{\text{FRIC}} \cdot l(t), \mu_{\text{static}} \cdot F_{\text{NORMAL}}\right) & \text{if } 0 < \nu_{\text{rel}}^t < \nu_{\text{rel}}^{t,\text{lim}} \\
0 & \text{otherwise}
\end{cases}
\]

(29)

At larger relative tangential velocities the dynamic friction force replaces the static one. Using a material-dependent \(\mu_{\text{dynamic}}\) dynamic friction coefficient:

\[
F_{\text{dynamic}}(t) = \begin{cases} 
\mu_{\text{dynamic}} \cdot F_{\text{NORMAL}}, & \text{if } \nu_{\text{rel}}^t > \nu_{\text{rel}}^{t,\text{lim}} \\
0, & \text{otherwise}
\end{cases}
\]

(30)

The final contact response is the sum of the normal and the tangential forces:

\[
F_{\text{CONTACT}}(t) = \begin{cases} 
e_A \cdot F_{\text{NORMAL}}(t) + e_f \cdot \left(F_{\text{static}}(t) + F_{\text{dynamic}}(t)\right), & \text{if } \nu_{\text{rel}}^t > 0 \\
e_n \cdot F_{\text{NORMAL}}(t), & \text{otherwise}
\end{cases}
\]

(31)

Let \(e_A\), \(p_A\) and \(e_B\), \(p_B\) denote the centre of mass and contact point locations of body \(A\) and \(B\), respectively. The contact torques acting on colliding bodies \(A\) and \(B\):

\[
\tau_{\text{CONTACT}}^A(t) = -F_{\text{CONTACT}}(t) \times (e_A - p_A)
\]

\[
\tau_{\text{CONTACT}}^B(t) = F_{\text{CONTACT}}(t) \times (e_B - p_B)
\]

(32)
3.9 A case study

In this section I will present a case study using the new six-legged prototype mobile robot Anton (developed at the RobotsLab [59] of the Otto von Guericke University, in cooperation with the Fraunhofer Institute for Factory Operation and -Automation in Magdeburg):

Figure 28: The construction of Anton

This prototype robot has the following properties:

Mechanics

• Aluminium construction with carbon-fibre filling pieces
• Walking mechanism: rectangular shape body consists of 3 modular segments, with 2 legs each and with 3 degrees of freedom per each leg
• Active degrees of freedom: 24
• Variable number of legs due modularity (4, 6, 8 etc.)
• Dimensions:
  o shoulder \( l_s=40 \text{ mm} \)
  o thigh \( l_t=100 \text{ mm} \)
  o shank \( l_s=180 \text{ mm} \)
  o total length \( L_B=944 \text{ mm} \)
  o length of a segment \( L_S=288 \text{ mm} \)
• Drives:
  o Faulhaber 1724, 24V DC motors in knees (6)
  o Faulhaber 2232, 24V DC motors in other joints (18)
Sensor system
• Angle (potentiometer) and IGRs in each motor
• 6 three-component force sensors mounted in the leg shanks
• Gyroscopic sensor in the first segment
• Stereoscopic camera

On-board control hardware
• Fast and flexible FPGA controller in each segment (control loop <0.1ms)
• Real-time Ethernet communication (EtherCat) via network processor NetX by Hilscher in each segment
• PC connection over Ethernet (EtherCat) (control loop 1ms)

I was involved in the virtual prototyping of this robot. I have tested the virtual prototype with numerous gait modes using the control algorithm of the robot, analysing contact forces in Modelica environment.

3.9.1 Using the presented workflow

The CAD model has been converted to a mechanical Modelica model using the workflow presented in Chapter 2. Within the same process I have extended the model by adding electromechanical models of the respective DC motors to all joints (chosen from the Faulhaber DC motor class of my Object Library).

The next figure shows Anton in the initial pose in my RobotMax tool. You can observe the axis alignment of the 24 revolute joints:

![Figure 29: Virtual Anton in the editor of RobotMax](image)

The mechatronic model has been exported to Modelica. Some information about the complexity of the Modelica models:
Anton\_CAD.mo:
- 1146 components
- DAE having 17813 scalar unknowns and equations
- 5679 unknown variables
- 2742 nontrivial equations (out of those 847 nonlinear)

Anton.mo: (this extends the Anton\_CAD.mo model by inheritance)
- 1614 components
- DAE having 20429 scalar unknowns and equations
- 8271 unknown variables
- 4388 nontrivial equations (out of those 871 nonlinear)

3.9.2 The control algorithm for the robot

Anton has a multi-level control algorithm that replaces the keyframe-based pre-generated motions and is doing online trajectory planning. The block schema is presented on the next figure:

![Block diagram of the control structure of Anton](image)

Figure 30: Control structure of Anton
Step cycle geometry block:

This block calculates a **pentagonal** geometry in a two-dimensional plane S₁-S₂ (shown with a loop of red arrows on the figure above). The endpoints of all legs follow this trajectory (with possibly different phases for the legs, of course).

- **external input vector “a”:**
  - \( \dot{u}_{\text{transfer}} \): longitudinal transfer speed of feet in air (real)
  - \( \tau \): time of a step cycle (real)
  - \( l_{\text{sum}} \): summary length of a step (real)
  - \( h_{\text{adapt}} \): height of the adaptation zone (real)
  - \( \text{Start} \): if false, the robot finishes movement (Boolean)
  - \( \text{Frozen} \): if true, the robot stops immediately (Boolean)

- **input vector “I”:** actual normal forces (3 x 6 = 18 real)

Step cycle control block:

This block generates trajectories for the legs in robot coordinate system, in various gait modes. By default it transforms the previously mentioned S₁-S₂ plane to the vertical (ZY) plane, parallel to the robot’s longitudinal axis (z).

The reference support polygon of the robot can be a rectangle or an ellipse: in the latter case the middle legs are touching the ground a little bit broader, than the front and rear legs, thus increasing stability.

- **input matrix “II”: (= output of previous block)**
  - \( S₁, S₂ \): two dimensional trajectory vector of each leg (2 x 6 = 12 real)
  - \( \text{Phase} \): phase of each leg (6 integer)

- **external input vector “b”:**
  - \( \text{ScaleH} \): horizontal scaling (real)
  - \( \text{ScaleV} \): vertical scaling (real)
  - \( R\text{Scale} \): direction (real) \( 0 = \) forward motion, \( \pm 1 = \) maximal turn left/right
  - \( R₁ \): support polygon offset right (real)
  - \( R₂ \): support polygon offset front (real)
  - \( \text{clearance} \): body ride height (real, negative!!!)
  - \( \phi \): body relative rotation-pitch (real)
  - \( \theta \): body relative rotation-roll (real)
RCS to LCS block:

This block transforms trajectory points from the robot coordinate system RCS to each leg’s local coordinate system LCS1-6.

- **input matrix “III”:** (= output of previous block)
  - \(X_r, Y_r, Z_r\): robot-centred reference position of each leg (3 x 6 = 18 real)
- **external input vector “c”:**
  - \(COG\): centre of gravity offset in Body Coordinate System (3 real)
  - \(HeadPos\): position of the head / manipulator in BCS (3 real)
  - \(HeadOri\): orientation of the head / manipulator in BCS (3 real)

IK block:

The last block performs the inverse kinematic calculations and outputs the reference angles for all 24 joints in the robot.

- **input matrix “IV”:** (= output of previous block)
  - \(X_l, Y_l, Z_l\): local reference position of each leg (3 x 6 = 18 real)
- **external input vector “d”:**
  - \(BodyAng\): ref. angles between neighbour segments (2 x 3 = 6 real)
- **output matrix “q”:**
  - \(q_i\): reference angle for joint #i (24 real)

I have converted the original S-functions of these blocks to C++ classes and created a C wrapper to use the functionality inside Modelica. I created a shallow Controller model in my CLAWAR.Anton Modelica package: it calls the C wrapper function with the actual input signal arguments (\(a, b, c\) and \(d\) vectors) each simulation step once, and the external C++ module (like Matlab S-functions) outputs the reference angles for all revolute joints of the robot. The output signals of the Controller connect to the Target input connectors of the respective actuated joints of the virtual Anton robot model.

In case of this complex trajectory-planning there are unlimited combinations of the input signals the controller can use (as some input variables are continuous ones), which all can result in different motions. For a simulation expert it is much more convenient to generate some test cases without caring about keyframes (section 2.9.1).

Using my implemented method, a powerful tool stays at our disposal for experimenting with various locomotion tasks of Anton in virtual reality. Using the force sensors of the real robot, there is a possibility to compare the virtual reality with the real world behaviour, thus allowing the validation of the dynamic model, including the contact processing.
3.9.3 An analysed real world motion

The physical Anton robot has a Matlab / xPC-Target-based teleoperation interface, through which the data of the 3-component force sensor of each leg can be monitored and logged in a file, on demand.

In the example locomotion scenario that I discuss here, the robot had to move forward on an even horizontal plane in tripod gait mode (3 legs support the body in a triangle arrangement, while 3 others could swing in the air freely) with a constant reference longitudinal translational speed \( v_{tr} = 0.1 \) m/s.

I acquired physical data from the force sensors of the front left, middle right and rear left feet. Although the data of the onboard sensors has to be interpreted natively in the coordinate system of their respective feet, the control algorithm of Anton assures that the legs are always vertical in global space, thus the sensor data can be directly interpreted in the Cartesian world coordinate system.

3.9.4 Comparison with the simulation

I ran the simulation in Dymola for 25 seconds with exactly the same control parameters as the real robot, while the virtual robot’s feet were also equipped with virtual force sensors. I included instances of the standard Modelica MultiBody library’s CutForce force sensor model from the Sensors package.

Some parameters used in my contact model (Thesis 2.2, page 58):

\[
S_{\text{foot, ground}}^{\text{CONTACT}} = 8000 \frac{N}{m} \quad \varepsilon = 0.8 \quad v_{rel}^{\text{lim}} = 1 \frac{cm}{s}
\]

\[
S_{\text{FRIC}} = 10000 \frac{N}{m} \quad \mu_{\text{static}} = 0.7 \quad \mu_{\text{dynamic}} = 0.25
\]

Some locomotion parameters of the robot (refer to section 3.9.2 on page 61):

\[
v_{tr} = 7 \frac{cm}{s} \quad \tau = 0.5 \quad h_{\text{adapt}} = 5 \text{ cm} \quad \text{clearance} = -18 \text{ cm}
\]

\[
l_{\text{sum}} = 7 \text{ cm} \quad R_1 = 23 \text{ cm} \quad R_2 = 28.8 \text{ cm}
\]

Because of its non-zero initial altitude and my soft contact model presented in this chapter, the virtual robots feet are allowed to penetrate the ground during the simulation. I executed the Start signal after 0.5 seconds to delay the locomotion planning algorithm a little bit.

The simulation results and the real force sensors’ data can be observed on the upper and lower part of Figure 31, respectively:
Figure 31: Simulated vs. real contact forces

It can be observed that the weight of the “ant-robot” *Anton* is balanced in a way that the front and middle legs carry a little bit more load than the rear ones, because on the head there are two (stereoscopic) cameras mounted.

As the *Start* command was executed after 0.5 seconds, it can be seen that the load on each feet started to change periodically. The robot was walking forwards in a tripod-gait mode, using the following way: in the first moment all six legs support the body, after the *Start* signal, the triple-support way of insect-like walking is initiated: while three legs are supporting the complete weight of the robot (approx. 7.8 kg) the other legs (lifted in the air) can swing forwards rapidly. Because of the relocation of the robot’s centre of mass, the load on the rear feet is always decreasing while it is increasing on the front ones. During the swing phases of the observed three feet (the front / rear left and the middle right ones) there is no relevant, measurable contact force, of course.

The bottom half of Figure 31 compares my simulation results with the real robot’s behaviour, validating my contact model that I introduced in Modelica environment with Thesis 2, showing a very good match to the reality.
3.10 Conclusion

This chapter presented my research on extending the standard Modelica implementation of mechanical models with collision processing. The collision forces are calculated in my method using temporary spring and damper elements acting during a contact both in normal and in tangential directions, considering also the restitution of the colliding materials. The restitution property of the materials is more tangible than the impact velocity-dependent damping factor that was used in others’ work.

The verification and validation of the contact model has been done. I have tested the collision processing with various motion tasks using a complex, six legged prototype mobile robot, its control algorithm and its virtual mechatronic model in Modelica / Dymola environment comparing the simulation results with the data of real force sensors of the real robot.
Chapter 4

Spatial visual feedback in teleoperation

Generally the phrase “teleoperation” can indicate circumstances in which a device or machine is operated by a person from a distance. Before a teleoperation mission involving expensive mechatronic (robotic) systems would start, it is common that the operating personal does some training in a simulated environment. Besides the fact that a high-level application of mechatronic simulation (refer to Thesis 1 and 2) is here demanded, it poses also a challenge for visualization. For navigation the correct localisation of possible obstacles demands significant visual depth information, as well.

4.1 Introduction

The 3D visual feedback is very important in teleoperation. The human operator must be able to navigate in the virtual reality by interpreting a large amount of visual information. The amount of visual information that can be sent during a teleoperation mission can be limited by the capacity of the transmission channel. In comparison to the binocular stereoscopic methods, the monocular ones need definitely fewer resources. Therefore the use of pure monocular visualization can also be considered, without loosing too much depth information that is necessary for the task.

The most people are able to navigate a vehicle in a dynamic virtual environment by using monocular information exclusively: namely the motion-parallax effect – meaning a steady change of angular position of two observations of the same object projected to the retina while the observer and the environment have a relative velocity – is established to present a relatively strong depth clue.

Thesis 3

[4][6]

Proceeding from the important role of motions during the interactive simulation of mobile mechatronic systems I recognized that besides the monocular visualization requires less resource compared to the binocular stereoscopic techniques, it has also enough strength in creating adequate depth impressions in the operator. □

During my literature research I did not find any relevant articles about the quantitative analysis of the intrinsic depth effect caused by the motion. For the applicability in mobile teleoperation a detailed study was demanded: thus it can be assured whether the monocular visualization has enough strength for that purpose.
4.2 Background

I will discuss the sensory background of human spatial vision first and then I give a short overview on various visualization methods and devices that can be used for stereoscopic imaging. Based on this brief introduction I present the details of a psychophysical experiment, which has been done at the department recently. The main goal of that research was a systematic comparison of the subjective perception of depth delivered by the aforementioned 3D techniques extended with dynamic motion.

With the help of my colleagues at the department, Tamás Urbancsek and Dr. Ferenc Vajda we created an experimental cell in the basement of the I building of the University, where a range of static or dynamic 3D scenarios were visualized with various methods for a group of voluntary people without informing them about the actual technology. The test persons were asked to evaluate questionnaires, concerning subjective impressions on their spatial sight while either wearing anaglyph / LCS / polarizer glasses or not. I will present the gained result that visual depth sensation delivered by a motion parallax technique compares to the efficiency of traditional 3D visualization techniques.

4.3 Components of visual information

Among the five basic sensory perceptions the vision stands on the first place along the overall amount of perceived information. There is a very broad literature on the anatomical structure [22][44][46] and the physiological functionality 0[24] of the human vision system, therefore I will not detail these here in this dissertation.

By studying human vision one can observe a really admirable phenomenon: our ability to perceive spatial, 3D information by means of only two-dimensional light patterns projected to our retinas. It is known that there is no direct depth perception: the retinas work separately on planar images and transfer the information to the visual cortex area of our brain. Only the neural fusion of two coherent planar images leads to our binocular depth sensing.

In the following sections I give two sensory categories visual perception that lead to depth sensing.

4.3.1 Monocular vision

Monocular vision is a form of visual perception, in which each eye is being used separately. By using the eyes in this way – opposed to binocular vision – the field of view can be increased (see animals, having eyes positioned on opposite side of their head), while depth perception is limited. Though there are a lot of monocular visual effects that influence depth sensing. The most of these are based on assumptions and learning, and are imprinted in the infant years.
Linear perspective

The central projection of parallel geometric structures – like roads or railway tracks – can cause an experience of convergence to a designated point on the horizon. This effect has a close connection to the relative size-based size consistency phenomenon.

Size consistency

Using the comparison of the nominal size of a projected image on our retina and the empirical information about the sizes of objects around us, we can decide whether an object in our sight is closer or farther. For the same reason the continuously decreasing size of a passing car leads to the information that it is driving away from us.

Overlapping

This effect can intervene when there are overlapping shapes in our sight, and some of them are hiding parts from other shapes. An assumption of ordering and relative distance is made, based on the fact that a closer object can hide some parts of the farther ones. The two middle examples on Figure 34 have no such inspiration.
Aerial perspective

Changing colour of objects can let you guess their actual distance: for example the bluish shade of a distant mountain is caused by small moisture particles in the air, where light is scattered and the colour of blue sky goes to domination. In foggy weather the mountains seem to be farther than in good weather.

Depth of field

Objects that are out of focus are blurry, and their grade of sharpness let you deduct their relative distance. One extra-retinal phenomenon – the accommodation – is strictly related to this effect: the tension of lens-focusing (or ciliary-) muscles changes the focal length of the eye (by acting on the zonular fibres), thus it has some depth information within small distances.

Lighting and shading

It is usually assumed that light comes from above, thus shadows and highlights can give you information about the shape of an object. Furthermore the brighter objects seem to be closer than the darker ones (“flashlight” effect).
Motion-parallax

If you move your eyes the object in different distances seem to move with different relative velocities. The objects that are closer are following the eyes’ direction, the farther ones are moving in the counter-direction and the objects exactly on the fixation distance are staying still. The motion-parallax itself is the relative change of angular position of two observations of the same object projected to the retina, while the observer and the object has a relative velocity.

4.3.2 Binocular vision

Binocular visual perception (or shorter: stereopsis) is very important for depth sensing. The importance of depth information caused by retinal disparities inside Panum’s fusional area (see later) is unquestionable. In case of binocular vision a single three dimensional point is projected to two 2-dimensional points on the retinas: the location of those points are different in each eye, but they are still cohesive. The location-differences are called disparity:

Figure 38: Disparity in binocular vision

There is a three-dimensional curve (named horopter: horos-: border; opter: observer) in the binocular human vision system, the points of which correspondingly stimulate the same (anatomically identical, with zero disparity) points of each retina [47]. The horopter is geometrically an intersection of a cylindrical surface with a hyperboloid [61]:

Figure 39: Horopter and the borders of binocular image fusion

Fusion is a neural process, where the images of the two retinas are forming a single image that allows stereoscopic vision. Panum’s fusional area is a set of
points in space (around the horopter, see Figure 39), which can stimulate points of the retinas of the observer in such a way that the fusion process can find the correspondence between these points. If this was successful, the retinal disparity carries the depth information [43]. Unsuccessful fusion of images leads to binocular rivalry and double sight.

4.4 The importance of motion

All of the monocular visual cues influence the relative depth sensing. As soon as the observer has a relative movement to the sight he / she is looking at, these cues will grow continuously. The changing of shadows and highlights, the varying overlapping status of objects can cause already a depth perception for the observer: this is actually independent from how many eyes are involved in sight.

I investigated some rules of thumb how the depth information of a dynamic 3D scene can be increased:

- Lighting conditions are very important in achieving good results.
- Big contrast is demanded: bright foreground in front of a darker background. Flat surface colours should be avoided.
- The more depth layers (at least foreground, middle- and background layers) are in the sight, the better quality of motion-parallax can be achieved. The relative movement of these layers can lead to guess their relative distances (in depth).
- Detailed textures should be used on the surfaces: the perspective distortions on high-resolution textures can be very important. Using wide angle camera projection, the linear perspective has even bigger priority.
- If you have objects in front of the focal plane, these will “pop-out” of the display. If such an object intersects any side of the viewport, there will be a rivalry between the virtual and the real background and the depth effect will be lost.
- Captions inside the focal plane can help to distinguish between objects that are closer and the ones that are farther from the viewer.
- Care should be taken to the fact that the brain of an observer needs at least 2-3 seconds in order to accommodate the new scene and process depth information out of it.

These assumptions were taken I was designing the trial scenarios in the psychophysical experiment at the department (see section 4.6).
4.5 Stereoscopic devices and techniques

In this section I emphasize most the stereoscopic visualization techniques that were involved in the psychophysical experiments at the department [4]. There is an important common feature of these devices: given a 3D scene that the observer is looking at, these are exploiting various separation methods, and thus only one respective image is transferred to each eye of the observer. The human brain interprets the disparities in the slightly different images as corresponding central projections of the same original 3D scene.

I categorize these devices and techniques by the way of image separation. I will call left and right images the respective pictures targeted for the left and right eye of the observer.

4.5.1 Spectral separation

These techniques divide the visible light spectrum into two or more disjoint intervals. The left and right images are passed through colour filters that are disjoint in their spectra, and the observer has to wear a goggle with the same spectral parameters. The less crosstalk between the left and right filter, the better quality of fusion can be achieved. Both presented techniques here are using passive separation (with spectacles) on the observer side.

**Anaglyph method**

This method uses only two spectral intervals: the most commonly applied filters are red and cyan (complementary colours are to be used in order to keep the overall white balance). This passive technique is one of the oldest ones (also works on the good, old paper media), thus there is a huge amount of images to be found over the Internet.

4.5.2 Time-domain separation

This technique presents the left and right images alternately, in rapid succession. From the observer’s point of view this method needs active synchronisation. Though only a single image is visible on the presenting monitor in a moment, the eyes are “remembering” the previous image for a short time. As the neural process of image fusion takes a finite time these additional circumstances allow a (continuous) depth sense for the observer.
**Liquid crystal shutter glasses**

The LCS glasses [56] are active programmed shutters: in a time frame one of the lenses is opaquely black, while the other is transparent: thus each eye can see only its respective image. The changes between opaque and transparent states must be strictly synchronised with the monitor hardware.

As the perceived image frequency is the half of the original vertical refresh rate of the monitor, this technique requires a display capable of minimum 100 Hz, thus it is better to use with CRT or DLP projectors.

4.5.3 **Polarization-based separation**

Polarization is the property of light (or other electromagnetic waves) that describes the direction of the transverse electric field. Thus, polarization is the direction of oscillation in the plane perpendicular to the (longitudinal) direction of travel. Polarization can be linear, circular or elliptical. If a linear polarized filter is in the way of a perpendicular linear polarized light, no light is transmitted. The same happens if a circular polarized filter is in the way of a circular polarized light of the opposite direction. Therefore, two different filters can separate two light sources.

Using this technique, polarizing filters are applied on both the right and the left images, but with different polarization, which is aligned with the polarization filters of the observer’s viewing glasses.

In passive methods the two images of the 3D view are projected over each other onto the same canvas by two projectors. The projectors must emit differently polarized light beams (where polarization can be either linear or circular). It is important, that the canvas must not change the polarization of the reflected light: a metallic coating is being used for this purpose. In linear case the polarization of the two projections can be perpendicular to each other (or can be counter-directed using circular polarization), so using the respective polarizing filters again in the glasses on the observer side, only a single image is visible for each eye at a time. Simple projectors can be used with polarizing filters for this purpose.

In active methods, a display creates two sorts of polarized light natively: this is nowadays only possible using special LEDs (put into LED matrix displays). The same passive, polarized goggles need to be used to view such displays.
4.6 My StereoVision application

At the Mobile- and Microrobotics Laboratory of the department I have developed a program – called StereoVision – that supported the psychophysical experiments on objective comparison of anaglyph, liquid crystal shutter and polarisation-based stereoscopic visualization methods when motion was optionally also taken into account. It conducted the process entirely and logged the subjective answers of each voluntary test person. All features of this software have been integrated later to RobotMax, my interactive mechatronic model authoring and visualization tool that I already introduced in Chapter 2.

The application has a main module that controls maximum two projectors that present a given 3D scenario by means of various visualization techniques. Using the user input module, a range of parameters of the visualization (the technique itself, the position and orientation of virtual cameras, etc.) can be changed interactively. This component is in close connection with a script interpreter module that allows automated control of some of these parameters, and monitors and logs the user’s behaviour making the anticipated objective evaluation later possible.

The program offers four well-known 3D visualization techniques: the motion-parallax (when only monocular cues are presented), the anaglyph (images for left and right eyes are separated in spectral domain), the liquid crystal shutter (images are separated in time domain) and the dual-view methods (image separation using polarization).

4.6.1 The important features

My StereoVision application is based on the managed version of the Direct3D 9.0c application programming interface (later referred to as D3D), as it is written entirely in C# language, under .NET environment. The DirectX is Microsoft’s library of hardware-accelerated APIs, the managed version of which allow reaching high performance even under the .NET environment – which is basically an interpreted, just-in-time compiled platform running over a virtual machine (i.e.: Common Language Runtime). There is only a minimal performance advantage for the native libraries, which is outperformed by the convenience and robustness of managed C# programming.

The geometry and material information of the individual scenarios are stored in the .X DirectX scene format, which can include triangular mesh shapes, material / texture information and optionally animation keyframes, too.

I implemented advanced (HLSL) shader techniques for visualization (see in Appendix, on page 101). I use the (C-like) HLSL high level shading language [45] to implement the 3D and stereoscopic techniques. The so called “shaders” are programs that run on the specific parts of the GPU (graphics processing unit) instead of the main processor. The shader codes are automatically
compiled into an efficient intermediate effect language (and later to machine code of the actual GPU) in runtime. Basically a vertex program is a graphics function used to perform mathematical operations (for example perspective projection to screen space) on the objects' vertex data (position, normal, etc.). The pixel- (also known as fragment-) program is a graphics function that calculates effects on a per-pixel basis after the output of the vertex shader (i.e. triangles) is rasterized. The basics of vertex- and pixel shader programming can be found in [17].

For binocular techniques I need to generate images for both eyes’ perspective and make them separately stimulate the left and right eyes, respectively. According to the 3D technique being used at a given time, either the composition of these images can appear as the final image, or I pick a single one to be displayed (for LCS glasses) or show both ones simultaneously (using two projectors and polarization lenses).

Although the scene being viewed at can be static, any camera movement can cause significant increase in the depth impression of the viewer (see section 4.4). I allow an option to use a small amplitude harmonic function to automatically rotate the camera around its target point to enhance this depth cue.

In case of the monocular method I can render the projection of the 3D scene directly to the screen in the usual way: for each object in the scene I set up the transformation matrices and the material parameters, and then feed the mesh data of the dynamic vertex buffers through the programmable graphics pipeline using my shader routines in order to get the shaded result.

In order to avoid flicker, everything is drawn first to a back-buffer image and after synchronization with the display hardware, this image can be presented on the screen (this is the well-known double buffer technique [33]). The camera that is used in this mode will be referred later to as the cyclopean or central camera.

For binocular methods two camera models are distinguished: the one with parallel viewing axes and the one with convergent eyes, crossing axes at the same target point in the space (also known as “toe-in”). In my actual implementation there is no support for asymmetric frustums, for the latter case I use the method shown in the following figure (when the cameras are rotated towards each other around the world’s vertical axis):

![Figure 42: The case of convergent cameras (viewed from above)](image-url)
The left and right eyes have a horizontal offset \( \text{(base)} \), the half of which is the distance from the central (cyclopean) eye’s origin for both eyes. I treat this value with a ± sign, so it is easy to switch the roles of the left and right eyes, when the current setup needed that (e.g.: by changing the order of the projectors dual projection mode using polarizer filters).

For the **convergent case** the left and right cameras are transformed from the central one the following way (the camera’s up axis is the unit vector that is vertical in camera space, pointing upwards):

\[
\begin{align*}
    z &= \text{eye}_{\text{center}} - \text{target}_{\text{center}} \\
    \text{offset} &= \frac{\text{base}}{2} \times \frac{z}{|z|} \\
    \text{eye}_{L,R} &= \text{eye}_{\text{center}} \pm \text{offset} \\
    \text{target}_{L} &= \text{target}_{R} = \text{target}_{\text{center}}
\end{align*}
\]

(33) \quad (34) \quad (35) \quad (36)

Additionally, the **parallel case** implies that the camera targets have to be relocated, too:

\[
\begin{align*}
    \text{target}_{L,R} &= \text{target}_{\text{center}} \pm \text{offset}
\end{align*}
\]

(37)

If I simply change the sign of the offset vector, I can switch the roles of the left and right cameras.

For the **anaglyph method** I have to generate a composite picture, as this one separates the left and right source images only in spectral domain (i.e. with color-filters). In my application for each frame I render the left and right images into dynamic textures, which have the required resolution that matches the display. In D3D these dynamic textures can be stored in the fast video memory. After setting up these new rendering targets, and adjusting the virtual camera to the view of the given eye, I can render 1-1 image into those textures using the usual way (through the programmable pixel pipeline using shaders). As I would like to composite the result image, finally I need to render a single rectangle (i.e. two triangles) that fills exactly the entire screen, thus I can calculate the color of each pixel of the result image using a simple and fast pixel shader.

When \( c_1 \) and \( c_2 \) are the source pixel colors (in RGBA order) at the current position, I use the following formula in the compositor pixel shader to calculate the color of the current pixel (Note that I neglect transparency here):

\[
c = \begin{bmatrix} c_{1R} & c_{2G} & c_{2B} & 1 \end{bmatrix}
\]

(38)

When I use the **LCS method** I render the image only from one eye’s perspective for each frame. As I can directly render to the screen (like I did in the monocular
case), this implies that this technique requires much less resource than the anaglyph one.

The LCS glasses have an infrared control hardware using the “vertical sync” signal of the graphics card’s video output. This synchronizes the switching between transparent and opaque states of the lenses. I also use this in my software, and tell the D3D device to swap the double-buffers and present the new frame right at the same time.

With this technique each time a frame is rendered, in the next frame the other eye’s view comes into consideration. I allow switching between the L-R and R-L cases with a keyboard hit, as being unable to predict which shutter is the opaque one, while presenting the first image destined for the left eye.

The last binocular technique is using the circular polarized lenses. As there are two projectors equipped with polarized filters displaying their own image on the same canvas, this method requires each frame rendering the scene twice, from both eyes’ perspective. I have to present these images simultaneously on the projectors connected to the two video outputs of the graphics card. For this purpose I have to use the dual monitor configuration in the display driver.

I encountered a problem that is in connection with the D3D API being used in the application. Namely, it is not possible to have a rendering device operating on more than one display in fullscreen mode. In the managed D3D API only a single fullscreen device can exist at a time: the one that can own the graphics hardware exclusively. Each time I want to use the polarizer glasses technique, my solution is to switch back to windowed mode, and create a rendering window, which has a height value that matches the native vertical resolution \((H)\), but has a width value that is two times the native horizontal resolution \((W)\) of a single display (i.e.: \(2560 \times 1024\) pixels when using \(1280 \times 1024\) resolution on the projectors). First I define a viewport with upper-left corner at \((0; 0)\) with dimensions \(W \times H\). I render the scene to that viewport the usual way, and then relocate the viewport to position \((H; 0)\) while its size remains \(W \times H\). From the other eye’s perspective I render the scene once again into the new viewport’s back buffer. As the \(2W \times H\) big window stretches exactly over the two displays, each half (a viewport’s image) will finally be shown on its representative monitor (projector).

Finally I have to note that in windowed mode no fixed framerate can be guaranteed, so this is not applicable for the shutter method (that needs exclusive fullscreen usage of the graphics hardware, because the goggles are synchronized with the screen vertical refresh signal). The anaglyph and motion-parallax methods in the application run always in full-screen mode, but this is only a convenience.
Scripting the psychophysical experiments

I let the group of persons evaluate a series of 3D scenarios in the aspect of 3D spatial impressions. During these psychophysical experimental tests the subsequent user input was disallowed and some of the scenario-parameters (3D scene, visualization technique, monocular camera / binocular camera with eye-base distance) were controlled by the script interpreter module of my application.

During each experiment a pre-generated text file is being read line by line to generate new 3D scenarios that will be presented next time to the user. He / she might be asked to take either the anaglyph glasses (red-cyan lenses), the LCS or the polarizer goggles on or off, before the new setting is shown. The user is asked to give scores for each setup, and the response is logged by the software into a file. After all experiments have finished, this made the offline evaluating with Matlab possible. It is known that sometimes simply the fact the user has to wear extra “placebo” goggles can also be an influencing factor for subjective judgment. Thus I generate also scenarios, where the left and right virtual cameras are coincident (i.e.: have zero horizontal disparity), and I still ask the user to wear the goggles, in order to analyze this effect.

4.7 The psychophysical experiment

I discuss the introduced psychophysical experiment here, which was done at the department a few years ago with the kind help of my colleagues. Many tests were successfully carried out on smaller groups of voluntary people: altogether we could count 44 participants.

4.7.1 The goal

The goal of the experiments was to quantize the psychophysical perception of depth delivered by the aforementioned 3D techniques combined with motion. While other researchers concentrated on the efficiency of 3D techniques in combination of the shadowing [25], or other particular applications [28], I investigated the efficiency of these techniques combined with the motion-parallax effect, which is always present between the environment and the dynamic mobile platform in case of teleoperation.

4.7.2 The environment

A small basement-room has been furnished to accomplish the experiment. The room contained a desk with a chair and the accessories of the experiments. At a particular time, only one participant and the experiment leader stayed in the room. The specification of hardware accessories was the following:
• PC: AMD Athlon64 Venice 3000+ CPU, 1024 MB memory, Inno3D Geforce 6600, PCI-E, 256 MB video card
• Projectors: Toshiba DLP (two pieces), 2 circular polarizing filter one per each (one polarizing clockwise and one counter-clockwise)
• Polarizer glasses (also one CW and one CCW)
• LCS glasses with infrared synchronizer
• Anaglyph glasses
• Polarization preserving canvas

Figure 43: The test room for the experiment

Both projectors have been installed with a circular polarized filter, so two images with different polarization are projected to the canvas. Through the polar filter glasses, both eyes can see only one image. Most surfaces modify the polarization of the incident light (usually by scattering). However, in this application this would cause an unusable result. There are polarization preserving canvases on the market, but they are rather expensive. By examining some inexpensive surfaces we found that cardboards coated with silvery paint retain polarization sufficiently, so we prepared the reflective canvas surface using this kind of cardboards.

4.7.3 The measurement model

As the depth perception is a subjective phenomenon, you could ask whether it is measurable at all or whether it is feasible to compare the results of multiple measurements. I will show that it is possible to build an appropriate measurement model.

The first problem is that people cannot quantize exactly their depth experiences even on an exactly defined scale, thus it can happen that one estimates the same experience differently at a later time. If a person’s actual subjective opinion is asked more than once, it can happen easily that the subsequent answers might be different, thus introducing a noise into the measurement. It is supposed that it is a noise with zero expected value.

A bigger issue is that the same scene generates different depth sensation level to the observer under the same circumstances but at a later time (i.e.: a repeated
scenario). The first impression with a novel technique may cause overestimation of the experience. Therefore, the measured value depends on the preceding (partially memorized) scenes viewed by the experimental person. Therefore the internal state of the observer person had to be taken into account, too.

I have presumed that the depth perception caused by every 3D scenario has an **intrinsic** component, which is independent from the observer person (and from his internal mental state caused by previous scenes) and defined as the expected value of the measurement of that scene.

A **constellation** is an objective observing situation, including all properties of the actual visualization technique, the 3D scene that is shown and other circumstances of the measurement, but the observer him/herself is not included. Figure 44 shows the measurement model of a single experiment:

![Measurement model diagram](image)

**Figure 44: The measurement model**

- \( x \) denotes the parameters of the constellation (which technique is used, is the virtual camera moving, is there a binocular disparity: see later, in section 4.7.4)
- \( c \) denotes all the other parameters of the constellation, which affect the intrinsic depth perception level and we have to set them constant during the series of experiments (for example ambient lighting).
- The quantity \( I \) denotes the intrinsic depth perception level. This value has to be evaluated at different constellation parameter values (\( x \)).
- The aggregated measurement noise (\( \xi \)) depends on the internal state of the observer (\( q(t) \)) the environmental effects (\( u(t) \)). Both components are time-varying (\( t \)).
- Every person has his internal habits a personal bias (\( b \)) and a personal interpretation of scale (\( a \)). These individual properties are assumed to be constant for each individual and they cause a linear transformation of the intrinsic depth sensation.
In this series of experiments the goal was to compare the intrinsic depth perception level of the different constellations by asking the observer person to give scores between 1 and 9 (the bigger the score the better the quality).

The questionnaire focused on two points:

- which depth effect quality level does a given constellation have $\Rightarrow z$
- how does it compare to the previous constellation $\Rightarrow z_{prev}$ (whereby $q$ is influenced, too)

Based on the model above, the following methods were used to eliminate the distortions:

- All observers have been brought to the same initial $q_0$ state in the beginning of an experiment by giving them the same a-priori information about the experiments (what does the questionnaire focus on, what kind of devices are involved, how long time will the experiment take)
- The other outer, environmental effects ($u$) were reduced by performing the experiments in a silent dark room, where only the projected image was visible.
- The vectors $x$ and $q$ were assumed to be independent.
- The $c$ constant parameters of the constellation were kept at the same level during the whole series of experiments.
- The order of successive constellations has been designed in a way that the distortion effects could be eliminated with statistical methods.

## 4.7.4 The constellation parameters

The $x$ constellation parameter vector can be split in three factors:

**Applied stereoscopic technique:** $x_1$
- Anaglyph technique (A)
- Shutter technique (S)
- Dual stereo technique (D)
- Normal monocular technique (M)

**Binocular cue:** $x_2$
- No binocular cue (coincident cameras) (0)
- Binocular cue with convergent camera model (T)
- Binocular cue with parallel camera model (P)

**Dynamism** $x_3$
- Moving scene (R)
- Still scene (N)
The following figure shows the domain of \( x \) as a graph (\( GX \)), with 20 labelled vertices and 52 edges. As all \( x_i \) factors are taken independent, there can be \( 4 \times 3 \times 2 = 24 \) combinations of \( x \). Note that the missing four vertices (MTN, MTR, MPN, MPR) denote illegal factor-combinations (by the definition of the \( M \) case – independent from camera movement – there can’t be any binocular cue: \( T \) or \( P \)). Adjacent vertices \( x_a \) and \( x_b \) (connected by an \( x_a \rightarrow x_b \) edge) represent two \( x \) parameter vectors that differ only in a single factor (i.e.: they have one Hamming distance).

![Figure 45: The GX graph in constellation parameter space](image)

Table 3 shows the factor-label assignment of \( GX \) on the left hand side and the adjacency matrix of \( GX \) on the right hand side.

| \( X_1 \) | \( X_2 \) | \( X_3 \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| A | 0 | N | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| A | 0 | R | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| A | T | N | 2 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| A | T | R | 3 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| A | P | N | 4 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| A | P | R | 5 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| S | 0 | N | 6 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| S | 0 | R | 7 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| S | T | N | 8 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| S | T | R | 9 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| S | P | N | 10 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| S | P | R | 11 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| D | 0 | N | 12 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| D | 0 | R | 13 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| D | T | N | 14 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| D | T | R | 15 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |

Table 3: Labelling and adjacency matrix of the GX parameter graph
These and others have been independently scheduled.

By defining the order of the constellations the following criteria had to meet:

- The intrinsic depth perception level has to be evaluated once in all of the vertices of $GX$: each constellation (a node of the graph) is shown once to each observer (a full factorial design is applied).
- As the questionnaire asks to compare the currently viewed scene to the previous one it is recommended that the successive constellations should differ only in a single factor: thus they should be adjacent.

These two criteria demand Hamiltonian paths in $GX$ for each observer. It was pointed out in section 4.7.3 that $x$ and $q$ have to be independent. This means that both the initial constellation (vertex in the parameter space graph) and the subsequent ones (over individual edges in the graph) had to take place independently and randomly as the order of scenes is designed for an observer.

A heuristic algorithm for searching Hamiltonian paths in the $GX$ graph fulfilling this independency criterion has been developed as a Matlab script. A HP Hamiltonian path in $GX$ – by definition – contains 19 edges and visits all 20 vertices exactly once.

Although in $GX$ there is not enough HP, I could find many quasi-Hamiltonian paths (QHP). Such a QHP contains two Hamiltonian paths from both $x_{start}$ to $x_i$ and from $x_i$ to $x_{end}$, where the $x_i$ and $x_j$ vertices are not necessary neighbours (it is not a real edge of $GX$).

Table 4 shows the order of constellations (each row is a QHP in $GX$) for the first 10 voluntary people. A row represents an order in which the given labelled constellations (refer to Table 3) must be presented to a given $O$ observer.

### Table 4: Design of experiments

<table>
<thead>
<tr>
<th>$O$</th>
<th>Constellation label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>17 13 12 14 8 10 6 7 9 11 5 3 1 19 18 0 2 4 16 15</td>
</tr>
<tr>
<td>2.</td>
<td>15 13 19 1 5 4 0 6 12 16 10 8 14 2 3 9 7 11 17 18</td>
</tr>
<tr>
<td>3.</td>
<td>18 6 10 11 7 19 13 1 0 12 14 16 17 15 9 3 2 4 5 8</td>
</tr>
<tr>
<td>4.</td>
<td>8 9 15 14 12 18 19 7 1 3 5 11 10 6 0 4 16 17 13 2</td>
</tr>
<tr>
<td>5.</td>
<td>2 8 6 18 12 13 7 9 3 15 17 11 5 1 0 4 10 16 14 19</td>
</tr>
<tr>
<td>6.</td>
<td>19 7 6 8 2 0 18 12 16 4 5 17 15 13 1 3 9 11 10 14</td>
</tr>
<tr>
<td>7.</td>
<td>14 15 3 5 2 0 1 7 13 17 16 12 18 6 8 10 11 9 19</td>
</tr>
<tr>
<td>8.</td>
<td>19 18 0 6 10 4 2 14 15 17 16 12 13 7 11 5 3 9 8 1</td>
</tr>
<tr>
<td>9.</td>
<td>1 13 15 14 16 10 4 0 12 6 18 19 7 9 8 2 3 5 11 17</td>
</tr>
<tr>
<td>10.</td>
<td>17 5 1 7 6 0 18 19 13 15 9 11 10 8 14 2 4 16 12 3</td>
</tr>
</tbody>
</table>
4.7.5 Preprocessing the results

The first question of the questionnaire demanded from the observers to give scores on an absolute scale from 1 to 9, where 9 was the best quality of perceived depth effect.

Every person has own habits: a personal bias (b: depending on how likely he uses bigger numbers, 0=no offset) and a personal interpretation of scale (a: scale factor for likelihood of overvaluing things, 1=normal). These properties are assumed to be constant for each individual and they cause a linear transformation of the results on the first questions’ answers $z_{abs}$.

All observers ($O_0...O_{43}$, $N=44$) have seen the same set of 20 constellations ($x_0...x_{19}$), and gave answers for the first question in the following $Z$ matrix:

$$Z = \begin{bmatrix} z_{0,0} & z_{0,1} & \cdots & z_{0,19} \\ z_{1,0} & z_{1,1} & \cdots & z_{1,19} \\ \vdots & \vdots & \ddots & \vdots \\ z_{43,0} & z_{43,1} & \cdots & z_{43,19} \end{bmatrix} \quad (39)$$

By analysing the $Z$ matrix – containing all the answers of each observer – the subjective personal distortion of the individual parameters $a$ and $b$ has to be determined and eliminated. A given constellation should have the same objective mean and variance in case of each person after normalization:

The subjective answers of the person $O_i$ (row of the $Z$ matrix) are in vector $z_i$:

$$z_i = \{ z_{i,0}, z_{i,1}, \ldots, z_{i,19} \} \quad i \in [0, N-1] \quad (40)$$

The expected value and the variance of those are the following:

$$\mu_i = E(z_i) = E(z_{i,j}) \quad j \in [0, 19]$$

$$\sigma^2 z_i = E((z_i - \mu_i)^2) \quad (41)$$

![Figure 46: Distribution of absolute, subjective answers](image.png)
On Figure 46 it can be seen how the mean and standard deviation of the answers of a first few observers looked like, according to (41), including the confidence intervals for 90% significance level. As all observers have seen all constellations, the differences of individual mean and deviation values are modelled as the effect of two personal factors (a and b).

After eliminating the effect of these distortion factors, the intrinsic (i.e.: objective) mean and variance values have to be the same for each observer.

The mean of all answers gives an estimation:

$$\mu_{\text{average}} = E(Z) = 5.263$$

The variance of all answers (N observers are taken to be independent):

$$\sigma^2_{\text{average}} = \frac{\sum_{i=0}^{N-1} \sigma_{z_i}^2}{N} = 2.42^2$$

The average mean and variance values are taken as default when a=1 and b=0.

The variance of the final results has two factors: an intrinsic factor \(\sigma_i^2\) and a \(\xi\) noise factor \(\sigma_{\xi}^2\). The average of all answers estimates the average value of the intrinsic depth perception level, while the standard deviance estimates the effect of the average aggregated noise affected by the individual scales of the observer persons.

As \(\xi\) has zero expected value, and is independent from the constellations, the variance of the result is the sum of these variance factors.

The normalization for one person (subjective \(\rightarrow\) objective transformation) is done using the following formula:

$$z_{i,\text{norm}} = \mu_{\text{average}} + (z_i - \mu_{\text{average}}) \frac{\sigma_{\text{average}}}{\sigma_{z_i}} \quad i \in [0, N-1]$$

In the model the objective (intrinsic) depth effect of a constellation is taken as the mean value of all observers’ normalized opinion:

$$z_{\text{norm}} = E_i(z_{i,\text{norm}}) \quad i \in [0, N-1]$$

4.7.6 Analysing the measurements

First I have analysed the question on the objective depth sensation delivered by various constellations.
**Absolute values – Intrinsic depth:**

According to the formula (45) the intrinsic depth sensation quality can be found in the 20 components of the $z^{\text{norm}}$ vector for each constellation, respectively. The following diagrams and table show the mean value of the normalized answers:

![Diagram](image1.png)  ![Diagram](image2.png)

**Figure 47: Answers for static and dynamic scenes**

**Table 5: Intrinsic depth perception levels**

<table>
<thead>
<tr>
<th>F(X1,X2,N)</th>
<th>0</th>
<th>T</th>
<th>P</th>
<th>F(X1,X2,R)</th>
<th>0</th>
<th>T</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3.08</td>
<td>4.24</td>
<td>4.40</td>
<td>A</td>
<td>5.23</td>
<td>5.82</td>
<td>5.93</td>
</tr>
<tr>
<td>S</td>
<td>3.32</td>
<td>5.29</td>
<td>5.83</td>
<td>S</td>
<td>5.99</td>
<td>6.78</td>
<td>7.17</td>
</tr>
<tr>
<td>D</td>
<td>3.87</td>
<td>6.44</td>
<td>6.64</td>
<td>D</td>
<td>5.38</td>
<td>7.13</td>
<td>8.10</td>
</tr>
<tr>
<td>M</td>
<td>2.97</td>
<td>-</td>
<td>-</td>
<td>M</td>
<td>4.79</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The smallest value is to be found at $F(M,0,N)=2.97$, at the still, monocular scenario. Surprisingly, the anaglyph technique has performed relatively weak, compared to [48]. One reason can be the bad quality spectral separation of the cheap plastic anaglyph filters that were involved in those cases.

According to the overall values you can see that my parallel camera model was more successful at the projected scenes – it got bigger average scores – than the convergent model. This was due to the big viewing distance between the observer and the canvas: naturally, when looking into the distance, the eyes tend to have parallel axes that do not cross at the object plane (which is in focus), but only much further on the horizon.

On a scale from 1 to 9 the motion ($F(M0N)\rightarrow F(M0R)$) improves +1.8 units in the depth perception levels compared to a still scene. Using auxiliary goggles the $F(X1,X2,R) - F(X1,X2,N)$ differences lay at $+1.8\pm0.9$ units.

The difference caused by binocular information using the $P$ projection model ($F(X1,0,N) - F(X1,P,N)$) is at $+2.5\pm1.2$ units. Thus it can be seen that the visual depth information presented by the motion-parallax technique is in many cases as big as that of the binocular visualization methods.

It is also obtained that by simply wearing a “placebo” auxiliary spectacle (in A, S or D modes) the observer might have an illusion he / she perceives binocular information, although only a central $0$ camera model was used in six constellations covering this psychic effect (e.g.: $S0N$ or $A0R$). These
constellations gained in average +0.4 units more with “placebo” goggles than without one.

Investigating the split of the results one can state that the small and big results have smaller, the middle results have greater standard deviance. The reason might be that the observer persons agreed in the obviously good and bad depth sensation results and were split by the middle ones but this effect can be interpreted with the saturation of the representation scale.

Adjacent scenarios – relative values:

In this case the answers lay also between 1 and 9; 5 was the neutral answer. The results were transformed later into the ±4 interval (0=neutral). These answers gave me a balanced set of information about the edges of GX. Although my heuristic (quasi)Hamiltonian path-search algorithm used random start nodes, each observer could score only 18 real edges in GX, while it has 52 edges altogether. Nevertheless the comparison of adjacent constellations was directly possible using these answers, as each edge of GX relates to a single factor change in the x input vector of constellations.

The expected values of the individual answers are not the same therefore these result values required another type of pre-processing. All these answers are related to an edge of the adjacency graph; by checking an opposite direction one could expect an opposed value. Therefore I built a statistics from these values with corrected sign.

The results can be summarized in the followings. The mean value of the scores of the 10 edges related to the change of the x3 factor (N→R) gave the result that the motion-parallax effect scored +2.05 (± 0.39 @ 90%) units more, thus improving the quality of depth perception. Similarly, by analysing the x1 factor, the usage of binocular visualization improves +1.91 (± 0.56 @ 90%) units.

4.8 Integrated visualization in RobotMax

The goal of the development of my RobotMax tool – which I already introduced in previous chapters – was to create an integrated mechatronic model authoring, simulation and visualization environment. I created two coupling modes between Dymola and my application in the following ways:

- Offline visualization: using the .mat output of Dymola, after the simulation has been completely ended
- Online visualization: using standard shared memory communication while the simulation is still running, offering interaction with the simulation
4.8.1 Offline visualization

During the simulation, Dymola can export the values of all signals existing in the model (including output-, auxiliary- and state variables + time-derivatives) either in Matlab binary or ASCII text format. Unfortunately this output file is exclusively locked until the simulation terminates, thus it is not possible to start parsing the results during the simulation from any 3rd party application.

As soon as the lock on the output file is released, RobotMax can load the given file and create keyframes for each signal with a given granularity (the maximum is the sample rate Dymola has stored the signal values). In this way the user can play an animation multiple times easily, while inspecting the result from various viewing angles, but here the simulation itself is not interactive.

4.8.2 Online visualization

The most recent improvement of RobotMax is the support for parallel, online visualization during the simulation, by establishing interprocess communication with Dymola over a shared memory module. It is used as a temporary buffer to hold the actual values of simulation signals to be presented.

I use semaphores to control parallel access of this shared memory from these external modules. On one side the data is fed from the Dymola / Modelica world (through write operations from my rigid body and joint models, storing their selected signals that are important to visualize), on the other side RobotMax continuously polls the SM to retrieve the signal’s actual value and refresh its internal model and re-render its viewports. Note that the writings can be much more frequent (e.g.: every 1 ms) than the read operations (e.g.: every 20 ms for 50 Hz refresh rate).

In this operation mode the visualization runs parallel with the simulation, thus it is suitable for displaying the real-time behaviour of the simulated mechatronic system, while the user can interact with the system, influencing the simulation, too.

4.9 Conclusion

Using statistical analysis of measurements gained from a series of psychophysical experiments I determined the quantitative strength of the motion among the depth clues of various visualization techniques.

Consequently, in a dynamic environment the motion parallax effect can compare to the effect of real stereoscopic visualization methods in creating depth impressions, thus in the application field of teleoperation the support of this monocular technique can be satisfactory.
Chapter 5

Outlook

5.1 A trend in virtual engineering

The tendency of integrating multi-domain simulation inside a modern CAE system started to grow recently. Dassault Systèmes (DS) has announced a strategy to make CATIA the world leader of integrated Product Lifecycle Management solutions. They have selected the open standard Modelica to be at the core of their open strategy. Hence in the near past they announced the acquisition of Dynasim: the Swedish company that is developing the Dymola product suite and is a leader in Modelica based modelling and simulation solutions. Albeit this acquisition the leaders of DS promised that Dymola will be kept as a stand-alone product in the future.

5.2 Further work

5.2.1 Pro/Toolkit: API for Pro/Engineer

An Application Programmers Interface (API), Pro/Toolkit allows Pro/Engineer functionality to be augmented and/or customized to meet the specific needs of PTC’s customer base using the "C" programming language. Pro/Toolkit provides the ability to customize the standard Pro/Engineer user interface, automate processes involving repetitive steps, integrate proprietary or other external applications with Pro/Engineer and develop customized end-user application for model creation, design rule verification or drawing automation. Unfortunately, Pro/Toolkit deservedly gained a reputation of being hard to work with. This state of the affairs is the result of PTC’s current vision, which dictates where and how this product ought to augment the functionality of Pro/Engineer. This vision, as a series of articles have proved [55], offers by no means the ideal toolkit users would hope for; rather, it leaves quite a few features to be desired.

5.2.2 Task 1: Complete workflow within Pro/Engineer

Using the Pro/Toolkit API I will make an effort to integrate the translation process (related to Thesis 1, presented in Chapter 2) into the Pro/Engineer environment, as a plug-in module.

For the purpose of visualizing the simulation results, an interface has to be designed between the new Pro/E module and Dymola: inter-process communication could make integrated design and optimization possible within Pro/Engineer, resulting in an even smoother virtual engineering workflow.
5.2.3 Task 2: Improve contact processing accuracy

Instead of the requirement of being (at least soft-) real-time, the accuracy of the mechatronic simulation can also have a priority. Besides finite element methods, there are other accurate methods to formulate the contact response between colliding objects. Instead of penetration depth and -velocity, the penetration area (e.g.: in the Hertz model [21]) or volume must be determined in such cases.

It has to be investigated, whether the accurate intersection surface / volume of two given construction parts can be determined on demand, using the Pro/Toolkit API during a simulation. If it will be supported, a more accurate contact response module could be integrated in the functional model of the construction (refer to Thesis 2 and Chapter 3 of this work).

5.2.4 Task 3: New Pro/Engineer components

In the latest Wildfire Pro/Engineer versions there is a possibility to add new components to an assembly, though those are not rigid bodies. The new elements can be springs, dampers or motors, but with rather basic parameterization, for this time. These components could also be incorporated in the model translation process, if there were support in the Pro/Toolkit API for their extraction. This effort (just like the previous tasks) will definitely need cooperation with the developers of the API.

5.3 Conclusion

These days the digital integration in virtual engineering has even more important role in the product lifecycle management: from the computer-aided design and -production planning to -manufacturing, it offers wide range of possibilities in the fields of analysis and optimization. The digital revolution establishes a competition on the market of small and medium-sized enterprises (SME), while presenting new challenges to them. It is often the case that the smaller undertakings in the supply chain do not have enough expertise in modelling and simulation of complex mechatronic systems, whereas it would be expected even more by their partner companies nowadays.

The scope of my work is along a well-defined virtual engineering workflow. Following the natural order of tasks in this process I present a highly automated solution to help the designers verify the product using multi-domain simulations without much manual effort. The presented workflow starts from the Pro/Engineer system, where the mechanical construction of a product is designed.
5.3.1 Significance of achievement

- **Thesis 1** (on page 4) declares that I created an automated translation method from a CAD construction to object-oriented Modelica models, which can be extended with components from other engineering domains in an independent way, making multi-domain optimization possible.

- **Thesis 2** (on page 44 and 58) states that in the mechanical domain I also introduced contact processing in Modelica using a penalty force-based approach on multi-shape rigid bodies, with temporary spring and damper elements and material-based response calculation. Using a real case study on a six-legged robot I validated my contact model with real sensor data.

- **Thesis 3** (on page 67) affirms that in the application of virtual teleoperation trainings the less resource-demanding monocular technique with the motion-parallax effect can be as strong as the intrinsic depth effect of other, relatively expensive stereoscopic visualization methods.

I created an own integrated mechatronic model authoring and visualization tool to encompass all of my research results under a common software environment. In an interactive way it allows the visualization of a real time mechatronic simulation that can run in the background.

*The solutions of my dissertation could make the design and optimization process more efficient, especially for the SME in the supply chain, which could be more competitive on the digital market in this way. These undertakings (without much physical modelling expertise) can better cooperate with the large factories, when they offer the model of their product (which has already been verified and optimized in the virtual reality using multi-domain simulation) far earlier than the physical production would begin. By integrating the supplied virtual components, a digital mechatronic system can be planned and optimized along various criteria in advance.*
Related publications


Other publications


Bibliography


[43] Qian, N.: “Binocular disparity and the perception of depth”; Neuron 18, 1997


[47] Vieth.: “Über die Richtung der Augen”; 1818


[59] RobotsLab – http://www.uni-magdeburg.de/ieat/robotslab


Appendices

Glossary

BUTE Budapest University of Technology and Economics
CAD Computer Aided Design
CAE Computer Aided Engineering
CAM Computer Aided Manufacturing
CRT Cathode Ray Tube
DAE Differential Algebraic Equation system
FEM Finite Element Method
LCD Liquid-Crystal Display
LCS Liquid-Crystal Shutter (glasses)
LED Light Emitting Diode
MBS Multibody System (interconnected set of rigid bodies and joints)
ODE Ordinary Differential Equation system / Open Dynamics Engine
PMXML Physical Modelling XML descriptor
Pro/E The Pro/Engineer system
PTC Parametric Technology Corporation
QHP Quasi Hamiltonian Path
SME Small- and Medium-sized Enterprises
SOLID Software Library for Interference Detection
SWIFT Speedy Walking via Improved Feature Testing
VRML Virtual Reality Modelling/Markup Language
XML Extensible Markup Language
Full results

The XML schema of the Object Library

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="object_library">
    <xs:annotation>
      <xs:documentation>Object Library</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="object_folder" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="object_folder">
    <xs:annotation>
      <xs:documentation>Object Folder</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="object_class" minOccurs="0" maxOccurs="unbounded" />
        <xs:element ref="object" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="model" type="xs:string" use="optional" />
    </xs:complexType>
  </xs:element>

  <xs:element name="object_class">
    <xs:annotation>
      <xs:documentation>Object Class</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded" />
        <xs:element ref="object" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="model" type="xs:string" use="optional" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```
<xs:element name="parameter">
  <xs:annotation>
    <xs:documentation>Class Parameter</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="required" />
    <xs:attribute name="type" type="xs:string" use="required" />
    <xs:attribute name="editable" type="xs:boolean" use="required" />
    <xs:attribute name="unit" type="xs:string" use="optional" />
    <xs:attribute name="comment" type="xs:string" use="optional" />
    <xs:attribute name="min" type="xs:float" use="optional" />
    <xs:attribute name="max" type="xs:float" use="optional" />
  </xs:complexType>
</xs:element>

<xs:element name="object">
  <xs:annotation>
    <xs:documentation>Object Instance</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="parameter" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" use="required" />
          <xs:attribute name="value" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
The HLSL shaders of *StereoVision* and *RobotMax*

```cpp
// Global variables
//------------------------------------------------------------------------------
// World transformation
float4x4 m_World;
// View * Projection transformation
float4x4 m_ViewProj;
// texture transformation
double m_Texture;
// texture amount
float4 m_TAmount;
// world space eye position
int n_Lights;
// world space light positions
float4 m_Light[8];
float4 m_LCol[8];

texture tex1;
texture tex2;
texture tex3;

float4 m_Ambient;
float4 m_Diffuse;
float4 m_Specular;
float4 m_Emission;
double m_Power;

uniform int column[15] = {0,2,3,5,7,0,2,4,5,7,0,2,4,5,7};

sampler tex1Sampler =
sampler_state {
    Texture = <tex1>;
    MipFilter = LINEAR;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

sampler tex2Sampler =
sampler_state {
    Texture = <tex2>;
    MipFilter = LINEAR;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

sampler2D tex3Sampler =
sampler_state {
    Texture = <tex3>;
    MipFilter = POINT;
    MinFilter = POINT;
    MagFilter = POINT;
    AddressU = WRAP;
    AddressV = WRAP;
};
```
struct VS_GOURAUD {
    float4 Position : POSITION;
    float3 UV : TEXCOORD0;
    float4 Data0 : COLOR0;
    float3 Data1 : TEXCOORD1;
    float3 Data2 : TEXCOORD2;
};

struct VS_PHONG {
    float4 Position : POSITION;
    float3 UV : TEXCOORD0;
    float3 Normal : TEXCOORD1;
    float3 View : TEXCOORD2;
    float3 Pos : TEXCOORD3;
};

struct VS_COLOR {
    float4 Position : POSITION;
    float4 RGBColor : COLOR0;
};

struct VS_QUAD {
    float4 Position : POSITION;
    float2 UV : TEXCOORD0;
};

VS_GOURAUD GouraudVS(float4 vPos : POSITION,
                      float3 vNorm : NORMAL,
                      float2 vTCoord : TEXCOORD0) {
    VS_GOURAUD Out;
    float4 P = mul(vPos, m_World);
    float4 N = normalize(mul(vNorm, m_World));
    float4 L1 = normalize(m_Light[0] - P);
    float4 L2 = normalize(m_Light[1] - P);
    float4 V = normalize(m_Eye - P);
    Out.Position = mul(P, m_ViewProj);
    Out.UV = mul(float3(vTCoord, 1), m_Texture);
    double diff1 = max(0, dot(N, L1));
    double diff2 = max(0, dot(N, L2));
    float3 R1 = normalize(2 * diff1 * N - L1);
    float3 R2 = normalize(2 * diff2 * N - L2);
    double spec1 = pow(max(0, dot(R1, V)), m_Power);
    double spec2 = pow(max(0, dot(R2, V)), m_Power);
    Out.Data0 = m_Emission;
    Out.Data1 = float3(diff1, diff2, 0);
    Out.Data2 = float3(spec1, spec2, 0);
    return Out;
}
float4 GouraudPS(float3 UV : TEXCOORD0,
float4 Data0 : COLOR0,
float3 Data1 : TEXCOORD1,
float3 Data2 : TEXCOORD2) : COLOR0 {
float4 color = lerp(m_Diffuse, tex2D(tex1Sampler, UV), m_TAmount);
    color = Data0 + color * (m_Ambient + m_LCol[0] * Data1.x + m_LCol[1] * Data1.y);
    color += m_Specular * (m_LCol[0] * Data2.x + m_LCol[1] * Data2.y);
    return color;
}

VS_PHONG PhongVS(float4 vPos : POSITION,
float3 vNorm : NORMAL,
float2 vTCoord : TEXCOORD0) {
    VS_PHONG Out;
    float4 P = mul(vPos, m_World); // vertex world position
    Out.Pos = P;
    Out.Position = mul(P, m_ViewProj);
    Out.UV = mul(float3(vTCoord,1), m_Texture);
    Out.Normal = normalize(mul(vNorm, m_World));
    Out.View = normalize(m_Eye - P);
    return Out;
}

float4 PhongPS( float3 UV : TEXCOORD0,
float3 N : TEXCOORD1,
float3 V : TEXCOORD2,
float3 P : TEXCOORD3) : COLOR0 {
float4 color = lerp(m_Diffuse, tex2D(tex1Sampler, UV), m_TAmount);
float4 res = m_Emission + color * m_Ambient;
    N = normalize(N);
    V = normalize(V);
for (int i = 0; i < n_Lights; i++) {
    float3 L = normalize(m_Light[i] - P);
    float diff = max(0, dot(N, L));
    float3 R = normalize(2 * diff * N - L);
    float spec = pow(max(0, dot(R, V)), m_Power);
    res += m_LCol[i] * color * diff;
    res += m_LCol[i] * m_Specular * spec;
}
    return res;
}
VS_COLOR ColorVS(float4 vPos : POSITION, 
                        float4 vColor : COLOR) {

    VS_COLOR Out;

    Out.Position = mul(mul(vPos, m_World), m_ViewProj);
    Out.RGBColor = vColor;

    return Out;
}

VS_COLOR ColorUnlitVS(float4 vPos : POSITION) {

    VS_COLOR Out;

    Out.Position = mul(mul(vPos, m_World), m_ViewProj);
    Out.RGBColor = m_Diffuse + m_Emission;

    return Out;
}

VS_QUAD QuadVS(float4 vPos : POSITION, 
                    float2 vTCoord : TEXCOORD0) {

    VS_QUAD Out;

    Out.Position = vPos;
    Out.UV = vTCoord;

    return Out;
}

float4 X3DPS(float2 UV : TEXCOORD0) : COLOR0 {

    double y = UV.y * 1024;
    double x = UV.x * 1280;

    double TR = 3 * floor(fmod(x, 5));
    double rT = floor(fmod(column[TR] + y, 8));
    double gT = floor(fmod(column[TR + 1] + y, 8));
    double bT = floor(fmod(column[TR + 2] + y, 8));

    double rowR = floor(rT / 2.0);
    double colR = rT - 2.0 * rowR;
    double rowG = floor(gT / 2.0);
    double colG = gT - 2.0 * rowG;
    double rowB = floor(bT / 2.0);
    double colB = bT - 2.0 * rowB;

    float4 col = 0;
    float2 M = float2(0.5, 0.25);

    col.r = tex2D(tex3Sampler, M * (UV + float2(colR, rowR))).r;
    col.g = tex2D(tex3Sampler, M * (UV + float2(colG, rowG))).g;
    col.b = tex2D(tex3Sampler, M * (UV + float2(colB, rowB))).b;

    return col;
}
// AnaglyphPS(float2 UV : TEXCOORD0) : COLOR0 {
    float4 col1 = tex2D(tex1Sampler, UV);
    float4 col2 = tex2D(tex2Sampler, UV);
    return col1 * float4(1, 0, 0, 1) + col2 * float4(0, 1, 1, 1);
}

// Default technique
technique Default {
    pass P0 {
        Sampler[0] = (tex1Sampler);
        MaterialAmbient = (m_Ambient);
        MaterialDiffuse = (m_Diffuse);
        MaterialSpecular = (m_Specular);
        MaterialPower = (m_Power);
        VertexShader = null;
        PixelShader = null;
    }
}

// Colored technique
technique Colored {
    pass P0 {
        VertexShader = compile vs_1_1 ColorVS();
    }
}

// Wireframe technique
technique Wireframe {
    pass P0 {
        VertexShader = compile vs_1_1 ColorUnlitVS();
    }
}

// Gouraud technique
technique Gouraud {
    pass P0 {
        VertexShader = compile vs_1_1 GouraudVS();
        PixelShader = compile ps_1_4 GouraudPS();
    }
}

// Phong technique
technique Phong {
    pass P0 {
        VertexShader = compile vs_1_1 PhongVS();
        PixelShader = compile ps_3_0 PhongPS();
    }
}

// Anaglyph technique
technique Anaglyph {
    pass P0 {
        VertexShader = compile vs_1_1 QuadVS();
        PixelShader = compile ps_1_4 AnaglyphPS();
    }
}

// X3D technique
technique X3D {
    pass P0 {
        VertexShader = compile vs_1_1 QuadVS();
        PixelShader = compile ps_3_0 X3DPS();
    }
}